Commun. Comput. Phys.
doi: 10.4208/cicp.210910.250311a

Vol. **11**, No. 1, pp. 48-64
January 2012

Multiscale Hemodynamics Using GPU Clusters

Mauro Bisson[1,*], Massimo Bernaschi[2], Simone Melchionna[3,4], Sauro Succi[2,5] and Efthimios Kaxiras[4,6]

[1] *Department of Computer Science, University of Rome "Sapienza", Italy.*
[2] *Istituto Applicazioni Calcolo, Consiglio Nazionale delle Ricerche, Rome, Italy.*
[3] *Istituto Processi Chimico-Fisici, Consiglio Nazionale delle Ricerche, Rome, Italy.*
[4] *Institute of Material Sciences and Engineering, École Polytechnique Fédérale de Lausanne, Switzerland.*
[5] *Freiburg Institute for Advanced Studies, School of Soft Matter Research, Albertstr. 19, 79104 Freiburg, Germany.*
[6] *Department of Physics and School of Engineering and Applied Sciences, Harvard University, Cambridge, MA, USA.*

Received 21 September 2010; Accepted (in revised version) 25 March 2011

Available online 5 September 2011

Abstract. The parallel implementation of MUPHY, a concurrent multiscale code for large-scale hemodynamic simulations in anatomically realistic geometries, for multi-GPU platforms is presented. Performance tests show excellent results, with a nearly linear parallel speed-up on up to 32GPUs and a more than tenfold GPU/CPU acceleration, all across the range of GPUs. The basic MUPHY scheme combines a hydrokinetic (Lattice Boltzmann) representation of the blood plasma, with a Particle Dynamics treatment of suspended biological bodies, such as red blood cells. To the best of our knowledge, this represents the first effort in the direction of laying down general design principles for multiscale/physics parallel Particle Dynamics applications in non-ideal geometries. This configures the present multi-GPU version of MUPHY as one of the first examples of a high-performance parallel code for multiscale/physics biofluidic applications in realistically complex geometries.

PACS: 02.70.Ns
Key words: Multi-GPU computing, hemodynamics, molecular dynamics, irregular domain.

1 Introduction

The behavior of blood in both capillaries and large coronary arteries has deep implications on the genesis of cardiovascular diseases such as atherosclerosis. Computational

*Corresponding author. *Email addresses:* bisson@di.uniroma1.it (M. Bisson), massimo@iac.rm.cnr.it (M. Bernaschi), simone.melchionna@epfl.ch (S. Melchionna), succi@iac.cnr.it (S. Succi), efthimios.kaxiras@epfl.ch (E. Kaxiras)

hemodynamics aims at studying flows in complex geometries, like those of blood vessels under stationary and pulsatile flow conditions. In the last few years, the study of hemodynamics has experienced an upsurge of activity due to the rapid advancement of methodological approaches and the availability of a steadily growing computing power, as also provided by high-performance commodity hardware, such as Graphics Processing Units (GPU). Blood is a complex fluid, composed of more than 99% in volume by two components, plasma and Red Blood Cells (RBC). Plasma is the solvent carrying simple Newtonian rheology, whereas RBCs play the role of basic building blocks, which are held responsible for shear-thinning and viscoelastic behavior. To the purpose of capturing the essence of blood dynamics, in particular close to the vessel walls and to morphological irregularities of the vessels, like the atherosclerotic plaques, it is imperative to look at the composite RBC-plasma system in its entirety, that is, by including the corpuscular nature of blood and evolve it concurrently with the continuum plasma component. For this reason, we adopt a multi-scale simulation approach that follows the two components on equal footing and in a concurrent fashion [2]. In our work, we leverage two distinct methods to handle plasma and RBCs and combine them in such a way to achieve a simple, yet effective, Janus-like representation of blood. Lattice Boltzmann (LB) is an efficient computational method to describe plasma as a fluid in the continuum within an Eulerian framework [3]. LB is a grid-based method, that uses a cartesian mesh and exchanges information related to the fluid among first and second mesh neighbors through the motion of fictitious molecules hopping and interacting on the sites of a regular lattice. LB shows an excellent scalability on high-end parallel computers that makes it very suitable for the simulation of large-scale systems, such as the complete coronary arterial system. Particle Dynamics (PD) is the method that handles the motion of suspended bodies in the Lagrangian (grid-free) framework. RBCs are represented as anisotropic particles that move, tumble and collide among themselves. RBCs are active scalars for the plasma, that is, they are responsible for a two-way exchange momentum with the solvent. The coupling is spatially local, rendering the concurrent evolution of plasma and RBCs an optimal choice for a bottom-up approach to the study of hemodynamics.

Coronary arteries constitute a system of interconnected vessels, presenting a non-trivial morphology (see Fig. 1), that surround the heart and carry oxygen to the heart muscle. The vessels are irregularly distributed in space and their layout calls for a highly sparse mesh to manage the active nodes only [4].

