GPU-centric resolved-particle disperse two-phase flow simulation using the Physalis method

Adam J. Sierakowski*

*Department of Mechanical Engineering, Johns Hopkins University,
3400 North Charles Street, Baltimore, MD 21218, USA*

Abstract

We present work on a new implementation of the Physalis method for resolved-particle disperse two-phase flow simulations. We discuss specifically our GPU-centric programming model that avoids all device-host data communication during the simulation. Summarizing the details underlying the implementation of the Physalis method, we illustrate the application of two GPU-centric parallelization paradigms and record insights on how to best leverage the GPU's prioritization of bandwidth over latency. We perform a comparison of the computational efficiency between the current GPU-centric implementation and a legacy serial-CPU-optimized code and conclude that the GPU hardware accounts for run time improvements up to a factor of 60 by carefully normalizing the run times of both codes.

Keywords: computational fluid dynamics, GPU, particle, Physalis method

1. Introduction

Particulate disperse two-phase flows, where a solid phase is dispersed throughout a fluid (liquid or gas) phase, are found in many contexts in both nature and technology, including dust storms, beach erosion, avalanches, fluidized bed reactors, and many others. Due to the complexity of the interactions between the two phases, there remain many open questions regarding the evolution of such systems in time, and only limited capacity for accurately predicting their behavior. Further, the wide range of relevant length scales from the particle size (micrometer to millimeter scale) to the system size (from meter to kilometer scale) renders full-scale simulation of disperse two-phase flow systems impossible on even the most capable of the current crop of supercomputers.

There exist numerous methods that incorporate varying levels of simplification in order to approximate disperse two-phase flows at realistic scales [5], from two-fluid continuum models [13] to discrete particle models [7]. All of these methods rely on the approximate closure of a simplified set of governing equations, and it is the pursuit of more accurate closure models that drives the development of first-principles resolved-particle disperse two-phase flow simulation tools. Many such tools exist, including those of Kidanemariam and Uhlmann [17], Mehrabadi et al. [19], Yang and Stern [27], Ji et al. [14], Wylie et al. [26], Apte et al. [1], Zeng et al. [29], and Simeonov and Calantoni [25]. The Physalis method, first put forth for three-dimensional incompressible Navier-Stokes flow in Zhang and Prosperetti [30], introduces the influence of particles to the fluid using an analytic solution for flow about a sphere, and achieves highly accurate particle-fluid coupling.

*Correspondence:

Email address: sierakowski@jhu.edu (Adam J. Sierakowski)

Sierakowski and Prosperetti [23] discusses the Physalis method in detail, and the present work expands upon the theory by presenting some of the finer implementation details.

In many ways, this work began as an experiment in applying the massively parallel architecture afforded by today’s graphics processing units (GPUs) to resolved-particle disperse two-phase flow simulations. Researchers have worked on solving computational fluid dynamics (CFD) on GPUs since Harris [12], and the results have so far indicated that these devices will be an important part of CFD for the foreseeable future [15, 16]. A recent review by Niemeyer and Sung [20] presents an excellent introduction to the benefits, challenges, and past work related to CFD on GPUs; they conclude that there remains much room for growth via the improvement of the underlying numerical methods used for CFD in a massively parallel context. Some researchers have completed work on extending single-phase CFD to particle-resolved flows on GPUs [28], but none have adopted particle treatments that are as accurate or as computationally demanding as the Physalis method.

By carefully adapting the numerical methods underling the Physalis method to suit the GPU hardware, we realize computational speedups of nearly 60 times when comparing the GPU-centric code to a legacy implementation of the Physalis method on a serial CPU. Here we discuss the data structures, methods, and algorithms used to achieve this computational efficiency. In Section 2, we begin by presenting our application, particle-resolved disperse two phase flow using the Physalis method. We then discuss in Section 3 the desired device-host communication model that make possible the disperse two-phase flow simulations considered below: the strict adherence to GPU-centric parallelization, where we avoid all data communication between the host and device. Sections 4 and 5 detail the GPU-centric parallelization schemes that we use for the flow solver and Physalis method, respectively. In these sections, we identify two paradigms for GPU parallelization that are essential for efficient simulation of disperse two-phase flow: (i) algorithms that operate on a fixed Cartesian grid and (ii) algorithms that operate on disperse particles. These two paradigms may be generalized for use beyond the Physalis method, with applications in finite-differencing and particle-based numerical methods, respectively. In Section 6, we discuss in depth the computational efficiency of the GPU-centric code when compared to a legacy CPU code, and we summarize our conclusions in Section 7.

2. The Physalis method

To provide context for this work, we first introduce our application: particle-resolved disperse two-phase flow using the Physalis method. Consider a simulation in which n_p spherical particles of radius a and density ρ_p (not all necessarily the same) move about a domain, interacting with each other both through hydrodynamic forces due to an interstitial fluid as well as through direct contact. The particles may be acted upon by external forces such as gravity and may be subject to incident laminar or turbulent flow. We solve the incompressible Navier-Stokes equations

$$\partial_t \mathbf{U} + (\mathbf{U} \cdot \nabla) \mathbf{U} = -\frac{1}{\rho_f} \nabla p + \nu \nabla^2 \mathbf{U} + \mathbf{g}; \quad 0 = \nabla \cdot \mathbf{U}, \quad (1)$$

for fluid velocity $\mathbf{U}$, pressure p , and external force per unit mass $\mathbf{g}$ with density ρ_f and kinematic viscosity ν in the presence of the moving particles with no-slip surfaces. Regenerating the computational mesh to respect these internal boundaries as particles move can be prohibitively expensive, so we instead choose to operate on a fixed discrete grid and call specific attention to the geometric mismatch between the particle shape and the fixed grid. The Physalis method communicates the effect of each particle to the flow solver in order to overcome this geometric mismatch. We present only the most synthetic overview of the Physalis method in this work, deferring to Sierakowski and

Prosperetti [23], Zhang and Prosperetti [30], Gudmundsson and Prosperetti [11] to provide a more detailed description.

