Revisiting Molecular Dynamics on a CPU/GPU System: Water Kernel and SHAKE Parallelization

A. Peter Ruymgaart and Ron Elber*

Department of Chemistry and Biochemistry, Institute for Computational Engineering and Sciences, University of Texas at Austin, Austin, Texas 78712, United States

S *Supporting Information*

ABSTRACT: We report graphics processing unit (GPU) and Open-MP parallel implementations of water-specific force calculations and of bond constraints for use in molecular dynamics simulations. We focus on a typical laboratory computing environment in which a CPU with a few cores is attached to a GPU. We discuss in detail the design of the code, and we illustrate performance comparable to highly optimized codes such as GROMACS. Besides speed, our code shows excellent energy conservation. Utilization of water-specific lists allows the efficient calculations of nonbonded interactions that include water molecules and results in a speed-up factor of more than 40 on the GPU compared to code optimized on a single CPU core for systems larger than 20 000 atoms. This is up 4-fold from a factor of 10 reported in our initial GPU implementation that did not include a water-specific code. Another optimization is the implementation of constrained dynamics entirely on the GPU. The routine, which enforces constraints of all bonds, runs in parallel on multiple Open-MP cores or entirely on the GPU. It is based on the conjugate gradient solution of the Lagrange multipliers (CG SHAKE). The GPU implementation is partially in double precision and requires no communication with the CPU during the execution of the SHAKE algorithm. The (parallel) implementation of SHAKE allows an increase of the time step to 2.0 fs while maintaining excellent energy conservation. Interestingly, CG SHAKE is faster than the usual bond relaxation algorithm even on a single core if high accuracy is expected. The significant speedup of the optimized components transfers the computational bottleneck of the MD calculation to the reciprocal part of particle mesh Ewald (PME).

INTRODUCTION

Extending time scales of molecular dynamics (MD) simulations makes it possible to address more complex molecular problems than was possible before. One approach to this problem utilizes advances in hardware to obtain faster propagation of the trajectory in time. Increases in computational speed of molecular dynamics algorithms have been obtained by utilizing multiple CPUs and/or specialized hardware such as graphics-processing units (GPU).[1−5] Parallelization is the dominant way nowadays to speed up MD using modern computer hardware. Speed-up is achieved while retaining the overall accuracy of the calculations. As simulations become more complex and are more expensive and lengthy to conduct, it becomes important to be able to assess their quality on the fly and avoid costly mistakes. One of the important measures of correctness is the conservation of the total energy (or other conserved entities). Moreover, if the focus of the calculations is on explicit dynamics and time scales, then the NVE ensemble (N number of particles, V volume, and E energy) is the most natural choice. To sample from this ensemble correctly and to accurately compute the dynamics, the energy must be conserved. This requirement is significant and not trivial to satisfy even for relatively short trajectories of only a few nanoseconds. If energy conservation is not adequately addressed, the phenomenon of energy drift (energy change linear in time) is observed. Under these conditions, even if some physical observables are reasonably well behaved, or if the energy is adjusted empirically, it is difficult to accept the results without significant reservations. One of the focuses of our code MOIL[5,6] is on the calculations of kinetics with the method of milestoning.[7−11] In milestoning, we launch a large number of short trajectories and use their statistics to estimate rates at long times. To ensure the correctness of our short trajectories, we insist on relative energy conservation ($|\Delta E/E|$), which is at most 10^{-2} per microsecond for the standard benchmark of DHFR.[5]

At present, our code is aimed at individual laboratories and typical laboratory hardware. Since our applications focus on launching a very large number of short trajectories, it makes sense for us to optimize the code of one trajectory for one node. We optimized our code for a CPU with four cores and a GPU. With this system size, we can easily and efficiently study systems of about 100 000 atoms. Simulations of systems larger than this size will require further adjustments of the code to overcome current memory limitations.

We chose to split the computations between the GPU and the CPU since we wanted a stable programming model (to avoid rewriting a significant portion of the code when new GPU hardware is introduced) and since we seek a programming platform that supports reasonably rapid code development. The GPU provides massively parallel computations for simple data and instructions and can be faster for specific tasks by a factor of several tens compared to a single core. However, it is not always the best choice, and consideration of memory and accuracy is important. The GPU is most efficient if single precision is used. However, single precision accuracy is not always sufficient. While recent GPU versions support double precision computations,

Received: April 24, 2012
Published: August 21, 2012

there is a cost using this option (e.g., the variables may not fit in a cache or available registers). The evolving support level of GPU makes it difficult to retain a stable programming model, and extensive code changes may be required for new hardware variants, something that we wish to avoid to the extent possible. Having an option of computing in double precision when deemed necessary on the CPU is therefore an advantage. Moreover, the GPU is significantly more difficult to code, and for a program that we routinely add and change, programming on the GPU is costly in human and time resources. Complex code is better developed on the CPU. Finally, it is likely that exploitation of heterogeneous computing is possible for the CPU/GPU pair in which some of the calculations are done on the CPU and some on the GPU to obtain greater efficiency. Ignoring this possibility (by insisting on doing everything on the GPU) is limiting. We demonstrate the possibility of asynchronous computing in the paper. We show a significant asynchronous speedup for SHAKE by SHAKing the water molecules on the CPU and the macromolecule on the GPU.

We believe the future of single node computing in general is heterogeneous. The AMD Fusion and Intel Sandybridge are already commonplace with AMD offering training for heterogeneous programming on its Fusion chips. Now also, NVIDIA joins the heterogeneous CPU−GPU market with "Project Denver". We also think memory transfer between the CPU and GPU will increase in performance in future systems. Calculations that are complex (require many registers), are few in number, and involve numerous atoms per calculation are best suited for the CPU, just like operations with many branch possibilities. PME, torsion, and improper torsion calculations fall in this category and can be performed on CPU cores at the same time as the nonbonded calculations on the GPU. We believe in using them both, and using them both at the same time. Let the total computational cost be T, and in our programming model a fraction $1 - \alpha$ is computed on the GPU and a fraction α on the CPU. The compute time on the GPU is $T(1 - \alpha)/G$ and on the CPU $T\alpha/C$ where the computational speeds of the CPU and the GPU are denoted C and G, respectively. We also consider an overhead o of the communication between the CPU and the GPU. If we use just the GPU, the calculation time is T/G. If we use both, the new time is $t = \max[T(1 - \alpha)/G, T\alpha/C] + o$. The parameter α is a subject of optimization. Can we indeed argue that the optimal α is always 1 or 0? For reasonably small o and diverse computing tasks, this is not the case.

The only argument supporting GPU-only computing is that the data transfer o is expensive. Indeed in the case of a simple Lennard-Jones fluid GPU-only simulation is sound. In this case, memory transfer between host and device (the only pitfall of heterogeneous computation) is costly if done at each step. However, in the far more complex simulations of explicitly solvated biopolymers, there are components that are not suited for massive parallelization with single precision. In particular, we illustrate in our first paper that particle meshed Ewald (PME) calculated in single precision increases the energy drift by a factor of about 10. Therefore, single precision PME should not be used in quantitative simulations. There are also terms such as torsions and improper torsions that run with similar efficiency on both the CPU and the GPU. The latter two require significant local resources (registers) and at the same time are relatively small in number, reducing benefits from parallelism. Hence, these arguments and experimentations led to our choice of using heterogeneous computing in a laboratory setting to speed up molecular simulations.

