Acceleration of Coarse Grain Molecular Dynamics on GPU Architectures

Ardita Shkurti[1], Mario Orsi[2], Enrico Macii[1], Elisa Ficarra[1*], Andrea Acquaviva[1*]

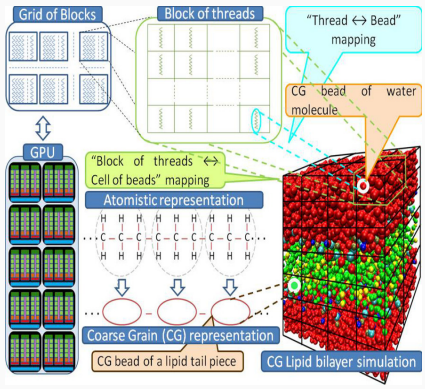
[1]Department of Control and Computer Engineering,
Politecnico di Torino, Corso Duca degli Abruzzi 24, 10129 Torino, Italy.
E-mail: {*ardita.shkurti, enrico.macii, elisa.ficarra, andrea.acquaviva*}*@polito.it*

[2]School of Engineering and Materials Science,
Queen Mary, University of London, Mile End Road, London, E1 4NS, United Kingdom.
E-mail: *m.orsi@qmul.ac.uk*

[*] Ficarra and Acquaviva are principal investigators

Keywords Molecular Dynamics, GPU Acceleration, Coarse Grain, Membrane Modeling, CUDA.

Abstract *Coarse grain (CG) molecular models have been proposed to simulate complex systems with lower computational overheads and longer timescales with respect to atomistic level models. However, their acceleration on parallel architectures such as Graphic Processing Units (GPU) presents original challenges that must be carefully evaluated. The objective of this work is to characterize the impact of CG model features on parallel simulation performance. To achieve this, we implemented a GPU-accelerated version of a CG molecular dynamics simulator, to which we applied specific optimizations for CG models, such as dedicated data structures to handle different bead type interactions, obtaining a maximum speed-up of* 14 *on the NVIDIA GTX480 GPU with Fermi architecture. We provide a complete characterization and evaluation of algorithmic and simulated system features of CG models impacting the achievable speed-up and accuracy of results, using three different GPU architectures as case study.*



Acceleration and Optimization of a Coarse Grain (CG) Molecular Dynamics Simulator is performed by porting its main computational kernels to Graphic Processing Unit (GPU) architectures using CUDA language. A detailed analysis on CG features and their impact on the achievable speed-up is reported for different GPUs, timestep values and arithmetic precision. Correctness of the simulation accuracy is demonstrated. Simulations in recent GPU architectures showed more stable trend of energy values compared to simulations in CPU.

1 Summary

In this work we characterize specific bottlenecks of CG models proposing specific strategies for handling them, through the implementation of an accelerated CG model. We identify the impact of interaction computation, data structures and memory transfers between CPU and GPU, on the optimization of the CG application. In particular, we evaluate the impact of additional data structures needed to take into account interactions among heterogeneous bead types. Then, we explore the speed-up achieved across various architectures characterized by different parallelism and computational capabilities. We analyze the impact of the complexity of the system, that is the amount of lipid beads, on the simulation performance. In addition, we investigate the impact of the timestep duration on the achievable speed-up. We use several techniques leveraging on arithmetic operations, data reuse and work merge to raise the simulator performance and subsequently for each of them assess the accuracy of the simulations.

2 Introduction

Most aspects of cell activity may be perceived observing the dynamics of cell membranes which are composed of patterns of lipid bilayers. Such aspects include, among others, the conformation of embedded proteins which are directly influenced by the bilayer structure, the biomembrane permeability, interaction with drugs and signaling (Orsi et al. [2008]).

The value of such biological processes brings to an increasing appeal for hardware and software optimized realistic models of simulation which may provide *in silico* access to biological cell mechanisms otherwise impossible to achieve through real experiments.

Molecular Dynamics (MD) is a technique of simulation in which sets of atoms and molecules interact with each other over a certain time interval according to known physics laws. It is unfeasible to describe analytically the properties of such complex systems, due to the considerable quantity of particles they are composed of. Hence, MD emerges as a virtual laboratory, that is as an interface between laboratory experiments and theory. Depending on the granularity of the particle model, MD can be distinguished between atomistic level (AL) and coarse grain (CG) level.

AL models (Wohlert and Edholm [2006], MacCallum and Tieleman [2006], Hgberg and Lyubartsev [2006]) require considerable computational resources, due to the necessity of interaction calculations between each atom and all other atoms in the system and the considerable amount of data involved in these calculations. Moreover, the reliability of the calculation of some essential features, such as diffusion coefficients or the lateral pressure profile, which require a large number of simulation steps, can be undermined by insufficient sampling.

CG modeling techniques have been proposed to address the shortcomings of atomic level models, in that they allow longer timescales and thus a better sampling of relevant properties especially for large macromolecules such as lipids (Orsi et al. [2008]), proteins (Dong and Zhou [2011]) and RNA (Orsi et al. [2008]). In a CG representation specific atom groups are organized in clusters and each of the clusters is considered as a unique particle called a *bead*. Hence, by lowering the number of system particles, computational requirements are reduced as well. Figure 1 illustrates the relationship between AL and CG modeling for a lipid tail piece, where

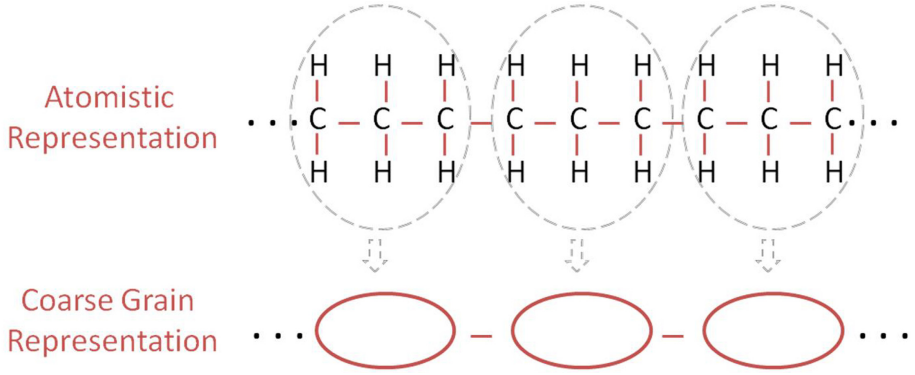


Figure 1: Coarse grain mapping of an atomistic representation of a lipid tail piece (at the top) to the corresponding coarse-grained bead (at the bottom).

27 atoms are clustered into only 3 beads. Typically, a lipid represented by about 100 particles in AL approaches is modeled by approximately 10 beads in CG representations (Orsi et al. [2008], Mller et al. [2006]).

A consequence of a bead-type organization is that interactions are now dependent on the bead type, compared to the single type of atom-atom interaction. While CG models are promising and may allow simulations at a longer time scale, the complexity of the biological systems they can simulate in reasonable time is still limited. There are several interesting biological phenomena requiring higher time scales than what currently reachable by CG simulators.

For this reason, techniques for accelerating CG simulators need to be developed. GPUs have emerged as a powerful architecture for MD acceleration (Bauer et al. [2011], Friedrichs et al. [2009], Stone et al. [2007], Schmid et al. [2010]). While previous work on GPUs mainly addressed AL models (Bauer et al. [2011], Rapaport [2011], Ganesan et al. [2011], Friedrichs et al. [2009], Anandakrishnan et al. [2010], Stone et al. [2011, 2010], Liu et al. [2008]), acceleration and optimization of CG models have only recently gained attention (Shkurti et al. [2010], Anderson et al. [2008], Zhmurov et al. [2010], Anandakrishnan et al. [2010], Sunarso et al. [2010], Shkurti et al. [2012], van Meel et al. [2008]).

Due to their features, CG models impose specific challenges that must be carefully addressed. In this paper we characterize these features from a computational viewpoint, evaluating the consequences on different GPU architectures. To the best of our knowledge, this is the first paper providing a comprehensive characterization and evaluation of CG acceleration on GPUs.

In this work we first provide background about the programming environment exploited and the simulated model. Thereafter, the main optimization methods are characterized followed from hardware specifications and simulation setup. Then, we show the results achieved and analyze the accuracy of the simulations. Afterwards, further detailed optimizations performed are described and the achieved results are discussed. Immediately before concluding the work, we also report a *state of the art* review.

3 CUDA Environment

The Compute Unified Device Architecture (CUDA) environment represents a parallel programming model and instruction set oriented to highly parallel computing. CUDA is designed for extending NVIDIA GPU programming to general purpose parallel applications (i.e. not only graphic). In GPUs much more transistors are assigned to data processing rather than to flow control or data caching with respect to general purpose CPUs. However, memory latency is hidden with compute-intensive calculations (CUDA [2011]).

The GPU is divided into a set of Streaming Multiprocessors (SM) in which hundreds of *threads* reside concurrently. In Figure 2, the 30 SMs of the GTX295 GPU architecture and their components are highlighted.

CUDA threads may access information from several memory spaces during their execution: *(i)* Each thread has its own private *local* memory; *(ii)* All threads of a CUDA block have a *shared* memory (on-chip) visible only at block level and with the same duration as the block; *(iii) Global* memory is accessible from all the threads; *(iv) Constant* and *texture* memory spaces are read-only and cached memory spaces accessible by all threads. In our work, we use all of these memory spaces but however we will not examine the details of each of them.

Threads are organized in *blocks* and only threads belonging to the same block can communicate with each other by means of shared memory or synchronization barriers. Instead, different block threads are independent and can be executed in the GPU in any order on any available GPU core. Each thread has a unique identifier in the block which in his turn has a unique identifier among the program blocks (CUDA [2011]).

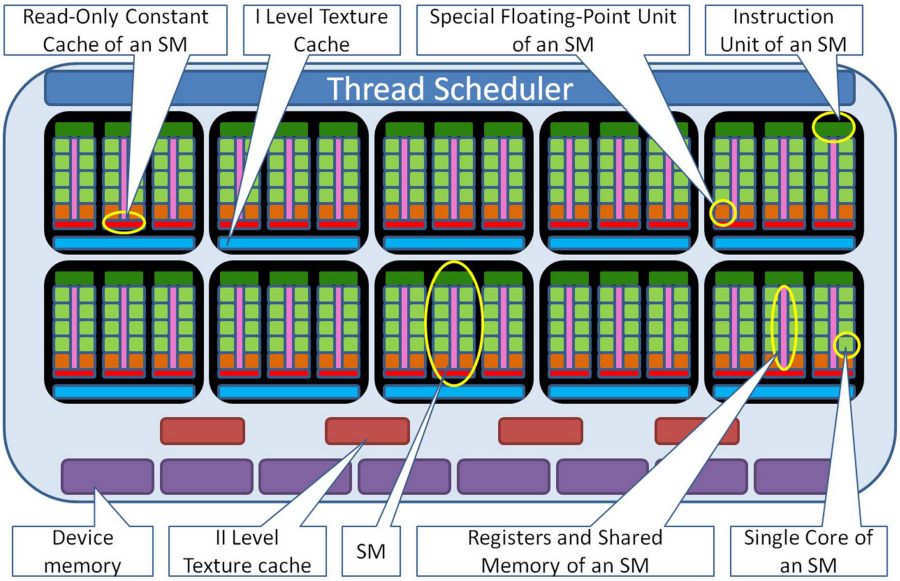


Figure 2: GTX295 GPU architecture components from a parallel processing point of view.

As a consequence, problems adapted to GPUs are divided into independent sub-problems to be scheduled in different thread blocks. Sub-problems are in their turn divided into even finer parts that can be solved by threads of the same block.

The minimum hardware execution unit in the CUDA environment is called *warp* and contains 32 threads. Each of the SM stores the information related to the instruction counter, register states and other thread context information for each active *warp*. When a warp is waiting for the result of a long latency operation, context switching is performed to another resident warp that is ready for execution, according to a priority mechanism. Therefore, the cost of context switching from a resident warp to another is negligible with respect to CPUs, and memory latency is hidden as well.

Furthermore, context conservation enables thread divergence, as it keeps independent thread information for each divergent flow. Clearly, since divergent threads are executed sequentially, a price is paid in terms of performance.

A large number of CUDA threads can be launched in parallel by means of *kernels*, representing C functions executed in parallel by all CUDA running threads (CUDA [2011]). Anyway, the number of threads and consequently warps residing concurrently in an SM is limited since they must share the limited resources available for that SM such as registers and shared memory.

