Acceleration of Coarse Grain Molecular Dynamics on GPU Architectures

Ardita Shkurti,*[a] Mario Orsi,[b] Enrico Macii,[a] Elisa Ficarra,[a]† and Andrea Acquaviva[a]†

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 (GPUs) 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 studies. © 2012 Wiley Periodicals, Inc.

DOI: 10.1002/jcc.23183

Introduction

Many aspects of cell activity may be perceived observing the dynamics of cell membranes that are composed of patterns of lipid bilayers. Such aspects include, among others, the conformation of embedded proteins that are directly influenced by the bilayer structure, the biomembrane permeability, interaction with drugs, and signaling.1

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

Molecular dynamics (MD) is a simulation technique 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 models2–4 require considerable computational resources, due to the necessity of interaction calculations between each atom and all other atoms in the system. Moreover, the reliability of the calculation of some essential features, such as diffusion coefficients and 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 allow the simulation of large size and timescales that can be inaccessible by AL models, especially for large macromolecules such as lipids.5,6 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. Typically, a lipid represented by about 100 particles in AL approaches is modeled by ~10 beads in CG representations.7,8

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. Although 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 simulations need to be developed. Graphic processing units (GPUs) have emerged as a powerful architecture for MD acceleration9–12 for both AL and CG models.10

Due to their features, CG models impose specific challenges that must be carefully addressed. In this article, 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 article providing a comprehensive characterization and evaluation of CG acceleration on GPUs.

In this work, we first provide background about the programming environment used and the simulated model. Thereafter, the main optimization methods are characterized,
followed by hardware specifications and simulation setup. Then, we show the results achieved and analyze the accuracy of the simulations. Afterward, further detailed optimizations performed are described. A comparison with related works is reported immediately before concluding the work.

**Compute Unified Device Architecture 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.[24]

The GPU is divided into a set of streaming multiprocessors (SM) in which hundreds of threads reside concurrently.

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 these memory spaces, but 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 turn has a unique identifier among the program blocks.[24]

As a consequence, problems adapted to GPUs are divided into independent subproblems to be scheduled in different thread blocks. Subproblems 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, as 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.[24] Anyway, the number of threads and consequently warps residing concurrently in an SM is limited, because they must share the limited resources available for that SM such as registers and shared memory.

**CG Simulation**

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

**Algorithm 1** Main steps of MD simulation

1: Input particles initial positions \( r(t=0) \) and velocities \( v(t=0) \); set timestep \( \Delta t \)
2: for timestep := 0 to Total number of timesteps do
3: \( F(i,t) \rightarrow -\frac{\partial V(r,t)}{\partial r_i}; \quad a_{i,t} \rightarrow F(i,t)/m_i \); \( a_{i,t} \rightarrow -\frac{\partial V(r,t)}{\partial t}; \)
4: \( r(t+\Delta t) \rightarrow r(t) + v(t) + \Delta t + 1/2 + a(t) \times \Delta t^2 \quad \cdots \)
5: Move time forward: \( t \leftarrow t + \Delta t \)
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.

At last, time is moved forward according to the timestep size 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, whereas in AL models a particle represents a single atom, in CG models a particle represents a bead, which is a cluster of atoms.

In this work, we perform an optimization and acceleration for CUDA environment of a CG simulator called BRAHMS.[1] BRAHMS can simulate lipid bilayers,[7,25–27] as well as more complex membrane systems,[28–30] BRAHMS was originally implemented in the C language as a serial code. The main MD algorithms were developed following a standard approach.[31] In particular, the Leapfrog integration algorithm is used to move the positions of beads and their velocities one timestep forward.[31] BRAHMS uses the neighbor list method together with a cell subdivision to optimize the calculation of interactions, following Rapaport in Ref. [31].

At every step, after the forward time motion of the system obtained through the Leapfrog integration algorithm (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, each time a bead is displaced above a

---

certain distance threshold, an update of the neighbor structure is necessary.

Then, the bonded force interactions among system particles are calculated, which are followed by the computation of nonbonded force interactions among system beads. In general, this is the most computationally intensive part of BRAHMS and MD codes.[9–11,14–21,23,31–35] After each simulation step, system properties such as temperature, energy, and pressure are updated. Pressure and temperature are controlled using the weak-coupling scheme described in Ref. [36].

