JOURNAL OF CHEMICAL INFORMATION AND MODELING

Article

pubs.acs.org/jcim

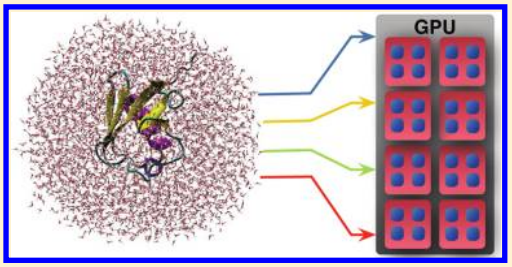
# Computation of Induced Dipoles in Molecular Mechanics Simulations Using Graphics Processors

Frederico Pratas,[†] Leonel Sousa,[†] Johannes M. Dieterich,[‡] and Ricardo A. Mata[‡,*]

[†]INESC-ID/IST, Technical University of Lisbon, Rua Alves Redol, 9, 1000-029 Lisboa, Portugal

[‡]Institut für Physikalische Chemie, Universität Göttingen, Tammannstrasse 6, D-37077 Göttingen, Germany

**ABSTRACT:** In this work, we present a tentative step toward the efficient implementation of polarizable molecular mechanics force fields with GPU acceleration. The computational bottleneck of such applications is found in the treatment of electrostatics, where higher-order multipoles and a self-consistent treatment of polarization effects are needed. We have implemented a GPU accelerated code, based on the Tinker program suite, for the computation of induced dipoles. The largest test system used shows a speedup factor of over 20 for a single precision GPU implementation, when comparing to the serial CPU version. A discussion of the optimization and parametrization steps is included. Comparison between different graphic cards and CPU−GPU embedding is also given. The current work demonstrates the potential usefulness of GPU programming in accelerating this field of applications.




## INTRODUCTION

The question of computational cost is a central issue in the simulation of biochemical systems. The size scales involved are extremely large, usually well above 1000 atoms, and one needs to propagate the system in a suitable time scale (at least in the nanoseconds regime). This requires thousands of computations in a system with thousands of interactions and coupled degrees of freedom. The tools of choice for such systems are parametrized force fields, which make use of empirical data to greatly simplify the description of the forces. The most widely used models to date rely greatly on 2-body energy functions. Bonded interactions can be quite accurately described with such a model because the conformational space of interest is rather small. However, the description of nonbonded interactions is a much more delicate task. In most current-use force fields, such as AMBER,[1] OPLS,[2] or CHARMM,[3] the electrostatics are represented through atom-centered point charges (PCs), which are kept throughout a simulation. These point charges are parametrized taking into account a given environment and describe in a crude way the distribution of electronic density in a molecular system.

The frozen PC description leads to several different faults in the model. The first problem is simply the lack of flexibility in such an approximation. Although electronic charge density can be reasonably described by PCs for large distance interactions, higher terms in the multipole expansion become important for close interacting molecular species. The other problem lies in the fact that any molecule is not allowed to change its charge distribution along the course of a simulation. For example, in the case of water, it is easily expected that the electronic density distribution will change depending on whether the molecule is acting as a single/double hydrogen bond donor and/or acceptor. The environment effect on the electrostatic properties of the molecule can be significant.[4]

In order to overcome the above-mentioned limitations, new force fields have been developed that include a more refined description of electrostatics.[5−15] This can be accomplished by adding higher-order multipole terms and creating an explicit dependence between the electrostatic terms and the surrounding environment. The AMOEBA model[16,17] is a preeminent example of this class of polarizable force fields. The electrostatics are described by multipole sites (usually atom-centered, but not exclusively); each site includes fixed monopole, dipole, and quadrupole terms. Polarization effects are taken into account by a dipole term that depends on the external electric field. This induced dipole can account for the effect of different environments on the site electrostatics. However, the computational cost of adding these terms is extremely significant. In particular, the iterative self-consistent calculation of thousands of induced dipoles gives rise to a computational bottleneck that limits its application to relatively small size and time scales. This is even aggravated by the fact that the water model is flexible, allowing only for time steps on the order of 1−2 fs.

One possibility to work around this bottleneck is to make use of hardware acceleration. The use of graphical processing units (GPUs) in the context of scientific computing has significantly increased in the past few years. This is due not only to the rapid increase in computational performance of these devices, but also to the availability of general purpose programming interfaces. GPU-compatible implementations of classical force fields have been developed, with significant computational

**Received:** November 29, 2011
**Published:** April 27, 2012





dx.doi.org/10.1021/ci200564x | *J. Chem. Inf. Model.* 2012, 52, 1159−1166

gains.[18−27] However, GPU codes are still not widely available in the case of polarizable force fields. This is certainly linked to the lesser popularity of these models, as they are not applicable to the systems sizes and time scales of greater interest for the community (above 20,000 atoms and pushing above the nanosecond time scale). Implementations based on the classical Drude oscillator model would be an exception.[28] The interactions are still between point charges, just as in the case of nonpolarizable force fields, and the computational cost may not be significantly different. Polarizable force fields with higher multipole interactions, nonetheless, could substantially profit from the use of GPUs.

Molecular dynamics programs today are highly efficient codes, with exceptional scalability in CPU distributed computing. The algorithms have also developed in line with the CPU architecture, favoring the use of cutoffs and sorting operations to restrict the calculation as much as possible to the most relevant interactions, thereby reducing the number of floating point operations. Such an approach is optimal for a CPU implementation but hard to transpose for use with a GPU. The latter works best performing the largest possible number of floating-point operations even if a large portion may be redundant. This is one of the reasons why a finely tuned CPU program running on a multicore machine can still compete with GPU-accelerated codes, as long as the system size is large enough to make the interaction matrix sparse. However, in the size range where polarizable force fields are used, such algorithms give little advantage because almost all interactions are significant. Not only that, the computational cost of evaluating a single interaction term is higher, so that there are almost no "tricks" that can be used to reduce the number of operations.