4 Coarse Grain Simulation

In Algorithm 1, the main steps of the MD simulation are reported. Here r, t, F, V, a, m, v, i represent respectively position coordinates, time in the simulation, force, particles interaction potential, acceleration, mass, velocity and bead indexes. For example, $r_{(i,t+\triangle t)}$ represents the position coordinates of bead i at the instant of time $t + \triangle t$. At the beginning of the simulation, initial positions and velocities are assigned to all beads, and a certain timestep value is chosen (line 1).

Algorithm 1 Main steps of MD Simulation

1: Give particles initial positions $r_{(t=0)}$ and velocities $v_{(t=0)}$; set timestep $\triangle t$
2: **for** $timestep := 0$ **to** $Total\ number\ of\ timesteps$ **do**
3: $F_{(i,t)} \leftarrow -\delta V_{(r_i,t)}/\delta r_i$; $a_{(i,t)} \leftarrow F_{(i,t)}/m_i$; $a_{(i,t)} \leftarrow \delta v_{(i,t)}/\delta t$; $v_{(i,t)} \leftarrow \delta r_{(i,t)}/\delta t$
4: $r_{(i,t+\triangle t)} \leftarrow r_{(i,t)} + v_{(i,t)} * \triangle t + 1/2 * a_{(i,t)} * \triangle t^2 + ...$
5: Move time forward: $t \leftarrow t + \triangle t$
6: **end for**

Then, forces acting on each bead are calculated (line 3) and atoms are moved according to the interaction potential of particles in the system (line 4). These forces refer to bonded interactions among particles due to chemical bonds and non-bonded interactions that include electrostatic and van der Waals forces. Van der Waals

forces account for all attractive and repulsive forces among molecules (or among pieces of the same molecule) that are not related to covalent bonds or to electrostatic interactions (McNaught and Wilkinson [1997]).

At last, time is moved forward according to the timestep value chosen (line 5). Actions from lines 3 to 5 are repeated until the number of the desired simulation steps is reached.

The steps described in Algorithm 1 apply to both AL and CG simulations. However, while in AL models a particle is represented by a single atom, in CG models a particle is represented by a bead, which is a cluster of atoms. In addition, force computation has to take into account the different types of pairwise interactions arising between beads, in CG models.

In this work, we perform an optimization and acceleration for CUDA environment of a CG simulator called BRAHMS[1]. BRAHMS can simulate lipid bilayers (Orsi et al. [2008, 2010]), as well as more complex membrane systems (Orsi et al. [2009, 2011]). BRAHMS was originally implemented in the C language as a serial code. The main MD algorithms were developed following a standard approach (Rapaport [2004]). In particular, Newton Leapfrog Equations are used to move the positions of beads (which in CG models represent clusters of atoms) and their velocities one time step forward. BRAHMS uses the *neighbor list* method to optimize the calculation of interactions: For the calculation of the non-bonded interactions among the beads of the system, a neighbor structure is needed to avoid considering a contribution for each pair of beads, which would lead to a *quadratic* time complexity. In this way, only bead pairs with a distance value under a cut-off distance value are considered as neighbor beads.

At every step, after the forward time motion of the system obtained through the equations of motion integration (generalized in line 4 of Algorithm 1), a check is performed to verify whether the structure containing the neighbor beads is to be updated or not. Indeed, an update of the neighbor structure is necessary each time a bead is displaced over a certain distance threshold.

Then, the bonded force interactions among system particles are calculated, which are followed from the computation of non-bonded forces interactions among system beads. This is the most computational intensive part of BRAHMS and of MD codes in general (Bauer et al. [2011], Rapaport [2011], Friedrichs et al. [2009], Stone et al. [2011, 2010], Liu et al. [2008], Shkurti et al. [2010], Anderson et al. [2008], Zhmurov et al. [2010], Sunarso et al. [2010], van Meel et al. [2008], Stone et al. [2007], Phillips et al. [2005], Nguyen et al. [2011], Harvey et al. [2009], Colberg and Hfling [2011], Rapaport [2004]). After each simulation step, system properties such as temperature, energy and pressure are updated.

BRAHMS includes CG beads treated as symmetric rigid bodies such as Gay-Berne (GB) (Gay and Berne [1981]) ellipsoid molecules (with two non-zero moments of inertia), non-symmetric rigid bodies such as water molecules modeled with the Soft Sticky Dipole (SSD) (Liu and Ichiye [1996]) potential (having three non-zero moments of inertia) and simple point-masses beads modeled with standard isotropic potentials (such as LJ and Coulomb) Orsi et al. [2008]. In addition to translational motion, rotational motion should be considered for rigid body beads as well, at every step, during forward time motion. Rigid-body constraints relate to this rotational motion of rigid-body beads.

5 GPU Optimization of Coarse Grain Models

Code profiling of BRAHMS was performed. In Table 1 the most computationally demanding parts of the code are reported. Consequently, we focused on the optimization of these three parts.

Part of application	Related percentage
Non-bonded forces computation	94.3%
Integration of equations of motion	2.3%
Neighbor structure generation	1.1%

Table 1: Bottlenecks and their percentage in terms of execution time (averaged over the total number of simulation steps) in the case of the sequential application.

Note that besides non-bonded forces calculations, which account for the largest execution time, the execution of the other two parts of the code on the GPU is required to avoid large CPU-GPU data transfer overheads.

Accordingly to the bottlenecks identified, we have implemented 4 *kernels* to be executed on the device (GPU): One kernel for the first integration step, one kernel for the neighbor structure generation followed by an auxiliary kernel for the verification of the correctness of the neighbor structure generation and finally the last kernel for the calculation of non-bonded forces and accomplishment of the second integration step.

[1] http://www.soton.ac.uk/~orsi/brahms/

5.1 Kernel Implementing the First Equation of Motion Integration

We implemented the integration algorithm on the GPU to avoid additional transfers of data between CPU and GPU needed by the non-bonded forces computation and neighbor structure generation kernels.

This first integration part of Leapfrog equations is performed at the beginning of each single step of the simulation while the second integration part (similar to the first one) is performed at the end, after eventual neighbor structure generation and forces calculation. Hence, we have implemented a kernel called *integration* (Algorithm 2) to perform the first Leapfrog equation while we have included the implementation of the second Leapfrog equation at the end of the kernel for non-bonded interactions computation called *forces*.

Algorithm 2 Algorithm that implements the first equation of motion integration.

```
1:  i ← blockIdx.x * blockDim.x + threadIdx.x
2:  if i < number of beads then
3:      velocity_(i,t+timestep/2) ← velocity_(i,t) + timestep/2 * force_(i,t)/mass_i
4:      position_(i,t+timestep) ← position_(i,t) + timestep * velocity_(i,t+timestep/2)
5:      if bead_i has an orientation then
6:          half − advance momenta of inertia of bead i
7:          calculations related to rotational motion of bead i
8:      end if
9:      if pressure is controlled then
10:         coordinates of bead i are rescaled
11:     end if
12:     boundary conditions of bead i are applied
13: end if
```

In Algorithm 2, i is a variable which links the identifier of a CUDA *thread* (among all active threads) and the identifier of a specific bead of the system simulated. The CUDA variables *blockIdx*, *blockDim* and *threadIdx* represent respectively the block identifier, the number of threads in a CUDA *block* and the identifier of the current thread in that block. We can obtain the unique identifier of a *thread* among all threads launched for the application by means of these CUDA variables. The calculation of the thread identifier (line 1) is followed by a check (line 2), to determine whether or not this identifier can be linked with any bead identifier[2]. In the affirmative case the calculations related to the specific bead the thread is linked to (the bead having identifier i), are performed (lines 3 to 12). $velocity_{(i,t)}$, $position_{(i,t)}$ and $force_{(i,t)}$ (lines 3 and 4) stand for the velocity, position in the system and forces acting on the bead identified by i at time t. *timestep* (lines 3 and 4) indicates the timestep of the simulation. The orientation (line 5) relates to the rotational motion of the beads considered as rigid bodies. In the MD context, pressure can be controlled during simulations (lines 9 and 10) by a uniform isotropic volume change brought about by rescaling the atomic coordinates (Orsi et al. [2008], Rapaport [2004]). Boundary conditions are applied at the end of the kernel (line 12), so as to capture the typical state of an interior bead (Orsi et al. [2008], Rapaport [2004]).

5.2 Kernel for the Generation of Neighbor and Interaction Type Structures

This kernel (called *cudaneigh*) is used to update neighbor list and data structures which are used by non-bonded forces kernel. The optimization of these data structures is then critical to improve the performance of the most demanding computation of the code.

In addition, accelerating this part of the code, even if its load in the sequential execution time amounts to 2%, prevents this contribution from becoming a bottleneck in the accelerated version.

5.2.1 Optimized Data Structures

Implementation and update of data structures within the GPU is needed to obtain maximum performance from GPU by avoiding data transfer overheads between CPU and GPU.

Cell structure

For the calculation of the non-bonded forces among the beads of the system, a neighbor structure is needed in order to avoid considering a contribution for each pair of beads, which would lead to a *quadratic* time complexity. For this reason the system is divided into cells of beads, where each cell has a cubic shape and the

[2]The number of total threads launched for this kernel on the GPU is *Number of blocks * Number of threads per block*. The *Number of blocks* is calculated as the ceiling of the value of $\frac{Number\ of\ beads}{Number\ of\ threads\ per\ block}$. If the number of beads is not fully divided by the number of threads per block, the last CUDA block of threads will have $ceiling(\frac{Number\ of\ beads}{Number\ of\ threads\ per\ block}) * Number\ of\ threads\ per\ block - Number\ of\ beads$ threads that are not linked to any beads of the system and should perform no calculations for our system.

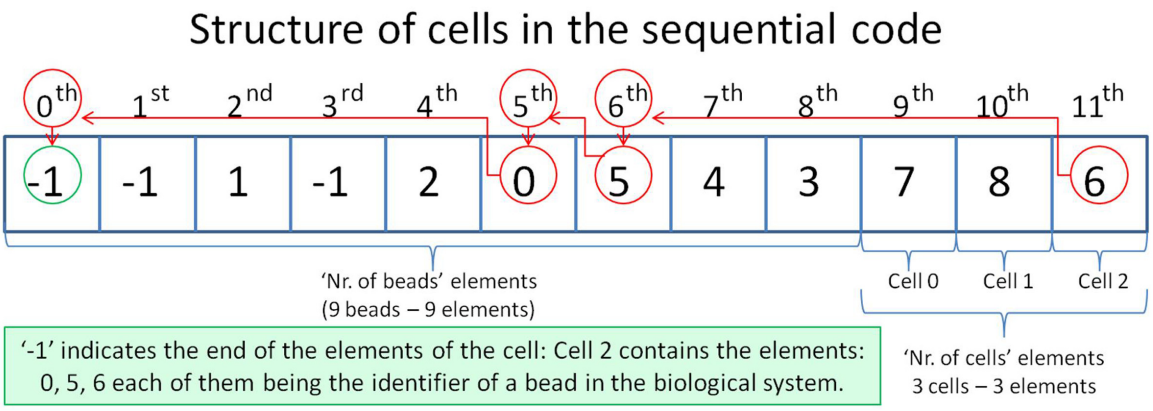


Figure 3: An example of the vector containing the system subdivision in cells in the CPU version implementation of the application code. The circles and arrows illustrate the way in which the elements of a certain cell are stored in the cells structure. The last $Number\ of\ Cells$ elements with vector indexes varying from $Number\ of\ beads$ to $Number\ of\ beads + Number\ of\ Cells - 1$, refer to the Cells of the system: from 0^{th} Cell to $(Number\ of\ Cells - 1)^{th}$ Cell. Each value has two roles: *(i)* It indicates one bead identifier belonging to the related cell and *(ii)* it indexes a cell vector position which contains a value having the same two roles. When one of these values is -1, the cell *list* ends.

edge size equal to the cut-off distance. Only bead pairs with a distance value lower than that of the cut-off are considered as neighbor beads. Hence, the search for neighbors is applied to bead pairs of the same cell and the eventual 26 adjacent cells (Rapaport [2004]).

In the CPU version, the cell structure is a monodimensional vector organized as a linked list with $Number\ of\ beads + Number\ of\ Cells$ elements. An example of the cell structure for the CPU version is reported in Figure 3.

