Computer Physics Communications 254 (2020) 107298

Contents lists available at ScienceDirect

Computer Physics Communications

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

Mirheo: High-performance mesoscale simulations for microfluidics☆,☆☆

Dmitry Alexeev, Lucas Amoudruz, Sergey Litvinov, Petros Koumoutsakos *

Computational Science and Engineering Laboratory, Clausiusstrasse 33, ETH Zürich, CH-8092, Switzerland

ARTICLE INFO

Article history:
Received 22 November 2019
Received in revised form 5 March 2020
Accepted 11 March 2020
Available online 23 March 2020

Keywords:
Microfluidics
High-performance computing
Dissipative particle dynamics
GPU computing
Red blood cell

ABSTRACT

The transport and manipulation of particles and cells in microfluidic devices has become a core methodology in domains ranging from molecular biology to manufacturing and drug design. The design and operation of such devices can benefit from simulations that resolve flow-structure interactions at sub-micron resolution. We present a computational tool for large scale, efficient and high throughput mesoscale simulations of fluids and deformable objects at complex microscale geometries. The code employs dissipative particle dynamics for the description of the flow coupled with visco-elastic membrane model for red blood cells and can also handle rigid bodies and complex geometries. The software (Mirheo) is deployed on hybrid GPU/CPU architectures exhibiting unprecedented time-to-solution performance and excellent weak and strong scaling for a number of benchmark problems. Mirheo exploits the capabilities of GPU clusters, leading to speedup of up to 10X in terms of time to solution as compared to state-of-the-art software packages and reaches 90%–99% weak scaling efficiency on 512 nodes of the Piz Daint supercomputer. The software Mirheo relies on a Python interface to facilitate the solution and analysis of complex problems. Mirheo is an open source, potent computational tool that can greatly assist studies of microfluidics.

Program summary

Program Title: Mirheo
Program Files doi: http://dx.doi.org/10.17632/n2dvz7htvn.1
Licensing provisions: MIT
Programming language: C++, CUDA, Python
Nature of problem: 3D simulations of microfluidic flows in complex geometries with suspended rigid bodies and deformable membranes such as cells, bacteria and microparticles.
Solution method: Dissipative particle dynamics are used to represent the fluid. Cell membrane dynamics are described through potentials for shear and bending energies that are discretized on a triangular mesh and by additional constraints on cell volume and membrane area. The model incorporates membrane viscosity and interactions between membranes and the surrounding fluid. Rigid objects and boundaries are represented by groups of particles with prescribed center of mass and rotation quaternion. Time integration is performed using the Velocity-Verlet algorithm.
Additional comments including restrictions and unusual features: The code runs on Nvidia GPU accelerators starting with the Kepler generation.

1. Introduction

Microfluidic devices are used to transport, control, analyze and manipulate nanoliter quantities of liquids, gasses and other substances in engineering applications [1–3] as well as in clinical research [3–6]. Fluid flows in microfluidic devices are characterized by Reynolds numbers that are low (10^{-3}–10^{1}) or moderate (10^{-1}–10^{2}) in high throughput settings. Fluid flows at the microscale are often modeled by the continuum Navier–Stokes equations discretized by various numerical methods [7–12]. These continuum models have been very successful in capturing several key features of microscale flows. At the same time they are limited in resolving phenomena at sub-micron resolution and related biochemical processes [13]. Moreover the deployment of continuum models in 3D is challenging for complex geometries and may lead to computations with low time-to-solution [14]. We note for example that state-of-the-art

☆ The review of this paper was arranged by Prof. D.P. Landau.
☆☆ This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).
* Corresponding author.
E-mail address: petros@ethz.ch (P. Koumoutsakos).

https://doi.org/10.1016/j.cpc.2020.107298
0010-4655/

simulations of red blood cells (RBCs) that use highly scalable boundary integral methods have only used a few hundred RBCs in two dimensions [15,16].

An alternative simulation approach for microfluidics is presented by dissipative particle dynamics (DPD) method. This method represents the fluids and the suspended objects as collections of particles. DPD is a stochastic, short-range particle method that aims to bridge the gap between molecular dynamics (MD) and the Navier–Stokes equations [17]. It has been used extensively to model complex fluids such as colloidal suspensions, emulsions and polymers [18–20] and has recently become a key method for the study of the blood rheology [21–24]. However, currently most of these simulations are carried out by non open source software based on LAMMPS [25] MD package. LAMMPS is notable for its flexibility and capability to model multiphysics but at the same time this may come at the expense of speed in domain specific settings such as MD.[1]

At the same time, a number of open source, however poorly documented, DPD codes [26–28] are available. By extensively exploiting graphics processing units (GPUs) to accelerate the most computationally-expensive kernels, they demonstrate much better performance than LAMMPS. In general, the appeal of moving some computations to the GPU is high in modern simulation codes [29]: while demanding highly parallel problems and careful implementations, the GPUs offer unmatched FLOPS performance and memory bandwidth, outperforming state-of-the-art server-grade CPUs by a factor of 5 to 10. However, transferring the existing code onto a GPU is usually not a trivial task. Computational challenges include architectural differences in the hardware such as cache size, width of the vector instructions and different control flow penalties. Another aspect of porting the application onto the GPU is the memory traffic through the PCI-E bus between the accelerator and the CPU. The bus only delivers a fraction of main RAM bandwidth, and if used extensively, may easily result in a performance bottleneck. Therefore porting parts of the application onto the GPU may be neither beneficial nor easy, and often starting from scratch is preferred.

Here we present Mirheo,[2] a high-throughput software for microfluidic flow simulations in complex geometries with suspended visco-elastic cell membranes and rigid objects, written exclusively for GPUs and clusters of GPUs (e.g. see Fig. 1). Mirheo is a successor of the uDeviceX software [30] with improved performance, usability and extensibility. Furthermore, Mirheo handles complex geometries, large number of suspended rigid bodies and cells, fluids with different viscosity and provides a flexible yet efficient and well-documented way to specify the simulation setup and parameters.

In the rest of the paper we first introduce the employed numerical method (Section 2), then go over details of our implementation and parallelization strategies (Section 3), followed by code validation (Section 4) and benchmarks (Section 5) before concluding (Section 6).

2. Numerical method

Mirheo is based on the DPD method [17,31]. The software accommodates flows in complex geometrical domains as well as deformable and rigid objects suspended in the fluid. More specifically, the supported objects are visco-elastic closed shells (representing cell membranes discretized on triangular meshes) and rigid bodies of arbitrary shape. The evolution of the system is governed by pairwise particle forces while enforcing the no-slip and no-through boundary conditions where applicable.

[1] http://www.hecbiosim.ac.uk/benchmarks.

[2] https://github.com/cselab/Mirheo.

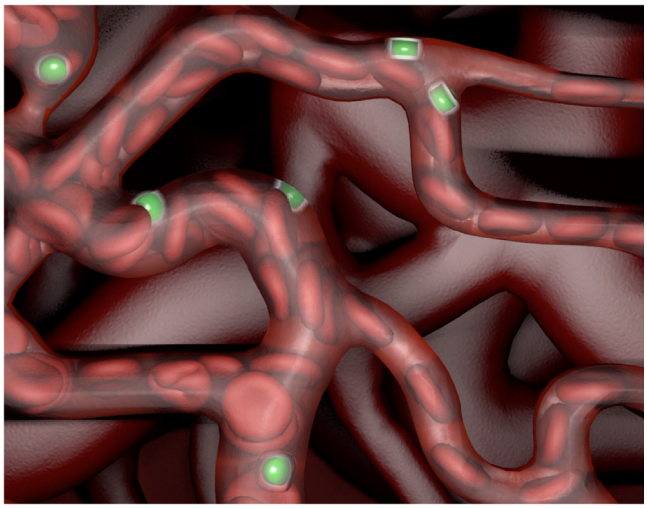

Fig. 1. Simulation of red blood cells and microscale drug carriers inside capillaries.