In this work, we present a GPU-accelerated implementation of the bottleneck step of AMOEBA runs, the induced dipole calculation. The implementation was carried out under the CUDA framework for use with NVIDIA cards. The following sections briefly summarize the algorithm and the GPU-specific changes that have been effected. We do not discuss the use of GPU acceleration in the MD steps. The rationale would be similar to classical forcefields, for which efficient implementations have already been presented.[19−21,24−26]

## GPU HARDWARE AND CUDA OVERVIEW

Originally, GPUs were small controllers or special-purpose accelerators used to build images in a frame buffer intended for output to a display. First GPU microarchitectures were very limited in terms of flexibility and functionality. In fact, they could only accelerate certain parts of the graphics pipeline, namely, conventional operations at vertex and pixel level. Thrusted by the gaming industry demands, market GPUs have kept up with the fast processor evolution observed in the past few years. In fact, GPUs have followed pretty much the same trends as other multiprocessors, i.e., an increasing number of processing cores with an increasing performance power. Nowadays, GPUs are affordable fully programmable platforms that can be used not only for graphical applications but also for other general-purpose applications (GPGPU).[29]

Although in general the number of cores in multiprocessors tends to increase, there are capital differences between CPUs and GPUs. General-purpose architectures are optimized for sequential code execution and include complex hardware structures such as branch prediction and cache coherent management structures. Modern GPUs are instead optimized for data parallel applications and can be seen as massive multiprocessors with many simple cores, typically 10× to 100× more than a general-purpose processor. In other words, the typical modern GPU architecture consists of numerous simple processing units supporting a highly multithreaded engine and organized in a pipeline manner. GPUs' microarchitectural optimizations include fine-grained SIMD parallelism, large data arrays, stream processing, and low latency floating-point operations. All of these features create a potential candidate architecture for processing applications with data and computation intensive requirements such as in the case of molecular mechanics.

Another important characteristic is that GPUs are coprocessors, i.e., they are not able to work independently and, therefore, are always attached to a host CPU, typically through an external bus such as PCIe. Moreover, GPUs have their own memory hierarchy structure. These two facts make of this platform a heterogeneous distributed multiprocessor and bring some inherent difficulties to the programmer. The most relevant is how to deal with the data partitioning and transfers in order to achieve a well performant solution, as we discuss in the rest of this article. Therefore, although the implementation provided in this work was programmed targeting NVIDIA GPUs, the majority of the problems discussed are general enough to be applied to other GPU architectures.

To develop our program, we have used the CUDA framework. CUDA is a complete environment for software development and execution, including a runtime API and parallel programming model. It was introduced by NVIDIA as a way to leverage the use of NVIDIA GPUs. CUDA supports several high-level programming languages such as C/C++ and Fortran. At its core, there are three key abstractions that are exposed to the programmer as a set of language extensions: (i) a hierarchy of thread groups, (ii) a hierarchy of shared memories, and (iii) synchronization mechanisms. These abstractions allow the programmer to partition the target application at different levels of complexity.

Overall, parallelism with CUDA is achieved by executing the same function, called kernel, $N$ times in parallel by $N$ different CUDA threads. Threads are organized in blocks, and groups of blocks form a grid. The grid represents the overall work that is to be executed in the GPU and split in a coarse-grained fashion through several blocks. Each block then represents a piece of fine-grained work that is expected to reside in the same stream multiprocessor. Blocks must be able to execute independently and in any order. Threads within the same block can cooperate via the shared memory, and the maximum number of threads per block is restricted by the amount of shared memory, registers, and other hardware resources available in the device. Threads are scheduled in groups called warps.

During execution, CUDA threads may access data at several distinct levels of the memory hierarchy: global, constant, texture, shared, and private local memory spaces. The global, constant, and texture memory spaces are persistent across kernel launches by the same application. Global memory resides in device memory and is not cached. Constant and texture memory also reside in the global memory but are cached. Besides these levels, there are the shared memory and the local private memory spaces. On the one hand, we have shared memory, which resides in the on-chip memory, is visible to all threads of a block, and has the same lifetime as the block. On the other hand, there is local memory, which is private to each thread and resides in the device memory. Local memory is

most commonly used by the compiler for register spilling, i.e., to store local variables that do not fit in the available registers. It is therefore extremely important that the programmer is able to manage the kernel in order to optimize the registers use.

Finally, the last exposed abstraction in CUDA is synchronization. Device synchronization mechanisms include implicit and explicit intrablock synchronizations. Implicit synchronizations are related to the fact that threads are scheduled in warps, and, therefore, threads within the same warp are always synchronous. Explicit synchronizations are introduced in the kernel by the programmer by using specific intrinsic extensions. These include memory fences and barriers that affect all threads within a block.

## ■ ALGORITHM

One of the most time-consuming steps in simulations involving the AMOEBA potential is the calculation of induced dipoles. These are responsible for the polarizable character of the force field and the main distinction from other potentials. The other code sections needed to work with this potential class can be improved with the same approaches used by other authors for nonpolarizable force fields. The full description of the potential is given in the original papers.[16,30,31] We will simply do a short overview, following the same nomenclature. In a MM run with the AMOEBA potential, multipole sites are generated at the atomic positions. In some cases, these can be shifted (e.g., along a bond), but these are parametrization issues. Nevertheless, in order to keep the generality of our discussion, we will restrict ourselves to the use of the term "site". For each site $i$, a vector $M_i$ is built, containing all fixed multipoles up to quadrupoles $M_i = (q, \mu_{i,1}, \mu_{i,2}, \mu_{i,3}, Q_{i,11}, Q_{i,12}, ..., Q_{i,33})$. These multipole sites will interact with other sites through the electric field they generate. At site $i$, an electric field component is given by the expression

$$E_{i,\beta} = \sum_j \Gamma^{ij}_{\beta} M_j \qquad \beta = 1, 2, 3 \tag{1}$$

The matrix $\Gamma^{ij}_{\beta} = (T^{ij}_{\beta}, T^{ij}_{\beta 1}, T^{ij}_{\beta 2}, T^{ij}_{\beta 3}, ...)$ is the multipole interaction matrix between two sites, and $\beta$ represents the three spacial dimensions. The induced dipole $\mu^{ind}_{i,\beta}$ is then given by the interaction between the field at the site, and its polarizability $\alpha_i$, with $\mu^{ind}_{i,\beta} = \alpha_i E_{i,\beta}$. Because all sites contribute to the electric field, even those in the same molecule, it is necessary to include a damping term in order to avoid overpolarization at short-range. This is built into the interaction matrix elements. The charge-dipole terms, for example, are given by

$$T^{ij}_{\beta} = -[1 - \exp(-au^3)]\frac{R_{ij,\beta}}{4\pi\varepsilon_0 R_{ij}^3} \tag{2}$$