Such a structure cannot be efficiently mapped on a GPU, as it would not provide *coalesced* accesses. Memory accesses are optimized for coalesced accesses where all threads in a warp access the same relative memory address, calculated as $base\ address + absolute\ thread\ identifier$.

To enable coalesced accesses to the cell structure, we organized it as a bidimensional matrix. In Figure 4 we show these matrix values for the same example that we reported in Figure 3. Its row identifier relates to the cell identifier, while it has a number of elements equal to $Number\ of\ Cells * Maximum\ beads\ per\ cell$, where this latter value indicates the maximum number of beads one cell can have in the simulated system. The elements of this matrix represent the identifier of the beads contained in the related cell. For example, the first row of the matrix refers to the cell of the system with identifier 0 which holds beads $1,\ 2,\ 4\ and\ 7$. In addition, we also introduced an auxiliary vector which has $Number\ of\ Cells$ elements and contains, for each element, the number of beads residing in the cell indicated by that element.

We leave the task of cell matrix and related auxiliary structure construction to the CPU. Upon completion, we transfer both to the device. The cell matrix is transferred to the global device memory. The related auxiliary structure is transferred to the *constant* device memory as it is frequently accessed from the GPU during the generation of the neighbor structure, when the iterations among all beads of the considered 27 adjacent cells are performed, to identify the neighbors of each bead.

Neighbor and Interaction Structures

After the cell structure, the neighbor and interaction structures are generated. They store information on bead pairs involved in the force computation and the type of bead-to-bead interaction. Hence, these structures are accessed by the non-bonded forces kernel, to retrieve information on neighbor beads and their interaction types.

Compared to its sequential counterpart (Orsi et al. [2008]) which does not provide coalesced accesses for the GPU, we implemented a dedicated structure to store information about bead interaction types. The size of the sequential version of the neighbor structure is $2 * Number\ of\ neighbor\ pairs$. To fill this structure an algorithm is used to detect the neighbors of each bead.

Each pair of neighbor beads is stored in the neighbor structure and, for each of them, their interaction type is stored in the interaction type structure.

Following the example in Figures 3 and 4, let us suppose that there are 7 pairs of neighbor beads among all possible pairs from the combination of the 9 beads of the system. Let these pairs be $(0,\ 3)$, $(0,\ 5)$, $(0,\ 6)$, $(1,\ 2)$, $(1,\ 4)$, $(3,\ 8)$, $(4,\ 7)$. Let us consider 6 types of interactions identified by numbers from 0 to 5. In Figure 5, we show the related structures for neighbors (upper side) and interaction types (lower side).

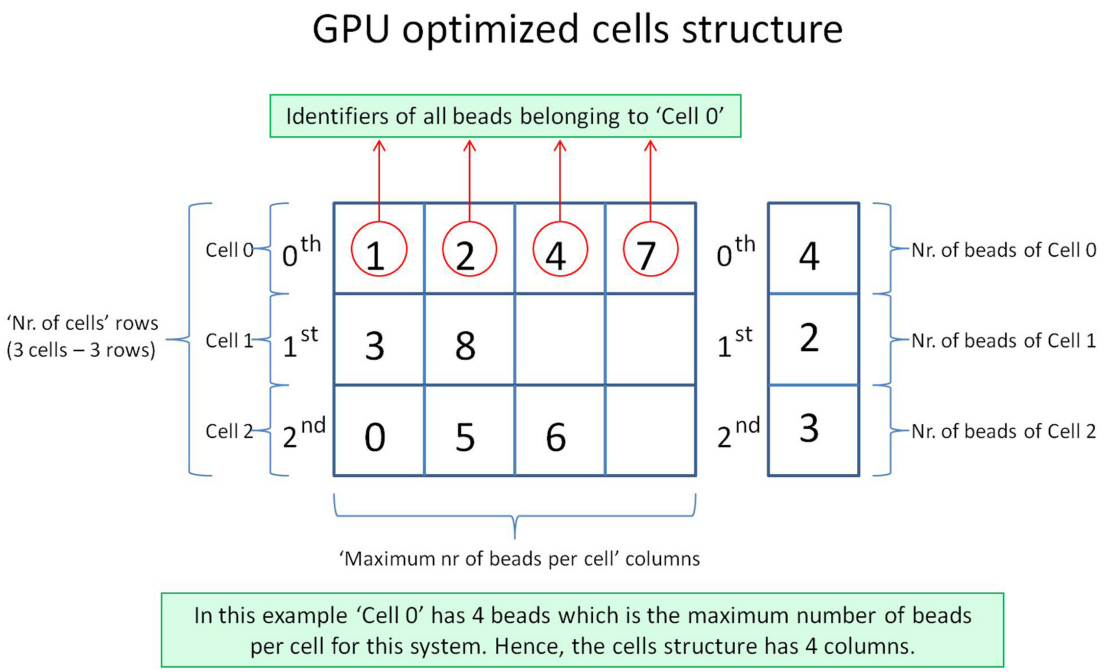


Figure 4: An example of the structure of the cells (on the left) implemented to be used for coalesced accesses from the GPU and the related auxiliary structure (on the right).

These structures do not provide coalesced accesses for the GPU. Consequently, an optimized version is proposed as shown in Figure 6 for the same system in the previous example.

In the neighbor structure (upper side) each column refers to a specific *bead* of the system and contains the identifiers of the neighbors of that *bead*. In the interaction type structure (bottom side) the interaction between the *bead* (having identifier equal to column index) and the neighbor in the corresponding position of the neighbor structure is stored.

We also introduced an auxiliary structure (in the middle of Figure 6) of size *Number of beads* where each element, associated (through its index) to one bead of the system, stores the number of neighbors of that bead.

Additional structures

In addition, to enable *coalesced* accesses by the GPU, we created separate structures for all beads positions, all beads velocities, all beads orientations and all beads types, as these data are frequently accessed.

5.2.2 Kernel Description

Neighbor and interaction type structures are generated using an algorithm similar to the one proposed in Anderson et al. [2008] for the neighbor structure construction.

However, the code reported in Anderson et al. [2008] considers a single bead type and thus does not take into account different interaction types. Moreover, it is specialized for simulations of polymer systems and includes integrators, neighbor lists, LJ potential and bonded forces but not rigid body constraints.

Furthermore, in Anderson et al. [2008], a one-dimensional texture type is used.

Since the texture cache is optimized for $2D$ spatial locality (CUDA [2011]), we have used two dimensional textures in this kernel as well as in the non-bonded forces computation kernel, when the accesses in device memory are not coalesced.

Moreover, as opposed to the description in Anderson et al. [2008], we make use of the cells auxiliary vector described in Paragraph *Cell structure*, to limit the loop iterations to the number of beads actually present in a cell. In this way we also save a check to determine whether a thread is related to any particular bead of the cell considered or not.

5.3 Non-Bonded Forces Computation

The non-bonded forces computation is implemented by the kernel *forces*, that exploits the optimized access to neighbor and interaction structures.

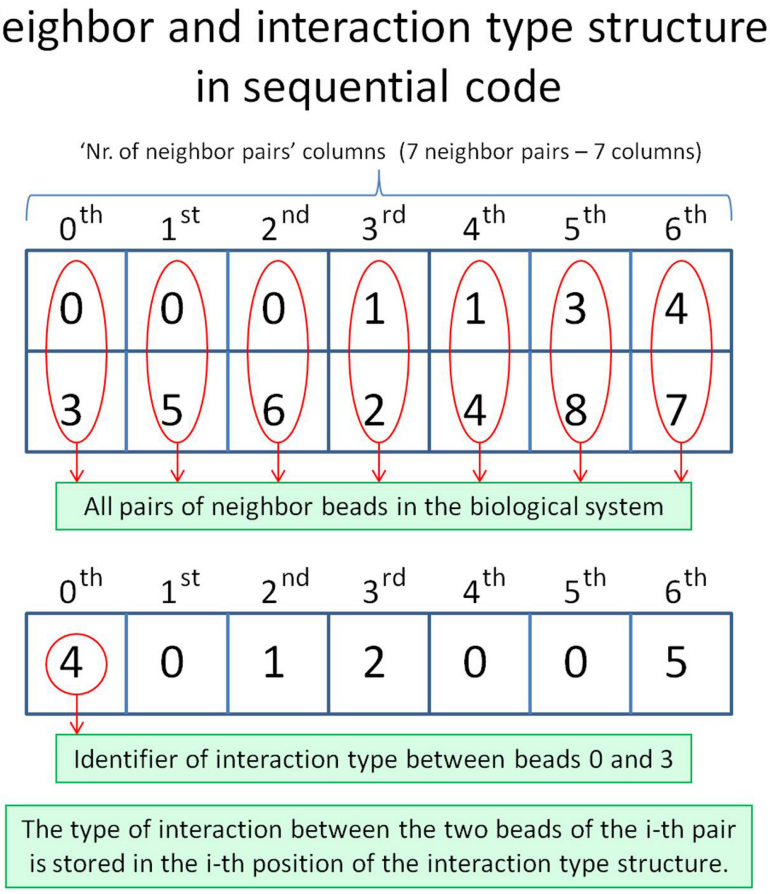


Figure 5: A neighbor structure example (at top) used in the CPU version and the related interaction structure (at bottom).

Compared to what has been done for single bead CG polymer models in Anderson et al. [2008], we use the additional interaction type structure, described in Paragraph *Neighbor and Interaction Structures*. Moreover we use $2D$ textures for random accesses in memory, to retrieve position and orientation coordinates.

The related pseudocode is reported in Algorithm 3.

In this Algorithm, we assign every thread of each block to a certain bead of the system (line 1) as in the integration kernel. Then, a check is performed (line 2) to control whether the current thread can be associated to a bead of the system or not, in order to avoid useless computations. In the former case, in Algorithm 3, a cycling among all neighbors[3] of bead **i** is carried out and for each neighbor **n** actions described in lines 4 to 18 are accomplished. The bead identifier of the $\mathbf{n}^{th}$ neighbor obtained from the neighbor structure is stored in variable **j** (line 4). Then, the position coordinates of bead **j** are fetched from texture memory (line 5). The distance between beads **i** and **j** is computed and stored in variable $\mathbf{r_{i-j}}$ (line 6) and the minimum image convention[4] is applied (line 7).

If the value of $\mathbf{r_{i-j}}$ is lower than a cut-off value (line 8), the *orientation vector* of bead **j** is fetched from texture memory (line 9) and either Algorithm 4 (line 11) or Algorithm 5 (line 13) is performed depending on whether the type of interaction between beads **i** and **j** is a water-water interaction or not (determined in line 10). The contribution of couple **i-j** is added to the related auxiliary variables $force_i$, $torque_i$ and pot_energy_i for the storage of values of force, torque and potential energy of bead **i** (lines 16 to 18).

After finishing the iterations among all neighbors of bead **i**, the state of the system and values of some of its thermodynamic properties, related to force, torque, potential energy, virial and orientation vector of bead **i** are updated (line 20).

The second Leapfrog equation for the motion integration related to bead **i** (line 21) which is very similar to kernel *integration* of Algorithm 2, has been implemented in Algorithm 3 after the computation of non-bonded forces acting on bead **i**. It accomplishes the second half-advance of inertia of bead **i** if bead **i** is considered as a

[3]The number of neighbors for each system bead is stored in the auxiliary neighbor structure, named *neighEl* and described in Paragraph *Neighbor and Interaction Structures*.

[4]The Minimum Image Convention is a form of Periodic Boundary Conditions in which each particle in the simulation interacts with the closest image of the remaining particles (Rapaport [2004]).