2.1. Dissipative particle dynamics

The DPD is a particle based method introduced by Hoogerbrugge [32] and further developed in [17,31]. In DPD the fluid is described by a set of particles in the 3D space. Each particle is characterized by its mass m, position $\mathbf{r}$ and velocity $\mathbf{v}$. Particles evolve in time according to the Newton's law of motion:

$$\begin{aligned} \frac{d\mathbf{r}}{dt} &= \mathbf{v}, \\ \frac{d\mathbf{v}}{dt} &= \frac{1}{m}\mathbf{F}, \end{aligned} \tag{1}$$

where $\mathbf{F}$ is the force exerted on the particle and t is time. The force fields are expressed in terms of the distance r between particles and they imply local interactions as they vanish after a cutoff radius r_c. The particles interact through central forces, which implies, by the Newton's third law, conservation of linear and angular momentum. The DPD forces acting on the particle indexed by i are written as

$$\mathbf{F}_i = \sum_j \left(\mathbf{F}^C_{ij} + \mathbf{F}^D_{ij} + \mathbf{F}^R_{ij}\right), \tag{2}$$

where the force is composed of a conservative, a dissipative and a random term. The conservative term acts as a purely repulsive force and reads

$$\mathbf{F}^C_{ij} = \alpha w(r_{ij})\mathbf{e}_{ij}, \tag{3}$$

where α is the interaction scaling factor, $r_{ij} = |\mathbf{r}_{ij}|$, $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$, $\mathbf{e}_{ij} = \mathbf{r}_{ij}/r_{ij}$ and

$$w(r) = \begin{cases} 1 - r/r_c, & \text{if } r < r_c, \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

The dissipative and random terms are given by

$$\begin{aligned} \mathbf{F}^D_{ij} &= -\gamma \left(\mathbf{v}_{ij} \cdot \mathbf{e}_{ij}\right) w_D(r_{ij})\mathbf{e}_{ij}, \\ \mathbf{F}^R_{ij} &= \sigma \xi_{ij} w_R(r_{ij})\mathbf{e}_{ij}. \end{aligned} \tag{5}$$

The random variables ξ_{ij} follow independent Gaussian distribution satisfying $\langle \xi_{ij}(t)\xi_{lm}(t')\rangle = \delta(t-t')\left(\delta_{il}\delta_{jm} + \delta_{im}\delta_{jl}\right)$, $\xi_{ij} = \xi_{ji}$ and $\langle \xi_{ij}\rangle = 0$. The parameters γ and σ are linked through the fluctuation–dissipation relation $w_D = w_R^2$ and $\sigma^2 = 2\gamma k_B T$ [17]. The dissipative kernel has the form $w_R(r) = w^k(r)$ with $k \in (0, 1)$ [33].

2.2. Objects representation

Rigid objects are modeled as groups of particles moving together as a whole. Their surface geometry is expressed either analytically or by a triangular mesh-based representation. The state of a rigid object is fully determined by its center of mass, orientation (stored as a quaternion), linear and angular velocities.

The visco-elastic incompressible membrane is modeled by a triangular mesh with particles as its vertices. The mesh is composed of N_v vertices, N_t triangles and N_s edges. The elastic potential energy of the membrane with constant volume and area is given by [34]:

$$U = U_{\text{in-plane}} + U_{\text{bending}} + U_{\text{area}} + U_{\text{volume}}. \tag{6}$$

$U_{\text{in-plane}}$ accounts for the energy of the elastic spectrin network of the membrane, including an attractive worm-like chain potential and a repulsive potential such that a nonzero equilibrium spring length can be obtained.

$$U_{\text{in-plane}} = \sum_{j=1}^{N_e} \left[\frac{k_s l_m \left(3x_j^2 - 2x_j^3\right)}{4(1-x_j)} + \frac{k_p}{l_j} \right], \tag{7}$$

where k_s is the spring constant, l_j is the length of the jth spring, $x_j = l_j/l_m$, l_m is the maximum spring extension and N_e is the number of mesh edges. k_p is computed such that the spring energy is zero at $l_j = l_0$. The bending energy term, U_{bending}, models the resistance of the lipid bilayer to bending. We implement two different energy models for the membrane dynamics. The first is attributed to Kantor and Nelson [35]:

$$U_{\text{bending}}^{KN} = \sum_{j=1}^{N_e} k_b \left[1 - \cos(\theta_j - \theta_0)\right], \tag{8}$$

where k_b is the bending constant, θ_j is the angle between two adjacent triangles (called dihedral) and θ_0 is the equilibrium angle. The second was developed by Jülicher [36]:

$$U_{\text{bending}}^{J} = 2k_b \sum_{j=1}^{N_v} \frac{M_j^2}{A_j}, \tag{9}$$

where

$$M_j = \frac{1}{4} \sum_{\langle j,k \rangle} l_{jk}\theta_{jk}.$$

U_{area} and U_{volume} are penalization terms accounting for area and volume conservation of the membrane:

$$\begin{aligned} U_{\text{area}} &= \frac{k_a (A^{\text{tot}} - A_0^{\text{tot}})^2}{2A_0^{\text{tot}}} + \sum_{j=1}^{N_t} \frac{k_d (A_j - A_0)^2}{2A_0}, \\ U_{\text{volume}} &= \frac{k_v (V - V_0^{\text{tot}})^2}{2V_0^{\text{tot}}}, \end{aligned} \tag{10}$$

where A_j is the area of a single triangle, $A^{\text{tot}} = \sum_{j=1}^{N_t} A_j$, V is the volume enclosed by the membrane and N_t is the number of triangles in the mesh.

The membrane viscosity is modeled by an additional pairwise interaction between particles sharing the same edge. This interaction contains a dissipative and random term with the same form as the DPD interaction with $w^R(r) = 1$.

2.3. Boundary conditions

Solid boundaries in the computational domain are represented via a signed distance function (SDF) whose zero value iso-surface defines the wall boundaries. A layer of frozen particles with thickness of r_c is located just inside the boundary. The no-through condition on the wall surface is enforced via a bounce-back mechanism [37]. These particles have the same radial distribution function as the fluid particles, and interact with the latter with the same DPD forces. This ensures the no-slip condition as well as negligible density variations of the fluid in proximity to the wall [38–40].

The fluid–structure interactions for the rigid objects are similar to the ones employed for the walls. The surface impenetrability is ensured by bouncing-back solvent particles off the rigid objects surfaces with linear and angular momentum conservation.

In order to maintain the no-slip and no-through flow boundary conditions on the membrane surface, we mainly follow the technique originally proposed in [34]. We also note a recent work on the development of advanced boundary conditions [41] for complex geometries. Here we assume that a membrane is always surrounded by fluid from both sides, with the same density and conservative potential, but the Mirheo software allows for different fluid viscosities. We then let the fluid particles across the membrane interact only with the conservative part of the DPD force, and in contrast, make the fluid–membrane interaction purely viscous. In that way we maintain constant radial distribution function in the liquids in proximity of the membrane, and with the appropriate choice of the liquid–membrane viscous parameter the no-slip condition is satisfied. The no-through condition is also enforced via the bounce-back mechanism.

3. Implementation details

The outline of Mirheo shares several key features with classical MD application with local interactions. However the introduction of the bounce-back mechanism, adoption of relatively large time-step and the necessity to operate on membranes consisting of hundreds and thousands of particles requires extra attention that distinguishes Mirheo from classical MD implementations. The software targets GPU-enabled clusters and use established technologies and libraries such as C++, CUDA, MPI, HDF5 and Python.

3.1. Algorithmic overview

The majority of design decisions in Mirheo have been dictated by the demands of the GPU architectures and the requirement for a robust and extensible code. The overall structure of Mirheo includes the following main components (see Fig. 2):

- Data management classes, called *Particle Vectors*. They store particles of a specific type and their properties. Objects, like rigid bodies or cell membranes, are also implemented as *Particle Vectors* for uniformity.
- Various handler classes that implement different actions and transformations of the *Particle Vectors*, e.g., integration, force computations, wall interactions.
- Plugins, which provide a convenient and non-intrusive way of adding functionalities to Mirheo.
- Coordinator classes, that perform initial simulation setup and time-stepping. These classes stitch together the *Particle Vectors*, handlers and plugins into an extensive simulation pipeline.
- Python bindings, that provide an interface to create and manipulate the data, handlers and other simulation details.