An increase in the size of the integration time-step is another way to enhance the efficiency of the calculation. Enlarging the time step (while retaining numerical accuracy) can be achieved by removing fast degrees of freedom from the system. A widely used example is the SHAKE algorithm,[12] which constrains bond lengths to their ideal values. Another approach is based on filtering high frequency modes.[13] Removal of bond vibrations, which are fast degrees of freedom, allows doubling the step of integration while retaining numerical precision. However, a combination of parallel architecture and the popular bond-relaxation SHAKE was proven hard to achieve. This is since bond relaxation is inherently serial and requires many iterations to adjust coordinate positions. An alternative SHAKE formulation is based on a direct and iterative solution to a set of approximate linear equations for the Lagrange's multipliers. This alternative is easier to parallelize and requires significantly fewer iterations of coordinate adjustments.[14−16] In the present manuscript, we consider a novel implementation of the matrix formulation of SHAKE in which the entire algorithm is carried out on the GPU with some components in double precision. The Lagrange multipliers are found using the conjugate gradient algorithm[14] in single precision, while the coordinate adjustments are conducted in double precision. This implementation brings more work to the GPU, freeing the CPU to asynchronously conduct other computations. Our implementation on the GPU of parallel matrix SHAKE is more efficient than the use of the CPU only in cases where the number of CPU cores is low compared to the system size. We use a typical laboratory tool of a CPU with four cores and a single GTX480. On a node with 12 cores and the same GPU, the constraints are better computed on the CPU. We therefore retain the option to run the same algorithm on multiple shared memory CPUs as defined by user input, illustrating the flexibility of heterogeneous computing.

In addition to the improvement in SHAKE parallelization, we also consider a special GPU routine (kernel) to handle nonbonded interactions between water molecules. We exploit the fact that most biological macromolecules simulated with an explicit solvent contain a large number of water molecules. Since all water−water interactions share Lennard-Jones and electrostatic parameters, we can significantly reduce memory access by keeping these parameters in fast memory. Moreover, since we use neighbor lists, making a special case for water, one stores a water index rather than three atomic indices, thereby reducing list size up to 3-fold. We recently reported a GPU implementation of the nonbonded force calculation using atomic neighbor lists.[5] Here, we build on this code and present a specialized GPU implementation of water nonbonded force calculations.

These two additions led to a speedup compared to our previous code (on the standard test case of DHFR) by about 3-fold overall and 4-fold in the GPU nonbonded force calculation. For each GPU implementation, an equivalent optimized function was implemented for execution on a multicore shared memory CPU (using the parallel library Open Multi Processing (OpenMP) in case a GPU is not available).

■ MATRIX SHAKE

In order to increase the time step while maintaining energy conservation, we remove bond displacements that are fast degrees of freedom from the equations of motion. It is easy to parallelize the constraints of only the fastest bonds that include hydrogen atoms since the number of coupled bonds is always small and decoupled. Here, we focus on the more general problem of parallelization of a network of bonds that is not factored to independent bond clusters. Examples are long peptide chains. Matrix SHAKE with conjugate gradient optimization to determine the Lagrange multipliers is an appropriate methodology to handle the above problem and was described in ref 14. We briefly review it and discuss specific adjustments appropriate for the heterogeneous computing system of the present manuscript. Consider the algorithm of velocity Verlet[17] with constraints (bold font denotes a vector):

$$\mathbf{x}(t+\tau) = \mathbf{x}_{i+1} = \mathbf{x}_i(t) + \mathbf{v}_i(t)\tau - \frac{1}{2}\mathbf{M}^{-1}[\nabla U(\mathbf{x}_i) + \sum_a \eta_a \nabla\sigma_a(\mathbf{x}_i)]\tau^2 \tag{1a}$$

$$\sigma_a = (\mathbf{x}_{a1} - \mathbf{x}_{a2})^2 - r^2_{\mathrm{eq},a} \tag{1b}$$

where τ is the time step and $\mathbf{x},\mathbf{v} \in \mathbf{R}^{3N}$ are coordinate and velocity vectors of the whole system. $\mathbf{x}_{a1}$ and $\mathbf{x}_{a2}$ are the three-dimensional coordinates of the two bonded atoms, and σ_a is the constraint on bond a; i.e. the distance square between the two points $\mathbf{x}_{a1}$ and $\mathbf{x}_{a2}$ is constrained to be equal to the square of the equilibrium distance ${r_{\mathrm{eq},a}}^2$. The Lagrange multipliers are η_a. $\mathbf{M}$ is a $3N \times 3N$ diagonal matrix where N is the number of atoms. The mass of a particle repeats three times in the matrix for the x, y, and z directions. For a corresponding equation for the velocities, see ref 14. We split eq 1a into a step without the application of the constraints; a Verlet step, which we call $\mathbf{x}^{(0)}$; and a correction when the constraints are taken into account:

$$\mathbf{x}^{(0)} = \mathbf{x}_i(t) + \mathbf{v}(t)\tau - \frac{1}{2}\mathbf{M}^{-1}\nabla U(\mathbf{x}_i(t))\tau^2 \tag{2a}$$

$$\Delta\mathbf{x}_{\mathrm{constr}} = 1/2\,\mathbf{M}^{-1}\sum_b \eta_b \nabla\sigma_b(\mathbf{x}_i)\tau^2 \tag{2b}$$

$\Delta\mathbf{x}_{\mathrm{constr}}$ is the adjustment to the coordinate enforced by the constraints such that $\mathbf{x}_{i+1} = \mathbf{x}^{(0)} - \Delta\mathbf{x}_{\mathrm{constr}}$. The constraints at step $i+1$ are approximated by a first order Taylor expansion near the coordinates $\mathbf{x}^{(0)}$ which are now adjusted by $\Delta\,\mathbf{x}_{\mathrm{constr}}$ to satisfy the constraint condition $\sigma(\mathbf{x}_{i+1}) = \mathbf{0}$:

$$\mathbf{0} = \sigma_a(\mathbf{x}_{i+1}) \approx \sigma_a(\mathbf{x}^{(0)}) - \nabla\sigma_a(\mathbf{x}^{(0)})\Delta\mathbf{x}_{\mathrm{constr}} \quad \forall\ a \tag{3a}$$

$$\sigma_a(\mathbf{x}^{(0)}) \approx \nabla\sigma_a(\mathbf{x}^{(0)})\Delta\mathbf{x}_{\mathrm{constr}} \tag{3b}$$

We can write the right-hand side of eq 3b $\nabla\sigma(\mathbf{x}^{(0)})\Delta\mathbf{x}_{\mathrm{constr}}$ in matrix form using eq 2b while factoring out the Lagrange multipliers and the time step:

$$A_{ab} = [\nabla\sigma_a(\mathbf{x}^{(0)})]^t \frac{1}{2}\mathbf{M}^{-1}\nabla\sigma_b(\mathbf{x}_i) \tag{4}$$

This matrix is asymmetric because $\sigma_a(\mathbf{x}^{(0)})$ is not equal to $\sigma_a(\mathbf{x}_i)$. We can use the matrix $\mathbf{A}$ to estimate the Lagrange multipliers $\lambda_b = \eta_b\tau^2$ of eq 2b using eqs 3 and 4:

$$\mathbf{A}^{-1}\sigma = \lambda \tag{5}$$

The solution of this system is approximate because σ is not linear in λ. The linear approximation made at eq 3a requires a Newton-like iteration until convergence. It converges provided that the initial point is sufficiently close to the right answer. Convergence is assumed when

$$|\sigma_a(\mathbf{x}_i^{(n)})| < \varepsilon\ \forall\ a \tag{6}$$

where ε is a small number. The convergence is required for all constraints. For every iteration step, we have

$$\mathbf{x}_{i+1}^{(n+1)} = \mathbf{x}_{i+1}^{(n)} - \frac{1}{2}\mathbf{M}^{-1}\sum_b \lambda_b^{(n)}\nabla\sigma_b(\mathbf{x}_i) \tag{7}$$