The effective distance $u = R_{ij}/(\alpha_i\alpha_j)^{(1/6)}$ is given as a function of the atomic polarizabilities, $\varepsilon_0$ is the vacuum permittivity, and $a$ is a dimensionless parameter that can be used to control the strength of damping. For large values of $a$, the damping is removed, and eq 2 reverts to its classical dipole−dipole interaction form. More elaborate expressions are needed for higher multipoles and can be obtained by derivation of the lower-ranking expressions. They are explicitly given in the original paper.[31] Furthermore, just as in the case of nonpolarizable force fields, electrostatics between neighboring atoms is scaled or fully removed. This is controlled by scaling prefactors and set according to connectivity and polarization groups classification.

In the case of a periodic simulation box, electrostatic contributions are taken into account through a regular Ewald sum. These expressions are available in ref 32. It should be noted that a classical Ewald sum is not the most adequate choice of algorithm for the calculation of long-range electrostatics. Particle Mesh Ewald (PME),[33] for example, will be much more efficient in dealing with large sized simulation boxes. We opted nevertheless to implement a regular Ewald sum for two main reasons. First of all, we did this in order to keep compatibility with the underlying serial code available at the time the project began (Tinker v4.2).[34] Second, it is not clear if in the application range of the AMOEBA potential, which are still relatively small sized systems, a GPU code would significantly profit from the use of PME. Some of the steps, such as charge location and spread, are difficult to efficiently implement on a multithreading platform.[20] In the following sections, we describe the general code used for the simulations, comparing the serial CPU code to the structure of a GPU accelerated version.

**Serial Implementation.** In Figure 1, the general structure of a serial code for the iterative computation of induced dipoles

```
for i = 1 to # of sites                                   Fixed Induction
    calculation of scaling prefactors
    for j = i + 1 to # of sites
        compute E_i due to M_j^fix
        compute E_j due to M_i^fix
    compute μ_i^ind due to fixed contributions

do                                                        Mutual Induction
    for i = 1 to # of sites
        calculation of scaling prefactors
        for j = i + 1 to # of sites
            compute E_i due to μ_j^ind
            compute E_j due to μ_i^ind
        update μ_i^ind
while(check for convergence)
```

**Figure 1.** General CPU program structure for computing AMOEBA induced dipoles.

is presented. It is composed of two blocks. In the first block, the electric field generated by fixed multipoles is computed at each site, and the resulting induced dipole is calculated. Because the induced dipole is proportional to the electric field, this value can be stored and only needs to be computed once. In the second block, the electric field at every site $i$, due to the induced dipoles at all other sites $j$, is computed and the induced dipole updated. This block is computed self-consistently because the induced dipoles change under the effect of other sites. In both blocks, the electric field generated by neighboring sites or in the same group as $i$ are scaled. These scaling prefactors depend on the parametrization and can even change between mutual and direct induction. These are precomputed (gathered) for each site $i$, as shown in Figure 1. Convergence is checked at the end of each cycle. By default, the RMSD of all induced dipoles is computed between the current and previous iteration. Besides the fact that the operation has to be repeated until self-consistency, the main difference to the fixed induction block is that each mutual induction cycle requires less floating-point operations because one is only dealing with dipoles. In the fixed block, monopoles, dipoles, and quadrupoles have to be computed. The scaling prefactors, as previously mentioned, are used to remove or scale electrostatic effects in neighboring atoms or within a given group. This information has to be


dx.doi.org/10.1021/ci200564x | J. Chem. Inf. Model. 2012, 52, 1159−1166

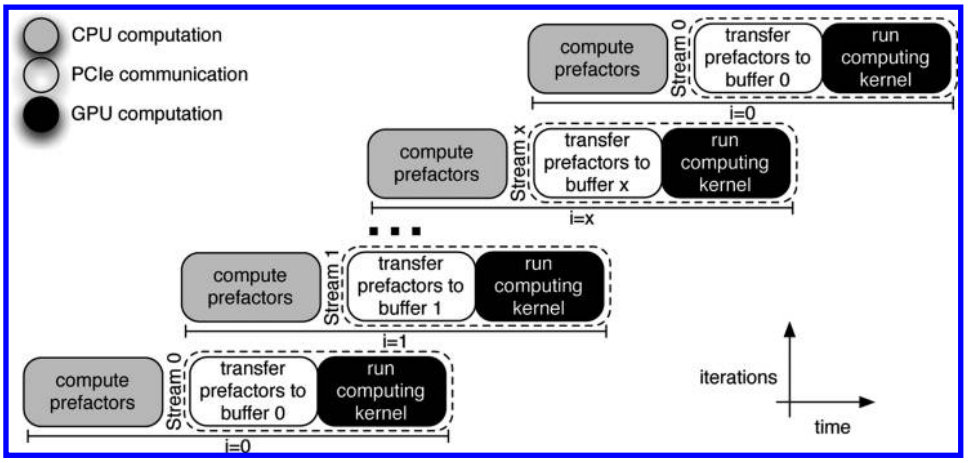



**Figure 2.** Scheme representing the overlap between CPU computations, communication, and the GPU kernel executions.

gathered for each individual site, so it is assembled within the loop, being kept in memory for the duration of the run.

In case one uses periodic boundary conditions with an Ewald summation, the electric fields from real and reciprocal spaces have to be computed. The former follows the same structure. In the latter case, one has a triple summation over the lattice vectors. We now proceed to discuss both cases for GPU-acceleration.

**GPU Implementation.** In discussing our GPU implementation, we start by restricting ourselves to the real space case, which is represented in Figure 1. As usual in pair-interaction GPU algorithms, it is more effective to replace the sum over unique pairs ($j > i$) by a redundant pair sum, with $j$ running over all sites.[21] The $j$ loops can then be performed by calling separate kernels. The redundant pair sum allows us to remove from each $j$ loop the calculation of $E_i$, thus eliminating the inner-loop write data dependencies.