We use MPI parallelization by employing domain decomposition into equal rectangular boxes, such that each MPI rank keeps only the local particles. To reduce communication between the MPI ranks, we assign all the particles of a single object to a single

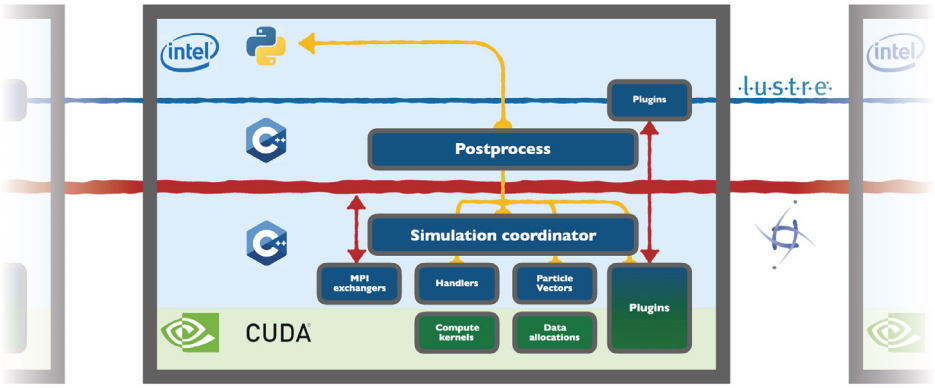

Fig. 2. Layout of the main Mirheo components within one computational node. Interconnect (here Infiniband) is in red, CPU part shaded in blue, GPU part — in green. Only the postprocess task performs heavy I/O. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

MPI process depending on the center of mass of the object. The core data (including particles, cell-lists, forces, etc.) are stored in the GPU RAM, while only the objects and particle adjacent to the subdomain boundaries require to be transferred to the CPU memory and communicated via MPI to the adjacent ranks (unless the user chooses to use GPU RDMA communication scheme). Moreover, we organize the particle data as structure-of-arrays (SOA) in order to optimize memory traffic, and also to allow dynamic addition of extra properties for each particle or each object.

The time-stepping pipeline of Mirheo is organized as follows:

1. Create the cell-lists for all the types of particles (*Particle Vectors*) and interaction cut-offs. In a typical simulation the cut-off radius is the same for all the pairwise forces involved and we observe that using a separate cell-list for different cut-off radii is beneficial in terms of overall performance.
2. Using the created cell-lists, identify the *halo* (or *ghost*) particles, that have to be communicated to the adjacent processes. The transfer itself is overlapped with the subsequent force computation. We will give more details about the overlap in the later section.
3. Compute forces due to the local particles.
4. After the halo exchange is completed, compute the forces in the system due to particles coming from neighboring processes.
5. Integrate the particles with the fused Velocity-Verlet. The rigid bodies need special treatment: we first calculate the total force and torque for each body, and then integrate their positions and rotational quaternions.
6. Bounce the particles off the walls, rigid bodies and membranes. The forces due to the bounce are saved in the objects and transferred to the next time-step.
7. Identify the particles and objects that have left the local subdomain and send them to the corresponding adjacent MPI rank.

3.2. Pairwise interactions and cell-lists

The nominal cost of computing all the pairwise forces in a system with N particles scales as $\mathcal{O}(N^2)$. However, in DPD the pairwise forces only affect the local neighborhood of each particle as the potential vanishes with the increased particle distance and the cost of force calculation is reduced to $\mathcal{O}(N)$. A common approach to restrict force computation to the particle pairs within a distance r_{cut} is to use the Verlet (or neighbor) lists (LAMMPS [25], NAMD [42], GROMACS [43], HOOMD-blue [44]). Such lists store for each particle the indices of all the other particles in the system within the distance $r_{cut} + \varepsilon$. The non-zero $\varepsilon > 0$ is introduced as the cost of building that structure is significant compared to the force evaluation, therefore it is advisable to rebuild it only once every few time-steps. For each simulation there exists an optimal ε that is governed by a balance between building the neighbor list (benefits from large ε) and force evaluation (benefits from smaller ε).

The potential of the conservative forces in DPD is bounded at zero distance, unlike typical potentials used in MD (e.g. Lennard-Jones). This feature leads to much larger typical time-steps in DPD than those used in MD. These two factors together result in fast changes of the particle neighborhoods, that would result in frequent Verlet list reconstruction and consequently a performance penalty.

In Mirheo we use cell-lists to accelerate the force computation. We first split the domain of interest into cubic cells with edge length r_{cut}, forming a uniform Cartesian grid. Then the cell-list data structure is defined as a two-way mapping of a particle onto a unique cell. The particle-cell mapping is trivial and can easily be computed from the particles coordinates. The construction of the inverse mapping, the cell-list itself, requires the particles to be sorted according to the index of the cell they belong to, and computing the positions in the particle array corresponding to each cell.

The force evaluation is typically the most time-consuming operation of each time-step. We map each particle to a GPU thread which scans the adjacent cells and calculates all the interactions for the given particle. The ordering of the particles in memory due to cell-lists increases data locality which accelerates the fetching of the particle data through cache. We observe that exploiting the symmetry of the forces yields in faster execution despite the additional atomic operations.

3.3. Particle bounce-back

An important part of a microfluidics simulation is maintaining the no-through flow boundary condition at the boundaries of rigid bodies and membranes. We enforce this condition by introducing a continuous "boundary" function of the particle coordinates that changes its sign on the impenetrable boundary. For example, for the wall that function is the SDF. By equating the "boundary" function with zero we obtain an equation whose solution gives the exact collision location. After this location is found, we place the particle into the collision point and reverse its velocity in the frame of reference of the surface. In order to reduce the computational cost, we exploit the cell-list that is built at the beginning of each time-step and only check the particles that

are located in the cells close to the zero level of the "boundary" function.

3.4. Efficient parallelization: compute/IO overlap

The vastly different scales of bandwidth provided by the GPU, PCI-E bus and the HDD storage make it necessary to overlap the intensive computation with different I/O operations performed by the code, such as MPI communications and dumping data on the disk [45]. This is achieved by the two asynchronous layers present in the Mirheo design.

The first asynchronous layer consists in running two MPI tasks per computational subdomain. The first task (*compute* task) performs the operations required to advance the system on the GPU. The second task (*postprocess* task) is responsible for all the heavy I/O and in-situ data post-processing on the CPU. This asynchronous design ensures perfect overlap of the disk operations with the simulation, improves code modularity and adds flexibility, since heavy data processing can be performed in parallel to the simulation, on the otherwise idling CPU. Fig. 9 shows the importance of the asynchronous HDF5 writes for the overall execution time.

The second asynchronous layer is implemented by a task scheduler which hides the MPI and PCI-E latencies. There is about thirty fine-grained tasks in one time-step. Maintaining their dependencies and concurrently executing some of the kernels is a tedious task. To facilitate this setup, we have implemented a GPU-aware task scheduler based on the Kahn's topological sorting algorithm [46], that supports task execution on concurrent CUDA streams. This scheduler facilitates the overlap of computation-intensive kernels (e.g. bulk force computations) with host-device memory transfers and inter-node communication (e.g. communication of the ghost particles).

3.5. Python interface

As software complexity increases to address multiple setups, the complexity of its usage is increasing accordingly. Software packages often provide custom syntax for their configuration files, or even introduce a simple programming language to help users [25,43]. We believe that implementing simulation setup through a well established programming language is superior with respect to the software specific approaches, as it benefits from the mature infrastructure and widespread usage of the language. With its flexibility and extensive support for scientific computations via comprehensive numerical libraries, Python proves to be one the best front-end languages [47] for complex codes like Mirheo. The *pybind11* project [48] allowed us to easily provide a C++/CUDA proxy into Python with minimal coding efforts.

We expose all our data holder classes, handlers, plugins and the coordinator class such that the user is able to assemble the specific simulation setup out of the few basic building blocks like a construction toy. Further advances of our approach include a very thin abstraction layer and the ease of documenting the functions available to the end users.

4. Validation

In this section we present a set of validation cases for Mirheo, in which we compare our results against available analytical solutions or previously published data. An additional large set of more fine-grained tests (for example, bounce-back tests, momentum conservation verification, etc.) is available with the source code. We note that Mirheo was developed using experiences from a previous code (uDeviceX [30]) co-developed by our group.