We define a corresponding matrix

$$\mathbf{A}_{ab}^{(n+1)} = \nabla\sigma_a(\mathbf{x}^{(n+1)})\frac{1}{2}\mathbf{M}^{-1}\nabla\sigma_b(\mathbf{x}_i) \tag{8}$$

And we also have

$$\sigma^{(n+1)} = \mathbf{A}^{(n+1)}\lambda^{(n+1)} \tag{9}$$

It is easier to understand the computational aspects of the linear equation (eq 9) for the constraints when the explicit expression for the bond constraints is used. Consider eq 1b: taking a spatial derivative, we have $\nabla\sigma_a = 2(\mathbf{x}_{a1} - \mathbf{x}_{a2})$, and a matrix element is therefore

$$\mathbf{A}_{ab}^{(n+1)} = 4(\mathbf{x}_{a1}^{(n+1)} - \mathbf{x}_{a2}^{(n+1)})^t\frac{1}{2}\mathbf{M}^{-1}(\mathbf{x}_{b1} - \mathbf{x}_{b2}) \tag{10}$$

Discussions on the properties of this matrix were given in the original paper of SHAKE[12] and in refs 14 and 18. We consider three cases: (i) The constraints "a" and "b" are identical. In this case we have $\mathbf{A}_{aa}^{(n+1)} = 2(1/m_{a1} + 1/m_{a2})(\mathbf{x}_{a1}^{(n+1)} - \mathbf{x}_{a2}^{(n+1)})^t(\mathbf{x}_{a1} - \mathbf{x}_{a2})$. (ii) The constraints "$a$" and "$b$" are not the same, but they share one atom (e.g., $a2 = b1$); we have $\mathbf{A}_{ab}^{(n+1)} = 2(\mathbf{x}_{a1}^{(n+1)} - \mathbf{x}_{a2}^{(n+1)})^t(1/m_{a2})(\mathbf{x}_{a2} - \mathbf{x}_{b2})$. (iii) No atoms are shared, $\mathbf{A}_{ab} = 0$.

From the explicit expression (eq 10) and the cases discussed above, it is clear that the matrix is sparse (bonds are coupled only if at least one atom is shared) and is recomputed every λ iteration. One alternative to the calculation of the matrix in the linear equation for the Lagrange multipliers is to approximate $\mathbf{A}_{ab}^{(n+1)}$ by its diagonal, which is the bond relaxation algorithm.[6] The disadvantage of the bond-relaxation algorithm is that it requires many iterations; iterations that must be executed serially. Bond-relaxation is therefore not effective for parallelization and for GPU implementation. From now onward, we consider only options that employ nondiagonal matrices. Besides using the exact matrix, we consider two approximate matrices: (a) a symmetric matrix and (b) a constant matrix (Figure 1).

Consider first the symmetric matrix option, we write:

$$A_{ab}^{s} = \nabla\sigma_a(\mathbf{x}_i)\frac{1}{2}\mathbf{M}^{-1}\nabla\sigma_b(\mathbf{x}_i) = 4(\mathbf{x}_{a1} - \mathbf{x}_{a2})^t\frac{1}{2}\mathbf{M}^{-1}(\mathbf{x}_{b1} - \mathbf{x}_{b2}) \tag{11}$$

Since $\mathbf{x}_{a1}$ is different from $\mathbf{x}_{a1}^{(n+1)}$ only by terms of order of τ^2 (the small time step), the difference between the exact and the symmetric matrix is small. The iterations of eq 9 converge with both matrices from eq 10 or eq 11 for sufficiently small τ. The number of iterations is however different. In practice, a smaller number of iterations is required to converge the values of the Lagrange multipliers using the asymmetric matrix.

Another option is to use a constant matrix that does not depend on the coordinate at all. Hence, a single matrix can be used throughout the simulations. The constant matrix was proposed in ref 14, and an implementation was discussed in ref 19. The basic idea is that the deviation of the constrained bond from ideality is expected to be small, so when constructing the symmetric matrix we have for nonzero elements

$$A_{aa}^{s} = 2(1/m_{a1} + 1/m_{a2}){r_{\mathrm{eq},a}}^2 \qquad A_{ab}^{s} = 2(1/m_{a2})r_{\mathrm{eq},a}r_{\mathrm{eq},b}\cos(\theta_{ab}) \tag{12}$$

where a and b bonds are sharing an atom ($a2 = b1$), and θ is the angle between the two bonds. For the protocol to be exact, the

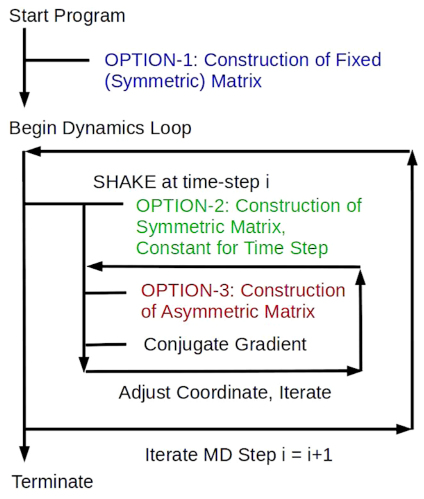
Start Program
OPTION-1: Construction of Fixed (Symmetric) Matrix
Begin Dynamics Loop
SHAKE at time-step i
OPTION-2: Construction of Symmetric Matrix, Constant for Time Step
OPTION-3: Construction of Asymmetric Matrix
Conjugate Gradient
Adjust Coordinate, Iterate
Iterate MD Step i = i+1
Terminate

Figure 1. An illustration of the different options in matrix SHAKE. Option-1 computes a fixed matrix that does not change throughout the calculations. It implies that the bond angles are exactly fixed[14] or approximately so.[19] The advantage is that the fixed matrix is computed only once throughout the simulation. The disadvantage is that the fixed matrix is approximate, and a relatively large number of iterations are required to determine the Lagrange multipliers accurately. We determine the Lagrange multipliers using a conjugate gradient (CG) algorithm. Building on the fixed matrix of ref 14, an implementation that does not constrain the angles was proposed.[19] This variation is likely to require an even larger number of iterations to converge to an accurate solution. In general, it is not possible to retain high accuracy and to avoid iterative solutions of the Lagrange multipliers (and coordinates) with the exception of very small systems for which analytical solutions are available (e.g., SETTLE[20]). In the second option, a symmetric matrix is constructed every time step. Since the matrix is sparse, the cost of generating it each time step is small. The reduction in the number of CG iterations to achieve high accuracy is significant. This results in a more efficient algorithm. The third option of generating an asymmetric matrix every CG step is not considered here since CG cannot use an asymmetric matrix without costly symmeterization, for example by using $\mathbf{A}^t\mathbf{A}$ instead of $\mathbf{A}$.

angle must be constrained as well. We can enforce it by adding an additional bond constraint between $a1$ and $b2$ atoms of eq 11.[14] Alternatively we can pick the equilibrium, constant angle ($\theta_{ab,eq}$) and plug it into eq 12 to compute an approximate but time independent constraint matrix.[19] Since bond angles are usually restrained with a stiff harmonic potential, their fluctuations are small, suggesting that the approximation of a constant matrix is sound (even though it is no longer possible to argue that the solution is accurate with a particular order of τ). A significant advantage is that the matrix is fixed and requires computation only at the beginning of the calculations. The disadvantage (similar to the discussion about the symmetric matrix) is that the number of iterations to convergence that must be conducted serially may be significantly larger and more costly than the number of iterations for matrices that are updated during the calculation. Below, we examine the options of symmetric and constant matrices.