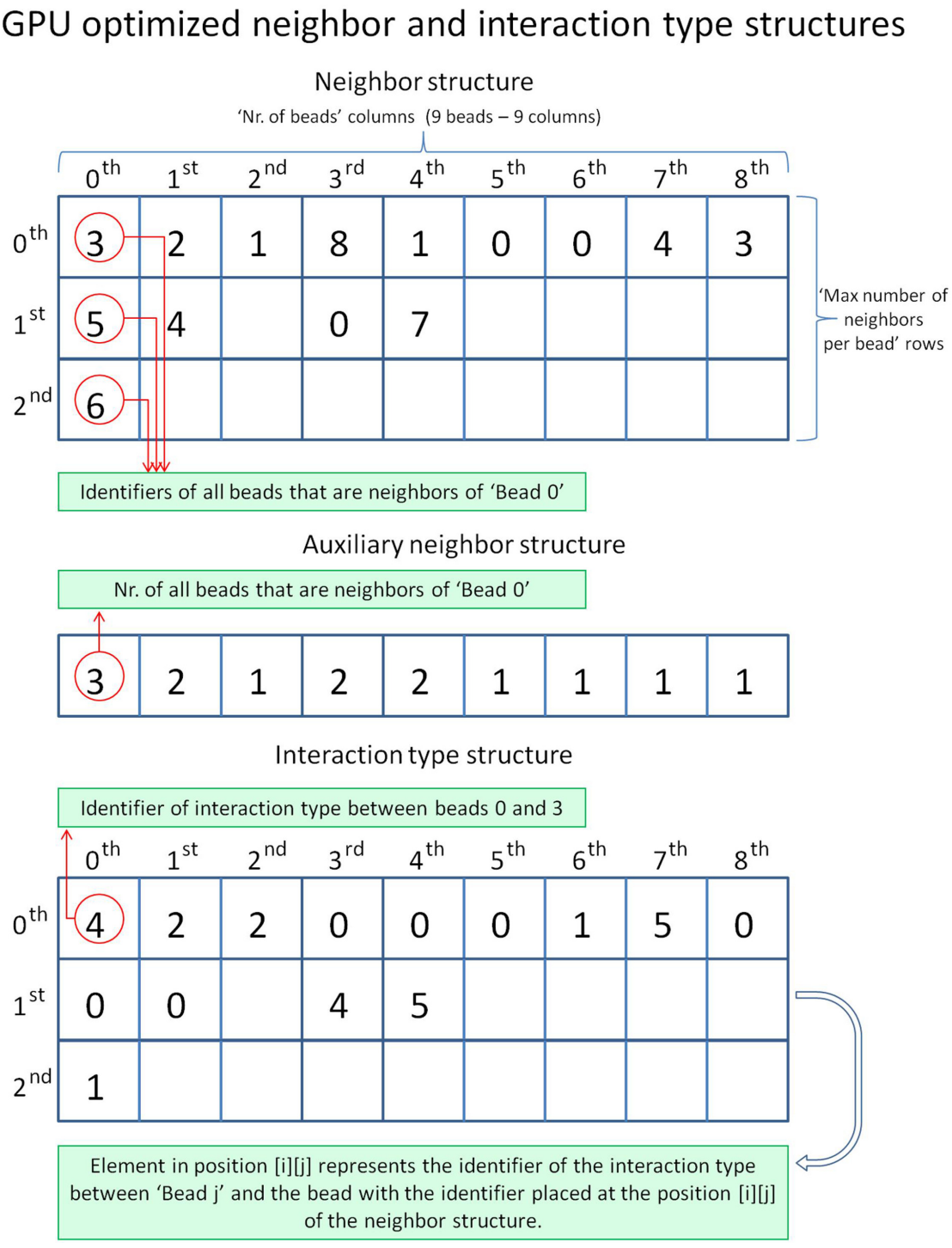


Figure 6: From top to bottom: a neighbor structure example used in the GPU version, the related auxiliary structure for storing the number of neighbors for each bead and the interaction structure which stores information about bead interaction types.

Algorithm 3 Pseudocode of non-bonded forces computation

```
1:  i ← blockIdx.x * blockDim.x + threadIdx.x
2:  if i < number of beads then
3:      for n from 0 to neighEl[i] do
4:          j ← identifier of n^th neighbor of bead i
5:          Texture fetching of position of bead j
6:          r_i−j ← distance_i−j
7:          Minimum image convention
8:          if r_i−j < max_cut − off then
9:              Texture fetching of orientation vector of bead j
10:             if interaction_i−j is a water − water interaction then
11:                 Forces and momenta calculations in Algorithm 4
12:             else
13:                 Forces and momenta calculations in Algorithm 5
14:             end if
15:         end if
16:         force_i ← force_i + force_i−j
17:         torque_i ← torque_i + torque_i−j
18:         pot_energy_i ← pot_energy_i + pot_energy_i-j
19:     end for
20:     Update of system state, virials and orientation vector related to bead i
21:     Second integration step
22: end if
```

rigid body and the second half-advance of the velocity of bead **i**.

In the CPU implementation, the third Newton law is applied by adding the contribution of the interaction $\mathbf{i}-\mathbf{j}$ to the $\mathbf{i}^{th}$ particle and by subtracting it to the $\mathbf{j}^{th}$ one; Instead, in the GPU implementation to avoid additional scattered accesses to memory and the corresponding impact on performance, we do not modify the $\mathbf{j}^{th}$ particle value for virials and torques but we correct that value once, for each of the beads after having considered the contribution of all their neighbors.

Algorithm 4 SSD interactions computation

```
1: if r_i−j < r_cut − off then
2:     Dipole-dipole interaction evaluation                ▷ SSD force field
3:     LJ potential evaluation
4:     Tetrahedral sticky term
5: end if
```

In Algorithm 4, $\mathbf{r_{i-j}}$ is compared to the cut-off distance (line 1), and if it is lower, then the SSD force field contribution (lines 2 to 4) on the state of the system and its thermodynamic properties are computed. The SSD water is a single-site model. The three atoms of individual water molecules are coarse-grained into a single interaction center, which comprises a point dipole (line 2) to account for electrostatics, a LJ core (line 3) providing excluded volume, and a tetrahedral *sticky* term (line 4) to model hydrogen bonding (Orsi et al. [2008]).

Algorithm 5 Computation of interactions different from SSD interactions

```
1:  if r_i−j < r_cut − off then
2:      LJ potential evaluation
3:      Switch(Interaction type)                ▷ Electrostatic interaction evaluation
4:      Case Charge_i − Dipole_j
5:      ...
6:      Case Dipole_i − Charge_j
7:      ...
8:      Case Dipole_i − Dipole_j
9:      ...
10:     Case Charge_i − Charge_j
11:     ...
12: end if
```

In Algorithm 5, if $\mathbf{r_{i-j}}$ is lower than the cut-off value (determined in line 1), the LJ potential (line 2) and electrostatic interaction (lines 3 to 11) are evaluated according to the interaction type dependent on bead **i** and

bead **j** types. We must take into account that we have up to four different execution flows (as showed in **Switch Case** of Algorithm 5) related to the six different types of beads modeled in our application.

6 Simulation Setup

To evaluate the impact of architectural characteristics on the achievable speed-up of the CG simulation, we considered three different NVIDIA GPU architectures, in particular: i) GeForce GTX295; ii) GeForce GTX480; iii) Tesla C2050.

In Table 2 the main characteristics of the employed architectures are reported. GTX295 has the smallest number of cores, while C2050 and GTX480 are more recent and have higher parallelism. The main difference concerns the amount of global memory (double for C2050 with respect to GTX480) and the memory bandwidth (larger for the GTX480).

	GTX295	GTX480	C2050
Number of SMs	30	15	14
Number of Cores	240	480	448
Global Memory (MB)	896	1536	3072
Memory Bandwidth (GB/sec)	112	177.4	144
Shared Memory per SM (kB)	16	48	48
Constant Memory (kB)	64	64	64
Processor Clock Rate (MHz)	1242	1400	1150
Max Registers per SM (k)	16	32	32
Max Threads per Block	512	1024	1024

Table 2: Technical specifications for the three nvidia architectures employed in this work.

The sequential simulation experiments, have been performed on a Processor Intel® Core™i7-920, equipped with 4 cores and 8 threads, a clock rate of 2.67 GHz, a $64-bit$ instruction set, 8 GB of RAM and a 25.6 GB/s maximum memory bandwidth. In this work, we refer to this device as *CPU*. Linux $2.6.26-2-amd64$ operating system runs on this processor.

The results reported in the next section refer to simulations with steps of 1 fs except when analyzing the speed-up depending on the timestep value. Neighbor structure generation takes place approximately each 100 timesteps, when the timestep duration is 1 fs. Therefore the neighbor structure contribution on total simulation time is very low. We consider systems from 840 to 218904 number of beads. As for complexity, we consider systems composed by heterogeneous types of beads. In particular, we consider either water systems uniquely containing water molecules or lipid systems containing lipid molecules in water solution. We developed a GPU version of the simulator customized to water systems, thus treating all the beads as a unique type (water beads), to evaluate the impact of additional structures used for handling interactions among the lipids. From now on we will refer to the version of the simulator which handles only water beads as *water only version* and to that which handles also the beads of lipid molecules as *complete version*. Notice that we can simulate water systems with either of the two versions.

The largest system simulated for the simulations reported in the next section, using: *(i)* The slowest version of the simulator that is the CPU version; *(ii)* Lipid system configuration; *(iii)* Timestep duration of 1 fs; *(iv)* 5000 simulation steps, takes about 200 minutes.

Speed-ups reported refer to kernels with 64 threads per block (except for *cudaneigh* kernel).

7 Results

In this section we present results about the speed-up of the GPU version of BRAHMS with respect to the CPU, considering: *(i)* Biological systems of different dimensions; *(ii)* Biological systems of different complexity (i.e. lipid, water); *(iii)* Different versions of BRAHMS GPU code (i.e. customized or not for water systems simulation); *(iv)* Three different GPU architectures; *(v)* Different timestep values.

Moreover we evaluate the impact of the arithmetic precision used for floating-point (FP) operations.

Moreover we evaluate the impact of the arithmetic precision used for floating-point (FP) operations.

7.1 Impact of System Complexity on Achievable Speed-up

In Figure 7, we report the comparison among the speed-ups (related to the CPU version) of two biological systems, lipid and water only, characterized by different complexity and simulated on the *GTX295* architecture. For water systems, we distinguish between water only and complete version. Single precision FP arithmetic is

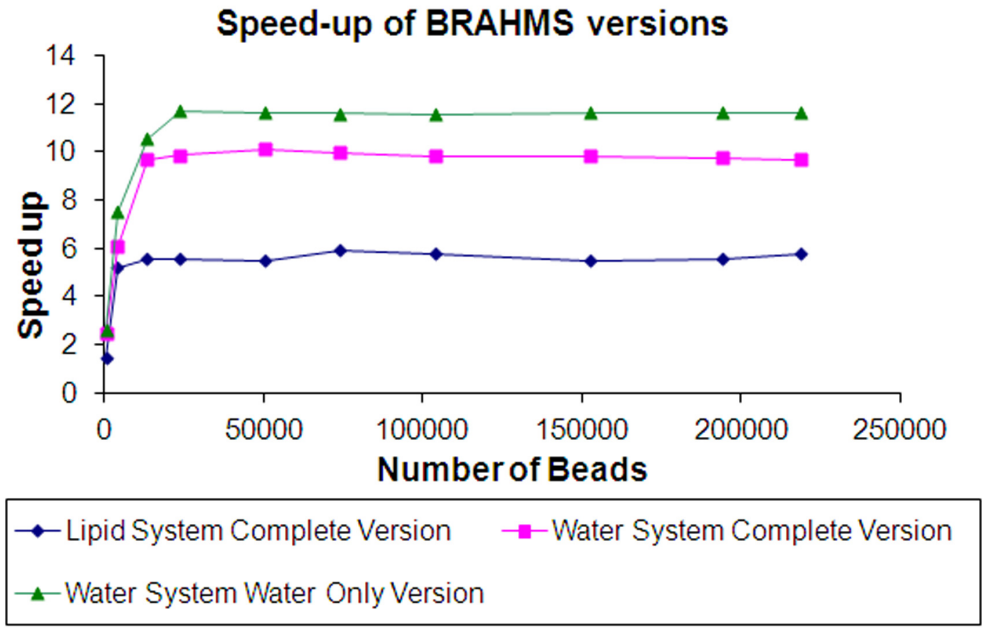


Figure 7: Comparison among the speed-up of different BRAHMS versions with respect to the CPU achieved on the GTX295 architecture.

used for these simulations. For a number of beads lower than $23K$ the speed-up is lower than the maximum achievable because the parallelism of the GPU is not completely exploited. The *water only version* is the fastest version with speed-ups up to *12x* compared to the CPU version. The higher performance of the water only version is mainly due to the absence of structures to handle different bead types. On the other hand, the lower performance of the complete version with lipid systems with respect to water systems is due to the fact that lipid systems involve six different types of beads with respect to the single type of beads present in water systems. For lipid systems, the computations performed during the simulations are dependent from the bead type. This leads to divergent execution flows among the threads, as we will show in the next section, thus leading to a performance hit.

Moreover, Figure 7 shows also the impact of lipid data structures, highlighted by the gap between the water-only and complete version for water systems.