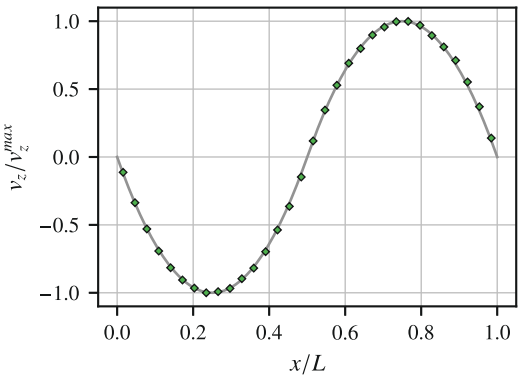

Fig. 3. Velocity profile in a periodic Poiseuille setup from simulations (symbols) and analytical solution (solid line). $L = 64$, $\rho = 8$, $a = 10$, $\gamma = 20$, $k_BT = 1.0$, $k = 0.5$, $\Delta t = 0.005$.

These experiences were instrumental in developing a software that is shown to outperform uDeviceX, a Gordon bell finalist in 2015 [30].

4.1. Viscosity of the DPD liquid for different parameters

Table 1 shows the measured viscosity of the DPD fluid with mass density ρ given specific parameters compared to the one presented in the literature. We choose to measure the viscosity using a pipe Poiseuille flow, such that $\eta = \rho f R^2/(8u_{avg})$. Here R is the radius of the pipe, f is the body force applied on each particle in order to form the pressure gradient, and u_{avg} is the average flow velocity. We assume small enough time-step in case it is not reported, and obtain the range of viscosities for body force going from 0.0003 to 0.05, resulting in good agreement with the previously reported values.

4.2. Periodic Poiseuille flow

Periodic Poiseuille flow was introduced in [51] as a convenient way to measure viscosity of a particle fluid without walls. The setup consists of a cubic domain ($L\times L\times L$) with periodic boundary conditions and the space-dependent body force that drives the fluid in the opposite directions:

$$\mathbf{f}(\mathbf{r}) = \begin{cases} (0,0,-f), & r_x \leqslant L/2, \\ (0,0,f), & r_x > L/2. \end{cases} \tag{11}$$

For a Newtonian fluid, the resulting laminar flow has a parabolic profile: $v_z(x) = \rho f(xL/2 - x^2)/2\eta$, where ρ is the fluid's mass density and η its dynamic viscosity. The simulation results are depicted in Fig. 3.

4.3. Taylor–Couette flow

Taylor–Couette flow consists of a fluid moving between two concentric cylinders, one of them rotating with respect to the other. Given the cylinders radii R_{in} and R_{out}, and their rotational velocities ω_{in} and ω_{out}, the resulting azimuthal velocity of the fluid is given by the following:

$$v(r) = r\,\omega_{in}\frac{\mu-\eta^2}{1-\eta^2} + \frac{1}{r}\,\omega_{in}\,R_{in}^2\frac{1-\mu}{1-\eta^2}, \tag{12}$$

where $\mu = \omega_{out}/\omega_{in}$ and $\eta = R_{in}/R_{out}$. The simulation results are depicted in Fig. 4.

Table 1
Comparison of the DPD fluid viscosity obtained from different parameters available in the literature with Mirheo. We simulate a Poiseuille flow inside a circular pipe of radius $R = 20$, length $L = 3R = 60$, and ran 10^6 time-steps. The pressure gradient was applied by adding body-force on each particle ranging from 0.0003 to 0.05. The viscosity was obtained by averaging the velocity over the last 2×10^5 steps.

DPD parameters							η_{Mirheo}	η_{ref}	Reference
a	γ	ρ	k	k_BT	r_c	Δt			
0.9375	115.6	4	1	0.05	1	0.01	4.4–4.6	4.7	[49]
6	20	3	0.15	0.1	1	0.002	8.0–8.3	8.1	[50]
4	8	3	0.15	0.1	1.5	0.001	24.7–26.3	26.3	[50]
4	40	3	0.15	0.1	1.5	0.0002	122.5–129.5	126	[50]
25	6.75	3	1	1	1	0.04	0.89–0.9	0.91	[31]
18.75	4.5	4	1	1	1	0.005	1.07–1.08	1.08	[33]
18.75	4.5	4	0.25	1	1	0.005	2.44–2.45	2.59	[33]
0	20.25	6	1	0.5	1	0.01	2.08–2.1	2.09	[51]

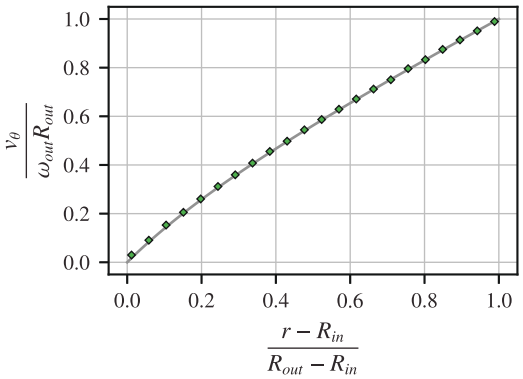

Fig. 4. Fluid velocity in the azimuthal direction against radial coordinate from simulation (symbols) and analytical solution (solid line), obtained with $R_{in} = 10$, $R_{out} = 32$, $\omega_{in} = 0$, $\omega_{iout} = 0.01$, $\rho = 10$, $a = 10$, $\gamma = 10$, $k_BT = 0.5$, $k = 0.125$, $\Delta t = 0.001$.

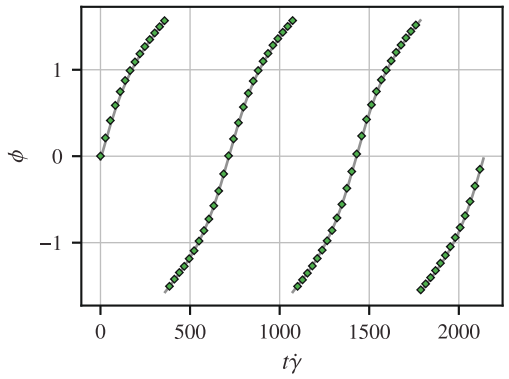

Fig. 5. Evolution of the inclination angle ϕ of the longer ellipsoid axis with respect to the flow direction with $a = 5$, $b = 3$, $\Gamma = 0.1$: Simulation (symbols) and Jeffery's theory (solid line). DPD parameters: $L = 64$, $\rho = 8$, $a = 25$, $\gamma = 50$, $k_BT = 0.5$, $k = 0.5$, $\Delta t = 0.005$.

4.4. Jeffery orbits in shear flow

We validate the rigid body dynamics by simulating the rotation of an ellipsoid in a simple shear flow. In the limit of small Reynolds number, the inclination angle of the longer ellipsoid axis is known to be following the Jeffery orbit [52] over time:

$$\phi(t) = \arctan\left(\frac{b}{a}\tan\frac{ab\,\dot{\gamma}\,t}{a^2 + b^2}\right), \tag{13}$$

where a and b are the longest and shortest axes of the ellipsoid and $\dot{\gamma}$ is the shear rate. We enforce the shear profile by moving two parallel plates with opposite velocities. The ellipsoid is kept in the middle of the computational domain throughout the entire simulation. The results are depicted in Fig. 5.

4.5. Flow past a sphere close to a wall

We study the drag and lift coefficients of a sphere translating in the otherwise quiescent fluid close to a flat wall. The setup is characterized by the fluid kinematic viscosity ν, sphere radius r, wall distance L and the translation velocity u. We perform the simulation in the sphere frame of reference by keeping the sphere center of mass forcedly fixed. We use periodic boundary conditions in x and y directions and introduce two walls orthogonal to the z direction. The walls are moved with the velocity $-u$ and the average fluid velocity in the domain is kept constant. The remaining non-uniform wake downstream the sphere is removed in a thin layer before the domain boundary to reduce the periodic boundary conditions effects. The sphere, however, is allowed to rotate freely.

The quantities of interest are the lift and drag coefficients of the sphere, which are computed by time-averaging of the fluid forces acting on the sphere. Due to the third Newton's law those forces are a simple negation of the forces that are required to keep the sphere in place. The non-dimensional expression of the above quantities read:

$$C_{l,d} = \frac{\langle F_{l,d}\rangle}{\frac{1}{2}\rho u^2 \pi r^2}, \tag{14}$$

where the angular brackets denote the time-averaging and the subscripts l, d represent wall-normal lift and wall-parallel drag, respectively. The simulation results are depicted in Fig. 6. Here we consider the case with $L = 2r$ and vary the Reynolds number $\text{Re} = Lu/\nu$ from 0.5 to 50.