To reduce the time required for the simulation of the whole set of coronary arteries, it is mandatory to resort to parallel processing. To this purpose, we use a domain-decomposition scheme such that fluid and RBCs are handled on each subdomain by an individual processor. This aspect entails the first, coarse-grained level of parallelism, handled by conventional message-passing libraries, such as MPI [5]. The highly irregular shapes of the partitioned domains are obtained by specialized software packages, such as METIS [8] or SCOTCH [9], that produce quasi-optimal, from both the load-balancing and communication view points, partitionings. The migration and force calculation of RBC across multiple irregular domains requires the definition of *ad hoc* algorithms for

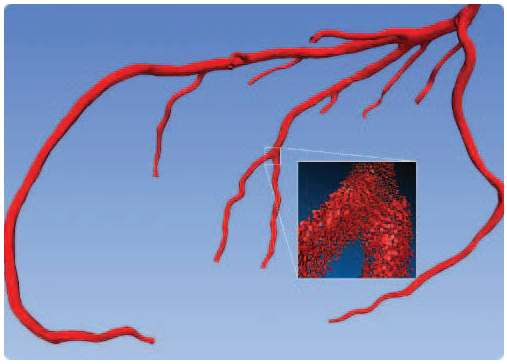

Figure 1: Geometry of a coronary artery system. The close-in shows a detail of the vessels with red blood cells visible.

the PD modules of the hemodynamic solver. In a previous paper [6], we described the issues emerging from parallel PD within irregular domains and proposed a general approach entailing a set of solutions. Hereafter, we present the related implementation issues, along with their solutions and the results obtained in selected test cases of specific hemodynamic relevance.

The second, fine-grained, level of parallelism is attained by employing Graphics Processing Units (GPUs) and distribution of tasks in threads. GPUs are advantageous and flexible hardware architectures for a wide class of computational problems and can be programmed in different optimized ways. With respect to high-end multicore architectures, GPUs may attain speed-ups that range between one and two orders of magnitude [7]. Currently, only CPUs are able to manage the MPI-level of communication, and the hybrid CPU/GPU computational paradigm may represent an optimal solution in terms of simplicity, flexibility and efficiency.

In the present paper, we describe our approach to multiscale hemodynamics and illustrate the several implementation issues related to the management of suspended RBCs within irregular domains. To address those issues, we have devised specialized data structures and the ensuing techniques for their management. The end result is a general data layout that provides excellent performances on CPU/GPU clusters, achieving quasi-ideal scalability on up to 32GPUs.

2 Multi-scale hemodynamics

Our approach combines two different levels of the description of matter: continuum fluids for the dynamics of blood plasma and individual particles for the representation of red blood cells and other minority suspended species. Fluid and particles are advanced *concurrently in time* and the exchange of information is computed on-the-fly [2].

Lattice Boltzmann (LB) is the method employed to reproduce the blood plasma dynamics. LB is based on the collective motion of fictitious particles defined on a regular

cartesian lattice [10] that reproduce hydrodynamics at the macroscale. The LB method presents several major advantages for the simulation in complex geometries, since the walls of the computational domain (in our case the blood vessels), are shaped via a staircase representation and not as body-fitted meshes, as employed in most Navier-Stokes simulations. Given the limited cost of handling the LB mesh, this representation of the vessels can be systematically improved by increasing the mesh resolution until the required accuracy (quality) is attained.

In LB, the probability of finding a plasma particle at location $\mathbf{x}$ and time t, and traveling with discrete speed $\mathbf{c}_p$, is encoded by the population $f_p(\mathbf{x},t)$. The index p indicates the direction of the traveling fluid particles. We employ the three-dimensional 19-speed cubic lattice (D3Q19) with mesh spacing Δx, with the discrete velocities $\mathbf{c}_p$ connecting mesh points to first and second mesh neighbors.

The fluid populations are advanced over a timestep Δt through the evolution equation

$$f_p(\mathbf{x}+\mathbf{c}_p\Delta t,t+\Delta t)=f_p(\mathbf{x},t)-\omega\Delta t(f_p-f_p^{eq})(\mathbf{x},t)+\Delta f_p(\mathbf{x},t). \tag{2.1}$$

The right hand side of Eq. (2.1) represents the effect of fluid-fluid molecular collisions, through a relaxation towards a local equilibrium,

$$f_p^{eq}=w_p\rho\Big[1+\frac{\mathbf{u}\cdot\mathbf{c}_p}{c_s^2}+\frac{\mathbf{uu}:(\mathbf{c}_p\mathbf{c}_p-c_s^2\mathbf{I})}{2c_s^4}\Big], \tag{2.2}$$

a second-order Maxwellian with density $\rho\equiv\sum_p f_p$ and plasma speed $\mathbf{u}\equiv\sum_p \mathbf{c}_p f_p/\rho$, where $c_s=1/\sqrt{3}$ is the speed of sound, w_p is a set of weights normalized to unity, and $\mathbf{I}$ is the unit tensor in Cartesian space. The last term in the r.h.s. of Eq. (2.1) encodes the coupling between fluid and suspended red blood cells, represented here as oblate ellipsoids in an hydrodynamic environment, and given by

$$\Delta f_p(\mathbf{x},t)=-\frac{w_p\Delta t}{c_s^2}\Big[\frac{\mathbf{G}\cdot\mathbf{c}_p}{c^2}+\frac{(\mathbf{G}\cdot\mathbf{c}_p)(\mathbf{u}\cdot\mathbf{c}_p)-c^2\mathbf{G}\cdot\mathbf{u}}{2c^4}\Big]. \tag{2.3}$$

Here, $\mathbf{G}$ is the forcing term containing the translational and rotational exchange of momentum induced by N moving red blood cells at position $\{\mathbf{R}_\alpha\}$. The forcing term is smeared over a mesh region extending over 32 mesh points around each RBC (this method will be the subject of a forthcoming publication). The drag force acting on each RBC is

$$\mathbf{F}_\alpha^D(\mathbf{R}_\alpha)=-\gamma^T\big[\mathbf{V}_\alpha-\widetilde{\mathbf{u}}(\mathbf{R}_\alpha)\big] \tag{2.4}$$

and the torque is

$$\mathbf{T}_\alpha^D(\mathbf{R}_\alpha)=-\gamma^R\big[\mathbf{\Omega}_\alpha-\widetilde{\mathbf{\Omega}}(\mathbf{R}_\alpha)\big], \tag{2.5}$$

with $\{\mathbf{V}_\alpha\}$ and $\{\mathbf{\Omega}_\alpha\}$ being the RBC velocities and angular velocities, and with $\widetilde{\mathbf{u}}$ and $\widetilde{\mathbf{\Omega}}$ being the fluid velocity and vorticity fields, smeared over the same 32 mesh points region.