Another complication that we need to consider is that we need to find the inverse of $\mathbf{A}$ to determine the Lagrange multipliers (eq 9). Calculation of an inverse is usually significantly more expensive than the construction of the matrix itself. How to estimate the Lagrange multipliers with explicit calculation of the inverse therefore attracted some attention. In LINCS,[21] an approximate inverse is computed. Here, we follow the algorithm outlined in ref 14 in which a conjugate gradient algorithm (CG) estimates the Lagrange multipliers. For a symmetric and non-negative matrix $\mathbf{A}$, we seek the minimum of a target function T as a function of the Lagrange multiplier λ

$$T = -\sigma^t\lambda + (1/2)\lambda^t\mathbf{A}\lambda \tag{13}$$

Note that since we are restricted to symmetric and non-negative matrices, we cannot use the exact matrix in eq 13 "as is," and we therefore use the symmetric versions. The symmetric matrix is indeed non-negative, as was shown in ref 14. Differentiating the target function T with respect to λ recovers eq 9. So solving eq 13 is equivalent to solving eq 9. Formally, CG is guaranteed to find the global minimum in L steps where L is the number of constraints. In practice, the number of iterations to determine the Lagrange multipliers is below 10. As is illustrated in the Results section, in our most efficient implementation, we fixed the number of iterations to seven. It provides good accuracy and at the same time avoids a convergence check and an "if" statement. Since the final decision about convergence is made by coordinate test, high accuracy of the Lagrange multipliers (which are calculated here in single precision) is less important.

We add one more twist to the straightforward application of CG using a preconditioner. The CG is solved trivially if $\mathbf{A}$ is diagonal or close to identity. A good preconditioner will bring the matrix closer to the identity operator. A common practical choice for a preconditioner is the following. Consider the matrix $\mathbf{D}$ that includes only the diagonal elements of $\mathbf{A}$ ($\mathbf{D}$ = diagonal($\mathbf{A}$)). We use $\mathbf{D}$ to bring the matrix $\mathbf{A}$ closer to identity and to reduce the number of iterations required.

Finally, it is important to emphasize that we have two loops of iterations. We have iterations to solve the Lagrange multipliers and iterations to adjust the coordinate vectors. The latter must be done in double precision or bond energy will not converge to less than 1×10^{-7} or so. We offer the option to run the entire algorithm on the GPU (determine the Lagrange multipliers and adjust the coordinates in double precision) without memory transfer before convergence.

Finally, we comment on the implementation on the GPU. The linear solver must be programmed from scratch manually, which is an additional challenge for nonsymmetric matrices. As we argue below, linear solvers for the GPU, based on function libraries, are not effective.

The Preconditioned Conjugate Gradient SHAKE Algorithm. The resulting SHAKE algorithm for constrained molecular dynamics[14] on the CPU or the GPU is below. Note the meaning of the three indices: time step i, SHAKE step n, and conjugate gradient step k. Step 4 is the CG algorithm where $\mathbf{r}$ (residual), $\mathbf{b}$ (direction of search step), and $\mathbf{x}$ are utility vectors. The scalars α (step magnitude) and γ are coefficients required in the preconditioner version of the conjugate gradient algorithm (PCG). In addition, max_shk limits the maximum number of SHAKE iterations, max_CG limits the maximum number of CG steps, ε is the maximum allowed constraint error (maximum of the error found in all bonds), and CG_tolerance is the conjugate gradient convergence tolerance for the determination of the Lagrange multipliers. The dashed lines indicate barriers. A barrier is a point in the code that enforces any threads that reached that point to wait. This is until all (other) threads have completed their tasks and reach the same point. Barriers are special breaks during parallel execution in which all the threads are forced to synchronize and are typically used at a time in which data sharing between the threads is required.

```
Algorithm: At time step i:
    1) Compute the matrix non-zero elements $A_{ab}$
    2) do while $(|\sigma_a(\mathbf{x}_i^{(n)})| > \varepsilon)$ and $(n < \text{max\_shk})$
        3. Evaluate constraints. If $|\sigma_a(\mathbf{x}_i^{(n)})| \le \varepsilon \quad \forall a$, stop.
        4. Solve the linear system $\sum_b A_{ab}\lambda_b = \sigma_a(\mathbf{x}_i^{(n)})$ to find $\boldsymbol{\lambda}^{(n)}$ by PCG:
            $\mathbf{r}_1 = \boldsymbol{\sigma} - \mathbf{A}\boldsymbol{\lambda}$
            $\mathbf{b}_1 = D^{-1}\mathbf{r}_1$
            $\mathbf{x}_1 = \mathbf{b}_1$
            - - - - - (end of PCG segment 1)
            do while $(|\mathbf{r}_k| > \text{PCG\_tolerance})$ and $(k < \text{max\_CG})$
                $\alpha = \mathbf{r}_k^T\mathbf{x}_k \,/\, \mathbf{b}^T\mathbf{A}\mathbf{b}$
                - - - - - (end of PCG segment 2)
                $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha\mathbf{A}\mathbf{b}_k$
                $\boldsymbol{\lambda}_{k+1} = \boldsymbol{\lambda}_k + \alpha\mathbf{b}_k$
                $\mathbf{x}_{k+1} = D^{-1}\mathbf{r}_{k+1}$
                $\gamma = \mathbf{r}_{k+1}^T\mathbf{x}_{k+1} \,/\, \mathbf{r}_k^T\mathbf{x}_k$
                - - - - - (end of PCG segment 3)
                $\mathbf{b}_{k+1} = \mathbf{x}_{k+1} + \gamma\mathbf{b}_k$
                - - - - -
            end do
        5. Compute an adjusted coordinate set:
            $\mathbf{x}_i^{(n+1)} = \mathbf{x}_i^{(n)} - (1/2)M^{-1}\sum \lambda_\beta^{(n)}\nabla\sigma_\beta(\mathbf{x}_i)$
        6. n = n + 1
    end do
```

Implementation of Parallel Conjugate Gradient SHAKE. We consider a symmetric or a constant **A** matrix. The PCG algorithm consists of matrix vector multiplications, vector dot products, and the product of a preconditioner matrix with a vector. In our case, the preconditioner matrix is diagonal and is stored as a linear array. For efficient execution, we exploit the sparsity of the matrix and minimize memory access. We store the sparse matrix in compressed row storage format.[22,14] This is the same format used by the cusparse NVIDIA GPU linear algebra library [NVIDIA CUDA user manual]. This storage is well suited for a parallel matrix vector multiplication algorithm.[23] The nonzero columns of each row are stored adjacent (row major order) in memory in the array **elem**. The length of **elem** is the number of nonzero matrix elements. The column number of each entry in **elem** is stored in the integer **col** array of the same length as **elem**. Similarly, the row number of each entry is stored in the integer **row** array of the same length as **elem**. The **cmprow** array is of length of the number of rows plus one. The elements of **cmprow** (except the very last) are the first nonzero column of **A** for each row (as indexed in **elem**). The very last element in the array stores the would-be-first column of a nonexisting additional row. This allows the creation of a loop such as shown in the matrix multiplication algorithm below to iterate over all rows without exceptions for the last row. Note that the indices follow C convention in which the first element is **elem**(0). An example for the compressed representation of a symmetric matrix follows below:

Compressed row storage example

$$\mathbf{A} = \begin{vmatrix} 3 & 2 & 0 & 0 \\ 2 & 1 & 5 & 0 \\ 0 & 5 & 6 & 0 \\ 0 & 0 & 0 & 4 \end{vmatrix} \quad \begin{aligned} \mathbf{elem} &= \{3, 2, 2, 1, 5, 5, 6, 4\} \\ \mathbf{col} &= \{0, 1, 0, 1, 2, 1, 2, 3\} \\ \mathbf{row} &= \{0, 0, 1, 1, 1, 2, 2, 3\} \\ \mathbf{cmprow} &= \{0, 2, 5, 7, 8\} \end{aligned}$$

The sparse matrix multiplication algorithm **b** = **Ax** is using the same arrays as described above. **b** is the result vector. All arrays are global variables. The integers row_start and row_end are the start and end row numbers assigned to the thread. In the GPU implementation, each thread loops over one row, and the outer loop disappears. In this case, row *j* equals the thread number.

```
for (int j=row_start; j<row_end; j++) {
        double elm = 0.0;
        for (int i= cmprow[j]; i< cmprow[j+1]; i++) {
                int c = col[i];
                double Aij = elem[i];
                elm += Aij * x[c];
                //---- if use symmetry, we must add the following here in the inner loop:
                if (c != row) b[c] += Aij * x[j]; //-- scattered write
        }
        b[j] = elm;
}
```

We do not take advantage of the symmetry of the **A** matrix because it requires scattered memory access and, in parallel, necessitates the use of a reduction. Threads need to write to elements in the result vector other than the rows they are assigned to in the load balance phase. The latter also requires local (thread private) copies of the result vector.