We obtain a good correspondence against the previous simulations carried out with spectral methods [53]. The discrepancy observed for $\text{Re} = 10, 20$ can be attributed to the smaller domain size employed in order to reduce the computational cost.

4.6. Cell stretching

We validate the RBC model described in Section 2.2 by simulating a RBC stretched by optical tweezers and comparing the force–extension curve with the experimental data [54]. We vary the force applied to the few opposite particles of a single cells membrane, and measure the axial and transverse diameters of the cell. The results are depicted in Fig. 7.

5. Performance

One of the advantages of the Mirheo software is the very low time to solution and nearly perfect weak scaling up to hundreds

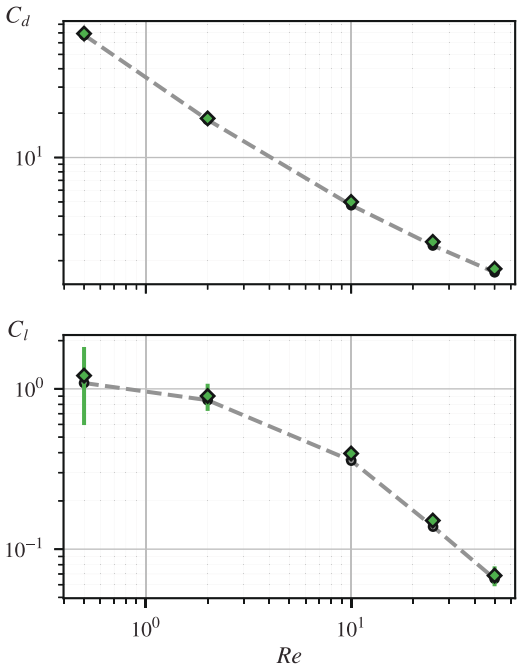

Fig. 6. Drag and lift coefficients for a sphere translating in a quiescent fluid at a distance $L = 2r$ from the infinite wall. Error bars represent 2 standard deviations of the mean estimate. DPD parameters vary to satisfy given Re and low enough Mach number Ma < 0.2. Symbols: DPD simulations obtained with Mirheo. Lines: spectral method [53].

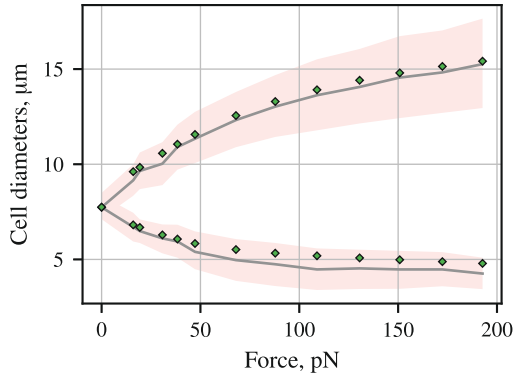

Fig. 7. Axial and transverse diameters of the RBC against stretching force. The simulation (symbols) employs parameters corresponding to the best fit [54] from experimental data (solid line).

of nodes. We benchmark our software against the state-of-the-art MD packages (HOOMD-Blue and LAMMPS) and the uDeviceX software [30] running on GPU clusters, and perform strong and weak scaling studies. All the timings were collected with the standard `nvprof` CUDA profiler and the internal high-resolution system clock. We used several hardware platforms to obtain the results: Piz Daint supercomputer (CSCS, Switzerland) with one Nvidia Tesla P100 per node, Leonhard cluster (ETHZ, Switzerland) with eight Nvidia GTX 1080Ti per node, Microsoft Azure platform with Nvidia Tesla V100 and a high-end consumer laptop with Nvidia GTX 1070.

5.1. Periodic Poiseuille flow

Our first benchmark is the periodic Poiseuille flow of the DPD particles, the least complex setup that is nevertheless representative for a wide class of problems where the object suspension is dilute. We consider a cubic domain of size $L \times L \times L$ per every GPU, filled uniformly with the DPD particles at a constant density ρ with the periodic body force f (see Section 4.2).

The reference benchmark employs $L = 64$ and $\rho = 8$, resulting in total of 2.1M particles, or 12.6M degrees of freedom, and roughly 34.1M interacting pairs per node assuming uniform particle distribution. The average time-step on 1 compute node of Piz Daint is 7.01 ms, which results in throughput of 4.9 billion interactions per second per GPU node.

Table 2 summarizes the performance comparison against uDeviceX code and HOOMD-blue on the Piz Daint supercomputer. Since the choice of the simulation parameters may affect the runtime, in Table 3 we show that our code consistently outperforms HOOMD-blue for various benchmark setups.

Fig. 8 (top) shows weak scaling capabilities of Mirheo running periodic Poiseuille benchmark on Piz Daint. Note that the reference point was chosen at $N = 8$ nodes, as the single node execution employs additional optimizations eliminating most MPI communications. Due to the good compute/transfer overlap, we reach almost perfect weak scaling for up to 1000 nodes. Strong scaling is not the primary scope of our code, as typically the problems of interest consist of very many particles. However, the strong efficiency of Mirheo is still good, see Fig. 8 (bottom). It also shows super-linear behavior that we believe is attributed to better spatial locality of smaller amount of data in the cache, which is the main bottleneck in computing interactions.

Fig. 9 shows benefits of the overlapping computations with I/O. We ran the periodic Poiseuille benchmark on Piz Daint with dumping HDF5 flow fields every 100 steps. The I/O bandwidth does not scale well with the number of nodes, reaching about 3 GB/s for ∼64 nodes. As one can observe, data dumps are overlapped with the computations, making the total runtime approximately maximum of the I/O and calculations. Note that in typical simulations, the dump frequency is much lower, such that we are never limited by the I/O performance.

5.2. Periodic whole blood flow

The second representative benchmark is the periodic Poiseuille flow of the blood, which includes cell membranes, different

Table 2
Wall-clock time in ms per one simulation time-step on the Piz Daint supercomputer (Nvidia P100 GPUs). 2^{21} particles per node, DPD parameters are the following: $a = 50$, $k_bT = 1$, $\gamma = 20$, $\delta t = 0.002$. The neighbor-list parameters of HOOMD-blue are tuned for the best performance.

	Nodes		
	1	27	64
HOOMD-blue	11.1	18.0	18.3
uDeviceX	10.4	11.3	11.6
Mirheo	**7.0**	**7.3**	**7.3**

Table 3
Performance comparison against HOOMD-Blue. Wall-clock time in ms per simulation time-step on the Piz Daint supercomputer (Nvidia P100 GPUs). The domain size is always 64^3, DPD parameters vary: $a = 10$, $\gamma \in \{1, 10, 100\}$, $\rho \in \{4, 8, 12\}$, $r_c \in \{0.8, 1.0, 1.2\}$. The neighbor-list parameters of HOOMD-blue are tuned for the best performance. Reported speedup is $t_{HOOMD}/t_{\text{Mirheo}}$.

k_BT	Δt	Speedup range	Average speedup
0.05	0.02	1.6 – 2.2	1.8
0.5	0.02	2.8 – 7.8	4.2
5.0	0.02	7.2 – 22.0	12.6
0.05	0.005	1.0 – 1.4	1.2
0.5	0.005	1.4 – 2.3	1.8
5.0	0.005	2.6 – 5.4	3.9
0.05	0.002	0.9 – 1.2	1.1
0.5	0.002	1.1 – 1.6	1.4
5.0	0.002	1.6 – 2.8	2.1

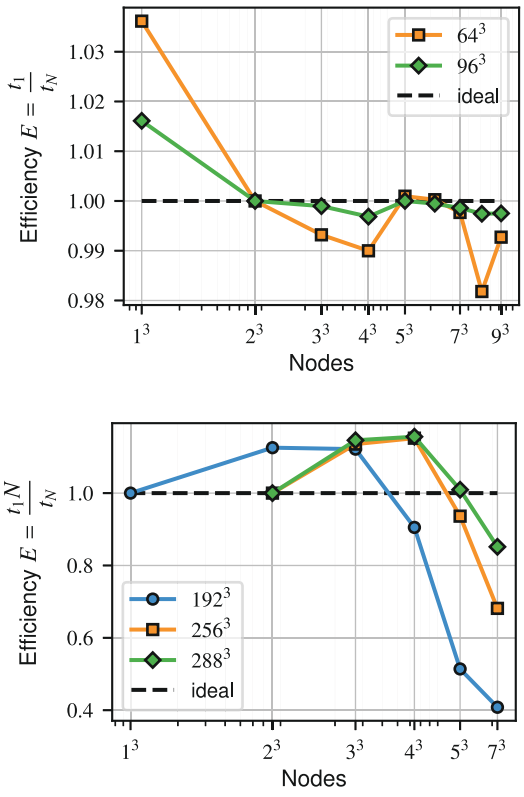

Fig. 8. Weak (top) and strong (bottom) scaling efficiencies of Mirheo running the periodic Poiseuille benchmark for different subdomain sizes. The number density is set to $\rho = 8$ for all the simulations.