The flow solution procedure at each point in time generally proceeds as follows. Given the solution from the previous time step computed using a fairly standard second-order finite difference projection method on a fixed, staggered Cartesian grid [3, 8], we construct a *particle cage*—the subset of discrete flow solver cells at which a particle’s influence is communicated—based on each particle’s current position (refer to Figure 5). Each particle’s influence is communicated by analytically posing velocity and pressure boundary conditions— $\tilde{\mathbf{u}}$ and $\tilde{p}$, respectively, transformed to the particle rest frame from $\mathbf{U}$ and p —on the cage using *Lamb’s solution* to Stokes flow about a sphere, which takes the form

$$\tilde{\mathbf{u}}(t, r, \theta, \varphi) = \sum_{l=0}^{\infty} \sum_{m=-l}^l A_{lm}(t) \tilde{\mathbf{u}}_{lm}(r) Y_l^m(\theta, \varphi); \quad (2a)$$

$$\tilde{p}(t, r, \theta, \varphi) = \sum_{l=0}^{\infty} \sum_{m=-l}^l B_{lm}(t) \tilde{p}_{lm}(r) Y_l^m(\theta, \varphi), \quad (2b)$$

where Y_l^m are spherical harmonics, A_{lm} and B_{lm} are time-dependent *Lamb’s coefficients* that vary for each particle, and the functions $\tilde{\mathbf{u}}_{lm}$ and $\tilde{p}_{lm}$ are radially-dependent simple algebraic functions known analytically. Due to the no-slip boundary condition at the surface of each particle, Lamb’s solution increases in accuracy as the distance from the surface decreases. We solve the Navier-Stokes equations (1) according to these boundary conditions (2) and check that the two solutions match at their interface by calculating Lamb’s coefficients based on the most recent flow information using the scalar product method posed in Gudmundsson and Prosperetti [11], which takes the form of spherical surface integration,

$$(Y_l^m, \tilde{p}) = \int_{-\pi}^{\pi} d\varphi \int_0^{\pi} \sin \theta d\theta \bar{Y}_l^m(\theta, \varphi) \tilde{p}(\theta, \varphi), \quad (3)$$

where the overbar denotes complex conjugation. If the flow conditions are changing rapidly between subsequent time steps, the local Lamb’s solution is unlikely to properly match the global Navier-Stokes solution on the first attempt, and the process must be iterated until the two solutions converge. The iterative nature of the Physalis method requires multiple (typically one to ten) solutions of the Navier-Stokes equations (1) for every time step, rendering computational expediency a top priority. Upon convergence, the global flow field matches with each local particle flow field and we then integrate particle motion forward in time using the hydrodynamic, interaction, and any other forces present. We then proceed to the next time step and repeat the process.

Calculation of the hydrodynamic forces proceeds directly from Lamb’s coefficients using only a simple algebraic relation. The direct interaction forces come in the form of contact and lubrication forces dependent on the gap between two particles α and β with centers at $\mathbf{X}_{\alpha}$ and $\mathbf{X}_{\beta}$, respectively, given by

$$h = |\mathbf{X}_{\alpha} - \mathbf{X}_{\beta}| - a_{\alpha} - a_{\beta}. \quad (4)$$

One such interaction force, the force on particle α in the normal direction that connects the particle centers

$$\hat{\mathbf{n}} = \frac{\mathbf{X}_{\alpha} - \mathbf{X}_{\beta}}{|\mathbf{X}_{\alpha} - \mathbf{X}_{\beta}|}, \quad (5)$$

takes the form

$$\mathbf{F}_\alpha^i = \sum_{\beta=1}^{n_p} \left[-k_n (-h)^{3/2} - \eta_n \mathbf{w}_{\alpha\beta} \cdot \hat{\mathbf{n}} \right] \hat{\mathbf{n}} - 6\pi\mu a_\alpha \sum_{\beta=1}^{n_p} \left[\frac{a_{\beta\alpha}^2}{(1+a_{\beta\alpha})^2} \left(\frac{a_\alpha}{h} - \frac{a_\alpha}{\varepsilon} \right) + \frac{a_{\beta\alpha} (1+7a_{\beta\alpha}+a_{\beta\alpha}^2)}{5(1+a_{\beta\alpha})^3} \ln \left(\frac{\varepsilon}{h} \right) \right] (\mathbf{w}_{\alpha\beta} \cdot \hat{\mathbf{n}}) \hat{\mathbf{n}}, \quad (6)$$

where k_n and η_n are nonlinear material-dependent spring and damping constants, $a_{\beta\alpha} = a_\beta/a_\alpha$, and the relative velocity is $\mathbf{w}_{\alpha\beta} = \mathbf{w}_\alpha - \mathbf{w}_\beta$. The first term models the force that occurs as two particles contact [see 5] and the second term models the lubrication effect that we fail to resolve using our fixed finite difference grid [see 25] where $\varepsilon = a$ is a length used to equip the model with compact support to assist in the calculations of Section 5. In addition to this normally-directed force, we also model tangential interaction forces and couples by similar relations.

3. GPU-centric parallelization

Parallelizing an algorithm for GPU operation presents unique challenges when compared to other methods of parallelization, such as OpenMP or MPI, because of a GPU’s massively parallel architecture. As the names imply, the differences between a graphics processing unit (GPU) and a central processing unit (CPU) stem from the historical applications of each device. Generating graphics on a screen inherently requires the high-speed data-parallel calculation of many thousands to millions of independent pixels, which leads to the architecture we see today that prioritizes bandwidth over latency. Thus, making broad generalizations, CPU and GPU architectures differ primarily in the number and computational capability of logical cores and in the memory hierarchy. While a CPU will contain one to eighteen logical cores that each operate between 2 and 3 GHz, a modern GPU may be comprised of more than 5,000 logical cores that each run just under 1 GHz. This is made possible by dedicating resources to computations at the expense of resources typically used to control work flow. In order to enable each of these many logical cores to access data, the memory layout of a GPU is significantly more complex than that of a CPU, as logical cores can access a number of different memory spaces (global, shared, constant, texture, registers, e.g.) at vastly different speeds. As a result, writing code for a GPU often requires more development effort than code for a CPU, but given an appropriately data-parallel application, the resulting computations can run up to two orders of magnitude faster [10, 24].

When running a program on a GPU, the primary data resides in the dedicated onboard memory space—referred to as *global memory*—and it must be copied across the PCI Express system bus from the CPU memory. This system bus presents the largest hurdle for GPU computing as the transfer rate across it from the *host*—the CPU—to the *device*—the GPU—is limited to 0.5 GB/s (for PCI Express 2.0), which is a rate significantly smaller than the memory bandwidth of 288 GB/s (for an Nvidia Tesla K40) on the GPU.

Recognizing the severe limitation that device-host communication poses to the throughput of a GPU code, we take a *GPU-centric* approach to our program development: we seek to avoid all device-host data communication. We contrast this to a *GPU-accelerated* approach, which offloads the most data-parallel (or, often, the most convenient) pieces of a (legacy) CPU code onto a GPU. Under a GPU-accelerated approach, though a particular algorithm ported to the GPU may experience accelerations of up to 100 times, frequent device-host communication will often consume much of the time saved. By taking a GPU-centric approach, we avoid this device-host data communication completely and realize computational throughput much nearer the actual speed of the GPU hardware.

To solve a generic spatial-temporal partial differential equation (PDE) (we will focus on a finite-difference solution to the Navier-Stokes equations below), our desired GPU-centric work flow is:

1. Define the initial condition, boundary conditions, coefficients, and forcing in host memory;
2. Copy all data to the device from the host;
3. Solve the PDE exclusively using the GPU without moving any data across the system bus;
- 3'. If desired, copy intermediate solutions to the host from the device for output to file;
4. Copy the final solution to the host from the device for output to file.