Parallelization. Steps 1 and 5 of the algorithm are easily parallelized. In step 1, when recomputing the matrix, we take advantage of the fact that the identity of the nonzero elements is fixed and that it was already stored in efficient sparse storage format. We consider only nonzero matrix elements that are divided precisely between open-MP threads on CPU and get the bond indices from the noncompressed column and row arrays. This can be done in parallel by dividing the number of matrix elements over the number of threads. Step 3 is divided between open-MP threads on the CPU or one constraint per thread on the GPU. On the CPU, each thread stores its own maximum σ error determined in a global array of length of the number of threads. Subsequently, on the CPU, all threads loop over this array and determine max error individually. On the GPU, a shared memory parallel reduction is done inside the function without global memory access. The GPU reduction is explained in more detail below. So on the CPU, the final reduction is not done in parallel as it is on the GPU. On the CPU, each thread performs its own final sum. In step 5 of the SHAKE algorithm (and eq 7), we sum up the adjustments of the coordinates. It is not possible to adjust the coordinates by looping independently over the constraints since it is possible that multiple constraints share an atom. Therefore, we loop instead over all atoms involved with one or more constraints and divide these between the threads. On the GPU, this is one atom per thread. To work simultaneously in the constraints and the atom representation, we keep three additional arrays, one array storing each atom number involved in any constraint and one array with pointers into a third array with the constraint numbers for that atom. The coordinate adjustments are in a nested loop, the outer loop over the atoms and inner loop (the sum of eq 7) over the constraints. This prevents the write conflicts between threads.

Parallel Preconditioned Conjugate Gradient. Dot products, while simple to calculate in series, are more challenging to do in parallel since communication is required to gather the results from all threads. To begin with, each thread calculates its share of elements of the dot product and accumulates the partial sum. To perform the global sum, all threads must complete their task, and a barrier for synchronization is required. In case of a dot product on several CPU threads, each thread must complete its own partial sum and write the result into a global array of length of the number of threads before a global sum can be attempted. Then, the final sum can be made. This final sum can also be done in parallel as we do on the GPU (see parallel reduction below) or as we do on the CPU; each thread can perform this final sum on its own whenever it is ready to use the data. The last option is not efficient on thousands of threads but is faster with a limited number of threads (say up to 128) because it does not require a second barrier. Therefore, we implement the parallel reduction only in the GPU code (see parallel reduction below).

The main obstacles to efficient parallelization are the thread synchronizations required to calculate the coefficients α and γ and update of the vector **b**. We therefore combined segments of the PCG algorithm (see algorithm above, dashed lines) in such a way that only three barriers remain, and at the same time memory (array) access is minimized. In general, the PCG code can be combined into three main segments (Supporting Information, Appendix A) that can be calculated without intermediate barriers in massive parallel calculation at a level of one thread per row of the matrix **A**. To reduce (array) memory access, for example in PCG segment 1, we declare a local utility variable q:

$$\text{double } q = \mathbf{Sigma}[n] - \mathbf{Ax}[n] \text{ (see Appendix A)}$$

This variable q is then used four times in the product with $\mathbf{D}^{-1}$, the assignment of vectors and the accumulating dot product. So this prevented us from accessing arrays **Sigma** and **Ax** again. Similarly, memory access is minimized in the other PCG segments (Supporting Information, Appendix A). This process works the same way on the GPU.

GPU Implementation. The entire algorithm is ported to the GPU in the GPU implementation. The outer Newton-like iterations need to be evaluated in double precision. In particular, the error vector determination and coordinate adjustment need to be done in double precision or the error will not converge to $\sim 10^{-10}$ or less as measured by relative error in bond lengths. The conjugate gradient itself is divided into the same segments as discussed above, and all steps are computed entirely on the GPU. There is no requirement for memory transfer between host and device during the determination of the Lagrange multipliers calculation nor during coordinate corrections resulting in significant speedup.

GPU Parallel Conjugate Gradient. The main difference between the CPU and GPU code is the granularity of parallelization. The CPU code is still serial in nature. On the GPU, each row of the vector **x** (and the matrix **A**) in **Ax** = **b** is represented by its own single thread. This completely eliminates all outer loops. In the matrix multiplication code segment shown above, the loop over row j is eliminated and row j is the thread number. We notice right away now there are no partial sums at all. Each thread only has the single value calculated in that thread. We take as an example $\mathbf{x}^T\mathbf{x}$. We could take the same approach as in the serial implementation. We place a barrier that enforces a wait until all threads have completed their single product $x_j{*}x_j$. And then we could have each thread sum all rows. This would give the correct result but with thousands of rows, very inefficiently. We therefore implement a shared memory parallel reduction algorithm at the end of each PCG segment, inside the kernel that is logarithmic in the number of array elements (Supporting Information, Appendix B).

A quick review of CUDA instructions necessary for the text below follows. Threads are divided into blocks. Within a block, threads can communicate through shared memory. Threads in a different block cannot access shared memory in another block. Individual threads also have local registers in which they can read/write scalars. Only the owner thread can access local registers. Global memory can be accessed by all threads, so it can also be used for communication between threads but is slow. Continuing with our example, we realize that each thread in a particular block has calculated its $x_j{*}x_j$ product; it can store this in fast and shared memory **shared** that can be seen by other threads in the same block. The shared memory array is the length of the fixed block size which is a factor of 32 (we use 128). The parallel shared memory reduction is conducted as follows. Each thread stores its value in **shared**:

$$\mathbf{shared}[j] = x_j{*}x_j$$

Reduction is first conducted within the thread block (see Figure 1). We start with 128 values in separate threads.

$$\text{if } (j < \text{blocksize}/2)\ \mathbf{shared}[j] += \mathbf{shared}[j + \text{blocksize}/2]$$

With this single statement, we have reduced our block data by half. The reduction, which is done in parallel, is exponentially fast:

```
for (int $n = 6$; $n <= 0$; $n--$)
    if ($j < 2^n$) $\mathbf{shared}[j] += \mathbf{shared}[j + 2^n]$
```

For a block size of 128, seven statements are needed to compute the sum of all elements of the block in the first position of the array **shared**. At this point, we need to communicate between blocks. We have up to 128 blocks that each have the sum of the dot product in the first thread of that block. We perform the last sum using the slow global memory. We define a global array **global** of length blocksize.

$$\text{if (thread} == 0)\ \mathbf{global}[\text{block ID}] = \mathbf{shared}[0]$$

At this point, we have a global array with up to 128 values, and one more reduction of one final block is required. Therefore, at this point we need a global thread barrier, which is done as follows (see also Appendix B in the Supporting Information). A counter is kept for each block during the global writes. When this counter reaches the number of total blocks, each has written its value, and the last block of threads to write is the one that will perform the final reduction. This last block will now load the global data back into shared:

$$\mathbf{shared}[j] = \mathbf{global}[j]$$

At this point, the reduction is just like what is shown above, except that only one block is used. Repeat for $n = 6$ to 0

$$\text{if } (j < \text{blocksize}/2^{9n})\ \mathbf{shared}[j] += \mathbf{shared}[j + 2^n]$$

When done

$$\mathbf{global}[0] = \mathbf{shared}[0]$$

The final single scalar result is now in **global**[0]. All other threads were waiting for this result; the waiting is enforced by the global barrier. Thread zero of this last block will release this global thread barrier.