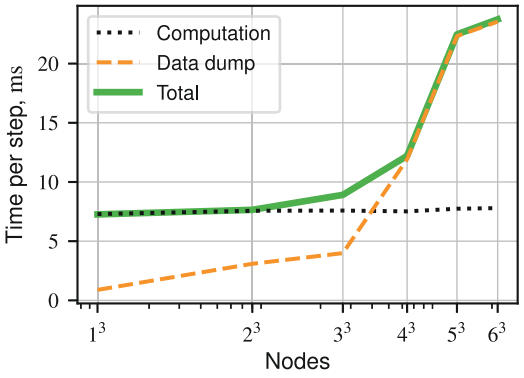

Fig. 9. Periodic Poiseuille benchmark on Piz Daint with data dumps every 100 steps.

viscosities between the plasma and the cytoplasm and runs with activated bounce-back mechanism. The latter incurs the biggest performance penalty, because it requires that objects are exchanged over MPI *after* the integration but *before* the bounce-back itself is performed. Therefore possibilities to overlap communication of objects and computations are limited in this case.

We consider a cubic domain of size $L \times L \times L$ filled uniformly with the RBCs at a specific volume fraction (or *hematocrit* level) Ht. The fluid density is ρ and the periodic force f is applied in the same manner as for the previous case.

Table 4 summarizes the performance comparison against the only GPU code known to the authors with roughly similar feature set: LAMMPS USER-MESO 2.0 [27]. We used the set-up available with the USER-MESO that runs whole blood at $Ht = 35\%$ and set

Table 4
Wall-clock time in ms per simulation time-step. Periodic whole blood at 35% hematocrit level, 1.6M particles per node.

	Nodes		
	1	8	27
LAMMPS USER-MESO 2.0	140.2	144.1	143.8
Mirheo	**9.8**	**13.6**	**13.7**

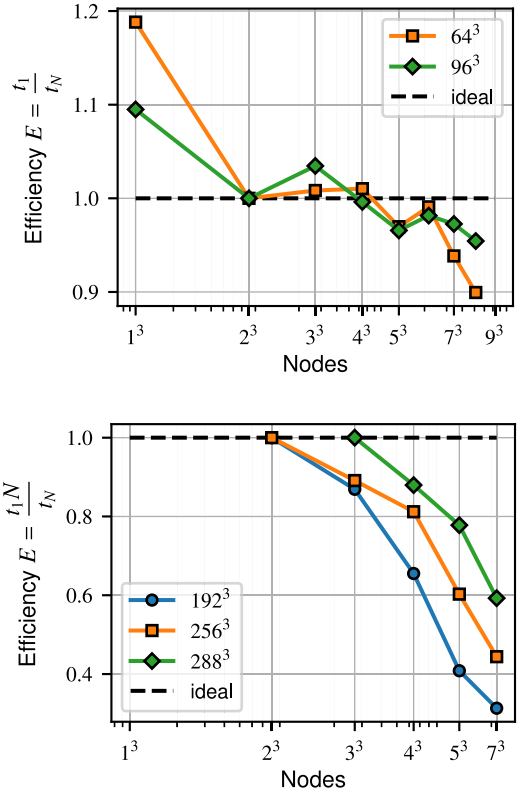

Fig. 10. Weak (top) and strong (bottom) scaling efficiency of periodic blood benchmark for different domain sizes. $\rho = 8$, $Ht = 40\%$.

the domain to roughly $76 \times 58 \times 58$. The present implementation outperforms USER-MESO by a factor of ~14x (~11x on many nodes), mainly attributed to the fact that LAMMPS, although evaluating forces on the GPU, keeps and uses a lot of supporting data on the CPU, incurring slow PCI-E traffic.

Fig. 10 (top) shows weak scaling capabilities of Mirheo running periodic blood benchmark at $Ht = 40\%$. The compute/transfer overlap worsens compared to the pure liquid flow, resulting in unstable execution time and deteriorated scaling. For the same reason we observe that the single node case benefits significantly more from the MPI calls elimination. Nevertheless, the code reaches 95% efficiency on 512 nodes for a bigger subdomain size. Strong scaling is also worse compared to the simple DPD case, but still yields in about 50% efficiency going from 8 to 216 nodes on a 256^3 domain size, see Fig. 10 (bottom).

5.3. Microfluidic device

To assess Mirheo at full complexity we simulate a part of a microfluidic device that captures circulating tumor cells (CTCs) from the whole blood [55]. The first stage of the device exploits deterministic lateral displacement principle [56] to separate the bigger and stiffer CTCs from the smaller and very flexible RBCs. In order to study the device and, later, to optimize the flow

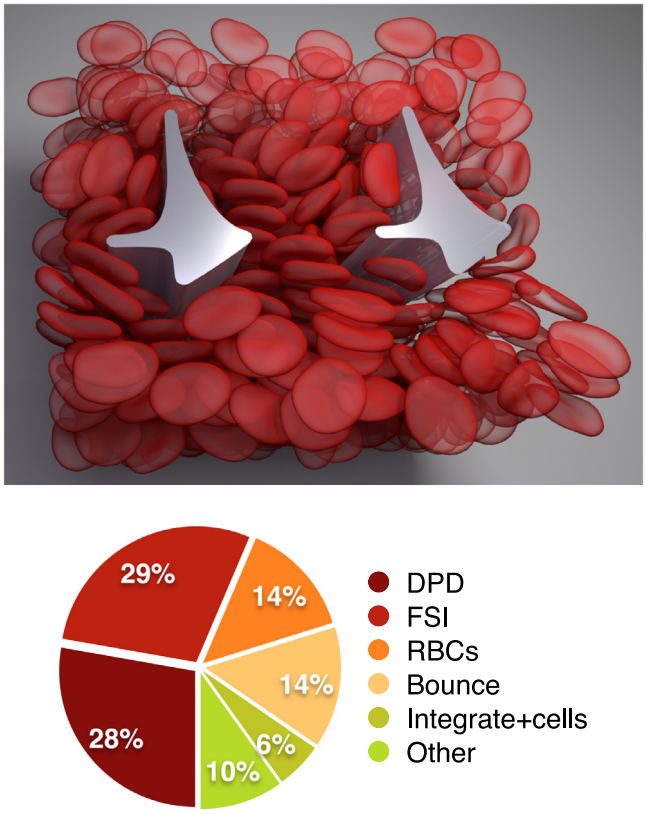

Fig. 11. Snapshot of the two-post periodic simulation (top) of a DLD device with irregularly-shaped obstacles and time distribution per kernel (bottom) on a single Piz Daint node. The domain size is $64 \times 56 \times 60$, particle density $\rho = 8$, hematocrit $Ht = 40\%$.

parameters and shape of the obstacles, we model its small part with two posts and impose periodic boundary conditions such that the domain replications correspond to the full device, see Fig. 11 (top). The setup features domain of complex shape, blood cells at $Ht = 40\%$ with cytoplasm 5 times more viscous compared to the solvent, and all the bounce-back mechanisms to prevent particle leakage.