BRAHMS includes nonspherical CG beads treated as symmetric rigid bodies such as Gay–Berne[37] (GB) ellipsoid molecules, nonsymmetric rigid bodies such as water molecules modeled with the soft sticky dipole[38] (SSD) potential and simple point-mass beads modeled with standard isotropic potentials (such as Lennard-Jones (LJ) and Coulomb).[37] The interaction potentials among all these bead types are taken into account while implementing the GPU version of BRAHMS. In addition to translational motion, rotational motion should be considered for rigid body beads as well, at every step, during forward time motion. To integrate the rotational motion, a rigid body integration was implemented.[39] 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 to account for electrostatics, a LJ core providing excluded volume, and a tetrahedral sticky term to model hydrogen bonding.[38] These three parts of the SSD model are generalized in lines 2–4 of Algorithm 4.

**GPU Optimization of CG 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. Note that besides nonbonded 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.

<p>| 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. |</p>
<table>
<thead>
<tr>
<th>Part of application</th>
<th>Related percentage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonbonded forces computation</td>
<td>94.3</td>
</tr>
<tr>
<td>Integration of equations of motion</td>
<td>2.3</td>
</tr>
<tr>
<td>Neighbor structure generation</td>
<td>1.1</td>
</tr>
</tbody>
</table>

According to the bottlenecks identified, we have implemented four kernels to be executed on the device (GPU): one kernel for the first part of the 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 nonbonded forces and accomplishment of the second part of the integration step.

In the rest of this section, we give computational details on the first, second, and last CUDA kernels implemented. Full details can be found in the GPU code itself, which is available at https://code.google.com/p/brahms-md/downloads/.

**Kernel implementing the first part of the integration**

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

We use a two-part integration algorithm.[31] The first part of the integration is performed at the beginning of each single step of the simulation, whereas the second part of the integration (similar to the first one) is performed at the end, after neighbor structure generation (if needed) and forces calculation. Hence, we have implemented a kernel called integration (Algorithm 2) to perform the first part of the integration, whereas we have included the implementation of the second part of the integration at the end of the kernel for nonbonded interactions computation called forces.

**Algorithm 2 Algorithm that implements the first part of the integration**

1: \( i \leftarrow blockIdx.x \times blockDim.x + threadIdx.x \)
2: if \( i < \text{number of beads} \) then
3: \( \text{velocity}(i, t + \text{timestep}/2) \leftarrow \text{velocity}(i, t) + \text{timestep}/2 \times \text{force}(i, t) / \text{mass}(i) \)
4: \( \text{position}(i, t + \text{timestep}) \leftarrow \text{position}(i, t) + \text{timestep} \times \text{velocity}(i, t + \text{timestep}/2) \)
5: if \( \text{bead has an orientation} \) then
6: \( \text{half – advance momenta of inertia of bead } i \)
7: \( \text{calculations related to rotational motion of bead } i \)
8: end if
9: if \( \text{pressure is controlled} \) then
10: \( \text{coordinates of bead } i \text{ are rescaled} \)
11: end if
12: \( \text{boundary conditions of bead } i \text{ are applied} \)
13: end if

In Algorithm 2, \( i \) is a variable that 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. In the affirmative case,
the calculations related to the specific bead the thread is linked to (the bead having identifier \( i \)), are performed (lines 3–12). \( \text{velocity}_{(i,t)} \), \( \text{position}_{(i,t)} \), and \( \text{force}_{(i,t)} \) (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 size 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 changing the volume through rescaling the atomic coordinates. Standard periodic boundary conditions are applied at the end of the kernel (line 12).

Kernel for the generation of neighbor and interaction type structures

This kernel (called \textit{cudaneigh}) is used to update neighbor list and data structures that are used by nonbonded 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.

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 nonbonded forces 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. For this reason, the simulation region is divided into cells, where each cell has an edge size larger than that of 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 26 adjacent cells.

In the CPU version, the cell structure is a monodimensional vector organized as a linked list with \( \text{Number of beads} + \text{Number of Cells} \) elements. An example of the cell structure for the CPU version is reported in Figure 1. 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 2, we show these matrix values for the same example that we reported in Figure 1. Its row identifier relates to the cell identifier, while it has a number of elements equal to \( \text{Number of Cells} \times \text{Maximum beads per cell} \), 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 that 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 nonbonded forces kernel, to retrieve information on neighbor beads and their interaction types.

Compared to its sequential counterpart that 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 $2 \times \text{Number of neighbor pairs}$. 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.