γ^T and γ^R are translational and rotational coefficients for the RBC-fluid coupling. For the sake of simplicity, in the present implementation, we neglect additional torques arising from the coupling with the elongational component of the fluid, as employed in a slightly different version of the model (that will be the subject of a forthcoming publication). The forcing $\mathbf{G}$ contains translational and rotational components, related to the drag terms $\{\mathbf{F}_\alpha^D\}$ and $\{\mathbf{T}_\alpha^D\}$ that locally preserve linear and angular momentum of the RBC-fluid composite system.

Pairwise repulsive forces prevent contacts between RBCs. The RBC-RBC interactions are handled via the Gay-Berne potential for oblate ellipsoids [11], according to the pairwise potential

$$u_{ij}^{GB}(q_{ij}) = 4\epsilon(q_{ij}) \times \Big[\Big(\frac{\sigma_0}{R_{ij} - \sigma(q_{ij}) + \sigma_0} \Big)^{12} - \Big(\frac{\sigma_0}{R_{ij} - \sigma(q_{ij}) + \sigma_0} \Big)^{6} \Big], \tag{2.6}$$

where $q_{ij} \equiv (R_{ij}, \hat{\mathbf{u}}_i, \hat{\mathbf{u}}_j)$ and with R_{ij} being the relative distance, $\hat{\mathbf{u}}_i$ and $\hat{\mathbf{u}}_j$ the principal directions of the i-th and j-th ellipsoids. The functions $\epsilon(q_{ij})$ and $\sigma(q_{ij})$ have lengthy expressions described in [11]. The potential u_{ij}^{GB} is purely repulsive and is set equal to zero beyond a orientation-dependent cut-off given by the condition

$$\Big(\frac{\sigma_0}{R_{ij} - \sigma(q_{ij}) + \sigma_0} \Big)^{6} > 2. \tag{2.7}$$

The state of the suspended RBCs is advanced in time concurrently with the LB solver, that is, the same timestep Δt is used for the LB fluid and the RBC dynamics. The rigid body dynamics is propagated via a second-order accurate time-stepping algorithm [12], properly modified to handle fluid-particle forces and torques.

3 Implementation

The numerical framework is implemented within the *MUPHY* software [13], a code recently developed to run multi-scale fluid simulations of different kinds. The original *MUPHY* (MUlti PHYsics/multiscale) code is written in Fortran 90 and uses MPI for the parallelization. *MUPHY* makes use of an indirect addressing scheme that has been described along the other main features of the code in [13].

In the parallel processing, the lattice representing the arteries is decomposed into subdomains. As the simulation starts, all mesh points and particles are distributed among processors so that each processor receives the subset corresponding to a subdomain. To maintain a high degree of data locality within each processor, it is necessary to use the same decomposition for both the LB and PD components. This strategy ensures that each processor handles the particles interacting with the fluid associated to the LB mesh assigned to it. An optimal load-balancing for the LB component cannot be achieved by using simple cartesian decompositions in such complex domain. However, a very satisfactory load-balancing is obtained by employing graph-partitioning tools like SCOTCH [9]

or METIS [8] on a graph representing the connectivity of the irregular lattice. These graph-decomposition tools work on a geometry-free representation of the lattice, i.e., a representation lacking any geometrical information and provide a (highly) irregular decomposition. An important aspect introduced in [6] is that of *cell tiling*. In order to efficiently manage RBC-RBC interactions and particles migration across subdomains, the irregular shapes are tiled by space-filling parallelepipeds. The tiling allows to carry out two basic functions: first to track and select particles that fall in proximity of the subdomain surface and, second, to minimize the handling and transfer of information across processors.

MUPHY has been originally developed for the IBM BlueGene systems [14]. More recently, its LB computational core has been ported to clusters of Graphics Processing Units (GPU), using the CUDA software environment, showing excellent results [15]. The porting of the PD core to CUDA allows to run hemodynamics simulations entirely on GPU clusters by avoiding a large amount of data traffic between CPUs and GPUs. In this case, the CPUs are only used to perform the domain decomposition and assist data transfer among GPUs.

At the early stage of development, we decided to design a new GPU version of the PD module instead of porting the existing CPU implementation. In fact, an existing code would have simply posed too many restrictions on the underlying data structures, whereas the new PD code is designed to exploit at its best the GPU capabilities. We thus developed a modular software architecture in such a way that new features are easily incorporated into the existing code.

Traditionally, one of the main issues related to the achievement of good performances on GPUs has been the requirement of having properly aligned accesses to the global memory (*coalesced* accesses in the CUDA jargon). However, devices with capability 1.3 and 2.0 (the *capability* defines the specific architecture of the GPU) can combine memory accesses by threads in a half-warp (that is a group of 16 threads) into a single memory transaction. This weaker notion of coalescence is such that memory access remains efficient as long as data lay in the same segment (that is a block, properly aligned, of 128 bytes), regardless of the memory access pattern. This feature highly mitigates the performance drop due to uncoalesced accesses with respect to older devices, where such accesses were always serialized on a per-thread basis. Since, at the time of this writing, the new devices are widely available whereas devices with capability <1.3 are disappearing, we limited our attention to fulfill the memory alignment requirements of the newer devices.