Here, we report time distribution for various parts of the code with completely synchronous GPU kernel execution (forced by setting `CUDA_LAUNCH_BLOCKING` environmental variable). The reason for using synchronous timings is that in the *production mode*, several independent computational kernels may overlap, thus sharing the GPU resources and yielding in longer individual execution times. So in order to accurately estimate the performance of each kernel, we use the synchronous mode, while to obtain the overall wall-clock time per simulation step we use the faster asynchronous one. The time chart in Fig. 11 (bottom) shows that fluid forces account for the biggest part of the execution time: 57%. Bounce-back and the internal membrane forces have an equal share of 14% each, while 6% of the time is taken by memory-intensive integration and cell-list creation. The remaining 10% of the time is spent in various helper kernels such as packing and unpacking ghost particles. With the help of the Nvidia profiler we see that the force kernels yield low FLOP/s performance and DRAM bandwidth, the latter being about 5%–10% (depending on the hardware) of the peak. Instead, we identify that main bottleneck for the force kernels is the L1 and L2 cache performance, with bulk and FSI kernels reaching around 60% of the aggregate cache bandwidth on the Nvidia P100 architecture.

Table 5
Wall-clock time in ms per one simulation time-step on different GPUs. All runs are single-node. PP means periodic Poiseuille benchmark, see Section 5.1, DLD means microfluidic device, see Section 5.3.

	PP	DLD
1070M	12.6	27.8
1080Ti	6.9	18.8
P100	7.0	15.9
V100	3.7	8.8

5.4. Memory footprint

The memory required by Mirheo is proportional to the number of particles required and domain size (for the cell-lists). The maximum number of particles that could fit on a V100 (16 GB) GPU for the cases described in Sections 5.1 and 5.2 are 1.23×10^8 (domain size of 249^3 with $\rho = 8$) and 1.06×10^8 (domain size of 216^3 with $\rho = 8$ and $H_t = 45\%$), respectively. Note that the latter case involves membranes, demanding larger buffers to hold the ghost particles. Since the primary objective for our test-cases is to run order of 10^6–10^8 iterations, we only use much fewer particles per GPU and therefore consider memory footprint not an issue for Mirheo.

5.5. Hardware comparison

As a last step of our performance analysis, we run the periodic Poiseuille (see Section 5.1) and the DLD benchmark (see Section 5.3) on different hardware platforms: a consumer-grade laptop with Nvidia GTX 1070, ETHZ Leonhard cluster with Nvidia GTX 1080Ti, Piz Daint supercomputer with Nvidia Tesla P100 and Microsoft Azure virtual machine with Nvidia Tesla V100. Table 5 summarizes the results. Together with lower-level kernel analysis, these results show that Mirheo performance benefits greatly from the better and newer hardware, with the most important factor being the size and the speed of the L1 and L2 GPU caches.

6. Summary

In this paper we have introduced the state-of-the-art, open-source GPU software package Mirheo, that implements the DPD method. The software is shown to outperform similar packages [27,30] deployed on massively parallel computing architectures. Mirheo can handle arbitrarily complex domains, many visco-elastic RBCs and rigid bodies of various shapes, parallelized over hundreds of GPU nodes. Such set of features, to the best of our knowledge, is not offered by any other particle-based simulation software. We have also presented a set of validation cases that exhibits good correspondence of simulation results with analytical, experimental or earlier numerical data.

We have presented a number of benchmarks showing that Mirheo provides unprecedented low time-to-solution, efficiently harnessing GPU capabilities. Our software outperforms the state-of-the-art competitors by factors of $\sim$1.5 (for pure DPD liquid) up to 11 (for dense blood) and reaches high weak and strong scaling efficiencies for up to 512 nodes of the Piz Daint supercomputer. Furthermore, Mirheo comes with the extensively documented Python interface, that offers a simple mechanism to combine the implemented features into a complex simulation. Mirheo is open-source and available at https://github.com/cselab/Mirheo, where the user can find installation instructions, documentation and an extensive set of examples.

Declaration of competing interest

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

Acknowledgments

We acknowledge support by the European Research Council (ERC Advanced Grant 341117) and computational resources granted by the Swiss National Supercomputing Center (CSCS) under project ID ch7.