Additional Structures. In addition, to enable coalesced accesses by the GPU, we created separate structures for all bead

**GPU optimized neighbor and interaction type structures**

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

These structures do not provide coalesced accesses for the GPU. Consequently, an optimized version is proposed as shown in Figure 4 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 Fig. 4) of size Number of beads where each element, associated (through its index) with one bead of the system, stores the number of neighbors of that bead.

**Neighbor and interaction type structures in sequential code**

<table>
<thead>
<tr>
<th></th>
<th>0th</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
<th>5th</th>
<th>6th</th>
</tr>
</thead>
<tbody>
<tr>
<td>0th</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>1st</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

All pairs of neighbor beads in the biological system

<table>
<thead>
<tr>
<th></th>
<th>0th</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
<th>5th</th>
<th>6th</th>
</tr>
</thead>
<tbody>
<tr>
<td>0th</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

Identifier of interaction type between beads 0 and 3

The type of interaction between the two beads of the i-th pair is stored in the i-th position of the interaction type structure.

**Figure 3.** A neighbor structure example (at top) used in the CPU version and the related interaction structure (at bottom). [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

**Figure 4.** 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 that stores information about bead interaction types. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]
positions, all bead velocities, all bead orientations, and all bead types, as these data are frequently accessed.

Kernel Description Neighbor and interaction type structures are generated using an algorithm similar to the one proposed in Ref. [19] for the neighbor structure construction.

However, the code reported in Ref. [19] is specialized for simulations of polymer systems and includes integrators, neighbor lists, LJ potential, and bonded forces and later in Ref. [33] also rigid body constraints.

Furthermore, in Ref. [19], a one-dimensional texture type is used.

In Ref. [24], the best performance of two-dimensional spatial locality of the texture cache is claimed when threads of the same warp, read texture addresses that are near from a two-dimensional viewpoint. Hence, we chose to use two-dimensional arrays for textures\(^6\) in this kernel as well as in the nonbonded forces computation kernel, where the accesses in device memory are not coalesced, for the following data: (i) position coordinates of beads; (ii) bead types that are used for the determination of the interaction type and related force field coefficients; and (iii) unit orientation vectors of beads that are used for the calculation of the rotational potential component of the energy.

Moreover, as opposed to the description in Ref. [19], 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 avoid a check to determine whether a thread is related to any particular bead of the cell considered or not.

Nonbonded forces computation

The nonbonded forces computation is implemented by the kernel forces, which exploits the optimized access to neighbor and interaction structures.

Compared to what has been done for CG polymer models in Ref. [19], 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.

Algorithm 3 Pseudocode of nonbonded forces computation
\begin{verbatim}
1: i ← blockIdx.x * blockDim.x + threadIdx.x
2: if i < number of beads then
3:   for n from 0 to neighbor[i] do
4:     j ← identifier of n\(^\text{th}\) 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 torques calculations in Algorithm 4
12:      else
13:        Forces and torques 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\(_\text{energy}_i\) ← pot\(_\text{energy}_i\) + pot\(_\text{energy}_j\)
19: end for
20: Update of system state, virials and orientation vector related to bead i
21: Second part of integration
22: end if
\end{verbatim}

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 with a bead of the system or not, to avoid useless computations. In the former case, in Algorithm 3, a cycling among all neighbors\(^6\) of bead i is carried out and for each neighbor n actions described in lines 4–18 are accomplished. The bead identifier of the n\(^\text{th}\) 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 \(r_{i,j}\) (line 6) and the minimum image convention\(^7\) is applied (line 7).

If the value of \(r_{i,j}\) 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\(_\text{energy}_i\), for the storage of values of force, torque, and potential energy of bead i (lines 16–18), respectively.

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 part of the 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 nonbonded forces acting on bead i. It accomplishes the

\(^6\)We use texture references as described in Ref.\([19,24]\) to define which part of texture memory is fetched. These texture references are bound through runtime functions to CUDA array regions of memory, before they can be read by kernels. We specify the dimensionality of the texture references as two-dimensional arrays using two texture coordinates. The specific code for the textures used in our application can be found in the GPU code of BRAHMS that we have uploaded in https://code.google.com/p/brahms-md/downloads/.

\(^7\)The number of neighbors for each system bead is stored in the auxiliary neighbor structure, named neighbor and described in paragraph ‘Neighbor and Interaction Structures.’