3.1 Domain structures

Some features of the LB implementation influence distinct aspects of the PD implementation, the most important one being related to the layout of the underling mesh. Given the irregular shape of the spatial domains, the mesh can not be stored in *full matrix* mode since this would imply a huge waste of memory (the bounding box of the domain can

be as large as $10^4 \times 10^4 \times 10^4$ with only 3-5% of the nodes actually used). Instead, the LB module relies on an indirect addressing scheme [15], by storing only active nodes (*fluid*, *wall*, *inlet* and *outlet*). Mesh nodes are stored in one-dimensional arrays, one for each LB discrete velocity. Nodes of the same type (*fluid, wall*, etc) are contiguous in the arrays. The arrays are complemented by a matrix that represents the connectivity of the lattice. For each node, the connectivity matrix contains the array indices of the neighboring nodes. The number of entries in the connectivity matrix depends on the LB model in use. Since we use the D3Q19 scheme, there are 18 populations connecting neighboring nodes, and, for each node, 18 neighbors are indexed (one population relates to fluid at rest and does not require any connectivity information).

The LB indirect addressing scheme does not allow to carry out efficiently one of the most important operations required by PD, that is finding the array index of the grid points (each one indicated as a triplet of integers (i,j,k)) covered by a particle at position (x,y,z). This operation is fundamental to implement the *membership test* that decides whether a particle is located inside a subdomain or not. To overcome this limitation, a new set of data structures has been devised. These data structures are of general use, in that they help in performing all PD operations that take into account spatial relations: particle-particle interactions, *frontier* and *migration management*. These structures are defined at the beginning of the simulation on the CPU, on the basis of the partitioning assigned to each processor and are subsequently transferred to the GPU memory, where they remain unmodified until the completion of the simulation.

The first data structure, that we name $COO2CELL$, is the *cell matrix* that we use to map points in space to cell indices. In order to avoid the full matrix representation, the cell matrix is stored by using the same indirect addressing scheme used to store the mesh. It is represented as two vectors: $COO2CELL[0][\,]$, containing the 1D-coordinate of internal, frontier and external cells, in ascending order, and $COO2CELL[1][\,]$, that contains the tiling indices of the cells. Given a point (x,y,z), the index *cid* of the containing cell is found by first computing the 1D-coordinate of the cell

$$1Dcoo \leftarrow \left\lfloor \frac{z}{c_z} \right\rfloor * m_x * m_y + \left\lfloor \frac{y}{c_y} \right\rfloor * m_x + \left\lfloor \frac{x}{c_x} \right\rfloor,$$

where (c_x,c_y,c_z) and (m_x,m_y,m_z) represent the size of the cells and their number along x, y and z directions, respectively. The 1D coordinate is then searched in the first array by using a binary search

$$j \leftarrow binsearch\big(COO2CELL[0][\,],1Dcoo\big),$$

and finally, the tiling *id* of the cell is obtained by accessing the j-th location of the second vector

$$cid \leftarrow COO2CELL[1][j].$$

The second data structure is the *neighbor matrix*, called $CNEIGH$. This is a two-dimensional matrix that contains for every *internal* and *frontier* cell the ids of the 26 neighboring cells (that can be *internal*, *frontier* and *external* cells). This matrix is not strictly

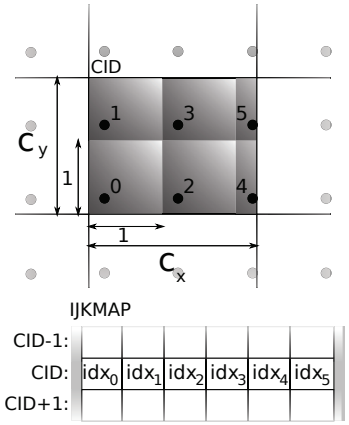

Figure 2: Entries in the IJKMAP matrix for the cell cid; idx_i is the index of the i-th mesh element in the array of nodes.

necessary, since it is a mere "repackaging" of information already present in COO2CELL. However, it simplifies the neighborhood scan when performing particle-particle interactions by avoiding the binary searches (see Section 3.4).

The third and last structure is the *map matrix*, called $IJKMAP$, that defines the association between particles and mesh data. It is a two-dimensional matrix containing, for every *internal* and *frontier* cell, the indices, in the array of LB nodes, of the mesh points inside that cell. Points inside the cells that are not part of the intradomain mesh are stored as -1. Indices are stored in lexicographic order, so that, given a mesh node (i,j,k), it is possible to easily compute the corresponding column index (see Fig. 2 for a 2D example). The index, in the LB array, of a mesh node at position (i,j,k) is:

$$nid \leftarrow IJKMAP[cid][lid],$$

where cid is the index of the cell containing the node and lid is defined by the linearized coordinates of the node in the cell frame of reference

$$i_{cell} \leftarrow \left\lfloor i - \left\lfloor \frac{i}{c_x} \right\rfloor \cdot c_x \right\rfloor, \quad j_{cell} \leftarrow \left\lfloor j - \left\lfloor \frac{j}{c_y} \right\rfloor \cdot c_y \right\rfloor, \quad k_{cell} \leftarrow \left\lfloor k - \left\lfloor \frac{k}{c_z} \right\rfloor \cdot c_z \right\rfloor,$$
$$lid \leftarrow \lceil c_x \rceil \cdot \lceil c_y \rceil \cdot k_{cell} + \lceil c_x \rceil \cdot j_{cell} + i_{cell}.$$

By using these data structures, the membership test is implemented as follows. Given a particle $p = (x,y,z)$, its coordinates are first rounded to the nearest integer, to find the nearest grid point g. Then, it is checked whether an index in the mesh vector for point g exists. If this is the case, the particle p is located inside the domain.