References

[1] D.J. Beebe, G.A. Mensing, G.M. Walker, Annu. Rev. Biomed. Eng. 4 (1) (2002) 261–286, http://dx.doi.org/10.1146/annurev.bioeng.4.112601.125916, URL http://www.annualreviews.org/doi/10.1146/annurev.bioeng.4.112601.125916.
[2] G.M. Whitesides, Nature 442 (7101) (2006) 368–373, http://dx.doi.org/10.1016/j.agee.2012.07.026, arXiv:arXiv:1011.1669v3.
[3] A.M. Streets, Y. Huang, Biomicrofluidics 7 (1) (2013) http://dx.doi.org/10.1063/1.4789751.
[4] T.M. Squires, S. Quake, Rev. Modern Phys. 77 (July) (2005).
[5] E.K. Sackmann, A.L. Fulton, D.J. Beebe, Nature 507 (7491) (2014) 181–189, http://dx.doi.org/10.1038/nature13118.
[6] J. Zhang, S. Yan, D. Yuan, G. Alici, N.T. Nguyen, M. Ebrahimi Warkiani, W. Li, Lab Chip 16 (1) (2016) 10–34, http://dx.doi.org/10.1039/c5lc01159k.
[7] D. Clague, S&Tr (2001) 4–11.
[8] D. Di Carlo, J.F. Edd, K.J. Humphry, H.A. Stone, M. Toner, Phys. Rev. Lett. 102 (9) (2009) 1–4, http://dx.doi.org/10.1103/PhysRevLett.102.094503, arXiv:NIHMS150003.
[9] I. Cimrák, M. Gusenbauer, T. Schrefl, Comput. Math. Appl. 64 (3) (2012) 278–288, http://dx.doi.org/10.1016/j.camwa.2012.01.062.
[10] N.M. Karabacak, P.S. Spuhler, F. Fachin, E.J. Lim, V. Pai, E. Ozkumur, J.M. Martel, N. Kojic, K. Smith, P.-i. Chen, J. Yang, H. Hwang, B. Morgan, J. Trautwein, T.A. Barber, S.L. Stott, S. Maheswaran, R. Kapur, D.A. Haber, M. Toner, Nat. Protoc. 9 (3) (2014) 694–710, http://dx.doi.org/10.1038/nprot.2014.044.
[11] T. Scherr, G.L. Knapp, A. Guitreau, D.S.W. Park, T. Tiersch, K. Nandakumar, W.T. Monroe, Biomed. Microdevices 17 (3) (2015) http://dx.doi.org/10.1007/s10544-015-9957-6.
[12] A. Grimmer, M. Hamidović, W. Haselmayr, R. Wille, ACM J. Emerg. Technol. Comput. Syst. (JETC) 15 (3) (2019) 26.
[13] J.B. Freund, Annu. Rev. Fluid Mech. 46 (1) (2014) 67–95, http://dx.doi.org/10.1146/annurev-fluid-010313-141349, URL http://www.annualreviews.org/doi/abs/10.1146/annurev-fluid-010313-141349.
[14] A. Rahimian, I. Lashuk, S. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, et al., Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society, 2010, pp. 1–11.
[15] B. Quaife, G. Biros, J. Comput. Phys. 274 (2014) 245–267, http://dx.doi.org/10.1016/j.jcp.2014.06.013.
[16] G. Kabacaolu, G. Biros, Phys. Rev. Fluids 3 (12) (2018) http://dx.doi.org/10.1103/PhysRevFluids.3.124201, arXiv:arXiv:1805.08849v2.
[17] P. Español, Phys. Rev. E 52 (2) (1995) 1734–1742, http://dx.doi.org/10.1103/PhysRevE.52.1734, URL http://link.aps.org/doi/10.1103/PhysRevE.52.1734.
[18] E.S. Boek, P.V. Coveney, H.N.W. Lekkerkerker, P. van der Schoot, Phys. Rev. E 55 (3) (1997) 3124–3133, http://dx.doi.org/10.1103/PhysRevE.55.3124.
[19] E. Moeendarbary, T.Y. Ng, M. Zangeneh, Int. J. Appl. Mech. 02 (01) (2010) 161–190, http://dx.doi.org/10.1142/S1758825110000469, URL http://www.worldscientific.com/doi/abs/10.1142/S1758825110000469.
[20] Y. Wang, Z. Li, J. Xu, C. Yang, G.E. Karniadakis, Soft Matter 15 (8) (2019) 1747–1757.
[21] D.J. Quinn, I.V. Pivkin, S.Y. Wong, K.-h.H. Chiam, M. Dao, G.E. Karniadakis, S. Suresh, Ann. Biomed. Eng. 39 (3) (2011) 1041–1050, http://dx.doi.org/10.1007/s10439-010-0232-y.
[22] P. Zhang, C. Gao, N. Zhang, M.J. Slepian, Y. Deng, D. Bluestein, Cellular Mol. Bioeng. 7 (4) (2014) 552–574, http://dx.doi.org/10.1007/s12195-014-0356-5.
[23] D.A. Fedosov, H. Noguchi, G. Gompper, Biomech. Model. Mechanobiol. 13 (2) (2014) 239–258, http://dx.doi.org/10.1007/s10237-013-0497-9.
[24] L. Lanotte, J. Mauer, S. Mendez, D.A. Fedosov, J.-M. Fromental, V. Claveria, F. Nicoud, G. Gompper, M. Abkarian, Proc. Natl. Acad. Sci. 113 (47) (2016) 13289–13294, http://dx.doi.org/10.1073/pnas.1618852114.
[25] S. Plimpton, J. Comput. Phys. 117 (1) (1995) 1–19, http://dx.doi.org/10.1006/jcph.1995.1039, URL http://www.sciencedirect.com/science/article/pii/S002199918571039X.
[26] Y.H. Tang, G.E. Karniadakis, Comput. Phys. Comm. 185 (11) (2014) 2809–2822, http://dx.doi.org/10.1016/j.cpc.2014.06.015, arXiv:1311.0402.
[27] A.L. Blumers, Y.H. Tang, Z. Li, X. Li, G.E. Karniadakis, Comput. Phys. Comm. 217 (2017) 171–179, http://dx.doi.org/10.1016/j.cpc.2017.03.016, arXiv:1611.06163.
[28] M.A. Seaton, R.L. Anderson, S. Metz, W. Smith, Mol. Simul. 39 (10) (2013) 796–821.
[29] M.J. Schulte, M. Ignatowski, G.H. Loh, B.M. Beckmann, W.C. Brantley, S. Gurumurthi, N. Jayasena, I. Paul, S.K. Reinhardt, G. Rodgers, IEEE Micro 35 (4) (2015) 26–36, http://dx.doi.org/10.1109/MM.2015.71.
[30] D. Rossinelli, G. Karniadakis, M. Fatica, I. Pivkin, P. Koumoutsakos, Y.-h. Tang, K. Lykov, D. Alexeev, M. Bernaschi, P. Hadjidoukas, M. Bisson, W. Joubert, C. Conti, Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '15, ACM Press, 2015, pp. 1–12, http://dx.doi.org/10.1145/2807591.2807677.
[31] R.D. Groot, P.B. Warren, J. Chem. Phys. 107 (11) (1997) 4423, http://dx.doi.org/10.1063/1.474784, URL http://scitation.aip.org/content/aip/journal/jcp/107/11/10.1063/1.474784.
[32] P.J. Hoogerbrugge, J.M.V.A. Koelman, Europhys. Lett. 19 (June) (1992) 155–160, http://dx.doi.org/10.1209/0295-5075/19/3/001.
[33] X. Fan, N. Phan-Thien, S. Chen, X. Wu, T.Y. Ng, Phys. Fluids 18 (6) (2006) http://dx.doi.org/10.1063/1.2206595.
[34] D.A. Fedosov, B. Caswell, G.E. Karniadakis, Biophys. J. 98 (10) (2010) 2215–2225, http://dx.doi.org/10.1016/j.bpj.2010.02.002.
[35] Y. Kantor, D.R. Nelson, Phys. Rev. A 36 (8) (1987) 4020.
[36] F. Jülicher, J. Phys. II 6 (12) (1996) 1797–1824.
[37] M. Revenga, I. Zúñiga, P. Español, I. Pagonabarraga, Internat. J. Modern Phys. C 09 (8) (1998) 1319–1328, http://dx.doi.org/10.1142/S0129183198001199.
[38] D.A. Fedosov, I.V. Pivkin, G.E. Karniadakis, J. Comput. Phys. 227 (4) (2008) 2540–2559.
[39] E.M. Kotsalis, J.H. Walther, E. Kaxiras, P. Koumoutsakos, Phys. Rev. E 79 (4) (2009) 45701.
[40] T. Werder, J.H. Walther, P. Koumoutsakos, J. Comput. Phys. 205 (1) (2005) 373–390, http://dx.doi.org/10.1016/j.jcp.2004.11.019.
[41] Z. Li, X. Bian, Y.-H. Tang, G.E. Karniadakis, J. Comput. Phys. 355 (2018) 534–547.
[42] J.C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R.D. Skeel, L. Kalé, K. Schulten, J. Comput. Chem. 26 (16) (2005) 1781–1802, http://dx.doi.org/10.1002/jcc.20289, arXiv:NIHMS150003.
[43] H.J. Berendsen, D. van der Spoel, R. van Drunen, Comput. Phys. Comm. 91 (1–3) (1995) 43–56, http://dx.doi.org/10.1016/0010-4655(95)00042-E, arXiv:arXiv:0803.4060v1.
[44] J.A. Anderson, C.D. Lorenz, A. Travesset, J. Comput. Phys. 227 (10) (2008) 5342–5359, http://dx.doi.org/10.1016/j.jcp.2008.01.047.
[45] P. Pacheco, An Introduction to Parallel Programming, Elsevier, 2011.
[46] A. Kahn, Commun. ACM 5 (11) (1962) 558–562, http://dx.doi.org/10.1145/368996.369025.
[47] A. Tosenberger, V. Salnikov, N. Bessonov, E. Babushkina, V. Volpert, Math. Model. Nat. Phenom. 6 (5) (2011) 320–332, http://dx.doi.org/10.1051/mmnp/20116512, URL http://www.mmnp-journal.org/10.1051/mmnp/20116512.
[48] W. Jakob, J. Rhinelander, D. Moldovan, Pybind11 — Seamless operability between c++11 and python, 2016.
[49] Z. Li, G. Drazer, Phys. Fluids 20 (10) (2008) http://dx.doi.org/10.1063/1.2980039.
[50] D.A. Fedosov, W. Pan, B. Caswell, G. Gompper, G.E. Karniadakis, Pnas 108 (29) (2011) 11772–11777, http://dx.doi.org/10.1073/pnas.1101210108/-/DCSupplemental.www.pnas.org/cgi/doi/10.1073/pnas.1101210108, URL http://www.pnas.org/cgi/doi/10.1073/pnas.1101210108.
[51] J.A. Backer, C.P. Lowe, H.C.J. Hoefsloot, P.D. Iedema, J. Chem. Phys. 122 (15) (2005) 154503, http://dx.doi.org/10.1063/1.1883163, URL http://scitation.aip.org/content/aip/journal/jcp/122/15/10.1063/1.1883163.
[52] G.B. Jeffery, Proc. R. Soc. Lond. Ser. A 102 (1922) 161–179, http://dx.doi.org/10.1098/rspa.1922.0078.
[53] L. Zeng, S. Balachandar, P. Fischer, J. Fluid Mech. 536 (2005) 1–25, http://dx.doi.org/10.1017/S0022112005004738, arXiv:arXiv:1011.1669v3. URL http://www.journals.cambridge.org/abstract{_}S0022112005004738.
[54] D.A. Fedosov, Multiscale Modeling of Blood Flow and Soft Matter (Ph.D. thesis), Brown University, 2010, p. 292, http://dx.doi.org/10.1080/01635580802395717.
[55] N.M. Karabacak, P.S. Spuhler, F. Fachin, E.J. Lim, V. Pai, E. Ozkumur, J.M. Martel, N. Kojic, K. Smith, P.-i. Chen, J. Yang, H. Hwang, B. Morgan, J. Trautwein, T.A. Barber, S.L. Stott, S. Maheswaran, R. Kapur, D.a. Haber, M. Toner, Nat. Protoc. 9 (3) (2014) 694–710, http://dx.doi.org/10.1038/nprot.2014.044, URL http://www.ncbi.nlm.nih.gov/pubmed/24577360.
[56] L.R. Huang, E.C. Cox, R.H. Austin, J.C. Sturm, Science 304 (5673) (2004) 987–990, http://dx.doi.org/10.1126/science.1094567.