A reduction (summation of array elements that are spread over a large number of threads) can be costly if communication is not handled efficiently. In the context of GPU programming, such reduction can be done in parallel if we stay within the boundary of a block—a subgroup of threads that share memory. For a block, the reduction requires only a logarithmic number of serial steps as a function of the length of the array. This is illustrated in Figure 2 for a block of threads on the GPU that share the same memory. A global barrier (for all blocks) is not provided currently in CUDA (only block-wise barriers are provided). The global barrier implemented between the block reductions and the final block reduction allowed calculation of an entire PCG step in one GPU kernel. The alternative is a separate kernel for each PCG segment. Calling separate kernels from the CPU introduces a global barrier anyway; however, the last barrier is a bit slower. Local storage of conjugate gradient intermediate results minimizes array memory access altogether just as in the CPU code (see example, double q = **Sigma**[n] − **Ax**[n] above and details in Supporting Information, Appendix B). We use a fixed block size of 128, allowing a maximum of 16 384 constraints. The fixed block size also allows complete unrolling of the reduction loop. The block size can be doubled to 256 (increasing the maximum to 65 536 constraints) or increased further as needed.

Water-Specific Force Kernels on the GPU. In molecular simulations with explicit solvent, the number of water interactions

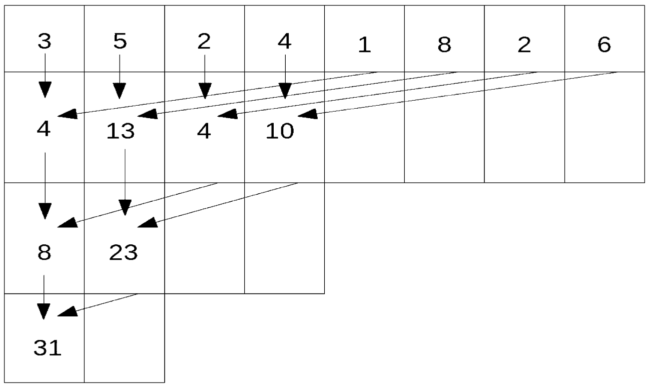

Figure 2. Parallel reduction (summation of array elements) in one block of eight GPU threads. In this example, the values in the top row of rectangles are examples of vector entries we wish to add. Three operations (in parallel) are required to accomplish this reduction of eight numbers. See also Appendix B. Doing the block-wise shared memory parallel reduction in the CG kernel avoids the intermediate storage of partial results in global memory. Only the final block reduction requires global memory access.

is significant, and a focus on optimizing this part, even with a specific code, seems like a reasonable approach. Indeed, the program MOIL[6] has used water-specific code for a long time. Here, we discuss the development of a new water code for the GPU that led to a dramatic speedup and was ported back to the CPU with a similar effect.

We start with the generation of atomic neighbor lists presented recently in our initial GPU implementation. Atomic neighbor lists are widely implemented in CPU based code. However, GPU memory limitations motivated a number of studies to seek different solutions.[24−26] In our previous paper, we show that atomic lists can be used accurately and efficiently on the GPU. The main simulation box is divided into rectangular cells or grid boxes, equal in size. Each atom is placed in a grid box according to its atomic coordinates. Since the neighboring boxes are predetermined, atomic neighbors are considered according to their own and neighboring boxes and are finally added to a list according to a cutoff distance criterion.

Force calculation can be done directly at the level of the grid boxes avoiding the use of lists. Why use atomic neighbor lists? The advantages of the atomic list approach are as follows. (1) Exclusion of nonbonded-interactions (of bonded atoms) is made at the list generation step, so we avoid exclusion tests during the force calculation. (2) Some parameter calculation can be done and stored during the list calculations (e.g., charge−charge multiplication) in a coalesced way. This not only saves a set of floating point calculations in the force kernel but also prevents the need to use local registers of the threads to store parameters for the atoms. It also prevents the need for shared memory storage of its neighbor atom parameters. (3) Since the cutoff distance is spherical, fewer interactions need to be calculated compared to direct box calculations.

A disadvantage of the neighbor list is that neighbor coordinates cannot be stored in shared memory (each atom has a unique list and the lists are too large), but this is compensated for by the significantly reduced number of interactions and frees the otherwise required shared memory for other uses. In particular, on Fermi based GPUs, a portion of this freed shared memory can be used as an L1 cache. This cache memory significantly increases coordinate access speed. The remaining portion of the freed shared memory stores a lookup table for a quadratic interpolation of the electrostatic force. We also keep the option of storing larger tables in texture memory, but we find the shared memory option significantly faster (almost 2 fold) even if the texture memory table access is completely cached. A third option is a medium-sized (up to 2000 knots or so) lookup table in constant memory; this option is still slower than the shared memory approach.

To conclude, the use of a neighbor list makes the resultant force kernel simpler. With the list, it has more resources available for other types of calculations. The contribution of the present manuscript is the division of input particles into nonwater atoms and water molecules resulting in multiple and specific force kernels, which are significantly faster. Our most complex force kernel (water−water interactions) allows the calculation of all nine terms of a water−water interaction while remaining around the maximum limit of 32 registers. Satisfying the limit of 32 registers allows for up to 66% occupancy on Fermi GPUs. For a water−water interaction, each thread represents one water molecule. It requires 18 local registers to store nine coordinates plus nine force accumulation registers. After separating the water molecules from the rest of the atoms, each grid box has two lists: a list of nonwater atoms and a list of water particles. The lists are generated in the same way as described previously.[5] We do not take advantage of the symmetry in the neighbor list in the GPU implementation (i.e., we compute both interactions, particles *i* and *j* and particles *j* and *i*). Memory access considerations motivate the doubling of computations (also see discussion by Stone et al.[25]). This means that we now have four respective kernels for neighbor list generation and four kernels for nonbonded force calculation. Each force kernel accesses its own neighbor list. The interaction types are as follows: (1) In the case of a nonwater atom interacting with another nonwater atom, each interacting pair can comprise two different atoms, and the geometric averages of the pair Lennard-Jones and product of charge parameters are stored in a single float4 of the neighbor list as previously described.[16] (2) In the case of a water molecule interacting with a nonwater atom, we again still use a float4 neighbor list matrix but store the nonwater atom parameters rather than the geometric pair averages stored in the previous case. We do not store products as is done in case 1 because additional memory would be needed. This is because there are two interaction types (nonwater with water oxygen and water hydrogen). We almost always choose to do an extra calculation over global memory access since the performance is typically bound by memory access. This is true for both CPU and GPU. (3) In the case of a nonwater atom with a water particle, only the integer number of the neighbor water molecule is stored in the respective neighbor matrix in order to obtain its coordinates in the force kernel. Note that this function is not needed in the CPU version of the code since on the CPU we do take advantage of the symmetry, and calculations of 3 are conducted in case 2. (4) With water interacting with another water molecule, we store the single integer of the number of neighbor waters in the neighbor matrix.