3.2 Particles structures

Each particle is characterized by a set of properties, such as position, velocity, angular velocity, etc. Particles are thus stored as a set of arrays, one for each of these properties,

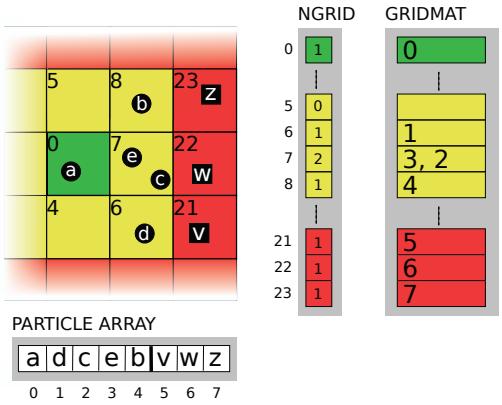

Figure 3: Layout of the *grid matrix*. Internal particles are represented with circles and external ones with squares. Internal, frontier and external cells are colored, respectively, in green, yellow and red.

according to the structure-of-arrays layout. This approach allows consecutive threads to access particles at consecutive memory addresses and it is part of GPU programming *best-practices*. The arrays contain both particles internal to the domain and the external ones at distance less than or equal to the cutoff distance from the domain boundary. Internal and external particles are separated: internal particles are located in the first part of the arrays, followed by external ones. This layout allows threads in charge of internal particles to access contiguous memory addresses. Particles position inside cells is stored in a *grid matrix*, called $GRIDMAT$. It is a 2D matrix that contains, for each cell, the indices of the particles located in the cell. The number of particles varies from cell to cell, so an auxiliary array NGRID stores the number of particles in each cell. Fig. 3 shows a simple example with 8 particles in a 2D grid.

3.3 Frontier management

We now present the GPU implementation of frontier and migration management, as outlined in our previous paper [6].

At the beginning of each iteration, the set of particles located within frontier cells is moved from GPU to CPU memory in order to be exchanged among processors by using MPI primitives. Since particle arrays are unsorted, frontier particles are first gathered in a GPU buffer and then transferred to CPU memory via the *cudaMemcpy* function. The gathering is done by using a map array V_{idx} that contains at position i the index of the frontier particle that is copied to the i-th location of the buffer.

The map is built by invoking a sequence of kernels (i.e., functions running on the GPU) starting from the data stored in the array of cell ids, V_{cell}. The first kernel fills up a binary array V_{mask}, such that $V_{mask}[i]=1$ if $V_{cell}[i]$ is a frontier cell, or 0 otherwise. Then, a parallel reduction kernel scans V_{mask} to compute the number n of frontier particles. If n is non-zero, the parallel prefix sum of V_{mask} is computed into a temporary array V_{ps}. Subsequently, the map array is created by setting $V_{idx}[V_{ps}[i]]=i$ for the entries i such that

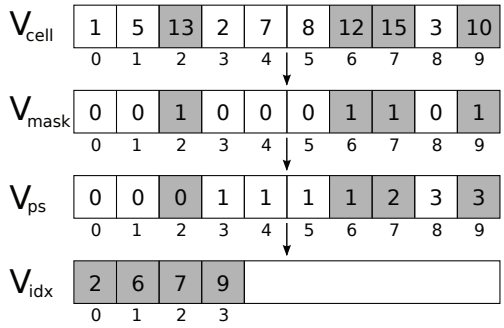

Figure 4: Steps performed to compute the map array required for gathering efficiently the set of frontier particles (subsequent steps are top to bottom and affect the arrays described in the text). Frontier cells are marked in grey.

$V_{mask}[i]=1$. Finally, a kernel with n threads is launched to gather data about the particles indexed by the first n locations of V_{idx} into the buffer. Fig. 4 illustrates an example of this process.

Once particles are copied to the CPU memory, they are exchanged among processors by using MPI point-to-point primitives. On the receiving end, particles are copied from CPU memory to a GPU buffer, their cell *id* is computed and only those particles located within external or frontier cells are moved into the corresponding data structures. This task is carried out by a kernel that runs one thread per received particle. Each thread calculates the cell *id* of its particle and, in case of either a frontier or an external cell c, uses the *atomicInc* function to compute the index where the particle is stored in the destination arrays. The *grid matrix* is also updated by using the CUDA *atomicInc* function to atomically increment the cell counter and store the index j at the corresponding column in the row c of $GRIDMAT$.

3.4 Particle-particle interactions

Every processor updates the particles data structures by using a kernel that implements the force calculation. A sequence of additional kernels propagates other variables of the particles related to linear velocities, angular velocities, torques, derivatives, rotation matrices, cell ids, etc.

Particle-particle interactions are computed by using a GPU implementation of the link-cell algorithm. For each internal particle i, cells CNEIGH[cid_i][$0,\cdots,26$] are searched to identify interacting pairs of particles. Particles located in the neighboring cells are accessed by looking up the corresponding rows of the GRIDMAT matrix. There are at least two ways to map data onto GPU threads for this task. A first possibility is to process pair interactions on a per-cell basis, by using the *grid matrix*. In this case, each thread block is assigned to a cell and each thread of the block computes the force acting on a particle inside the cell. This approach allows threads of the same block to cooperate while scanning the neighborhood of the cell. More in detail, since threads of the same block are

in charge of particles in the same cell, the shared memory of the GPU is used to cache memory accesses to neighboring particles. This is done by copying in shared memory the current neighboring cell and by having threads to scan synchronously the shared copy for interacting pairs. However, this approach may result in a huge waste of resources, since cells typically contain a number of particles much lower than the GPU *warp size* (32 threads, corresponding to the basic scheduling unit). As a consequence, many threads in a *warp* would be idle, as there would be an insufficient number of particles for all of them.