The fixed induction block, for example, can be accelerated by executing two kernels. A first kernel, running over all sites computes $E_j$, replacing the $j$ loop shown. The sites are blocked, with each $j$ being associated with one independent thread. At the end of each $i$ iteration, the kernel accumulates the $E_j$ results with the partial results of the previous iterations, as each $E_j$ is stored in a different memory position. Self-interaction is removed by setting the corresponding contribution to zero.

The last step, where $\mu_{i,\beta}^{\mathrm{ind}} = \alpha_i E_{i,\beta}$ is computed, is in fact a loop over all sites, and makes use of a second kernel. The latter is only executed once, with each site again being associated to a single thread. It is also worth noting that performing this final loop in the GPU reduces drastically the amount of data that otherwise would have to be transferred through the slow PCIe connection (bus between CPU and GPU).

The computation of the scaling prefactors (e.g., to avoid the interaction between neighboring atoms) is carried out in the CPU. There are several reasons for not including this operation in the GPU. First of all, it would not be very efficient because it would require transferring each site connectivity data. This data could only fit in the global memory, which is only slowly accessed. Although this could be done more efficiently by sorting the sites and separating them into blocks, the calculation itself is a run over small loops. This, again, is not very efficient in the GPU. Another option would be to precompute the whole data and store it in the GPU global memory, but that would incur unacceptable memory space as the prefactors array size would be proportional to the square of the number of sites. Instead, these quantities are precomputed in a per iteration basis, and their transfer to the GPU memory is overlapped with the CPU and GPU calculations. A schematic representation is given in Figure 2. For the system sizes considered in this study, we found the overlap to be significant. This is further discussed when timings are presented.

Some further points should be noted. The first one would be that by removing the computation of $E_i$, the parallelism inside the $j$ loop is increased, as there are no write data dependencies, i.e., all the executions can be reduced trivially. The contributions can all be added, and all threads are computing the electric field at the same site, without the need for inner synchronization points. Moreover, the calculation of the $\mu_j^{\mathrm{ind}}$ is now also performed fully in parallel by calling the second kernel only once at the end of the $i$ loop. These are important factors for performance. The second reason is that the number of operations in each thread is also reduced. Although this may seem at first as a drawback on the GPU because we would like to have as much computation as possible with minimum data transfers, it is not. The resources available on the simple GPU processing cores (namely, register and shared memory) are not enough to support the complex computations required without having to do spilling.

The mutual induction block is parallelized in a similar fashion. The major difference is that in order to check for convergence, the sum of the square deviation of the induced dipoles has to be computed. In order to do this efficiently, a partial two step reduction is used. In the first step, the partial square deviation of the sites in each block is computed, the reduction being executed in a binomial fashion using groups of 16 threads or 32 according to the respective GPU warp size[35] in order to avoid explicit synchronizations. In the second part, the partial reduction (of the block values) is finalized in a serial fashion.

In the case of an Ewald sum, the CPU implementation is structured differently, the indices running over the reciprocal lattice dimensions. As it is well-known, due to symmetry, the only terms needed in the sum are limited to

$$\sum_{h} = \sum_{\substack{j'=0 \\ k'=0 \\ l'=1,l_{\max}}} + \sum_{\substack{j'=0 \\ k'=1,k_{\max} \\ l'=-l_{\max},l_{\max}}} + \sum_{\substack{j'=1,j_{\max} \\ k'=-k_{\max},k_{\max} \\ l'=-l_{\max},l_{\max}}} \tag{3}$$

The dashed letters stand for the reciprocal lattice indices ($j'$, $k'$, $l'$). Although we have separated the total sum into three


dx.doi.org/10.1021/ci200564x | J. Chem. Inf. Model. 2012, 52, 1159−1166

different sums, a CPU implementation will execute a single sum using three nested loops (accounting for the necessary exceptions). Inside each loop, the necessary factors are computed, products of trigonometric functions according to the $h$ vector. Although eq 3 bears little resemblance to a typical CPU code structure, it is more adequate to discuss our GPU implementation. The latter is structured as follows. The outer $j'$ loop is handled by the CPU. Inside the loop, three kernels are called. The first kernel computes the factors that depend exclusively on $j'$. The execution of this kernel is extremely fast. Such an operation could also be handled by the CPU, but by executing it on the GPU, we avoid once more the slower PCIe data transfers. The second kernel performs the bulk of the calculations, calculating the electric field generated at each site. In its execution, each block is indexed to a $h$ vector. At this point, it is of interest to consider the splitting of the sums used in eq 3. For $j' = 0$, $k' = 0$, only $l_{max}$ blocks are needed. For $j' > 0$, a much larger number of lattice vectors are computed at each call, so one has to explicitly consider this while building the kernels to achieve optimal performance. Again, in the $j' = 0$, $k' = 0$, one-dimensional blocking is used with $l'$ as the index. The threads are indexed to the multipole sites. Several batches of sites are computed in each block sequentially until the full list is looped. Reductions are again performed in a binomial fashion. When the number of lattice vectors per call is increased, 2-dimensional blocking is used, with both $k'$ and $l'$ as block indices.

The third kernel finally computes the change to the induced dipole; reducing the reciprocal electric field results in a parallel fashion, with each thread working sequentially on a given site. This kernel has been introduced to improve parallelism, and because all the information can be kept inside the GPU between the two calls, the calculations can be split without any performance reduction. The computation of the real space part is similar to what has been discussed previously.

## DISCUSSSION

**Performance.** We start by considering the speedups achieved in the case of nonperiodic simulations. There can be significant performance differences relative to the use of an Ewald sum, so these will be discussed later in the text. The first test example is a single configuration of the ubiquitin protein embedded in a water droplet. The system consists of 11,885 atoms (which includes 3,551 water molecules). The speedups for the full self-consistent cycle with two different computing systems is given in Figure 3. We have taken the timings of a single processor run as reference (Intel X5550 2.67 Ghz).