All of the above-mentioned functions are also implemented with Open Multi-Processing (OMP) for parallel execution on a shared memory system with multiple CPU threads in the absence of a GPU. The simplicity and performance of the GPU functions prompted us to port this code back to the CPU for multicore execution in the absence of a GPU. On the CPU, we do take advantage of symmetry in the neighbor matrix, so no pair interaction is calculated twice. The CPU has a limited number of threads, and it makes sense to have a copy of the force vectors for each thread. This allows the scattered writes to neighbor atom force that we avoid on the GPU. Therefore, on the CPU, a reduction over all thread results is required after the completion of the nonbonded force calculation on all threads. This reduction is parallelized in a trivial way: each thread adds *n* vectors of force of its subset of atoms where *n* is the number of threads. On the GPU, one thread represents either one atom in cases 1 and 3 above or one water molecule in cases 2 and 4 above, while on the CPU one thread has an outer loop iterating over multiple nonwater atoms or water molecules. Neighbor matrices are of the same types, and they can be generated on the CPU or on the GPU. Importantly, however, for coalesced access, the CPU matrix versions are the transpose of the GPU version. In massive parallel computing, the first row of the neighbor matrix is the first neighbor of all atoms or water molecules. Switching the index generates the other type: for CPU serial execution, the first row contains all the neighbor atoms of the first atom or water. More discussion on this topic can be found in refs 4 and 7.

Since all water molecules have the same composition and atomic parameters, it makes sense to store these parameters in constant or local memory rather than include them as interacting pair information in the neighbor lists. We precalculate the five water hydrogen−hydrogen, oxygen−oxygen, and hydrogen−oxygen pair interaction for charge (q_iq_j) and Lennard-Jones (A_iA_j and B_iB_j) products (see #define, Appendix C). This significantly reduces memory access and storage requirements of the neighbor matrix in cases 3 and 4 above since a neighboring water is represented simply by one integer. In addition, since one GPU thread now represents a water molecule in two of the interaction types discussed above (cases 2 and 4), one set of neighbor coordinates read can be used for the calculation of three interactions: one with each of the three atoms of the water belonging to the thread. This means 3-fold less coordinate access in these kernels compared to the nonwater atom approach used exclusively before.

The NB Force Kernel in More Detail (See also Supporting Information). The nonbonded force kernel for nonwater interactions is essentially identical to the one previously reported.[5] The new water−water kernel is shown in its entirety in the Supporting Information. It is an interesting case to illustrate the trade-off of the computing versus memory access. The efficiency is dominated by memory access and less by computing, especially for the GPU. In the procedure described below, we therefore emphasize reduced memory access at the expense of more calculation. This leads to a significantly more efficient code.

In the current implementation of the water−water kernel, one thread represents one water molecule. The coordinates and forces of one water molecule are loaded into 18 local registers. Quadratic force lookup coefficients are read from texture memory and stored in shared memory.[5] The neighboring water molecules are read from the global memory matrix in a coalesced way in which the next thread accesses the exact adjacent memory location of the neighbor list, resulting in substantial performance gain. For a more extensive discussion on coalesced memory access with atomic neighbor matrix molecular dynamics, see refs 5 and 27. The neighboring water is identified only by its index since all of the energy parameters are stored in registers. Information that we miss is the grid-box position of the neighboring water. This information is useful since it identifies neighbors that are related by translational symmetry. Nevertheless, to avoid memory access, we found it beneficial to reexamine the Cartesian displacements between the water oxygens instead. The translation only adds a logical test, which is not very expensive, leading to an overall computational gain. Once the single integer number denoting a water neighbor is obtained from the water−water neighbor matrix, the coordinates of the neighboring oxygen are fetched in a single float4 read, from global (L1 cache) or optionally from texture memory as discussed in ref 5. The difference vector with respect to the oxygen of the prime water molecule is computed. The elements of the difference vector are (*rx*, *ry*, *rz*). If any of these elements (say, *ri*) exceeds half box length, the neighbor must result from a translational symmetry operation. The displacement is shifted to make the length of all the vector elements smaller than half the box size. The extra cost of the shift was found to be less significant than memory access (if we attempt to store and transmit the grid-box identity and/or the difference vector).

After accessing the neighbor oxygen coordinate, the O−O interaction force can be calculated, as well as the H1−O and H2−O interaction forces. This is before further neighbor coordinates are read from global memory. Each set of coordinates (one of the three neighbor water atoms) allows for three pair calculations with each of the atoms of the water belonging to the thread (see Supporting Information, Appendix C, underlined comment for neighbor coordinate read). During these three calculations, only shared memory is accessed in the force lookup. The nine calculations have the effect of partial unrolling of the neighbor loop. In loop unrolling, the statements inside a loop are written explicitly and thereby reduce the loop iterations. Loop iterations carry some overhead.

RESULTS

Systems. The systems investigated and benchmarked in this study varied in their size (from 2690 atoms to 99 905) with consistent improvements in performance, as outlined in this section. Important simulation parameters are given in Table 1.

Table 1. Test Systems[a]

	no. of atoms	no. of bond constraints	PME-DP grid	cutoff d-l(freq) d-f	no. of water molecules
Pepta	2690	43	16 × 16 × 16	10.0(8) 9.0	882
TRPzipper	5847	227	32 × 32 × 32	9.99(8) 9.0	1875
Helix	20187	388	48 × 48 × 48	9.8(8) 9.0	6588
DHFR	23536	2523	48 × 48 × 48	10.0(8) 9.0	7012
DOPC-NaCl	38802	6784	48 × 48 × 96	10.0(8) 9.0	10502
1IHF	91765	5494	64 × 64 × 64	9.97(8) 9.0	28763
Myosin	99905	7691	64 × 64 × 64	10.0(8) 9.0	30776

[a]All seven systems listed below use explicit solvation. Unless otherwise noted, all results are obtained with tabulated conditions on a four-core AMD Phenom 965 desktop PC equipped with a GTX 480 GPU. The solvation box sizes in Ångstroms are 30.1 × 30.1 × 30.1 (Pepta), 40.0 × 40.0 × 40.0 (TRPzipper), 59.0 × 59.0 × 59.0 (Helix), 62.2 × 62.2 × 62.2 (DHFR), 65.0 × 65.0 × 120.0 (DOPC-NaCl), 99.8 × 99.8 × 99.8 (1IHF), and 90.0 × 112.5 × 105.0 (Myosin). Cutoff list (update frequency) calculation: "d-l" is the cutoff distance used in the atomic neighbor list generation. Update "frequency" is the number of steps between list updates, and "d-f" is the cutoff distance used in the force calculation. All of the water molecules are constrained with a specialized version of Matrix SHAKE for individual water molecules (the inverse of the symmetric and constant **A** for a water molecule is pre-computed at the beginning of the calculations and used ever after).

Performance. The new matrix SHAKE implementation for satisfying the macromolecule bond constraints is faster than the serial "bond relaxation" approach even on a single core. Timing results for SHAKE on a single core are listed in Table 2, where the last row is the bond relaxation method. Scaling seems to improve with system size.

We comment that system sizes considered are small to overcome the parallelization overhead. A factor that leads to less than optimal scaling is the relatively low load per thread. Even in our largest linked system with 7691 nonseparable bonds, we only have 641 bonds per core on 12 cores. This is a relatively small amount of work to justify significant communication. On the GPU, at one constraint or atom per thread, this translates to low occupancy (only a portion of the card is being used). Thus, while we see significant speedup in the SHAKE calculations while adding more cores, the speedup is slower than linear (see Figure 3). Hence, the parallelization of a globally linked system is significantly more complex than parallelization of decoupled bonded molecules (e.g., water molecules). Decoupled systems can be divided trivially between cores, scale perfectly, and are not included in the simulation below.