Under this GPU-centric approach, the CPU orchestrates the operation of the GPU by configuring and launching *kernels*—GPU code—but it does not operate on the PDE data. After the initial problem specification, the PDE solution proceeds exclusively on the GPU without any device-host data communication. We draw specific attention to Step 3', which provides the ability to save intermediate solution time steps to file, though our GPU-centric work flow can proceed without any host-device data communication at all.

Compared to traditional programming methodologies, this GPU-centric approach represents a paradigm shift towards data-parallel operation. Recognizing that the design of the underlying computing hardware in many ways steers the choice of the numerical methods applied, we choose our methods carefully and deliberately to take full advantage of that hardware. For example, a GPU contains multiple *streaming multiprocessors*—essentially, groups of computational cores—that access thousands of *registers* used to schedule thousands of *threads* that access *shared memory* at significantly higher rates than the *global memory*, which is accessible by all cores from all streaming multiprocessors. The Cuda GPU programming interface, which we use exclusively in this work, mimics this structure by grouping threads into *threadblocks* when configuring a kernel call; each threadblock corresponds roughly to a streaming multiprocessor. This design introduces two important concerns for programming a GPU: (i) using shared memory efficiently and (ii) avoiding *thread divergence*, which occurs when the kernel being run by a streaming processor splits into different paths due to a conditional statement. Since all of the computational cores inside a streaming multiprocessor must perform the same operations simultaneously, thread divergence is particularly detrimental because the streaming multiprocessor reverts to completing the kernel in a serial manner.

This paper provides much of the implied reasoning behind the numerical methods put forth in Sierakowski and Prosperetti [23], as they lend themselves particularly well to GPU-centric operation. We have released our implementation of these methods, written in a combination of C and Cuda, as an open-source project available at <http://PhysalisCFD.org> [22]. We think of the current work as an intermediate step towards large-scale many-GPU computations that combines the traditional OpenMP and MPI parallelization methods, such as domain decomposition, at the higher levels with GPU-centric methods at the lower levels of computation in order to fully utilize the resources available on many current high performance computing systems.

4. Flow solver parallelization

The flow solver solves the incompressible Navier-Stokes equations (1) according to any combination of inhomogeneous Dirichlet, homogeneous Neumann, and periodic external velocity boundary conditions with possible inflow/outflow and pressure-gradient forcing on a Cartesian grid in a domain the shape of a right prism. A simulation runs in a GPU-centric manner on a single GPU; the 12-GB memory space available on an Nvidia Tesla K40 allows for up to $298^3 \approx 26.5 \times 10^6$ discrete

cells (the CUSP sparse matrix library [6] used to solve the pressure-Poisson problem requires a 100% memory overhead).

The methodology put forth in this section, especially the ideas behind GPU parallelization on a fixed grid discussed in Section 4.1, may be generalized to the solution of any PDE on a fixed grid. Section 4.2 describes the benefits of the use of ghost cells for assisting with computations near boundaries and Section 4.3 introduces a method for joining simulations in two separate GPUs using MPI communication.

4.1. Finite differences

Let us consider specifically one subset of data-parallel algorithms required for solving a generic PDE, such as (1). The PDE may contain a spatial derivative such as the convective term of (1),

$$[(\mathbf{U} \cdot \nabla) \mathbf{U}] \cdot \mathbf{e}_x = (\nabla \mathbf{U} \mathbf{U}) \cdot \mathbf{e}_x = \frac{\partial(UU)}{\partial x} + \frac{\partial(UV)}{\partial y} + \frac{\partial(UW)}{\partial z}, \quad (7)$$

which we discretize as

$$\begin{aligned} [(\nabla \mathbf{U} \mathbf{U}) \cdot \mathbf{e}_x]_{i,j,k} &\approx \frac{(U_{i+1,j,k} + U_{i,j,k})^2 - (U_{i,j,k} + U_{i-1,j,k})^2}{4\Delta x} \\ &+ \frac{(U_{i,j+1,k} + U_{i,j,k})(V_{i,j+1,k} + V_{i-1,j+1,k}) - (U_{i,j,k} + U_{i,j-1,k})(V_{i,j,k} + V_{i-1,j,k})}{4\Delta y} \\ &+ \frac{(U_{i,j,k+1} + U_{i,j,k})(W_{i,j,k+1} + W_{i-1,j,k+1}) - (U_{i,j,k} + U_{i,j,k-1})(W_{i,j,k} + W_{i-1,j,k})}{4\Delta z} \end{aligned} \quad (8)$$

in Cartesian coordinates, operating on vector field variable (velocity) $\mathbf{U} = (U, V, W)$, where

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} + \begin{pmatrix} i \\ j \\ k \end{pmatrix} \cdot \begin{pmatrix} \Delta x \\ \Delta y \\ \Delta z \end{pmatrix}, \quad (9)$$

with (x_0, y_0, z_0) a reference corner of the domain, (i, j, k) integer-valued discretization indices, and $(\Delta x, \Delta y, \Delta z)$ discretization lengths in each direction. Note that (8) is formulated to account for interpolation required by our use of a staggered grid flow solver.

To compute the value of the derivative at each discrete point in space, $[(\nabla \mathbf{U} \mathbf{U}) \cdot \mathbf{e}_x]_{i,j,k}$, we must visit each point and use its value and those of the surrounding points from three planes of U ($i+1$, i , and $i-1$) and two planes of V (i and $i-1$) and W (i and $i-1$). The planes and their index references are depicted in Figure 1. This operation lends itself to one obvious GPU parallelization scheme: dedicate one GPU thread to each discrete point and compute the derivative by accessing the data of the neighboring points using each thread.

The GPU hardware permits the implementation of this general idea, but it requires a slightly more complex approach, especially since GPU shared memory proves valuable in algorithms that access the same memory many times. The maximum allowable sizes of threadblocks, as well as the availability of registers and shared memory, prevents us from implementing the simple kernel configuration of breaking the domain up into three-dimensional threadblocks. Instead, we use two-dimensional threadblocks in the manner depicted in Figure 2:

1. Choose a predominant direction in which to perform the computation;
2. Divide the domain into two-dimensional blocks of threads (we use 16×16) in a plane normal to this predominant direction;
3. Launch the kernel that computes the derivative;

Figure 1: The staggered-grid velocity field discretization used in calculating finite differences on the GPU. The highlighted arrow in the center is $U_{i,j,k}$, the location at which the stencil is centered. Note that many internal vectors have been removed for clarity.

- (a) Slide along the predominant direction;
- (b) Copy the appropriate planes of U , V , and W to shared memory from global memory;
- (c) Compute (8);

We find that, by choosing a predominant direction and sliding the two-dimensional data-parallel plane along it, we efficiently use the available GPU hardware. The 16×16 threadblock size strikes a good balance between hardware utilization and (register and shared memory) resource availability. This scheme represents the first of two GPU parallelization paradigms essential to this work, namely, GPU parallelization on a fixed grid.