This gap is quantified in Table 3, in terms of minimum, medium and maximum percentage of speed-up lost by the complete version with respect to the water-only version. In this table, this analysis is performed considering also double precision arithmetic.

Arithmetic used	Average	Max	Min
Single precision	28.85%	36.34%	12.34%
Double precision	13.65%	15.71%	8.04%

Table 3: The percentage impact of lipid structures on the overall speed up performance, respectively when single precision arithmetic and double precision arithmetic is considered for FP operations, on GeForce GTX295 architecture.

A smaller gap is observed (15.71% *vs.* 36.41%), when double precision arithmetic is employed. The larger impact of lipid structures in case of single precision arithmetic computation is justified because the gap is mainly due to a constant overhead for memory accesses and additional control statements associated to beads types and interactions. Since single precision arithmetic simulations are shorter, this constant overhead has a larger relative contribution in this case.

7.1.1 Branch Penalty Analysis

We report a quantification of the execution flow divergence of threads in Tables 4 and 5, where a summary of the profiling of each kernel for both water and lipid systems containing 100K beads is reported for the GTX295 architecture.

	Integration	Cudaneigh	Forces
Speed-up vs CPU (times)	11.78	5.31	13.11
Execution time % on CPU	2.61	1.15	94.64
Execution time % on GPU	2.08	2.04	67.67
Average branches	201	18034	32652
Average divergent branches	1	853	2607
Div branches/Tot branches %	0.50	4.73	7.98

Table 4: Profiling of three kernels for a simulation of a $100k$ beads water system.

	Integration	Cudaneigh	Forces
Speed-up vs CPU (times)	10.00	4.38	7.58
Execution time % on CPU	1.98	1.11	93.99
Execution time % on GPU	1.16	1.48	72.54
Average branches	203.00	46025.00	79277.60
Average divergent branches	3.00	2540.00	9204.80
Div branches/Tot branches %	1.48	5.52	11.61

Table 5: Profiling of three kernels for a simulation of a $100k$ beads lipid system.

A higher number of average branches and divergent branches is observed for each kernel of the lipid system compared to the corresponding kernel of the water system. We note that corresponding to the speed-up gap between water and lipid system simulations ($13.109x$ vs. $7.577x$) there is an increasing percentage of divergent branches over the total number of branches (7.98% for water vs. 11.61% for lipids).

In both simulations, the impact of forces kernel on the total simulation time decreases after optimization, remaining the most time consuming section of the code.

Analysis of Biological System Complexity

The reason of a greater speed-up achieved for water system simulations with respect to lipid simulations is illustrated in Figure 8.

A descriptive explanation is provided, illustrating the execution time of the force fields calculations of water and lipid beads. Two simulations of the same number of beads are shown: One having only water beads and the other water and lipid beads. The former is longer than the latter when both are executed in a CPU architecture, due to a more complex[5] force field of the water-water than lipid-lipid and water-lipid interactions. Different thread execution flows due to different force fields for the calculation of the interactions among the beads are verified if at least one of the threads in a warp is related to a lipid bead and the other threads to water beads or viceversa. Divergent execution flows within a warp are serialized and this is the reason that despite the water system simulation is longer than a lipid system simulation in the CPU, the water system simulation is shorter with respect to a lipid system simulation when executed in the GPU. For simplicity, in Figure 8 only one force field (green rectangles) is reported when lipid beads are present in the system simulated, in addition of the water force field. In our case 4 supplementary force fields due to the 5 added lipid bead types are considered.

7.2 Architectural Impact on Speed-up

In Figure 9 we report for lipid systems and water systems, the speed-ups[6] achieved by the three considered architectures for an increasing system size. The double precision arithmetic has been used for FP operations.

For lipid systems we achieve speed-ups up to $7x$, $6.43x$ and $2.43x$ for respectively GTX480, Tesla C2050 and GTX295 architectures, while for water systems the respective speed-ups achieved are $13.69x$, $12.8x$ and $5.28x$.

The GTX295 architecture is the slowest architecture, as we expected, due to its lower number of cores and hence of parallelism. The small performance gap between GTX480 and C2050 can be justified because GTX480 has a higher parallelism with respect to Tesla C2050, having one additional SM, resulting in 32 additional cores. Nevertheless, Tesla C2050 has two DMA engines for bidirectional PCIe communication while GTX480 has only one enabled.

[5]The complexity of non-bonded forces calculations will be illustrated in detail in the next section.

[6]The speed-ups reported concern the GPU implemented part of the application because the servers where the GPU cards are situated have different CPU architectures.

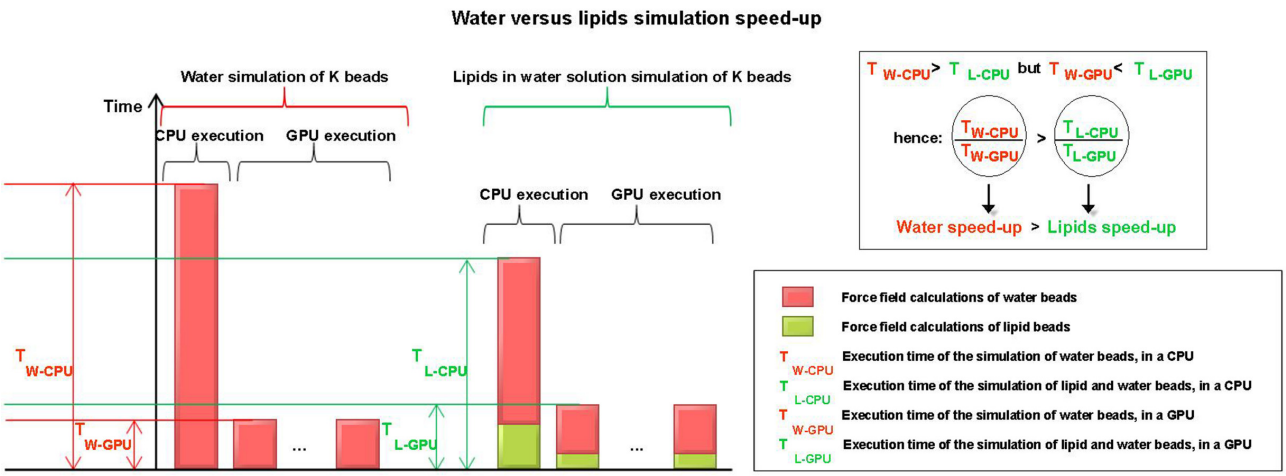


Figure 8: Comparison among CPU and GPU execution times of 2 simulations having the same number of beads K: The system of the first simulation contains only water beads while the system of the second simulation contains water and lipid beads. The repeated blocks in both GPU execution parts represent the parallel running of tasks on the GPU.

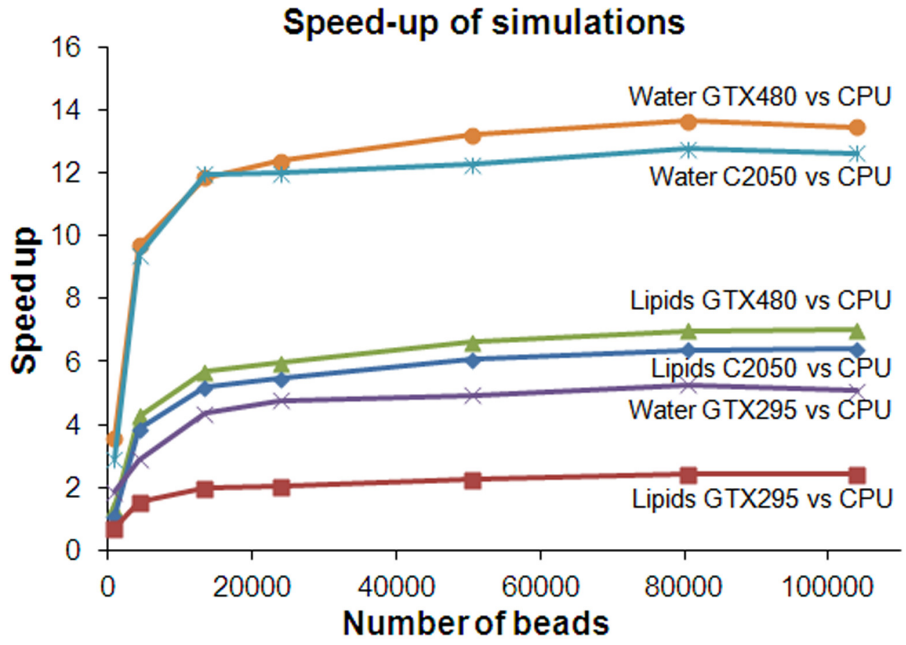


Figure 9: Speed-up of water and lipid in water solution system simulations among different architectures with respect to the CPU.

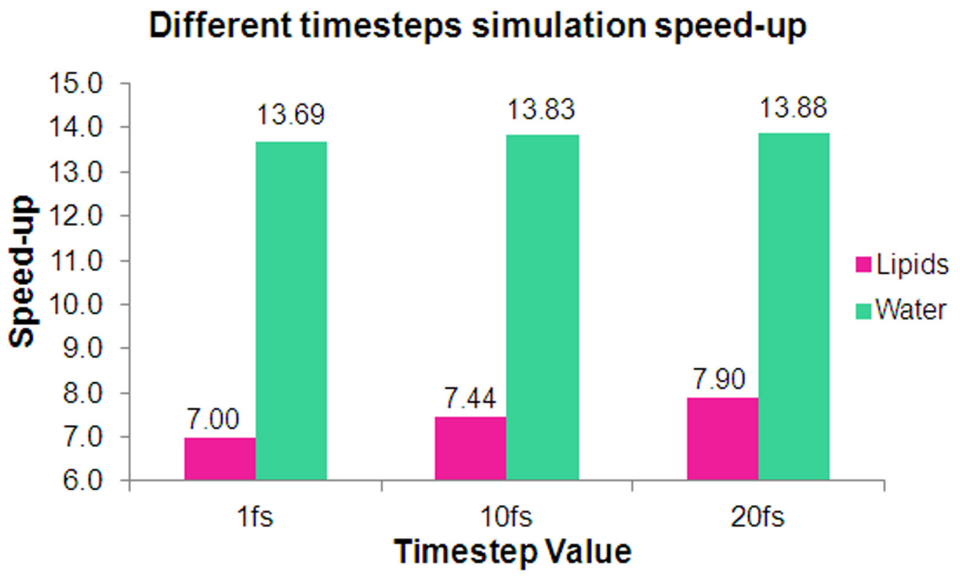


Figure 10: Speed-up of $80k$ bead systems of lipid in water solution and water NVT simulations on GTX480 architecture with respect to the CPU, when different timestep sizes are considered.

7.3 Impact of Timestep Value on Speed-up

In Figure 10 we show the speed-up achieved for: *(i)* $80k$ bead systems of lipids in water solution; *(ii)* $80k$ bead systems of water; *(iii)* three different timestep values: 1 fs, 10 fs and 20 fs. The simulations have been performed with constant volume and temperature ensemble (NVT) and the GTX480 GPU architecture has been used.

The speed-up achieved is 7, 7.44 and 7.9 for lipid systems and 13.69, 13.83 and 13.88 for water systems, for each of the timestep values respectively.

We notice that the greater is the timestep value the greater is the speed-up achieved. Indeed, for the same number of timesteps, the construction of the neighbor structure occurs more often for a simulation with a larger timestep value than for a simulation with a smaller timestep value. The construction of the neighbor structure is verified around each 100, 10 and 5 steps for 1 fs, 10 fs and 20 fs timestep simulations respectively. In fact, in the 20 fs timestep case the probability for a bead to cover the threshold distance and as a consequence trigger the neighbor structure generation, is greater than in the simulations with timestep duration of 1 or 10 fs. In this way, the calls to the neighbor (and related structures) construction algorithm corresponding to *cudaneigh* kernel, increase proportionally with the increasing of the timestep value. Instead, the non-bonded forces calculation and motion integration algorithms, corresponding to the other 2 GPU kernels, undergo the same number of calls, one call for each timestep, regardless of the timestep value. Hence, considering the increased calls to the neighbor construction algorithm, the percentage of the CPU version of the application that is parallelized for the GPU is greater for a larger timestep value, leading to a greater speed-up factor.