We thus followed a different approach, whereby interactions are processed on a per-particle basis. In this case, the grid of threads is directly mapped onto the arrays of particles. Threads are assigned to particles according to their global *id* and the search for interacting pairs proceeds in an independent fashion. Each thread scans the cell neighbors and, for each interacting pair, it computes the contribution to the total force. Cooperation is not easily achieved because the indices of the particles inside the arrays are not related to their positions in space, so that consecutive threads may have to handle particles located in different cells.

Algorithm 1 shows the pseudocode for this implementation. Lines 1 and 2 compute the size of the grid of threads in execution and define the indexing scheme for particles. Each thread handles the particle corresponding to its linear global index inside the grid. Then, each thread loops over the internal particles. Lines 4, 5 and 6 initialize the force acting on particle *tid*, read its position and the index *cid* of the cell containing it, respectively. Line 7 starts the loop through the 27 cells in the neighborhood of cell *cid*. The index of the current cell is read from $CNEIGH$ (line 8) and the number of particles located inside this cell is read from the $NGRID$ array (line 9).

The innermost loop (line 10) runs over the particles located in the current neighboring cell. The index of the j-th particle is first read from the $GRIDMAT$ matrix (line 11) and its data fetched from the array of particle positions (line 12). In this version, we only consider particle positions, but in real practice, much more data need to be fetched from memory (orientation matrices, universal ids, etc). The test at line 13 prevents the evaluation of the force between a particle and itself. Lines 16 and 17 compute the force between the current pair and add it to the total. Here we do not exploit the action-reaction principle $f_{ij} = -f_{ji}$ to avoid accumulating forces belonging to different threads. Doing so would require atomic operations for floating-point numbers that are not available on the Tesla line of GPUs[†].

Finally, after all neighboring cells have been scanned, the force computed for particle *tid* is saved into the corresponding array and the thread slides to the next internal particle to be processed.

Particles data are read from global memory by using texture fetches via the *tex1Dfetch* function to take advantage of the caching capability of the texture. This results in a performance increase of 5-10% with respect to direct memory fetches. We also tried to sort

[†]The latest generation CUDA architecture, named Fermi, implements atomic operations for floating point numbers.

Algorithm 1. Search for interacting particle pairs and computation of pairwise forces.

Require: CNEIGH, NGRID, GRIDMAT
Require: V_r is the array of particles positions.
Require: V_{cell} is the array of cell indices.
Require: V_{force} is the array of forces.
Require: N_{int} is the number of internal particles.

```
1: n ← gridDim.x · blockDim.x
2: tid ← blockIdx.x · blockDim.x + threadIdx.x
3: while (tid < N_int) do
4:    f_tot ← 0
5:    r_tid ← tex1Dfetch(V_r[tid])
6:    cid ← tex1Dfetch(V_cell[tid])
7:    for c = 0 to 26 do
8:       ncell ← CNEIGH[cid][c]
9:       N_ncell ← NGRID[ncell]
10:      for j = 0 to N_ncell − 1 do
11:         idx_j ← GRIDMAT[ncell][j]
12:         r_j ← tex1Dfetch(V_r[idx_j])
13:         if (r_j == r_tid) then
14:            continue
15:         end if
16:         k ← |r_tid − r_j| ≤ r_max ? 1 : 0
17:         f_tot ← f_tot + k · force(r_tid, r_j)
18:      end for
19:   end for
20:   V_force[tid] ← f_tot
21:   tid ← tid + n
22: end while
```

the arrays of particles by the cell *id* in order to limit the cache miss rate and improve the coherence of memory accesses. However, numerical tests showed that the cost of keeping particles in sorted order (periodic sorting via the radix sort implementation, provided by the *cudpp* library [16]) is greater than the performance gain provided by the slightly higher cache hit rate, so we resorted to using texture lookups on unsorted arrays. Threads executing this kernel may diverge in the innermost loop, in case of cells containing a different number of particles. Kernel profiling, however, shows that divergent warps have a negligible impact in this kernel.

3.5 Particle migration

Once the update is completed, particle arrays and the *grid matrix* need to be synchronized, the reason being that the particles inside the arrays may have moved outside the

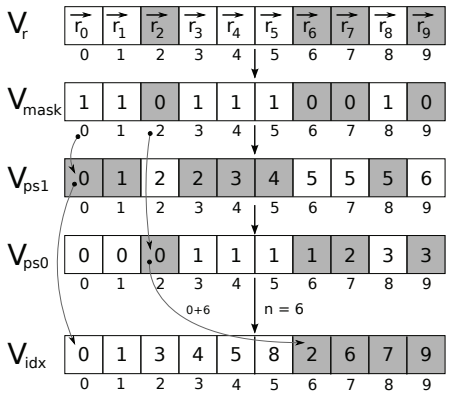

Figure 5: Steps performed to compute the map array used to separate internal and external particles inside the second buffer of arrays. Outgoing particles are marked in grey (subsequent steps are top to bottom and affect the arrays described in the text).

subdomain or to different cells. Consequently, the indices stored in the *grid matrix* are no longer valid due to the reallocation among cells.

The arrays and the *grid matrix* are set back in a coherent state in the migration management phase. In this phase, particles that moved outside the domain are identified and exchanged with neighboring processors and newcomers are inserted in the data structures. The search for departing particles is done in a similar way to frontier management. A new map vector V_{idx} is built, in order to permute updated particles data in a second buffer of vectors. In such way, internal particles are placed at the beginning of the arrays, followed by those that moved outside.