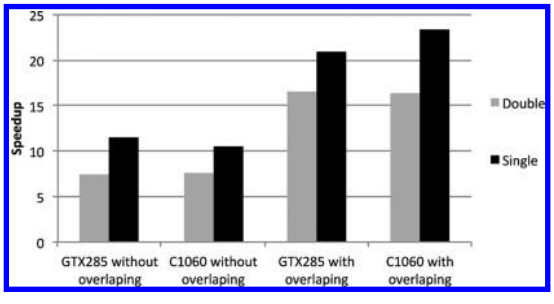



**Figure 3.** Speedups for the ubiquitin protein in a water droplet, using different systems and synchronization procedures. Results are given relative to a single processor run, using either single or double precision arithmetics.

Although we understand that such a choice (single processor use) does not meet well with the standards of today high-performance computing for molecular dynamics simulations, there are several reasons behind such a choice. The original code to which our library was interfaced is a serial implementation, and we opted to keep the original program as reference. Parallelizing the code is a possibility, and this has even been discussed in a previous communication.[36] However, it would be unfair to take the code "as is" and present it as a high-performance code. The algorithm chosen, although suitable for a single processor execution, would not be up to standards for distributed computing.

The results show timings for two different systems. The first corresponds to a quad-core Intel Q9550 2.83 GHz processor and GTX285 Nvidia GPU, and the second system is equipped with dual quad-core Intel X5550 2.67 GHz processors and a C1060 Nvidia GPU. The timings for both systems are relatively close. Also shown are the results obtained with different synchronization procedures, with and without the overlap shown in Figure 3. On the one hand, the first two groups of columns show the speedup obtained by pure computational acceleration with the inherent associated overheads of using the GPU, mainly the memory transfer delays. On the other hand, the second group of columns shows the speedup obtained when the different sections of the computation are performed in pipeline as depicted in Figure 2. In the latter case, we are able to overlap the execution of at least three iterations, one gathering the prefactors on the CPU, a second iteration transferring the prefactors to the GPU, and finally a third iteration running the kernel on the GPU. The results clearly show the advantage of the latter procedure.

The difference in speedup between single and double precision is somewhat surprising. Usually, one would expect an improvement in the order of 2−8 times increase in computation speed. The lower limit would correspond to the case of a purely memory-limited implementation, and the upper limit to a purely computation-limited implementation due to the ratio between single and double precision arithmetic units. In the GPU, the ratio is 8:1, and in current CPUs it is 2:1, meaning that the change from double to single precision can have a much larger impact in GPU performance. In the case at hand, one only observes an increase of about 30%. There are several reasons behind this fact. First of all, not all the data used is represented in double precision. Some inputs are integers, and the same is valid for the instructions, as part are not purely arithmetic. Second, the execution of the kernels has become so fast that the remaining steps (e.g., the scaling connectivity prefactors calculation in the CPU) become the dominant timing factor, thus reducing the overall speedup gain. In fact, we have obtained the sequential contributions of each part of the execution (Figure 4) showing that the kernel is able to achieve a 2× difference, but the transfers and the prescaler calculation are not as fast as one would expect. The difference between the direct and mutual induction kernels should also be noted. As mentioned while discussing the serial implementation, the computation of induced dipoles is divided into two main steps. In the first set of calculations, the induced dipole generated by fixed multipoles is computed (direct induction). This step only has to be carried out once. In the second step, a series of cycles is performed where the electrical field generated by other induced dipoles is taken into account (mutual induction). This has to be performed self-consistently. It is shown in Figure 4 that the direct induction kernel is able to achieve a speedup of

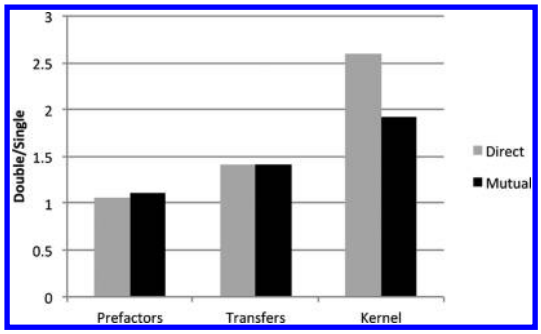



**Figure 4.** Comparison of speedups executing single versus double precision for independent sections of code (C1060 GPU system).

about 3×, while only about 2× are obtained for the mutual. This difference can be explained by the fact that the direct kernel computation is more arithmetic bounded than the mutual (calculations are performed up to quadrupole). Overall, a maximum speedup factor of 23× can be observed for this system, making use of single precision arithmetics.

We now turn to results for periodic simulation boxes. We have considered three systems. As a reference, we have chosen a simple box with 800 water molecules (28.81 Å in length). For comparison, we consider a simulation box with the double amount of molecules (1600, with 36.30 Å in length) and a simulation box holding a capped histidine aminoacid, solvation ions (3 sodium and 4 chloride ions), and 981 water molecules (31.0 Å in length and 2980 atoms in total). The latter is very modestly sized but already includes strongly charged species and serves as a model system for biomolecular simulations.

The speedups are shown in Figure 5. We show separately the performance gains for the real and reciprocal space calculations.

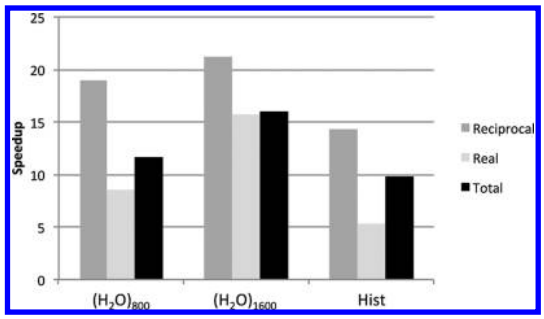



**Figure 5.** Speedups for the water box and histidine in solution simulations. Results are given relative to a single processor run (C1060 GPU system).