Since a water system simulation is longer than a lipid system simulation in the CPU, for the same number of beads (as shown in Figure 8), the relative impact in the simulation time of the higher frequency of neighbor structure construction is smaller in the case of a water system simulation. Indeed, in Figure 10 we can notice a difference of 1.38% between the speed-up obtained for a timestep of 1 fs and a timestep of 20 fs in the case of water systems versus a 12.89% difference for lipid systems.

8 Accuracy of the Simulations

In Table 6, we report mean and standard deviation values of temperature of simulations of lipids as well as of water, for $23k$ bead systems of lipids in water solution, when CPU, GTX295, C2050 and GTX480 GPU architectures are used to perform the simulations. The length of the simulations is 1 ns, performed after system equilibration, with NVT ensemble.

The timestep used is 1 fs. Thus, 1 million steps have been performed for each simulation. For these simulations, we show also the percentage of difference between mean values of temperature of all GPU architectures

Lipids Temperature [°C]				
	CPU	GTX295	GTX480	C2050
Mean value	30.0018	30.0019	30.0023	30.0023
Diff. vs CPU mean value [%]	0.0000	0.0003	0.0017	0.0017
Standard deviation value	0.3119	0.3194	0.3129	0.3129
Water Temperature [°C]				
	CPU	GTX295	GTX480	C2050
Mean value	30.0030	30.0097	30.0000	30.0000
Diff. vs CPU mean value [%]	0.0000	0.0220	0.0100	0.0100
Standard deviation value	0.2921	0.2752	0.2951	0.2951

Table 6: Mean and standard deviation temperature for $23k$ beads of lipid in water solution NVT simulations.

employed for this work and the CPU. The maximum difference of mean temperature values obtained in the GPU architectures with respect to the CPU is observed for the GTX295 architecture with a value of 0.022% in the case of water temperature.

While Table 6 indicates a correct conservation of the temperature during 1 million of 1 fs steps, presenting identical values of CPU, GTX295, C2050 and GTX480 architectures, the situation observed for system potential energy is different. Table 7 reports mean and standard deviation values of potential energy for the same simulations whose temperature values we analyzed before.

	CPU	GTX295	GTX480	C2050
Mean value [kJ/mol]	-33.0969	-30.6396	-33.3228	-33.3228
Diff. vs CPU mean value [%]	0.0000	7.4200	0.6800	0.6800
Standard deviation value	0.1055	1.4017	0.0951	0.0951

Table 7: Mean and standard deviation values of potential energy for $23k$ beads of lipid in water solution NVT simulations.

The GTX295 architecture has a mean value of the system potential energy that differs 7.42% from the mean value obtained in the CPU, with a standard deviation of 1.4017 kJ/mol. The drift of potential energy when the GTX295 architecture is used for the simulation is considerable for more than 40 ps of NVT simulation with steps of 1 fs. Instead, according to Table 7, C2050 and GTX480 architectures report smaller fluctuations of system potential energy than the CPU version.

To further confirm the correctness of the CG simulator implemented for the GPU architectures, we have performed constant volume and energy ensemble (NVE) simulations, to verify the conservation of system total energy throughout the simulations. In Figure 11, we show the values of total and potential system energy for 1 ns NVE simulations of 23K beads of lipids in water solution, with timestep of 1 fs.

The two small rectangular charts inside Figure 11 represent an enlarged portion of each case, for timesteps from 50 to 150 ps and reduced energy values range as shown. They highlight the drift of energy values in the case of GTX295 with respect to energy values in other architectures. We noticed an average difference of 25.28%, 0.72% and 0.65% among system total energy values on respectively GTX295, GTX480 and C2050 architectures versus total system energy value on the CPU.

Therefore, we conclude that for NVE simulations performed in our CG simulator lasting more than 30 ps with a timestep duration of 1 fs, the use of GTX295 architecture is not suitable, while the use of 2.0 and upper compute capability GPU architectures leads to acceptable total energy results with a difference with respect to CPU total energy results of only about 0.7%.

We can also observe in Figure 11, even a more stable trend of the total and potential system energy for simulations on C2050 and GTX480 GPU architectures with respect to the simulation on the CPU. This trend is confirmed from the calculated standard deviation values for total system energy in CPU, C2050 and GTX480 architectures for these simulations that has respectively values: 0.1514, 0.0428 and 0.0464 kJ/mol.

While the three architectures we evaluated support compute capability of 1.3 and more, thus being compliant with the *IEEE 754* standard, we observe a difference in the results of CPU versus GPU, after thousands of simulation steps. This is expected in general as: *(i)* GTX295 architecture is a 1.3 compute capability architecture and it does not support the IEEE-compliant implementation for some particular instructions of single-precision FP numbers, such as division and square root (CUDA [2011]) that are frequent in the code of our simulator. This is the major reason of the great difference reported among the simulations performed on the GTX295 architecture and those performed in the CPU; *(ii)* Small changes, due to rounding errors caused

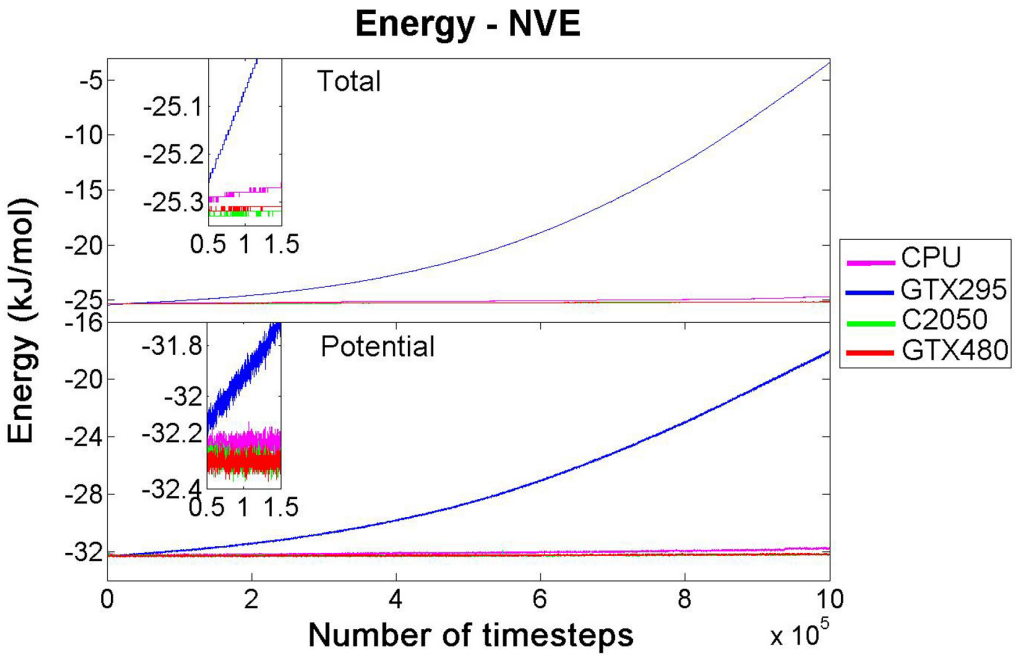


Figure 11: Comparison of total and potential energy values among CPU and GPU architectures for NVE simulations of $23k$ beads of lipids in water solution.

from a different order of arithmetic FP operations in the CPU with respect to GPU architectures[7], lead to accumulated differences throughout the simulations and finally to different results.

Indeed, the MD simulation systems are chaotic, which means that if two theoretically identical simulations are being considered, whatever minimal difference occurs, in the rounding off or instructions order for example, it grows exponentially throughout the simulation. This means that the trajectory of a certain bead in the CPU simulation will inevitably diverge exponentially from the trajectory of the same bead in the simulation on the GPU. Anyway, this is acceptable as long as the ensemble average, as for instance the average temperature or the pressure, is coherent after a reasonably large number of steps.

However, the simulations on the different architectures explored are coherent from an ensemble viewpoint, as observed from the experiments performed. The simulations on the GPU architectures are stable, except for the GTX295 architecture case when the simulations last more than several picoseconds and timesteps of 1 fs are used for them. Their energy, pressure and temperature resulted to have very similar values to energy, pressure and temperature of the simulations performed on the CPU. Mean values related to these properties, both of GPU and CPU simulations, are coherent.

9 Further Optimizations on Forces Kernel

Since *forces* kernel is the most computation demanding part of the CG simulator, we implemented several techniques in order to increase its performance in terms of speed-up achieved.

9.1 Data Reuse and Work Merging

When we split in two different kernels the calculation of non-bonded forces and the accomplishment of the second equation of motion integration, (instead of performing both in a unique kernel: *forces* kernel), a performance deterioration up to 11% is observed.

We explain the improvement of the performance when using a unique kernel to implement both non-bonded forces computation and second integration step, because of the following reasons: *(i)* A unique kernel, in our case enables precomputed data reuse[8], such as beads forces, torques and rotation matrix, avoiding additional

[7] Threads and blocks of threads in the GPUs are scheduled at different times while in the CPU there is a unique sequential execution flow.

[8] We have also noticed in Stone et al. [2007] the reuse of in-register data among GPU optimization techniques used, making in the related MD application case a reuse of atom coordinates and precomputed distance vector components.

data reads/writes from/to global memory; *(ii)* The overhead due to an additional kernel initialization, launch and release for each step of the simulation, is obviated.

9.2 Fixed-point Arithmetic and Intrinsic CUDA Functions

We leveraged on arithmetic operations to increase the simulator performance by means of: *(i)* **Fixed-point arithmetic** implementation for FP operations instead of FP arithmetic; *(ii)* **Intrinsic CUDA functions**, to perform addition, multiplication, division and square root among FP arithmetic operations.

The intrinsic CUDA functions used to implement addition, multiplication, division and square root, are only supported in device code. Intrinsic functions are less accurate but generally faster versions than the standard respective CUDA functions (CUDA [2011]).

In Table 8, we show speed-ups achieved for water and lipid simulations performed on GTX295, C2050 and GTX480 architectures. FP operations of *forces* kernel have been carried out by means of: *(i)* Floating-point arithmetic; *(ii)* Fixed-point arithmetic; *(iii)* Intrinsic CUDA functions: __fadd_rn, __fmul_rn, __fdiv_rn, __fsqrt_rn.

Water Systems			
	Floating	Fixed	Intrinsic
GTX295	5.28	6.13	2.66
GTX480	13.69	19.61	27.71
C2050	12.8	16.42	14.33
Lipid Systems			
	Floating	Fixed	Intrinsic
GTX295	2.43	2.81	0.52
GTX480	7	9.81	12.91
C2050	6.43	8.66	7.27

Table 8: Speed-ups achieved when fixed-point and intrinsic functions are used for floating-point arithmetic operations.

We have a maximum speed-up of $27.71x$ for water simulations and $12.91x$ for lipid simulations, when intrinsic CUDA functions and GTX480 architecture are employed for the simulations. In this case an improvement of 102.41% and 84.43% is achieved compared to FP implementation, for respectively water and lipid simulations. From Table 8 we can observe that in all combinations of architectures and fixed-point arithmetic or intrinsic CUDA functions used, we achieve improvement with respect to the FP CUDA implementation except when we employ GTX295 architecture and intrinsic CUDA functions. This worsening of performance for the latter case is due to the fact that some intrinsic CUDA functions such as __fadd_rn and __fmul_rn compile to tens of instructions for devices of compute capability $1.x$ such as GTX295 architecture. In Papadopoulou et al. [2009], the throughput for single precision FP multiplication is verified to be higher (11.2 *ops/clock*) than the intrinsic CUDA function for the multiplication __fmul_rn (10.4 *ops/clock*) when characterizing GT200 GPUs. Instead, the same intrinsic functions, map to a single native instruction for devices of compute capability 2.0 such as C2050 and GTX480 architectures (CUDA [2011]). Hence, for 2.0 compute capability architectures explored, using intrinsic CUDA functions would lead to higher performance in terms of speed-up as confirmed from the experiments shown in Table 8.

Despite the higher performance for both fixed-point arithmetic and intrinsic CUDA functions, accuracy of the simulations is prohibitive.

We observed the mean values of total system energy calculated over the first 10K steps of NVE simulations with 1 fs timestep duration, for 23K bead systems of lipids in water solution in the case of: *(i)* CPU, GTX295, GTX480, C2050 architectures; *(ii)* Floating-point arithmetic, fixed-point arithmetic and intrinsic CUDA functions. We noticed that mean values of total system energy for the different combinations of architectures and optimizations performed are different in the case of fixed-point arithmetic and intrinsic CUDA functions, with respect to the floating-point CPU version. A maximum divergence with respect to floating-point CPU results of 112.4% and a minimum divergence of 104% are verified when fixed-point arithmetic and intrinsic CUDA functions are used. We also noticed a 31.3% difference of mean values of total system energy between FP arithmetic and fixed-point arithmetic in CPU simulations. After only few timesteps this discrepancy would become even greater and lead to the explosion of the simulation.