The map is computed by invoking a sequence of kernels starting from the data stored in the array of particle positions V_r. A first kernel computes a mask vector V_{mask}. Each thread applies the membership test to a particle and sets the corresponding location of V_{mask} to 1, if the particle remained inside the domain, otherwise sets it equal to 0. Then, a parallel reduction kernel scans the mask vector to compute the number n of the particles that remained in the domain. As for the frontier management, a prefix sum vector V_{ps1} is computed to assign destination indices to internal particles. However, in this case, a second prefix sum vector V_{ps0} is built on a copy of V_{mask} by swapping ones and zeroes. This vector is necessary to compute destination addresses for external particles that follow the internal ones in the second buffer of arrays. The permutation vector V_{idx} is finally built by setting:

$$\begin{aligned} V_{idx}[V_{ps1}[i]] &= i, && \text{if } V_{mask}[i]=1, \\ V_{idx}[V_{ps0}[i]+n] &= i, && \text{if } V_{mask}[i]=0. \end{aligned}$$

Fig. 5 illustrates an example of this process.

The permutation vector is used to separate particles inside the second buffer, by copying at position i the particle at position $V_{idx}[i]$ in the first buffer. External particles are then moved to CPU memory and the *grid matrix* is rebuilt with the data of the new n particles.

External particles are exchanged among processors via MPI calls and the received particles are moved to GPU memory to identify newcomers. The procedure is basically the same as for the frontier management. The membership test is applied to each particle and only those particles that, actually, moved inside the domain are inserted into the corresponding arrays and into the *grid matrix*.

4 Performance tests

To evaluate the performances of our implementation, we ran a set of tests on a cluster equipped with 32 NVIDIA GPUs. The cluster has 16 computing nodes connected by Infiniband. Each node has an Intel Xeon Quad-Core E5520, equipped with 24GB of RAM and connected to a pair of NVIDIA Tesla C1060 GPUs (capability 1.3), each one having 4GB of global memory.

The testcase is the system representing an artery bifurcation derived from real-life tomographic data for which we simulate fluid and particle dynamics and it is shown in Fig. 1. The ensemble of RBC consists of 5×10^5 particles, immersed in the LB solvent and with a mesh made of $\sim$6 million active fluid nodes.

At first, we compared the GPU and CPU implementations of the code and measured the strong scaling of the system, by running the same testcase on an increasing number of GPUs, ranging from 1 to 32.

The speedup, measured on the total running time of the simulation (LB, PD and LB-PD coupling), obtained by the GPUs with respect to the CPUs, is shown in Fig. 6. With any number of processors the GPU implementation is more than 14 times faster than the CPU reference code. The resulting curve is not smooth, due to the domain decomposition. Given the irregularity of the domain, the SCOTCH graph-partitioning tool produces partitionings with a balancing whose quality slightly varies with the number of subdomains.

Fig. 7 shows a plot of the parallel efficiency of the complete simulation (including LB, PD and LB-PD coupling) for both GPU and CPU implementations. Both versions achieve an efficiency above 84% with any number of processors. However, while the CPU version almost immediately reaches the minimal efficiency, the GPU code runs with an efficiency greater than 96% up to 14GPUs from where it begins to decrease down to 84%, with 32GPUs. This effect is due to the relative small size of the test case. On the other hand significantly larger test cases would not fit in memory with few GPUs.

Fig. 8 reports the timings of the simulation components including the coupling between LB and PD, and the breakdown in terms of computation of pairwise forces, frontier and migration management. To measure the kernels execution times without the overhead of MPI transfers, frontier and migration management timings have been split into i) the time required to gather and move data from GPU to CPU memory and ii) the time required to exchange data among processors. It is apparent that the coupling between the LB and PD methods incurs the largest cost, regardless of the number of processors, while the computation of pairwise forces takes at most one third of the total time. The cost of

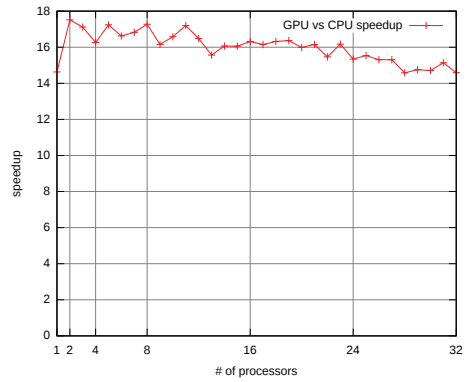

Figure 6: CPU run time/GPU run time for the complete simulation (PD, LB and coupling) plotted vs number of processors.

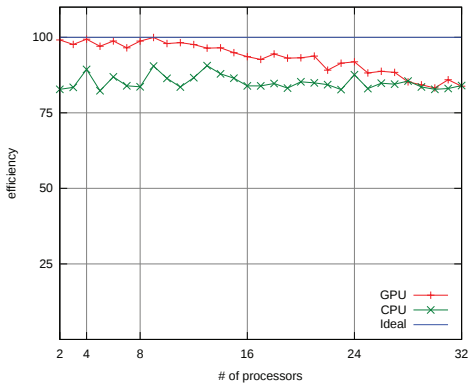

Figure 7: Parallel efficiency of both GPU and CPU implementations of the complete simulation (including the modules for PD, LB and PD-LB coupling).

frontier and migration management (both data gathering and MPI transfer) is negligible up to 8GPUs.

The high cost of the LB-PD coupling is due to the large number of GPU global memory writes that cannot be combined together. As a matter of fact, each particle exerts a feedback on eighteen LB populations belonging to 32 nodes covered by the particle. Since the subdomains assigned to each GPU are rather irregular and the number of particles is much smaller than the number of mesh points, it is highly unlikely that the populations updated by consecutive particles lay within the same memory segment. For such reason almost every memory write is translated into a single memory transaction.