\(^7\)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\([31]\).
second half-advance of inertia moment of bead i if bead i is considered as a 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 i – j to the i\textsuperscript{th} particle value and by subtracting it to the j\textsuperscript{th} particle value. Instead, in the GPU implementation to avoid additional scattered accesses to memory and the corresponding impact on performance, we do not modify the j\textsuperscript{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\textsubscript{i,j} < r\textsubscript{cut} \textend{doc}\then\textend{doc}
2: Dipole–dipole interaction evaluation \textend{doc}\SSD\textend{doc} force field
3: LJ potential evaluation
4: Tetrahedral sticky term
5: end if

In Algorithm 4, r\textsubscript{i,j} is compared to the cut-off distance (line 1), and if it is lower, then the SSD force field contribution (lines 2–4) on the state of the system and its thermodynamic properties are computed.

Algorithm 5 Computation of interactions different from SSD interactions
1: if r\textsubscript{i,j} < r\textsubscript{cut} \textend{doc}\then\textend{doc}
2: LJ potential evaluation
3: Switch (Interaction type) \textend{doc}\Electrostatic interaction evaluation\textend{doc}
4: Case Charge\textsubscript{j} \textend{doc}= \textend{doc} Dipole\textsubscript{j}
5: …
6: Case Dipole\textsubscript{i} \textend{doc}= \textend{doc} Charge\textsubscript{j}
7: …
8: Case Dipole\textsubscript{i} \textend{doc}= \textend{doc} Dipole\textsubscript{j}
9: …
10: Case Charge\textsubscript{i} \textend{doc}= \textend{doc} Charge\textsubscript{j}
11: …
12: end if

In Algorithm 5, if r\textsubscript{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 the Switch Case part of Algorithm 5) related to the six different types of beads modeled in our application.

Simulation Setup

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

In Table 2 the main characteristics of the used architectures are reported. GTX295 has the smallest number of cores, whereas 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).

The reasons we consider three different GPU architectures in our work (two GF100 generation GPUs and one G200 generation GPU) include the observation of: (i) the scalability of our application among different GPU architectures; (ii) handling of resource limits from different GPU architectures; and (iii) accuracy of simulations among different GPU architectures.

The sequential simulation experiments have been performed on a Processor Intel® Core™ i7-920, equipped with four cores and eight 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.

We defined speed-up as the ratio between the total execution time of a simulation run in a single CPU core with respect to a single GPU. CPU execution does not apply MPI parallelization, and it is just a sequential execution.

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 size. The pressure was maintained at 1 atm with a time constant of 0.02 ps for simulations with timestep size of 1 fs and an isothermal compressibility of 4.6 e−5 atm−1. The temperature was maintained at 30°C with a time constant of 0.01 ps. We increase the timestep size and time constants by the same factor, when the dependency of the application performance from the timestep size is investigated. For example, for simulations with a timestep size of 10 fs, we use time constants of 0.2 and 0.1 ps for the pressure and temperature, respectively. Neighbor structure generation takes place approximately each 100 timesteps, when the timestep size is 1 fs. Therefore, the neighbor structure contribution on total simulation time is very low. We consider systems from 840 to 218,904 number of beads. As for complexity, we consider systems composed by heterogeneous types of beads. In particular, we consider either water systems (only 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

| Table 2. Technical specifications for the three NVIDIA architectures used in this work. |
|----------------------------------------|--------|--------|--------|
| GTX295                                 | GTX480 | C2050  |
| Number of SMs                          | 30     | 15     | 14     |
| Number of cores                        | 240    | 480    | 448    |
| Global memory (MB)                     | 896    | 1536   | 3072   |
| Memory bandwidth (GB/s)                | 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   |
simulator that 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 later in this section, using: (i) the slowest version of the simulator that is the CPU version; (ii) lipid system configuration; (iii) timestep size of 1 fs; and (iv) 5000 simulation steps, takes about 200 min.

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

Results

In this article, 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; (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 floating point (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. 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.

11We report detailed information on different execution flows related to bead types and water and charge representations in subsection “Nonbonded Forces Execution Flows.”

Finally, 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.

To quantify the scalability over architectural variants, we first observe speed-ups and then identify the impact of different architecture features and application characteristics on the performance achieved. By means of the speed-ups, we analyze how the different generation architectures handle different resource limits such as floating point arithmetic calculations, memory exploitation, and data transfers.