We can finally deduct that the very high contrast in the results is due to: *(i)* The accuracy lack of fixed-point arithmetic; *(ii)* The accuracy lack of the intrinsic CUDA functions; *(iii)* The different order of execution of the operations in GPU architectures with respect to the CPU.

10 Discussion

A discussion is needed to evaluate speed-up results obtained in this paper. According to other papers related to acceleration of CG models (Anandakrishnan et al. [2010]), speed-up is in general much smaller than for AL models. In this paper, we give a detailed explanation of specific characteristics of the simulation model we target, devising general considerations about the impact of modeling techniques on the achievable speed-up. These aspects can be summarized in the following contributions: i) A relevant role is played by the force field for pair potentials, that includes particular representation for water and charges, which depends on the type of interaction. This leads to frequent divergent branches that impact the overall speed-up[9]; ii) A second aspect is related to the complexity of the force fields considered, which requires a large amount of local physical resources (i.e. registers), that have to be distributed among all threads of a CUDA block thus limiting the total number of threads per CUDA block, finally impacting the performance; iii) The adoption of transcendental functions to perform the square root operations. For these kinds of functions the normal 8 and 32 FP units for respectively the $1.x$ (such as GTX295) and $2.x$ (such as GTX480 and C2050) compute capability architectures are not suitable. Instead, 2 and 4 special function units for single-precision FP transcendental functions are available for respectively the 1.x and 2.x compute capability architectures (CUDA [2011]). Hence, when these functions have to be performed the parallelism offered by CUDA architectures is not exploited entirely; iv) Finally, the pair interaction potentials computation causes a relatively large amount of scattered memory accesses, causing a considerable texture cache miss rate. That is because neighbors of a bead are not spatially clustered. This issue could be addressed through additional optimizations obtained by adapting techniques developed for simple polymer simulations (Anderson et al. [2008]) that cannot be applied as is for CG models characterized from rigid body constraints and not homogeneous beads types and interactions. These optimizations will be the object of future work; v) The CPU used for comparison represents a single core but *high performance architecture*.

In the rest of this section, we provide a more detailed analysis of the most intensive and critical part of code, the *forces* kernel, highlighting the conditions controlling the execution flow, that are necessary to account for the forces computation among heterogeneous bead types and solvent.

10.1 Non-bonded Forces Execution Flows

We report the flowchart of the simulator code, for non-bonded forces calculations in Figure 12. This part corresponds to lines from 8 to 18 of Algorithm 3. In this code, conditions related to the electrostatic interactions and soft sticky dipole potential are highlighted.

In the flowchart, condition A (corresponding to line 8 of Algorithm 3) checks whether the neighbor $pair_{\mathbf{i\text{-}j}}$ has a distance within the cut-off radius in order for its non-bonded potential contribution to be considered for further non-bonded forces calculations. Then, condition B that refers to line 10 of Algorithm 3 follows and checks whether the $pair_{\mathbf{i\text{-}j}}$ interaction type is an interaction among water beads. The right part of the flowchart, representing the execution flow of Algorithm 4, including conditions from C to H refers to potential calculation of $pair_{\mathbf{i\text{-}j}}$ in the case both $bead_{\mathbf{i}}$ and $bead_{\mathbf{j}}$ are water beads. If at least one of them is not a water bead, the non-bonded forces potential will be calculated according to the left part of the flowchart including conditions from K to Q which show the execution flow of Algorithm 5.

In the flowchart, percentages for two different simulated systems are associated to each condition except for condition A, whose *"yes" branch* is taken as reference for the following statistics.

Each percentage reported indicates the ratio between the times the *"yes" branch* is taken, with respect to the times the *"yes" branch* of condition A is satisfied. Only for conditions B and Q the percentage of *"no" branches* is reported as well.

The two percentages reported for each condition refer to two different simulated systems. The right (and green) percentage refers to a $23K$ beads simulation of lipids in water solution while the left (and red) percentage refers to a $23K$ beads simulation of water only.

It can be observed that from condition A the execution flow may reach the end point of the flowchart in 23 different ways that lead to divergent execution flows among threads of the same warp. The percentage range goes from 100% to 3.36% for water simulations and from 76.53% to 0.27% for lipid simulations.

The reported statistics show that the execution flow is very heterogeneous, with the various branches taken depending on the system configuration. This results in a significant probability of having a fraction of the 32 threads of a warp being idle waiting for others to complete. In particular, since the *"yes" branches* are on the longest path and they are associated to low percentages, this implies that most of the threads wait for the completion of a small number of threads.

[9] We report detailed information on different execution flows related to bead types and water and charge representations in the following subsection

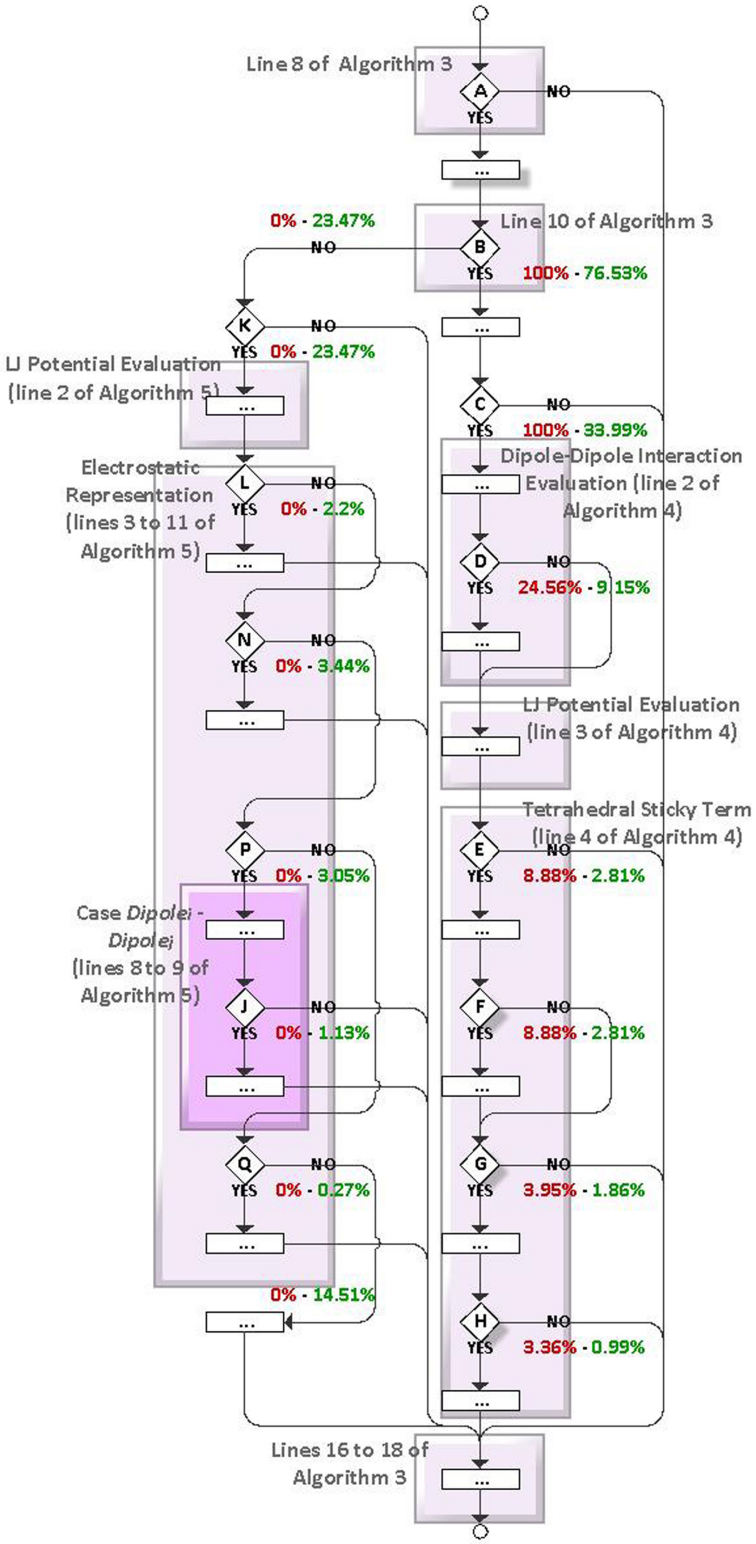


Figure 12: Execution flow of non-bonded forces computation for each $pair_{\mathbf{i\text{-}j}}$ of total neighbor bead pairs. Each percentage reported indicates the ratio between the times the related condition *"yes" branch* or *"no" branch* is verified, with respect to the times the *"yes" branch* of condition A is satisfied. Percentages in red and on the left of each percentage couple refer to a $23k$ beads water simulation while green ones on the right refer to a $23k$ beads lipids simulation.

10.2 Size of Simulated Systems

A final consideration regards the size of simulated systems. The required device memory, allocated on the GPU at the beginning of each simulation, increases linearly with the beads number up to the maximum system

memory that is limited by 896 MB, 1536 MB and 3072 MB device memory capability for respectively GTX295, GTX480 and C2050 architectures. In detail, we can simulate up to $475K$, $814K$ and $1630K$ beads water systems and $211K$, $363K$ and $726K$ beads lipid systems on respectively GTX295, GTX480 and C2050 architectures. Because of the smaller memory occupation of water bead structures, we can simulate larger water systems than lipid systems.

11 Related Work

Several MD tools have been proposed in the literature. In particular, some of them have been specifically optimized for GPU acceleration. ACEMD is a biomolecular dynamics software package, designed specifically for a single workstation with multiple GPUs (Harvey et al. [2009]). HOOMD is another MD simulations software specifically implemented for running on GPUs (Anderson et al. [2008]). It is specialized in simulations of polymer systems. NAMD has been the first MD simulation package to include GPU acceleration (Stone et al. [2007]).

In general, these implementations outperform traditional CPU cores by factors ranging from 10 to 20 and 100 in some ideal cases (Stone et al. [2010]). However, while large speed-ups have been obtained for AL models (Bauer et al. [2011], Rapaport [2011], Ganesan et al. [2011], Friedrichs et al. [2009]), CG models, very recently explored in literature, show much lower speed-up values. For these, a complete characterization like the one performed in this work was missing. Furthermore, most of the CG models use only one CG bead type (Sunarso et al. [2010], Anderson et al. [2008], Nguyen et al. [2011]), or are limited to the acceleration of specific and not complex parts of the model (van Meel et al. [2008], Colberg and Hfling [2011], Zhmurov et al. [2010], Anandakrishnan et al. [2010], Liu et al. [2008]). In the present work it has been shown that the overhead introduced by the conditionals and the data structures needed to handle different CG bead types severely impacts the achievable speed-up.

In Table 9 we outline relevant work about CG model optimization which is the focus of this work.

The electrostatic potential computation has been parallelized on an ATI Radeon 4870 GPU (800 cores), in Anandakrishnan et al. [2010]. This potential computation implemented by means of an analytical linearized Poisson-Boltzmann method, is reported to be 182 times faster for the atomistic simulation of a virus capsid structure with respect to the same conditions of simulation in an Intel Core 2 Duo E6550 processor and 22 times faster when a CG technique is used to partition the charges, for the same architectures. We can observe that the speed-up factor obtained for the CG simulation is several times lower than the speed-up factor achieved for the AL simulation of the same system using the same conditions.

An acceleration of the time integration of particles using a simple LJ potential and a cut-off distance is reported in Liu et al. [2008], showing speed-up factors of about 7 to 11 over optimized routines from LAMMPS (Plimpton [1995]).

In Zhmurov et al. [2010] an 85−fold speed-up is achieved implementing Langevin simulations of proteins through a CG self-organized polymer model and using the GeForce GTX295 GPU. Langevin dynamics characterizes the viscous aspect of a solvent without fully modeling an implicit solvent. This model does not consider the electrostatic screening and the hydrophobic effect that are carefully characterized instead in our model (Orsi et al. [2008]).