All results presented are for single precision code. The figure shows that the real space calculation is the one that profits the least from the GPU acceleration. Comparing the results of Figures 3 and 5, it is clear that the speedups are strongly dependent on the system size. The speedup on going from the $(H_2O)_{800}$ to the $(H_2O)_{1600}$ calculation almost doubles, a reflection of the increasing occupation of the GPU mainly for the real space calculation. This is further justified by the fact that the number of *h* vectors has been kept constant, thus only affecting the depth of the calculations on the GPU. On the other hand, in the real space calculation, for twice the data, we have a thread parallelism increase of 4 fold, which means 4 times more computation and a more efficient utilization of the GPU computational power.

The speedups for the reciprocal space calculation vary between 14× and 22×, depending on the system. Test calculations have also been run with varying Ewald fractions on the $(H_2O)_{800}$ system. Variations between 0.2 and 0.5 lead to speedups of 10.1× and 12.4×, respectively. We found it in general to be rather robust, even with such large variations in the number of *h* vectors. The speedups for the full induced dipole calculation (Figure 5) vary between 10× and 16×. This is a sizable decrease in the computational timings. However, it may fall a bit below expectations. According to Amdahl's law and in order to further improve the application, we mostly consider sections of the code which have been left to CPU serial execution. The calculation of the prefactors, as shown previously, has become a major contribution to the total computing time. Therefore, the first step to be taken would be to reduce or even remove the time overheads due to the prefactor calculation. This improvement would allow us not only to optimize the CPU time but also to reduce the transfers between the CPU and GPU, thereby reducing the memory pressure accelerating the execution on the GPU side as well.

**Accuracy.** In this section, we analyze the effects of reducing the GPU kernels to single precision. The first problem to consider is the fact that we introduce single precision code into an iterative self-consistent procedure. Such a change could lead to poorer convergence. The regular criterium requires that the RMSD between adjacent cycles to be as low as $1.0 \times 10^{-6}$ D. The second point to take into account would be the value of the induced dipole itself, which is obtained at the end of the cycle. Finally, there is the question of how the use of single precision arithmetics affects the gradient. One should take into account that we have implemented GPU acceleration solely in the dipole cycle, and all other parts of the program are still using double precision arithmetics. This means that any changes to the gradient are exclusively due to a change in the value of the induced dipoles.

In order to assess these questions, we have carried out short NVT simulations (at 298 K) on two of the model systems used in the previous section. We consider the simulation boxes with 800 water molecules and the histidine plus ions in solution. A time step of 1 fs was used, with a regular Ewald sum for electrostatics. After a short equilibration run of 10 ps, the systems were sampled in 1 ps intervals.

The main criteria for the accuracy in a molecular forcefield implementation is the gradient calculation. In order to evaluate the effect of single precision arithmetics, we have computed the gradient on each of the snapshots, determining the relative error in the total gradient vector $\Delta_{tot} = |\vec{F}^D_{tot} - \vec{F}^S_{tot}|/|\vec{F}^D_{tot}|$, and in the gradient vector terms connected to polarization $\Delta_{pol} = |\vec{F}^D_{pol} - \vec{F}^S_{pol}|/|\vec{F}^D_{pol}|$. The results are plotted in Figure 6.

It is visible in the figure that the errors are quite similar in both systems, even considering the added ions in the case of the histidine system. The relative deviation in the polarization gradient vector are kept below $10^{-6}$, which clearly reflects the small errors introduced by the single precision code in the induced dipole cycle. This is also well below the common accepted maximum relative error of $10^{-3}$ for biomolecular simulations.[37] The individual gradient components show very little deviations. In the case of the water simulation, for example, we found a maximum error of $4.2 \times 10^{-5}$ kcal/mol/Å (by comparing all Cartesian components of each snapshot). The second point to note is that we have not observed any degradation in the cycle convergence. In all calculations, the same number of iterations was observed. Single precision is


dx.doi.org/10.1021/ci200564x | J. Chem. Inf. Model. 2012, 52, 1159–1166

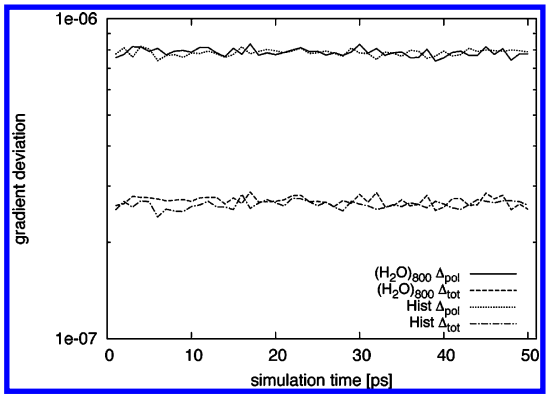



**Figure 6.** Relative deviations in the total gradient and induced polarization gradient terms. Values are given for snapshots of the $(H_2O)_{800}$ simulation box and the solvated histidine system.

sufficient to reach the default threshold of $1.0 \times 10^{-6}$ D in the RMSD. Observing the errors in the graph, one can conclude that the effect of using lower precision arithmetics affects the results below the accepted convergence parameters. The induced dipole cycles run without any degradation in performance up to a RMSD criteria of $1.0 \times 10^{-7}$ D. Below this value, and depending on the system/snapshot, the cycles will start to fail in convergence due to the numerical noise.

As noted before, all the errors discussed in this section are due to the use of single precision in the induced dipole calculation. The gradients were still computed with double precision arithmetics. The strict use of single precision, including the gradient evaluation, would be ill-advised. Just as in the case of nonpolarizable force fields, it should lead to energy drifts and the loss of energy conservation.[38]

## CONCLUSIONS

In this paper, we have presented a partial GPU implementation for the iterative computation of induced dipoles in molecular mechanics simulations. The model used corresponds to the AMOEBA potential and is interfaced to the Tinker program package in a hybrid computational platform composed by the CPU and the GPU, which acts as an accelerator connected through the PCIe bus. The hybrid platform is exploited in a very efficient way by overlapping computation with communication in a pipeline approach.