For the interpretation of the numerical results, we should take into account that the CPU used for comparison represents a single core but high performance architecture.

Impact of system complexity on achievable speed-up

In Figure 5, we report the comparison among the speed-ups (related to the CPU version) of two biological systems, lipid and water only, characterized by different complexities and simulated on the GTX295 architecture. For water systems, we distinguish between water only and complete version. Single-precision FP arithmetic is 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. The higher performance of the water only version is mainly due to the absence of structures to handle different bead types. Furthermore, taking into account simultaneously all the interaction potentials involved in the interactions among six different bead types, leads to having many more different variables in the kernel function code of the complete version with respect to the water-only version. Each of the variables is allocated in the registers but when the registers number per kernel is lower than the number of variables, then, local memory is exploited.

Figure 5. Comparison among the speed-up of different BRAHMS versions with respect to the CPU achieved on the GTX295 architecture. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]
to store the variable values that cannot be allocated in the registers. Local memory has much higher access times than registers, and this leads to a decrease of performance. 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 later in this section, thus leading to a performance hit.

Moreover, the impact of lipid data structures on the performance, is highlighted in Figure 5 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.

A smaller gap is observed (15.71% vs. 36.34%), when double-precision arithmetic is used. 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 with beads types and interactions. As single-precision arithmetic simulations are shorter, this constant overhead has a larger relative contribution in this case.

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 GeForce GTX295 architecture.

A higher number of average branches and divergent branches are observed for each kernel of the lipid system compared with the corresponding kernel of the water system. We note that corresponding to the speed-up gap between water and lipid system simulations (13.11x vs. 7.58x), 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 6. A descriptive explanation is provided, illustrating the execution time of the force fields calculations of water and lipid beads. Two simulations with the same number of beads are considered: 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 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 vice versa. 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 6 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, four supplementary force fields due to the five added lipid bead types are considered.

Architectural impact on speed-up

In Figure 7, we report for lipid systems and water systems, the speed-ups achieved by the three considered GPU 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 7.00x, 6.43x, and 2.43x for, respectively, GTX480, Tesla C2050, and GTX295
architectures, whereas for water systems the respective speed-ups achieved are 13.69x, 12.80x, and 5.28x.

The higher performance of the newer toward older generation GPUs is known and due mainly to the higher parallelism offered from the newer ones. In addition to the parallelism offered from the GPU architecture, the performance improvement of an application in a new generation GPU is strongly dependent on the characteristics of the application and how well these characteristics fit the improved architectural features introduced from new generation GPUs. In our case, the performance achieved is affected especially from the following aspects: (i) the quantity of floating point arithmetic operations included in the code; (ii) the exploitation of physical resources such as shared memory, texture memory and registers; and (iii) the use of CUDA intrinsic functions and transcendental functions in the application. All these aspects are different in different generation GPU architectures. Comparing the speed-ups, we have an immediate feedback of the good exploitation of the improved architecture features of new GPU generations.

Indeed, GTX480 shows to perform three times better than GTX295. This means that in our application, we take advantage of the better architecture characteristics of the new generation and not only of the higher parallelism. In fact, there are 480 cores in GTX480 versus 240 cores in GTX295 and the relative performance of GTX480 versus GTX295, due to the higher parallelism, is expected to be two times better.

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, whereas GTX480 has only one enabled.

Impact of timestep size on speed-up

In Figure 8, 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 sizes: 1, 10, 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 7.00, 7.44, and 7.90 for lipid systems and 13.69, 13.83, and 13.88 for water systems, for each of the timestep sizes, respectively.

We notice that the greater is the timestep size 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 size than for a simulation with a smaller timestep size. The construction of the neighbor structure is verified around each 100, 10, and 5 steps for 1, 10, 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 size 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 size. Instead, the non-bonded forces calculation and integration algorithms, corresponding to the other two GPU kernels, undergo the same number of calls, one call for each timestep, regardless of the timestep size. 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 size, leading to a greater speed-up factor.

As a water system simulation is longer than a lipid system simulation in the CPU, for the same number of beads (as shown in Fig. 6), 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 8, 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.

Time per day of simulations and normalized impact

The total time per day of simulations depends on the size and type of the system simulated and the timestep size used. For systems of 24k beads of lipids in water solution\textsuperscript{14} we simulate about 23 ns/day using a GTX480 GPU versus 3.8 ns/day when a single CPU core is used, with a 10-fs timestep. Instead, for systems with 24k beads of water, we obtain simulations of 43 ns/day using a GTX480 GPU versus 3.5 ns/day on a single CPU core, still using a 10-fs timestep.