The timings breakdown of the CPU implementation is similar to that of the GPU code. It is worth noticing that in spite of the above mentioned issue for the LB-PD coupling, the GPU is almost one order of magnitude faster than the CPU in executing the coupling part of the code.

Regarding the performance of the PD component alone (without the LB-PD coupling), most of the execution time is taken by the computation of forces. With one processor,

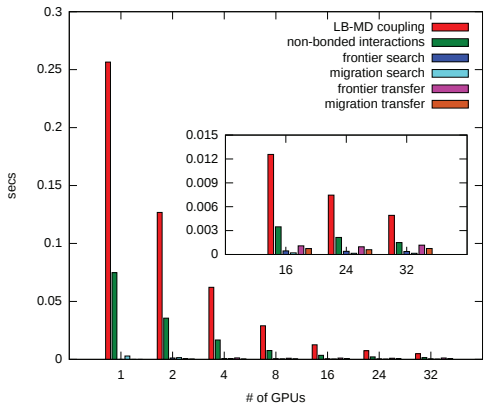

Figure 8: Time required to perform the main operations of PD and the coupling with the LB method vs number of GPUs. The inset contains a detail of the timings for 16, 24 and 32GPUs.

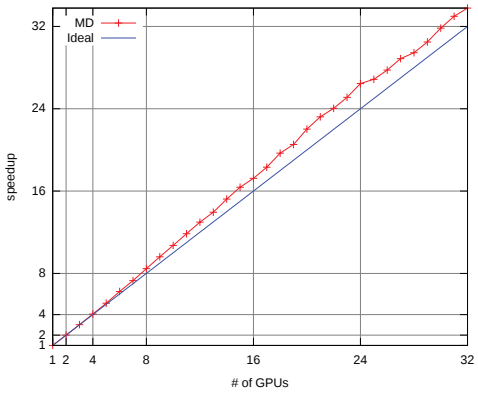

Figure 9: Speed-up of the PD implementation including the PD-LB coupling.

the GPU implementation is more than 20 times faster than the CPU one. The speed-up increases with the number of processors up to a factor of 34, achieved with 32 processors.

The time required by the gathering and transfer of data in frontier and migration management is negligible up to 16 processors. Starting with 24GPUs, the time required by frontier and migration management becomes comparable with that of forces computation and coupling, given the reduced workload of each GPU.

Fig. 9 shows a plot of the speed-up of the complete PD computational component with the LB-PD coupling activated. The speed-up is super linear with any number of GPUs.

5 Conclusions

To the best of our knowledge, the work presented in this paper represents the first effort to design and implement a general method to concurrently perform parallel Lattice Boltz-

mann and Particle Dynamics inside irregular domains. The two simulation modules are coupled in a concurrent and efficient way by executing most of the computational load on a cluster of GPUs. For the single core vs. single GPU comparison, the measured speed-up is in excess of one order of magnitude in favor of GPU computing. This speed-up is highly satisfactory, especially in view of the complex data layout and computational method. When going to multi-GPU parallel computing, handled by a MPI/CUDA programming paradigm over the hybrid CPU/GPU architecture, the parallel efficiency stays close to the ideal slope up to 32GPUs. The computational method is fully integrated in MUPHY, one of the first examples of high performance parallel code for the simulation of multi-physics/scale bio-fluidic phenomena in realistically complex geometries.

References

[1] M. Fyta, S. Melchionna, E. Kaxiras and S. Succi, Multiscale coupling of molecular dynamics and hydrodynamics: applications to DNA translocation through a nanopore, Multiscale Modeling and Simulation, 5 (2006), 1156–1173.
[2] M. Fyta, S. Melchionna, S. Succi and E. Kaxiras, Multiscale simulation of nanobiological flows, Comput. Sci. Eng., March/April, 2008.
[3] R. Benzi, S. Succi and M. Vergassola, The lattice Boltzmann equation: theory and applications, Phys. Rep., 222(3) (1992), 145–197.
[4] S. Melchionna, M. Bernaschi, S. Succi, E. Kaxiras, F. J. Rybicki, D. Mitsouras, A. U. Coskun and C. L. Feldman, Hydrokinetic approach to large-scale cardiovascular blood flow, Comput. Phys. Commun., 181 (2010), 462–472.
[5] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, 1994.
[6] M. Bisson, M. Bernaschi and S. Melchionna, Parallel molecular dynamics with irregular domain decomposition, Commun. Comput. Phys., 10 (2011), 1071–1088.
[7] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone and J. C. Phillips, GPU Computing, Proc. IEEE, 96(5) (2008), 879–899.
[8] `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.`
[9] `http://www.labri.fr/perso/pelegrin/scotch/.`
[10] S. Succi, The Lattice Boltzmann Equation for Fluid Dynamics and Beyond, Oxford University Press, USA, 2001
[11] J. G. Gay and B. J. Berne, Modification of the overlap potential to mimic a linear site-site potential, J. Chem. Phys., 74 (1981), 3316–3319.
[12] A. Dullweber, B. Leimkuhler and R. McLachlan, Symplectic splitting methods for rigid body molecular dynamics, J. Chem. Phys., 107 (1997), 5840–5851.
[13] M. Bernaschi et al., MUPHY: A parallel MUlti PHYsics/scale code for high performance bio-fluidic simulations, Comput. Phys. Commun., 180 (2009), 1495–1502.
[14] `http://www.ibm.com/systems/deepcomputing/bluegene/.`
[15] M. Bernaschi, M. Fatica, S. Melchionna, S. Succi and E. Kaxiras, A flexible high performance Lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries, Concurrency Prac. Ex., DOI: 10.1002/cpe.1466 (2009).
[16] `http://gpgpu.org/developer/cudpp.`