The calculations on model systems performed on the CPU−GPU hybrid platform show that significant speedups can be achieved. In the case of nonperiodic simulations, this can be up to 23×, regarding CPU only implementation and single precision CPU−GPU implementation. The obtained speedup for the double precision implementation is only 30% lower, which is mainly justified by the fact that the execution of the kernels on the GPU has become so fast that the remaining steps (e.g., the scaling prefactors calculation in the CPU) becomes dominant in the total required time. This leaves room for further improvements, as our preliminary results show that the use of single precision code has a minimal effect on the computed dipoles. We have also mainly focused on the fine-grained parallelization of the algorithm. Therefore, we have restricted ourselves to the implementation details for a single GPU multiprocessor architecture. For very large molecular systems where the memory size of the GPU may become a limitation, extra speedup gains could be achieved by splitting the data and work through more than one GPU processor. This could be easily implemented because the only data dependencies are at the reduction stage. In that case, we would only require to break the sum over sites throughout the different GPUs and add an extra reduction step on the CPU side to merge the partial results of each GPU. Modern platforms allow us to connect several GPU devices to the same host CPU and communicate with them in parallel. Therefore, the only extra time necessary would be the new accumulation step performed on the CPU side.

The code has been only partially implemented for use with GPU accelerators. Energy and gradient evaluations, on the basis of the converged dipoles, have not yet been considered. However, the rationale is similar to the work presented here and previous works on classical MD simulations.[18] A preliminary version for nonperiodic simulations is already available in our lab but has been left out of the discussion. We would like to emphasize the fact that the code section that has been presented still relies on the CPU, overlapping computations and data transfer with the kernel calls, which we believe to be the most sensible option even when considering the computation of forces. Contrary to regular force fields, which can be effectively run almost exclusively on the GPU,[24] the amount of data required to work with the AMOEBA model makes such an approach inviable. The advantages of using a hybrid solution have been highlighted in the most recent developments of GPU-accelerated MD code, such as in the case of the GROMACS program package.[39] The code presented will be made freely available following an update to the latest Tinker program version.

## AUTHOR INFORMATION

### Corresponding Author

*E-mail: rmata@gwdg.de.

### Notes

The authors declare no competing financial interest.

## ACKNOWLEDGMENTS

This work was supported by FCT (Fundação para a Ciência e Tecnologia) through the PIDDAC Program funds (INESC-ID multi annual funding). R.A.M. and J.M.D. gratefully acknowledge financial support from the German Excellence Initiative, through the Free Floater Research Group program of the University of Göttingen.

## REFERENCES

(1) Cornell, W. D.; Cieplak, P.; Bayly, C. I.; Gould, I. R.; Merz, K. M., Jr.; Ferguson, D. M.; Spellmeyer, D. C.; Fox, T.; Caldwell, J. W.; Kollman, P. A. A second generation force field for the simulation of proteins and nucleic acids. *J. Am. Chem. Soc.* **2003**, *117*, 5179−5197.

(2) Jorgensen, W. L.; Tirado-Rives, J. The OPLS potential functions for proteins, energy minimizations for crystals of cyclic peptides and crambin. *J. Am. Chem. Soc.* **1988**, *110*, 1657−1666.

(3) MacKerell, A. D., Jr.; Bashford, D.; Bellott, M.; Dunbrack, R. L., Jr.; Evanseck, J. D.; Field, M. J.; Fischer, S.; Gao, J.; Guo, H.; Ha, S.; Joseph-McCarthy, D.; Kuchnir, L.; Kuczera, K.; Lau, F. T. K.; Mattos, C.; et al. All-atom empirical potential for molecular modeling and dynamics studies of proteins. *J. Phys. Chem. B* **1998**, *102*, 3586−3616.

(4) Cieplak, P.; Dupradeau, F.-Y.; Duan, Y.; Wang, J. Polarization effects in molecular mechanical force fields. *J. Phys.: Condens. Matter* **2009**, *21*, 333102.

(5) Thole, B. T. Molecular polarizabilities calculated with a modified dipole interaction. *Chem. Phys.* **1981**, *59*, 341−350.

(6) Saint-Martin, H.; Hernandez-Cobos, J.; Bernal-Uruchurtu, M. I.; Ortega-Blake, I.; Berendsen, H. J. A mobile charge densities in