For the convective term, as well as other differential operators such as Laplacians and gradients, using shared memory expedites the computations since (8) indicates that each discrete data point will be read up to 15 times by 15 different threads. In Figure 3, we see that the ratio of the duration of the kernel run time for the global memory kernel to that of the shared memory kernel for this operation is greater than 2.5 for the largest resolution. This ratio increases initially as the problem grows to fill the GPU hardware until $N = 128$ but then stops increasing as the GPU has become fully saturated with work. Note that using shared memory is not always the best choice: a global memory read is required to copy data to shared memory, and it becomes valuable only when reading data multiple times. We will apply this basic scheme to various operations in the flow solver parallelization below.

Figure 2: Laplacian grid GPU parallelization scheme. Thread blocks are denoted by solid colors, with overlapping shared memory ghost cells denoted by combined colors. This data-parallel plane slides along the third Cartesian direction inside the GPU kernel.

4.2. External boundary conditions

In order to avoid thread divergence, we take care to ensure that all threads perform the same operations at all grid nodes. When a finite difference stencil, e.g. the convective derivative (8), must be computed at the domain boundary, such as the location $(i = 0, j, k)$, the code should behave differently because the data in the $(i - 1)$ position does not exist and therefore cannot be read. As such, we use ghost cells to simplify and standardize our differential operator calculations, which buffers the field variable data structure with extra data points across the boundary so that the stencil may be computed on the boundary using the same set of operations as at every other point in the domain. Our method generally proceeds by performing the finite-difference operations required everywhere in space and then overwriting the results on the boundaries and in the ghost cells in order to adhere to the appropriate boundary conditions. These functions that enforce the boundary conditions follow the general parallelization scheme described above but work only on the boundary plane of data without sliding across the entire domain.

4.3. Concurrent precursor

In conjunction with a typical inflow/outflow simulation with Dirichlet velocity boundary conditions on the inflow—e.g. $\mathbf{U} = (\bar{U}, 0, 0)$ —Neumann conditions on the outflow, and periodic

Figure 3: The runtime and speedup associated with using shared memory to compute the convective derivative (8) for various domain resolutions (the total number of discrete points is N^3).

conditions on the four remaining boundaries, we have implemented a mechanism for introducing turbulent perturbations $\mathbf{U}'$ to the inflow. We generate the turbulent perturbations by concurrently simulating triply-periodic homogeneous isotropic turbulence on a separate GPU (or, if desired, on the same GPU at the expense of simulation size) and sampling the velocity field from it. In this domain, the turbulence is maintained using the linear forcing of Rosales and Meneveau [21] with an additional enhancement provided by Carroll and Blanquart [4]. We refer to this homogeneous isotropic turbulence simulation as the *precursor domain* and the inflow/outflow simulation as the *experimental domain*. Using a plane parallel to the inflow boundary of the experimental domain called the *perturbation plane*, we slide across the precursor domain at the desired inflow velocity in such a way to introduce the mean flow—e.g. $\mathbf{U} = (-\bar{U}, 0, 0)$ —and interpolate the precursor domain velocity field to the perturbation plane [see 2]. We use the turbulent perturbations, which are now combined with the mean inflow velocity, as the inflow Dirichlet boundary conditions—e.g. $\mathbf{U} = (\bar{U} + U', V', W')$. Figure 4 illustrates this mechanism. The two simulations proceed together in time using identical time step sizes. The maximum Taylor-scale Reynolds number $\text{Re}_\lambda = \lambda |\mathbf{U}'| / \nu$ that we have successfully simulated in this way is $\text{Re}_\lambda \approx 100$ on a grid of size 298^3 .

In order to enforce the precursor turbulent perturbations on the experimental inflow velocity, we communicate between the two GPUs using the following procedure:

1. Interpolate the precursor velocity field to the perturbation plane;
2. Copy the perturbation plane to the precursor host from the precursor device;
3. Using MPI, communicate the contents of the perturbation plane to the experimental host;
4. Copy the perturbation plane to the experimental device from the experimental host;
5. Apply the perturbation plane to the inflow boundary condition.

In order to best optimize the code by controlling the timing of data transfers and synchronizing them with computations, we perform these steps explicitly instead of relying on Cuda unified memory, which limits precise control.

As the turbulence is maintained by adding a forcing term to the Navier-Stokes equations (1) that is proportional to the turbulent perturbations about the mean velocity, the method cannot handle the presence of particles (to be discussed in the following section) as their influence on the fluid is unnaturally amplified by the linear forcing. Due to this limitation, Physalis particles may only be introduced into the experimental domain, where simulations often take the form of decaying turbulence, as if in a wind tunnel with decaying grid-induced turbulence.

Figure 4: The concurrent precursor. The plane cutting through the homogeneous isotropic turbulence precursor simulation translates through the domain at the velocity of the mean inflow for the experimental domain. The turbulent velocity field from the precursor plane provides turbulent fluctuations about the mean flow for the inlet to the experimental domain via MPI communication between two GPUs.

5. Physalis method parallelization

In addition to storing the particle data such as material properties and Lamb’s coefficients in a Lagrangian sense, we maintain five Eulerian field variables spanning the entire domain that assist with applying particle boundary conditions while avoiding thread divergence (see Section 5.2). The first, **phase**, is a cell-centered phase mask where the value is set to either the particle number if the center of the cell is contained within a particle and -1 otherwise. This field is not meant to reconstruct the particle volume perfectly, but instead to denote which fluid cells are contained within the particle cage. A related field, **phase_shell**, is cell-centered and has a value of 0 everywhere except for the outermost set of cells in **phase**, where it is set to 1. This field specifies the location of the pressure component of the particle cage. The remaining three fields constitute a vector field denoting the locations of the velocity components of the cage. Each of these face-centered components, **flag_{u,v,w}**, is set equal to 1 everywhere except for the locations of the velocity components of the cage, where it is set equal to -1 (it is also used for specifying external domain boundary conditions, where it is set to 0).

As currently implemented, the number of particles n_p in a simulation remains constant for the duration of the simulation. Particles may cross periodic boundaries, but they cannot enter or leave the domain through Dirichlet or Neumann boundaries. Further, the particle translation and rotation flags enable the user to define whether a particle may move freely about the domain, move without rotation, rotate without moving, or remain fixed in space.

This section focuses specifically on the Physalis method and also provides in Sections 5.1 and 5.2 examples of the general grid parallelization paradigm discussed above. Section 5.3 discusses our GPU parallelization of the Lebedev quadrature integration scheme. Finally, Section 5.4 introduces the second GPU parallelization paradigm, *particle parallelization*, which may be generalized to any

particle-based numerical method.