Moreover, to enable a better comparison with previous literature, we introduce a parameter that we call normalized impact. We calculate the normalized impact value as the ratio between the execution time of the application implemented on the GPU and the product of number of beads and number of timesteps of the simulation. The lower is the value of the normalized impact, the better is the performance of the application.

We observed that the normalized impact values for systems with more than 23k beads are very similar: They have a maximum difference of 3%. We averaged the values of the normalized impact for different system sizes with more than 23k beads, because for simulations of systems with fewer beads, the parallelism offered from the GPUs was not fully exploited. In Figure 9, we report the averaged values of the normalized impact for lipids in water solution simulations and water simulations for the four different architectures.

For example for systems with more than 23k beads of lipids in water solution, we have the following values for the normalized impact: 10.14, 4.18, 1.58, 1.45 \times 10^{-6} per bead per step, for each of the four architectures considered respectively. For water system simulations with more than 23k beads, the normalized impact values for each of the four architectures are respectively: 10.23, 1.94, 0.8, 0.75 \times 10^{-6} per bead per step. The GTX480 GPU architecture needs less time per bead and per step to complete the simulations with respect to the other architectures. Moreover, the normalized impact value is lower for water system simulations than for lipid in water solution system simulations for GPU architectures and greater in the case of the CPU. The reason of this behavior is explained in paragraph “Analysis of Biological System Complexity.”

Spatial sort of beads in the application

Sorting is one of the optimizations performed in previous MD works. We also applied beads sorting in our tool. More specifically, we implemented the sorting of beads as suggested in Ref. [19]. This sorting is based on spatial sort rules, where we assign to each bead a value depending on its position in the

\textsuperscript{14}For example, in Ref. [26], 24k beads simulations are used to calculate important physical properties such as a quantitative characterization of the internal distributions (profiles) of lateral pressure and electrical potential.

\textsuperscript{15}Threads and blocks of threads in the GPUs are scheduled at different times, whereas in the CPU there is a unique sequential execution flow.
system, determined from the permutation rules explained in Ref. [40]. We call this value Hilbert parameter. We perform a sort of all the elements of the structures containing information on beads according to the Hilbert parameter. We have also implemented an additional kernel to perform the sort of the neighbors of each bead according to the Hilbert parameter assigned to the bead. This sort occurs after the kernel of the neighbor and related structures generation has been executed.

We performed simulations using the sorting algorithm on different system configurations. The resulting execution times are in most of the cases marginally degrading the total execution time (almost 1%). As such, we decided to remove this sorting step from our application. In future work, we plan to investigate alternative sorting methods.

Nonbonded forces execution flows

We report the flowchart of the simulator code, for nonbonded forces calculations in Figure 10. This part corresponds to lines from 8 to 18 of Algorithm 3. In this code, conditions related to the electrostatic interactions and SSD potential are highlighted.

In the flowchart, condition A (corresponding to line 8 of Algorithm 3) checks whether the neighbor pair \( i/C_0 j \) has a distance within the cut-off radius in order for its nonbonded potential contribution to be considered for further nonbonded forces calculations. Then, condition B that refers to line 10 of Algorithm 3 follows and checks whether the \( i/C_0 j \) 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 \( i/C_0 j \) in the case both bead \( i \) and bead \( j \) are water beads. If at least one of them is not a water bead, the nonbonded 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 with 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, whereas 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.

Figure 10. Execution flow of nonbonded forces computation for each pair \( i/C_0 j \) 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.

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, as 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.
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, 1536, 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.

Accuracy of the Simulations

In Table 6, we report mean and standard deviation values of temperature of lipids as well as of water, for 23k bead simulations 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, one million steps have been performed for each simulation. For these simulations, we show also the percentage of difference between mean values of beads temperature in all GPU architectures used 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%. 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.

Although Table 6 indicates a correct conservation of the temperature during one million of 1 fs steps, presenting almost identical values in 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.

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 the 50 to 150 ps interval of the simulation time 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 with our CG simulator lasting more than 30 ps with a timestep size of 1 fs, the use of GTX295 architecture is not suitable, whereas 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.

![Figure 11](image-url)
Although 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[24] 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 from a different order of arithmetic FP operations in the GPU with respect to GPU architectures, 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 simulations 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, on both GPU and CPU simulations, are coherent.