harmonic oscillators (MCDHO) molecular model for numerical simulations: The water−water interaction. *J. Chem. Phys.* **2000**, *113*, 10899.
(7) Donchev, A. G.; Ozrin, V. D.; Subbotin, M. V.; Tarasov, O. V.; Tarasov, V. I. A quantum mechanical polarizable force field for biomolecular interactions. *Proc. Natl. Acad. Sci. U.S.A.* **2005**, *102*, 7829−7834.
(8) Åstrand, P.-O.; Linse, P.; Karlström, G. Molecular dynamics study of water adopting a potential function with explicit atomic dipole moments and anisotropic polarizabilities. *Chem. Phys.* **1995**, *191*, 195−202.
(9) Yu, H.; van Gunsteren, W. F. Charge-on-spring polarizable water models revisited: From water clusters to liquid water to ice. *J. Chem. Phys.* **2004**, *121*, 9549.
(10) Kaminski, G. A.; Stern, H. A.; Berne, B. J.; Friesner, R. A. Development of an accurate and robust polarizable molecular mechanics force field from ab initio quantum chemistry. *J. Phys. Chem. A* **2004**, *108*, 621−627.
(11) Patel, S.; Brooks, C. L., III CHARMM fluctuating charge force field for proteins: I. Parameterization and application to bulk organic liquid simulations. *J. Comput. Chem.* **2004**, *25*, 1−15.
(12) Harder, E.; Kim, B. C.; Friesner, R. A.; Berne, B. J. Efficient simulation method for polarizable protein force fields: Application to the simulation of BPTI in liquid water. *J. Chem. Theory Comput.* **2005**, *1*, 169−180.
(13) Lamoureux, G.; Harder, E.; Vorobyov, I. V.; Roux, B.; Mackerell, A. D., Jr. A polarizable model of water for molecular dynamics simulations of biomolecules. *Chem. Phys. Lett.* **2006**, *418*, 245−249.
(14) Wang, Z. X.; Zhang, W.; Wu, C.; Lei, H.; Cieplak, P.; Duan, Y. Strike a balance: Optimization of backbone torsion parameters of AMBER polarizable force field for simulations of proteins and peptides. *J. Comput. Chem.* **2006**, *27*, 781−790.
(15) Gresh, N.; Cisneros, G. A.; Darden, T. A.; Piquemal, J.-P. Anisotropic, polarizable molecular mechanics studies of inter- and intramolecular interactions and ligand-macromolecule complexes. A bottom-up strategy. *J. Chem. Theory Comput.* **2007**, *3*, 1960−1986.
(16) Ren, P.; Ponder, J. W. Consistent treatment of inter- and intramolecular polarization in molecular mechanics calculations. *J. Comput. Chem.* **2002**, *23*, 1497−1506.
(17) Ponder, J. W.; Wu, C.; Ren, P.; Pande, V. S.; Chodera, J. D.; Schnieders, M. J.; Haque, I.; Mobley, D. L.; Lambrecht, D. S.; DiStasio, R. A., Jr.; Head-Gordon, M.; Johnson, M. E.; Head-Gordon, T. Current status of the AMOEBA polarizable force field. *J. Phys. Chem. B* **2010**, *114*, 2549−2564.
(18) Stone, J. E.; Phillips, J. C.; Freddolino, P. L.; Hardy, D. J.; Trabuco, L. G.; Schulten, K. Accelerating molecular modeling applications with graphics processors. *J. Comput. Chem.* **2007**, *28*, 2618−2640.
(19) Friederichs, M. S.; Eastman, P.; Vaidyanathan, V.; Houston, M.; Legrand, S.; Beberg, A. L.; Ensign, D. L.; Bruns, C. M.; Pande, V. S. Accelerating molecular dynamic simulation on graphics processing units. *J. Comput. Chem.* **2009**, *30*, 864−872.
(20) Harvey, M. J.; Fabritiis, G. D. An implementation of the smooth particle mesh Ewald method on GPU hardware. *J. Chem. Theory Comput.* **2009**, *5*, 2371−2377.
(21) Anderson, J. A.; Lorenz, C. D.; Travesset, A. General purpose molecular dynamics simulations fully implemented on graphics processing units. *J. Comput. Phys.* **2008**, *227*, 5342−5359.
(22) Meel, J. A. V.; Arnold, A.; Frenkel, D.; Portegies, S. F.; Belleman, R. G. Harvesting graphics power for MD simulations. *Mol. Simul.* **2008**, *34*, 259−266.
(23) Liu, W.; Schmidt, B.; Voss, G.; Müller-Wittig, W. Accelerating molecular dynamics simulations using graphics processing units with CUDA. *Comput. Phys. Commun.* **2008**, *179*, 634−641.
(24) Stone, J. E.; Hardy, D. J.; Ufimtsev, I. S.; Schulten, K. GPU-accelerated molecular modeling coming of age. *J. Mol. Graphics Modell.* **2010**, *29*, 116−125.
(25) Brown, W. M.; Wang, P.; Plimpton, S. J.; Tharrington, A. N. Implementing molecular dynamics on hybrid high performance computers—Short range forces. *Comput. Phys. Commun.* **2011**, *182*, 898−911.
(26) Eastman, P.; Pande, V. J. Efficient nonbonded interactions for molecular dynamics on a graphics processing unit. *J. Comput. Chem.* **2010**, *31*, 1268−1272.
(27) AMBER 11 NVIDIA GPU Acceleration Support. http://ambermd.org/gpus (accessed April 13, 2012).
(28) Jiang, W.; Hardy, D. J.; Phillips, J. C.; MacKerell, A. D., Jr.; Schulten, K.; Roux, B. High-performance scalable molecular dynamics simulations of a polarizable force field based on classical drude oscillators in NAMD. *J. Phys. Chem. Lett.* **2011**, *2*, 87−92.
(29) Luebke, D.; Harris, M.; Krüger, J.; Purcell, T.; Govindaraju, N.; Buck, I.; Woolley, C.; Lefohn, A. GPGPU: General Purpose Computation on Graphics Hardware. *ACM SIGGRAPH 2004 Course Notes*; New York, 2004; p 33.
(30) Ponder, J. W.; Case, D. A. Force fields for protein simulations. *Adv. Protein Chem.* **2003**, *66*, 27−85.
(31) Ren, P.; Ponder, J. W. Polarizable atomic multipole water model for molecular mechanics simulation. *J. Phys. Chem. B* **2003**, *107*, 5933−5947.
(32) Aguado, A.; Madden, P. A. Ewald summation of electrostatic multipole interactions up to the quadrupolar level. *J. Chem. Phys.* **2003**, *119*, 7471−7483.
(33) Darden, T.; York, D.; Pederson, L. Particle mesh Ewald: An N.log(N) method for Ewald sums in large systems. *J. Chem. Phys.* **1993**, *98*, 10089.
(34) Ponder, J. W. *TINKER: Software Tools for Molecular Design*; 4.2 ed.; Washington University School of Medicine: Saint Louis, MO, 2003.
(35) Lindholm, E.; Nickolls, J.; Oberman, S.; Montrym, J. NVIDIA Tesla: A unified graphics and computing architecture. *Micro, IEEE* **2008**, *28*, 39−55.
(36) Pratas, F.; Mata, R. A.; Sousa, L. *Iterative Induced Dipoles Computation for Molecular Mechanics on GPUs*, Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, Pittsburgh, PA, 2010; pp 111−120.
(37) Bowers, K. J.; Chow, E.; Xu, H.; Dror, R. O.; Eastwood, M. P.; Gregersen, B. A.; Klepeis, J. L.; Kolossvary, I.; Moraes, M. A.; Sacerdoti, F. D.; Salmon, J. K.; Shan, Y.; Shaw, D. E. *Scalable Algorithms for Molecular Dynamics Simulations on Commodity Clusters*, Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, Tampa, FL, 2006; DOI: 10.1145/1188455.1188544.
(38) Ruymgaart, A. P.; Cardenas, A. E.; Elber, R. MOIL-opt: Energy-conserving molecular dynamics on a GPU/CPU system. *J. Chem. Theory Comput.* **2011**, *7*, 3072−3082.
(39) Hess, B.; Kutzner, C.; van der Spoel, D.; Lindahl, E. GROMACS 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation. *J. Chem. Theory Comput.* **2008**, *4*, 435−447.