5.1. Particle cage construction

At the beginning of every time step, we reconstruct the particle cages—that is, we fill in the `phase`, `phase_shell`, and `flag_{u,v,w}` fields—in order to update their shapes according to the new positions of the particles. Figure 5b illustrates the locations denoted by `phase_shell` and `flag_{u,v,w}` as dots and arrows, respectively. As governed by the time step size, particles move, at most, a fraction of a discrete cell at any time step, so that it is not strictly necessary to reconstruct the cages at every time step. However, we choose to perform the reconstruction at every time step in order to limit the magnitude of the force fluctuations that appear as a result of changes in the cage shape. While it may seem appropriate to reconstruct only the cages of particles that have moved and to reuse the previous cages of the remaining particles, we draw particular attention to the deficiency underlying this idea in a more direct setting in Section 5.2.

Figure 5: An example of the Physalis particle cage that specifies the locations of the analytic boundary conditions.

We reproduce here the cage generation algorithm as described in Sierakowski and Prosperetti [23] with some additional annotations:

1. Mark all cells of `phase` with cell centers inside the particle surface with the particle number (and -1 if fluid); the outermost shell of these cells, `phase_shell`, constitutes the set of cage cells;
2. The centers of the `phase_shell` cage cells, marked with 0 if fluid and 1 otherwise, are the locations at which the Dirichlet conditions for the pressure-Poisson equation are imposed;
3. All faces of the cage cells `flag_{u,v,w}` with an outward normal with respect to the particle are part of the set of points where the velocity boundary conditions are imposed, and are marked with -1 (and 1 otherwise);
4. The set of velocity nodes is completed by including the centers of all faces common to two neighboring cage cells by marking them with -1.

Thus, reconstructing the cages requires one sweep of the domain for each cage field variable, and then one extra for each of the `flag_{u,v,w}` in order to complete Step 4. This reconstruction technique, being fully GPU-parallelized using the grid parallelization scheme described in Section 4.1,

typically constitutes less than 2% of the total runtime and significantly reduces force fluctuations when applied at every time step.

5.2. Internal boundary condition application

Though unintuitive at first glance, the prioritization of bandwidth over latency in GPU hardware means that we can sometimes achieve faster performance overall by doing a significant amount of extra work. Generally speaking, when performing a serial (or lightly parallelized) calculation, limiting the amount of work to be done is almost always the correct choice. When performing a GPU (or massively parallelized) calculation, however, this is not always the correct answer because it may lead to underutilization of the hardware or strong thread divergence thereby increasing overall runtime.

When applying the particle boundary conditions (2) at the locations designated by the cage with `phase_shell` and `flag_{u,v,w}`, we find that it is an inefficient use of the GPU hardware to limit the work to be done by updating only a subset of the flow velocity and pressure nodes denoted by the cages. Our boundary condition application algorithm applies the grid parallelization scheme of Section 4.1 instead of the more obvious, but less efficient, particle parallelization scheme described below. The particle boundary condition application works in this way:

1. For each pressure and velocity field component, visit each grid cell in the manner of the grid parallelization scheme;
2. For each thread, upon arriving at a given cell, determine to which particle the cell belongs (if any) using `phase`;
3. In a temporary location, calculate the boundary condition to be applied using (2) depending on the particle number and, if the cell is a fluid cell, perform the same calculations using a dummy particle;
4. Write the temporary value to the cell only if the cell is a cage node, according to `phase_shell` and `flag_{u,v,w}`.

By applying the grid parallelization scheme and visiting all velocity cells, we achieve a faster overall boundary condition application algorithm despite doing a significant amount of work that is ultimately discarded because we fully utilize the GPU hardware and avoid thread divergence.

5.3. Lebedev quadrature scalar products

Lamb's coefficients are key to coupling between the particle and flow solutions. They determine the velocity and pressure boundary conditions (2) applied at the cage, which the flow solution must obey. In turn, Lamb's coefficients contain information derived from the flow solution, and we determine their values through scalar products (surface integrals) of the form (3).

There exist many ways of performing the required integrations, and we have settled on a 26-point Lebedev quadrature rule [18] that exactly integrates spherical harmonics up to order seven of the form

$$\frac{1}{4\pi} \int_{-\pi}^{\pi} d\varphi \int_0^{\pi} \sin \theta d\theta f(\theta, \varphi) \approx A \sum_{i=1}^{26} f(a_i), \quad (10)$$

with node weights A and positions a_i . Figure 6 illustrates the positions of the quadrature nodes relative to a particle. This method is superior to many other spherical surface integration methods because it relies on a relatively small number of evenly spaced quadrature nodes on the integration surface and avoids the common problem of integration mesh singularities.

Here, we discuss our method of parallelizing Lebedev quadrature on a GPU. The quadrature nodes, attached to the particle, are free to reside at any location in space, while the flow data

Figure 6: The Lebedev quadrature node positions associated with a particle. The arrows indicate the three Cartesian directions.

that it is integrating appears at discrete locations according to the flow solver grid. The first step, therefore, is to interpolate the flow data to the quadrature nodes. Given 26 nodes per n_p particles, we configure our first kernel call with n_p threadblocks of 26 threads each. For each thread, representing one quadrature node of one particle, we interpolate from the discrete field to the quadrature node using tri-linear interpolation and store the values in a temporary array.

In a subsequent step, we calculate (10) for six different integrands, representing the three sets of complex-valued Lamb's coefficients. Given a truncation order $0 < L < 4$ for the reconstruction of Lamb's solution from the coefficients, each set requires $M = \frac{1}{2}(L+1)(L+2)$ coefficients, so we have $6M$ total coefficients to calculate for each particle (though some are identically zero). We change our kernel configuration to have a two-dimensional threadblock of size $(n_p \times 6M)$ of 26 threads each. The kernel call then proceeds as follows:

1. Calculate at a node the integrand of (10) [given in 23];
2. Ensure that all threads have completed their work;
3. Use the zeroth thread (associated with node zero) in each block to perform the sum (10);
4. Use the zeroth thread of the second dimension of the threadblock (the $6M$ coefficients) to manipulate the surface integrals as put forth in Sierakowski and Prosperetti [23] to yield Lamb's coefficients.

As the number of items to sum is small in all cases, we perform the sums directly in the kernel using a subset of the available threads because it forgoes the overhead associated with a parallel reduction.

5.4. Force & couple calculation and trajectory integration

Once we have obtained Lamb's coefficients, the hydrodynamic force $\mathbf{F}_h$ and couple $\mathbf{L}_h$ are computed directly from the A_{lm} and B_{lm} . We parallelize this computation by assigning a thread to each particle and computing $\mathbf{F}_h$ and $\mathbf{L}_h$ independent of the others. This represents an application of our second essential GPU parallelization paradigm, namely, GPU parallelization across particles.

The calculation of the interaction forces such as (6) requires more attention, however. Both the lubrication and contact models require the calculation of the $O(n_p^2)$ particle-particle interaction n-body problem as all pairs must interact. Relying on the compact support of (6), we can ignore the interaction between α and other particles that are far away. In this way, we operate only on the neighbors of α within $\varepsilon \sim a_\alpha$ and significantly decrease the total number of required interactions.