A GPU-based MD simulation has been developed in Sunarso et al. [2010] for the study of flows of fluids with anisotropic molecules such as liquid crystals. A 50 fold speed-up is achieved on a GTX200 series GPU with respect to an Intel Core i7 940 2.93 GHz processor, for a CG simulation. In addition to the difference between the macroscopic flow under application of electric field simulations and lipid bilayer simulations, two main differences in Sunarso et al. [2010] are present with respect to our work: *(i)* In the CG application in Sunarso et al. [2010] only one CG mechanical bead type is present: ellipsoid molecules that are treated as symmetric rigid bodies. Hence, the interaction potential is simplified as well. BRAHMS includes CG beads treated as symmetric rigid bodies, non-symmetric rigid bodies and point-masses beads. *(ii)* The cell list in Sunarso et al. [2010] is updated at every time step of the simulation. Intermolecular forces and torques are calculated using the cell-list algorithm (used mainly to populate neighbor structures) and taking advantage of the GPU shared memory per block suitable for this algorithm. In our work the neighbor list update frequency depends on the timestep value. For the simulations presented with timestep of 1 fs, it is updated approximately every 100 timesteps. Hence, the non-bonded forces calculations are separated from the neighbor structure generation algorithm where the very fast shared memory exploitation is appropriate.

Velocity Verlet integrator, cell-list algorithm and only the simple LJ potential are used in van Meel et al. [2008] to implement MD of CG simulations. Single precision floating point operations are used on an NVIDIA GeForce 8800 GTX GPU, achieving speed-ups from 2 to 40 with respect to a 3.2 GHz Intel Xeon processor, depending on the density of particles.

In Colberg and Hfling [2011], a soft-sphere MD application has been implemented for the GPU, to perform simulations of LJ fluids. The velocity Verlet integrator has been used and only the LJ potential has been considered in the application. Moreover, compared to our work, only one type of CG particle is present. Speed-

Work ref.	Main similarities	Main Differences	Speed-up vs. CPU
Anandakrishnan et al. [2010]		- AL level - Acceleration of electrostatic potential computation only - Virus capsid simulation - Intel Core 2 Duo E6550 CPU - ATI Radeon 4870 GPU - 800 GPU cores	182
	- CG level	- Acceleration of electrostatic potential computation only - Virus capsid simulation - Intel Core 2 Duo E6550 CPU - ATI Radeon 4870 GPU - 800 GPU cores	22
Liu et al. [2008]	- CG level - Cut-off applied	- Time integration acceleration - No electrostatics handling	7 - 11
Anderson et al. [2008]	- CG level	- 1 CG bead type - Polymer systems - Xeon 80546K 3 GHz CPU - 1 GB RAM CPU - GeForce 8800 GTX GPU - 128 GPU cores	30
Zhmurov et al. [2010]	- CG level - GeForce GTX295 GPU - 240 GPU cores	- Langevin Dynamics - Protein simulations - 1 CG bead type - Xeon E5440 CPU - Single FP precision GPU	85
Sunarso et al. [2010]	- CG level - Anisotropic molecules considered - Cut-off applied - 240 GPU cores	- Flow under application of electric field simulations - 1 CG bead type - Cell structure built each step - Intel Core i7 940 2.93 GHz CPU - GeForce GTX280 GPU	50
van Meel et al. [2008]	- CG level - Molecular Dynamics - Cut-off applied	- No electrostatics handling - Intel Xeon 3.2 GHz CPU - GeForce 8800 GTX GPU - 128 GPU cores - Single FP precision GPU	2 - 40
Nguyen et al. [2011]	- CG level - Rigid body constraints - Complex beads shapes - GeForce GTX480 GPU - 480 GPU cores	- 1 CG bead type - No electrostatics handling - 5 to 90 particles per rigid body	2.5 - 40
Colberg and Hfling [2011]	- CG level - Molecular Dynamics - 240 GPU cores	- LJ fluids simulations - No electrostatics handling - 1 CG bead type - AMD Opteron 2216 HE 2.4 GHz CPU - GeForce GTX280 GPU - Single FP precision GPU	4 - 80

Table 9: Comparison of state-of-the-art related works with our work reporting main similarities and differences from our work.

ups of 4 to 80 folds are achieved in an NVIDIA GeForce GTX 280 GPU with respect to a 2.4 GHz AMD Opteron 2216 HE processor, for the application, using single FP precision.

12 Conclusions

In this work we presented an optimized, accelerated version of a coarse grain molecular dynamics simulator on GPU architectures. We described the optimization in terms of computation and data structures, specifically targeted to CG models, taking into account bead type heterogeneity.

We compared molecular systems of different complexity, composed of water and lipids, observing that lipid systems achieve a speed-up almost 2 times slower than water systems.

We performed an evaluation of the impact of CG features on the speed-up, and explored how these features affect the achievable acceleration on three different GPU architectures and in case of single and double precision arithmetic. We obtained a maximum speed-up of 13.88 for water systems and 7.9 for lipids on the GTX480 architecture for 20 fs timestep simulations. Moreover, we analyzed the dependance of the achievable speed-up from the timestep value, obtaining slightly greater speed-up values for simulations with larger timestep

duration. We performed a detailed analysis of CG features and their impact on the achievable acceleration, to devise guidelines for writing more efficient CG MD codes.

Finally, we demonstrated the correctness of the accuracy of the simulations in the GPU architectures and discovered a more stable trend of total and potential energy values for simulations on C2050 and GTX480 architectures with respect to the simulations in CPU.

References

R. Anandakrishnan, T. R. W. Scoglandb, A. T. Fenleyc, and et al. Accelerating electrostatic surface potential calculation with multiscale approximation on graphics processing units. *J. Mol. Graphics Modell.*, 28(8): 904–910, 2010.

J. A. Anderson, C. D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *J. Comput. Phys.*, 227:5342—5359, 2008.

B. A. Bauer, J. E. Davis, M. Taufer, and S. Patel. Molecular dynamics simulations of aqueous ions at the liquid vapor interface accelerated using graphics processors. *J. Comput. Chem.*, 32(3):375–385, 2011.

P. H. Colberg and F. Höfling. Highly accelerated simulations of glassy dynamics using gpus: Caveats on limited floating-point precision. *Comput. Phys. Commun.*, 182(5):1120—-1129, 2011.

CUDA. *NVIDIA CUDA C Programming Guide*, 2011. URL http://developer.download.nvidia.com/compute/cuda/$4_0_$r

Q. Dong and S. Zhou. Novel nonlinear knowledge-based mean force potentials based on machine learning. *IEEE/ACM Trans. Comput. Biol. Bioinf.*, 8(2):476–486, 2011.

M. S. Friedrichs, P. Eastman, V. Vaidyanathan, and et al. Accelerating molecular dynamic simulation on graphics processing units. *J. Comput. Chem.*, 30(6):864—-872, 2009.

N. Ganesan, B. A. Bauer, T. R. Lucas, and et al. Structural, dynamic, and electrostatic properties of fully hydrated dmpc bilayers from molecular dynamics simulations accelerated with graphical processing units. *J. Comput. Chem.*, 32(14):2958—2973, 2011.

J. G. Gay and B. J. Berne. Modification of the overlap potential to mimic a linear site-site potential. *J. Chem. Phys.*, 74(6):3316–3319, 1981.

M. J. Harvey, G. Giupponi, and G. De Fabritiis. Acemd: accelerating biomolecular dynamics in the microsecond time scale. *J. Chem. Theory Comput.*, 5(6):1632—1639, 2009.

C. J. Högberg and A. P. Lyubartsev. A molecular dynamics investigation of the influence of hydration and temperature on structural and dynamical properties of a dimyristoylphosphatidylcholine bilayer. *J. Phys. Chem. B*, 110(29):14326–14336, 2006.

W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig. Accelerating molecular dynamics simulations using graphics processing units with cuda. *Comput. Phys. Commun.*, 179(9):634–641, 2008.

Y. Liu and T. Ichiye. Soft sticky dipole potential for liquid water:? a new model. *J. Phys. Chem.*, 100(7): 2723–2730, 1996.

J. L. MacCallum and D. P. Tieleman. Computer simulation of the distribution of hexane in a lipid bilayer:spatially resolved free energy, entropy and enthalpy profiles. *J. Am. Chem. Soc.*, 128(1):125—-130, 2006.

A. D. McNaught and A. Wilkinson. Blackwell Science, International Union of Pure and Applied Chemistry, 2nd edition, 1997.

M. Müller, K. Katsov, and Michael Schick. Biological and synthetic membranes: What can be learned from a coarse-grained description? *Phys. Rep.*, 434:113—-176, 2006.

T. D. Nguyen, C. L. Phillips, J. A. Anderson, and S. C. Glotzer. Rigid body constraints realized in massively-parallel molecular dynamics on graphics processing units. *Comput. Phys. Commun.*, 182(11):2307—2313, 2011.

M. Orsi, D. Y. Haubertin, W. E. Sanderson, and J. W. Essex. A quantitative coarse-grain model for lipid bilayers. *J. Phys. Chem. B*, 112(3):802–815, 2008.

M. Orsi, J. Michel, and J. W. Essex. Coarse-grain modelling of dmpc and dopc lipid bilayers. *J. Phys.: Condens. Matter*, 22(15):155106, 2010.

M. Orsi, N. G. Noro, and J. W. Essex. Dual-resolution molecular dynamics simulation of antimicrobials in biomembranes. *J. R. Soc., Interface*, 8(59):826–841, 2011.

M. Orsi, W. E. Sanderson, and J. W. Essex. Permeability of small molecules through a lipid bilayer: a multiscale simulation study. *J. Phys. Chem. B*, 113(35):12019–12029, 2009.

M. Papadopoulou, M. Sadooghi-Alvandi, and H. Wong. Micro-benchmarking the gt200 gpu. Technical report, Technical report, Computer Group, ECE, University of Toronto, 2009. URL http://www.eecg.toronto.edu/$\sim$moshovos/CUDA08/arx/microbenchmark_report.pdf.

J.C. Phillips, R. Braun, W Wang, and et al. Scalable molecular dynamics with namd. *J. Comput. Chem.*, 26 (16):1781–1802, 2005.

S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *J. Comput. Phys.*, 117(1):1—-19, 1995.

D. C. Rapaport. Cambridge University Press, New York, 2nd edition, 2004.

D. C. Rapaport. Enhanced molecular dynamics performance with a programmable graphics processor. *Comput. Phys. Commun.*, 182:926—-934, 2011.

N. Schmid, M. Bötschi, and W. F. van Gunsteren. A gpu solvent–solvent interaction calculation accelerator for biomolecular simulations using the gromos software. *J. Comput. Chem.*, 31(8):1636–1643, 2010.

A. Shkurti, A. Acquaviva, E. Ficarra, and et al. Gpu acceleration of simulation tool for lipid-bilayers. In *IEEE Int. Conf. Bioinf. Biomed. Workshops*, pages 849–850, 2010.

A. Shkurti, A. Acquaviva, E. Ficarra, and et al. Characterization of coarse grain molecular dynamic simulation performance on graphic processing unit architectures. In *Proceedings of the International Conference on Bioinformatics Models, Methods and Algorithms*, pages 339–342, 2012.

J. E. Stone, D. J. Hardy, B. Isralewitz, and K. Schulten. *GPU algorithms for molecular modeling*, chapter 16, pages 351–371. Chapman & Hall/CRC Press, 2011.

J. E. Stone, D. J. Hardy, I. S. Ufimtsev, and K. Schulten. Gpu-accelerated molecular modeling coming of age. *J. Mol. Graphics Modell.*, 29:116—-125, 2010.

J. E. Stone, J. C. Phillips, and P. L. Freddolino. Accelerating molecular modeling applications with graphics processors. *J. Comput. Chem.*, 28:2618—-2640, 2007.

A. Sunarso, T. Tsuji, and S. Chono. Gpu-accelerated molecular dynamics simulation for study of liquid crystalline flows. *J. Comput. Phys.*, 229(15):5486—-5497, 2010.

J. A. van Meel, A. Arnold, D. Frenkel, and et al. Harvesting graphics power for md simulations. *Mol. Simul.*, 34(3):259–266, 2008.

J. Wohlert and O. Edholm. Dynamics in atomistic simulations of phospholipid membranes: Nuclear magnetic resonance relaxation rates and lateral diffusion. *J. Chem. Phys.*, 125(20):204703, 2006.

A. Zhmurov, R. I. Dima, Y. Kholodov, and V. Barsegov. Sop-gpu: Accelerating biomolecular simulations in the centisecond timescale using graphics processors. *Proteins: Struct., Funct., Bioinf.*, 78(14):2984–2999, 2010.