Further Optimizations on Forces Kernel

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

Data reuse and work merging

When we split into two different kernels, the calculation of nonbonded forces and the accomplishment of the second part of the 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 nonbonded forces computation and second part of integration, because of the following reasons: (i) A unique kernel, in our case enables pre-computed data reuse,\textsuperscript{111} such as data on beads forces, torques, and rotation matrix, avoiding additional 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.

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 and (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.[24]

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)\textsuperscript{†} floating-point arithmetic; (ii) fixed-point arithmetic; and (iii) intrinsic CUDA functions: \_fadd\_rn, \_fmul\_rn, \_fdiv\_rn, and \_fsqrt\_rn.

\textsuperscript{†}We have also noticed in Ref.\textsuperscript{111} 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.

<table>
<thead>
<tr>
<th>Table 8. Speed-ups achieved when fixed-point and intrinsic functions are used for floating-point arithmetic operations.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>Water systems</td>
</tr>
<tr>
<td>GTX295</td>
</tr>
<tr>
<td>GTX480</td>
</tr>
<tr>
<td>C2050</td>
</tr>
<tr>
<td>Lipid systems</td>
</tr>
<tr>
<td>GTX295</td>
</tr>
<tr>
<td>GTX480</td>
</tr>
<tr>
<td>C2050</td>
</tr>
</tbody>
</table>

We have a maximum speed-up of 27.71\% for water simulations and 12.91\% for lipid simulations, when intrinsic CUDA functions and GTX480 architecture are used 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 use 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 Ref. [41], 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.[24]
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 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 size, for 23k bead systems of lipids in water solution in the case of: (i) CPU, GTX295, GTX480, and C2050 architectures and (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 GPU 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; and (iii) the different order of execution of the operations in GPU architectures with respect to the CPU.

Comparison with Related Work Reported in the Literature

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. HOOMD is another MD simulations software specifically implemented for running on GPUs. It is specialized in simulations of polymer systems. NAMD has been the first MD simulation package to include GPU acceleration. LAMMPS is an MD simulator that includes force field potentials for the simulation of soft and solid-state materials, CG models, and mesoscopic systems. Several of the force fields enclosed in LAMMPS have been implemented for GPU acceleration.

In general, these implementations outperform traditional CPU cores by factors ranging from 10 to 20 and 100 in some ideal cases. However, most of the CG models are limited to the acceleration of specific and not complex parts of the model. In this 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.

It is relevant to discuss how our simulator would perform if applied to some of the systems studied in the literature with alternative GPU simulators. While making accurate predictions is difficult, because of the many variables involved (system size, simulation protocol, hardware used, etc.), it is reasonable to expect the speed-up factors obtained in this work to be transferable. So, for example, it would be possible to simulate the GB liquid crystals of Sunarso et al. with a speed-up factor comparable to that obtained for our lipid systems. In some cases, such as for simple systems like the LJ mixtures of Colberg and Hofling we would expect the speed-up of our simulator to improve, as we would avoid the computationally heavy overheads related to the nonspherical potentials. It is also important to stress that we can only simulate systems modeled by force fields that are available in BRAHMS. For example, to simulate the polymer systems of Nguyen et al. which make use of the WCA potential, we would need to implement such an additional model. On the other hand, as far as we are aware, BRAHMS is the only GPU simulator that implements the tetrahedral octupolar potential (required by the SSD model), so in this respect it offers a unique capability. In general, each of the available GPU simulators implements only a subset of all possible molecular models, will be particularly suitable to tackle certain types of simulations, and will also inevitably be affected by particular limitations. It is, therefore, important to be aware of the differences between the various available programs so as to choose the most appropriate one for a particular problem of interest.

Conclusions

In this work, we presented an optimized, accelerated version of a CG MD 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 complexities, composed of water and lipids, observing that lipid systems achieve a speed-up almost two times lower 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 ~14 for water systems and 8 for lipids on the GTX480 architecture. Moreover, we analyzed the dependence of the achievable speed-up from the timestep size, obtaining slightly greater speed-up values for simulations with larger timestep size. 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 investigated the numerical 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.

Keywords: molecular dynamics · GPU acceleration · coarse grain · membrane modelling · CUDA


Received: 13 July 2012
Revised: 19 October 2012
Accepted: 25 October 2012
Published online on 28 December 2012