Figure 7: A two-dimensional abstraction of the nearest-neighbors interaction algorithm. The red particle must interact with all particles within ε (dashed radius), and the algorithm selects those in green. The red box contains the red particle's center, and it must be interacted with all particles whose centers fall inside the eight neighboring boxes (26 in three dimensions), highlighted in green.

We implement a GPU-centric n-body algorithm that relies on a spatial partitioning of the domain to generate lists of particles near enough to interact with each other (this is similar to a sample code distributed with the Nvidia Cuda Toolkit). We recognize this as an advanced application of our second GPU parallelization paradigm. It is important to draw attention to the wealth of research on the topic of n-body simulation of which the method described below is but an example. For more advanced and/or specialized methods, see, e.g., Fortin et al. [9].

Consider the situation depicted in Figure 7 with 64 particles with finite radius a distributed throughout a domain, uniquely numbered zero through 63. Taking ε to be the compact support length beyond which (6) may be safely ignored, we divide the domain into segments called *bins* of size $2a + \varepsilon$, which represents the minimum distance required to ensure particles residing in two adjacent bins properly interact. We set up an empty list of length n_p that will associate bin numbers with the particles they contain. We then proceed as follows:

1. Thread over all particles to determine in which bin the center of each particle resides;
2. Carrying along the associated particle number, sort the particles by bin index using a GPU sorting algorithm;
3. Thread over all bins to determine the first and last particle in each bin (to assist with parallelization);

4. Thread over all particles to compute (6);
 - (a) Loop over all particles residing in the same bin as the particle in question;
 - (b) Loop over all particles residing in the 26 neighboring bins;
 - (c) Sum the influence of all neighboring particles.

We draw particular attention to Step 2, which is required to sidestep race conditions resulting from the simultaneous placement of multiple particles each bin. Following this process, we observe $O(n \log n)$ complexity (due to the bin sorting algorithm), significantly improving upon the $O(n^2)$ all-pairs algorithm. Figure 8 illustrates the superiority of the spatial partitioning algorithm by varying the number of randomly positioned particles in a domain of constant size. The dashed lines, which plot the isolated kernel run times, show that the spatial partitioning method significantly improves upon the run time of the all-pairs method, with gains recognized at as few as $n_p = 8$ and a factor of nearly 10,000 improvement for $n_p = 4096$. The solid lines indicate that for simulations with more than approximately $n_p = 500$, the brute-force all-pairs calculation is more time consuming than the solution of the pressure-Poisson problem, and that the spatial partitioning method yields more than a factor of 10 improvement for the overall simulation with $n_p = 4096$. We can see from the plot that the space-partitioned n-body implementation consistently accounts for less than 0.1% of the total calculation time. Further, we see that this method appears to perform slightly better than the expected $O(n \log n)$ complexity, presumably because the GPU spends, proportionally, less time launching kernels and more time computing as more particles are added. Using the space-partitioned particle-based operation scheme, in contrast to the cage generation and boundary condition application examples above, we find a situation where limiting the total amount of work does indeed pay off and significantly improves the overall performance, but only because we are able to saturate the GPU hardware with computations.

Figure 8: A comparison of the spatial partitioning and all-pairs algorithms by varying the number of randomly placed particles in a domain of constant size. The computation times for both the n-body function alone and the total simulation are shown. Overlaying the plots are dashed lines representing $O(n^2)$ and $O(n \log n)$ scaling.

6. Computational efficiency

In order to discuss the computational efficiency of this GPU-centric implementation of the Physalis method, we compare it to the best available alternative: a legacy implementation of the

method. This legacy implementation, which follows the approach of Zhang and Prosperetti [30] with the scalar product improvements of Gudmundsson and Prosperetti [11], is a combination Fortran and C++ code optimized for serial CPU operation. We recognize the myriad limitations underlying comparisons of GPU-optimized codes with serial-CPU-optimized codes, but we remain convinced of the value of this study provided we sufficiently control for those limitations.

It is important to note that there are more differences between the GPU and the CPU codes than simply the hardware on which they run, as each code is optimized for its respective hardware. In addition to low-level differences such as choosing underlying data structures best suited for each programming model, there also exist significant differences in the high-level numerical methods employed in each code. Sierakowski and Prosperetti [23] describes these differences and explains the rationale behind them. As such, we perform a series of simulations controlled to make the comparisons as fair as possible.

We perform ten simulations of various size. Each simulation uses the following conditions:

- Periodic boundary conditions in all three directions of a box of size L^3 ;
- Initially quiescent flow driven by a normalized pressure gradient $a^3\mu^{-1}\nu^{-1}\nabla p \cdot \mathbf{e}_z = 10$;
- Fixed particles (no translation or rotation) distributed on a uniform grid with resolution $a/\Delta x = 8$.

We vary the flow domain resolution $(L/\Delta x)^3$ from 32^3 to 256^3 by powers of two, with arrays of one, eight, 64, and 512 particles for each, if they fit into the domain with at least a particle diameter separation. For example, a 32^3 domain can only fit one particle at resolution $a/\Delta x = 8$, so we perform only one such simulation. In the 256^3 domain, however, we can fit up to 512 particles, so we perform simulations with all four sets of particles. For all simulations, the particles are distributed in such a way that they evenly fill the entire domain; the 256^3 simulation with 512 particles looks exactly the same as if we repeated the 32^3 simulation with one particle and tiled it eight times in each of the Cartesian directions. Four of these configurations are illustrated in Figure 9a.

We run the simulations for four time steps using each code with a time step size of $\Delta t \approx 0.0013$, which corresponds to a CFL number of approximately 0.5 [8]. We limited the number of time steps to four because of the time required to run the largest cases on the CPU. We consider the average timing results in Table 1, where we denote each simulation with a number and a letter—C for CPU code and G for GPU code. Given the differences between the numerical methods underlying each code, we record the number of iterations required for Lamb’s coefficient convergence N_i and report averages for the runtime of each code as both iteration $\bar{t}_i$ and time step $\bar{t}_s$ averages. For the GPU code, we list the average time for initial condition and intermediate solution data transfers $\bar{t}_D$ in addition to the total run time t_T . The data transfers happen once in each direction during t_T , which, with only four time steps, over-emphasizes their affect as we typically require the storage of intermediate time steps as infrequently as every 10 to 100 time steps, depending on the simulation. The final column of Table 1 lists the ratio of the run time for the CPU to the run time for the GPU to provide an estimate of their relative computational efficiency.

Every simulation is run on an in-house development cluster with the following relevant specifications:

- Intel Xeon E5-2650v2 CPU;
- DDR3 1866 MHz RAM;
- Nvidia Tesla K40 GPU.

(a) A representation of the ten simulations used for comparison. Each simulation varies in resolution along each side and in number of particles.

(b) The average iteration time $\bar{t}_i$ and CPU-to-GPU ratio listed in Table 1.

Figure 9: The comparison between the current work, a GPU-centric implementation on a single GPU, and a legacy serial CPU implementation of the Physalis method.

Both codes run with the same numerical accuracy convergence parameters, and as shown in Sierakowski and Prosperetti [23], the GPU code is at least as accurate as the CPU code, and is even superior in some measures. The GPU simulations use Bluebottle version 2.0.2 [22].

We present in Table 1 two different measures of efficiency: ratios of total run time t_T and ratios of average Lamb’s coefficient iteration run time $\bar{t}_i$. The first measure provides an estimate of the real value of the GPU code as a whole by summarizing the true run time improvement between the two codes. Generally speaking, the GPU code is always faster, except for Case 1 where the CPU code runs nearly three times faster. This is because the problem is too small for the GPU’s bandwidth to make up for its latency, *i.e.* time spent configuring and calling kernels. We see a trend develop as the GPU code becomes progressively faster as the problem size grows. We can see from Table 1 that this trend becomes quite significant: the GPU code runs over 150 times faster than the CPU code for the largest case tested (Case 10). We recognize, however, that this metric fails to separate the differences in the numerical methods from those in the hardware. Specifically, we see that although the two codes require approximately the same number of Lamb’s iterations to find a solution in the smaller cases, as the cases grow in both resolution and in the number of particles, the CPU code requires significantly more iterations for convergence.

We can obviously attribute some of this factor of 150 to this major difference in the number of Lamb’s iterations, so we also compute the ratio for the average Lamb’s coefficient iteration run time $\bar{t}_i$ (plotted in Figure 9b). This metric provides a fairer comparison between the two codes and concludes that the GPU code runs over 60 times faster than the CPU code for Case 10. We should be clear that this metric still includes some differences in numerical methods. To highlight one such difference, notice that the run time per iteration actually decreases as the number of particles increases for a given simulation resolution. This can be attributed to our changing from using a fast Fourier solver in the CPU code to a conjugate gradient solver for the pressure-Poisson problem with Dirichlet pressure particle boundary conditions in the GPU code. We discuss this change extensively in Sierakowski and Prosperetti [23] but, put simply, the pressure-Poisson problem becomes easier

case	$(\frac{L}{\Delta x})^{\frac{1}{3}}$	n_p	timing (s)		averages (s)			ratios (C/G)	
			$\bar{t}_D$	t_T	N_i	$\bar{t}_i$	$\bar{t}_s$	t_T	$\bar{t}_i$
1C	32	1	–	6.18	23	0.269	1.55	0.351	0.367
1G			0.0922	17.6	24	0.733	4.40		
2C	64	1	–	53.7	23	2.33	13.4	2.59	2.14
2G			0.151	20.7	19	1.09	5.19		
3C	64	8	–	67.6	24	2.82	16.9	3.57	3.27
3G			0.165	19.0	22	0.862	4.74		
4C	128	1	–	636	23	27.6	159	12.5	9.24
4G			0.715	50.9	17	2.99	12.7		
5C	128	8	–	690	24	28.8	173	15.5	12.3
5G			0.722	44.4	18	2.34	11.1		
6C	128	64	–	1470	47	31.3	368	37.8	14.5
6G			0.732	39.0	18	2.17	9.75		
7C	256	1	–	7760	23	338	1940	45.1	33.3
7G			2.91	172	17	10.1	43.1		
8C	256	8	–	8290	24	346	2070	63.9	45.3
8G			3.13	130	17	7.63	32.4		
9C	256	64	–	14100	41	345	3530	125	54.9
9G			3.38	113	18	6.27	28.2		
10C	256	512	–	21900	47	466	5480	157	63.3
10G			3.58	133	19	7.36	34.9		

Table 1: A summary of the results of the computational efficiency comparison between the current GPU-centric implementation and the legacy serial CPU implementation of the Physalis method. The case number denotes whether the simulation was performed using the CPU (C) or the GPU (G). Here, $\frac{L}{\Delta x}$ is the resolution on each side of the domain, n_p the number of particles, $\bar{t}_D$ the average data transfer time, t_T the total runtime, N_i the number of Lamb’s iterations for all four time steps, and $\bar{t}_i$ the time for one Lamb’s iteration.

to solve as more particles are added (e.g., the solver takes only 28% of the number of conjugate gradient iterations for Case 10G compared to Case 7G).

We recognize the imperfections underlying these comparisons, but maintain that these two measures provide insight into the value of the GPU code. In terms of the real-world significance of the new code presented here compared to the legacy serial-CPU implementation, we see a run time improvement of a factor up to 150. A more fair, but still imperfect, comparison of the average Lamb’s iteration time, which represents one complete unit of calculations performed by the code, yields a run time improvement of a factor up to 60. As this measure removes as many of the differences between the numerical methods employed by each code as possible without damaging their respective hardware optimizations, we conclude that this factor of 60 is the most reliable benchmark at our disposal. This comparison indicates the high potential value of combining the GPU-centric code with traditional OpenMP/MPI parallelization methods, as the lowest-level computations performed on the GPU could run up to 60 times faster than on current OpenMP/MPI codes. This approach has the potential to significantly improve the overall run time of the code as a whole, although device-to-host, MPI, and host-to-device communications are likely to consume a

significant piece of this run time improvement in traditional OpenMP/MPI parallelization methods.

7. Summary and conclusions

We have presented work on a new implementation of the Physalis method for resolved-particle disperse two-phase flow simulations. In Section 2 we gave an overview of the Physalis method and in Section 3 we discussed specifically our GPU-centric programming model, where we avoid all device-host data communication during the simulation. Sections 4 and 5 detailed our GPU-centric algorithms, including two GPU parallelization paradigms essential to the efficient operation of the code: grid and particle parallelization schemes. We illustrated the application of these GPU-centric parallelization paradigms while recording insights about how to best leverage the GPU's prioritization of bandwidth over latency. We performed a comparison between the computational efficiency of the current implementation and a legacy serial-CPU-optimized code in Section 6 and discussed its relevance while acknowledging the limitations underlying such a comparison. When normalizing for the differences between the two codes as well as possible, we find that the GPU hardware runs the simulation up to 60 times faster than the serial-CPU hardware.

We believe that this work shows the relevance of GPU hardware for resolved-particle two-phase flow simulation, and even for general-purpose computational fluid dynamics when utilized well. This technology provides the ability to carry out larger simulations with more particles in less time, which is critical to the future development of improved closure models for reduced-order modeling of disperse two-phase flow systems of realistic scale. By further combining this GPU-centric methodology with traditional OpenMP and MPI domain decomposition methods to combine multiple GPUs for the simulation of disperse two-phase flow systems, we expect that the future holds the ability to simulate hundreds of thousands of resolved particles in fluid flows in limited computational time.

Acknowledgements

The author wishes to acknowledge Prof. Andrea Prosperetti as the driving force behind the theory of the Physalis method. Further, the author wishes to thank Dr. Claire VerHulst for her insights during the development of this work, as well as Daniel Willen for his assistance in implementing the n-body space partitioned particle interaction scheme, and Dr. Laura Lukassen and Yayun Wang for their help testing the code.

This work was supported by the US National Science Foundation under grants CBET1258398 and CBET1335965. The author further acknowledges support from the Johns Hopkins University Modeling Complex Systems IGERT Fellowship Program, under National Science Foundation grant DGE0801471.

References

- [1] Apte, S. V., Martin, M., Patankar, N. A., 2009. A numerical method for fully resolved simulation (FRS) of rigid particle-flow interactions in complex flows. *J. Comput. Phys.* 228 (8), 2712–2738.
- [2] Botto, L., Prosperetti, A., 2012. A fully resolved numerical simulation of turbulent flow past one or several spherical particles. *Phys. Fluids* 24 (1).

- [3] Brown, D. L., Cortez, R., Minion, M. L., 2001. Accurate projection methods for the incompressible Navier-Stokes equations. *J. Comput. Phys.* 168 (2), 464–499.
- [4] Carroll, P. L., Blanquart, G., 2013. A proposed modification to lundgren’s physical space velocity forcing method for isotropic turbulence. *Phys. Fluids* 25 (10).
URL <http://scitation.aip.org/content/aip/journal/pof2/25/10/10.1063/1.4826315>
- [5] Crowe, C. T., Schwarzkopf, J. D., Sommerfeld, M., Tsuji, Y., 2012. *Multiphase Flows with Droplets and Particles*, 2nd Edition. CRC Press, Boca Raton, USA.
- [6] Dalton, S., Bell, N., Olson, L., Garland, M., 2014. Cusp: Generic parallel algorithms for sparse matrix and graph computations. Version 0.5.0.
URL <http://cusplibrary.github.io/>
- [7] Deen, N. G., Annaland, M. V. S., der Hoef, M. A. V., Kuipers, J. A. M., 2007. Review of discrete particle modeling of fluidized beds. *Chem. Eng. Sci.* 62 (1-2), 28–44, fluidized Bed Applications.
- [8] Ferziger, J. H., Perić, M., 2002. *Computational Methods for Fluid Dynamics*, 3rd Edition. Springer, Berlin.
- [9] Fortin, P., Athanassoula, E., Lambert, J.-C., July 2011. Comparisons of different codes for galactic n-body simulations. *Astron. Astrophys.* 531, A120.
- [10] Govindaraju, N., Lloyd, B., Dotsenko, Y., Smith, B., Manferdelli, J., 2008. High performance discrete fourier transforms on graphics processors. In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE, pp. 1–12.
- [11] Gudmundsson, K., Prosperetti, A., 2013. Improved procedure for the computation of Lamb’s coefficients in the physalis method for particle simulation. *J. Comput. Phys.* 234, 44–59.
- [12] Harris, M. J., 2004. Chapter 38: Fast fluid dynamics simulation on the GPU. In: *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Pearson Higher Education, New Jersey, USA.
- [13] Jackson, R., 2000. *The Dynamics of Fluidized Particles*. Cambridge University Press, Cambridge.
- [14] Ji, C., Munjiza, A., Avital, E., Xu, D., Williams, J., May 2014. Saltation of particles in turbulent channel flow. *Phys. Rev. E* 89, 052202.
- [15] Khajeh-Saeed, A., Perot, J. B., 2012. Computational fluid dynamics simulations using many graphics processors. *Comput. Sci. Eng.* 14 (3), 10–19.
- [16] Khajeh-Saeed, A., Perot, J. B., 2013. Direct numerical simulation of turbulence using GPU accelerated supercomputers. *J. Comput. Phys.* 235 (0), 241–257.
- [17] Kidanemariam, A. G., Uhlmann, M., 7 2014. Direct numerical simulation of pattern formation in subaqueous sediment. *J. Fluid Mech.* 750.
- [18] Lebedev, V. I., 1976. Quadratures on a sphere. *USSR Comput. Math. Math. Phys.* 16 (2), 10–24.

- [19] Mehrabadi, M., Tenneti, S., Garg, R., Subramaniam, S., 5 2015. Pseudo-turbulent gas-phase velocity fluctuations in homogeneous gassolid flow: fixed particle assemblies and freely evolving suspensions. *J. Fluid Mech.* 770, 210–246.
- [20] Niemeyer, K. E., Sung, C. J., 2014. Recent progress and challenges in exploiting graphics processors in computational fluid dynamics. *J. Supercomput.* 67 (2), 528–564.
- [21] Rosales, C., Meneveau, C., 2005. Linear forcing in numerical simulations of isotropic turbulence: Physical space implementations and convergence properties. *Phys. Fluids* 17 (9). URL <http://scitation.aip.org/content/aip/journal/pof2/17/9/10.1063/1.2047568>
- [22] Sierakowski, A. J., 2015. Bluebottle: GPU-centric fluid-particle direct numerical simulations using the Physalis method. <http://PhysalisCFD.org>, v2.0.2.
- [23] Sierakowski, A. J., Prosperetti, A., 2016. Resolved-particle simulation by the physalis method: Enhancements and new capabilities. *J. Comput. Phys.* 309, 164–184.
- [24] Silberstein, M., Schuster, A., Geiger, D., Patney, A., Owens, J. D., 2008. Efficient computation of sum-products on gpus through software-managed cache. In: *Proceedings of the 22nd Annual International Conference on Supercomputing. ICS '08*. ACM, New York, NY, USA, pp. 309–318. URL <http://doi.acm.org/10.1145/1375527.1375572>
- [25] Simeonov, J. A., Calantoni, J., 2012. Modeling mechanical contact and lubrication in direct numerical simulations of colliding particles. *Int. J. Multiphase Flow* 46, 38–53.
- [26] Wylie, J. J., Koch, D. L., Ladd, A. J. C., 4 2003. Rheology of suspensions with high particle inertia and moderate fluid inertia. *J. Fluid Mech.* 480, 95–118.
- [27] Yang, J., Stern, F., 2015. A non-iterative direct forcing immersed boundary method for strongly-coupled fluid-solid interactions. *J. Comput. Phys.* 295 (0), 779–804.
- [28] Zaspel, P., Griebel, M., 2013. Solving incompressible two-phase flows on multi-GPU clusters. *Comput. Fluids* 80 (0), 356–364, selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011.
- [29] Zeng, L., Balachandar, S., Najjar, F. M., 2010. Wake response of a stationary finite-sized particle in a turbulent channel flow. *Int. J. Multiphase Flow* 36 (5), 406–422.
- [30] Zhang, Z., Prosperetti, A., 2005. A second-order method for three-dimensional particle simulation. *J. Comput. Phys.* 210 (1), 292–324.