An interesting feature of the combination of the heterogeneous environment of multiple CPU cores and a GPU is the possibility to conduct asynchronous calculation. For example the GPU may work on conducting the SHAKE algorithm for the coupled system of bonds of the macromolecule(s), while at the same time the CPUs may work on adjusting water coordinates. A possibility of partitioning the workload is given in Table 3. Using the GPU and exploiting asynchronous calculations as mentioned earlier are providing an additional speed advantage. On the larger systems, the GPU constrained calculations are done at about the same speed as five CPU cores but at the same time free CPU cores for potentially picking up other tasks such as in this case water SHAKE.

More detailed benchmarks of SHAKE options and the use of a fixed matrix, versus a matrix that is updated every time step, is provided in Table 2. In all cases reported below, convergence was assumed when the relative errors of bond lengths were smaller than 10^{-10} and the corresponding errors for velocities did not exceed 10^{-8}. Significantly less accurate convergence criteria did not conserve energy to the threshold we desired (less than 1% change in energy for a microsecond simulation of DHFR in the NVE ensemble). Significant experimentation with the SHAKE parameters led to the following conclusions:

1. The use of a constraint matrix **A**, which is updated every time step, led to execution that was about ∼30% faster than the use of a constant matrix, which is precomputed in the beginning of the calculations. The constant matrix is of course cheaper to compute, but it requires a larger number of SHAKE iterations.
2. Conjugated gradient (CG) iterations on the GPU, to determine the Lagrange multipliers, are conducted in single precision. In principle, CG iterations can continue until high precision Lagrange multipliers are obtained. This however is not necessary and less efficient. The final judge of convergence is the accuracy of the bond lengths (in double precision), so the algorithm is still successful if the (approximate) Lagrange multipliers are not the most accurate but are successful in generating accurate bonds.
3. The CG calculations are conducted with a fixed number of iterations (and not necessarily until exact convergence of the Lagrange multipliers is achieved). This choice was

Table 2. Use of Fixed Constraint Matrix A versus a Matrix Which Is Updated Every Time Step (See Also Figure 1)[a]

method	DHFR	1IHF	DOPC	myosin
M-up PCG-fix-it=7	8(56*), 2.69	8(56*), 2.95	6(42*), 1.73	7.1(49.6*), 2.72
M-fix PCG-fix-it=7	13.6(95.5*), 3.81	12.8(89.7*), 3.89	10.7(75.1*), 2.57	12.6(88.1*), 3.77
M-up PCG-var.-it=7	11.8(63.4), 2.85	10.9(66.1), 2.99	8.0(47.7), 1.75	10.3(61.3), 2.77
M-fix PCG-var.-it=7	17.0(82.9), 3.99	16.0(81.5), 3.91	13.7(66.5), 2.59	16.1(79.0), 3.83
M-up PCG-var.-it=32	8.8(91.2), 3.16	7.5(92.6), 3.35	6.0(67.3), 1.96	7.25(84.8), 3.15
M-up GPU PCG-fix-it=7	7.8(55*), 1.07	8(56 *), 0.68	6(42.1), 0.44	7.1(49.6*), 0.53
M-fix GPU PCG-fix-it=7	13.7(95.9*), 1.6	13(90.7 *), 0.86	10.7(75.1*), 0.61	12.6(88.1*), 0.86
SHAKE-B (bond relaxation method)	61.4(n/a), 4.36	66.6(n/a), 4.38	21.62(n/a), 1.44	61.1(n/a), 4.0

[a]Explanation of symbols: M-up is an **A** matrix which is updated every time step. M-fix is a fixed matrix which is generated once at the beginning of the calculations and used ever after. PCG-fix-it=7 is the use of a parallel conjugate gradient algorithm with a fixed number of iterations (in this case seven). Similarly, PCG-var-it=7 means that the number of iterations was set to a maximum of seven, but the calculation was terminated if convergence was detected at a smaller number of iterations. This may be expected to save time; however, the check (for convergence) is less efficient on the GPU and on the CPU. SHAKE-B is the standard use of SHAKE by the bond relaxation method.[12] The numbers reported below each example name are reported in the format x.xx(yy.y)z.zz and means the following: x.xx is the average number of overall SHAKE iterations that are required for convergence. Note the significantly larger number of iterations required for coordinate adjustments when the matrix is fixed. yy.y is the total number of conjugate gradient iterations required per time step. If "*" is added to the left of the number, then the number CG of iterations was fixed; otherwise the number of iterations was determined according to convergence of the conjugate gradient algorithm ("var" option). It is not surprising that the number of CG evaluations is smaller for the "var" option. However, consider the most important measure, which is z.zz and is the time of executing the SHAKE algorithm in microseconds per bond. The shortest times obtained are for the "M-up PCG-fix-it=7" option.

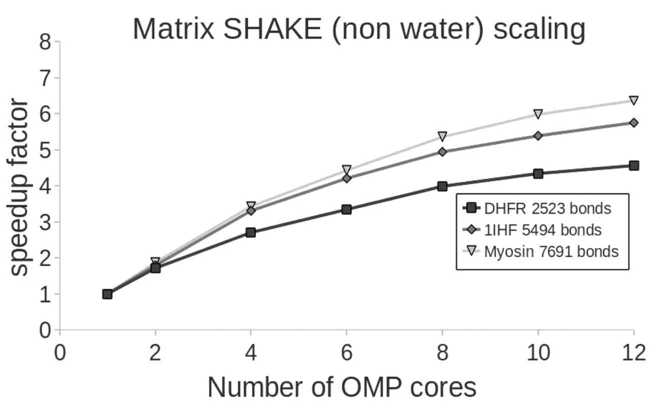

Figure 3. Speedup of the conjugate gradient SHAKE algorithm (coordinate + velocity) with the number of Open-MP cores used. Results are shown for DHFR, 1IHF, (protein + DNA), and myosin: the constraints for water are done in a separate calculation. These data were obtained on a 12-core node. The load of SHAKE bonds per thread in the case of DHFR is only ~200 on the 12 cores. Scaling improves with larger systems, but even our largest test system, myosin, balances to only 641 bonds per core, making further improvement with system size likely. Fastest GPU results (one GTX 480) are 5-fold faster than one CPU core so translate to at least five CPU cores for the largest systems.

made following efficiency considerations. A fixed number of iterations converges faster to the correct bond length than CG calculations that accurately determine the Lagrange multipliers. We note that testing for convergence is inefficient, even on the CPU. Instead, we conduct a fixed number of CG steps (typically seven, unrolled) that provide a sound next-approximation for the coordinate adjustments.

4. In principle, velocity SHAKE requires no iterations and should be satisfied with a single round of Lagrange multipliers. However, our calculations are using approximate, single precision Lagrange multipliers, and therefore sometimes iterations are required (until the above convergence criterion is satisfied).

As we discussed in our first paper on GPU/CPU implementation of MOIL,[5] our code conserves well the energy of the system and therefore provides adequate sampling of configurations and trajectories from the microcanonical ensemble. We illustrate the energy conservation in Table 4 and Figure 4. In principle, the SHAKE algorithm allows the use of a larger time step than before. In Table 4, we report energy conservation for the full SHAKE implementation as well as for the option of SHAKE-L, which means that only bonds that include light atoms (like hydrogen atoms) are included in the set of constrained bonds. SHAKING all bonds in the DHFR system shows an improved

Table 3. SHAKE on GPU vs Multi-CPU SHAKE, System Myosin, ~100 000 Atoms, 7691 Non-Water Bonds[a]

	CPU cores	time water	time other bonds	gain asynch	total time SHK
Matrix SHAKE: CPU	4	5.75	5.70	none	11.45
Matrix SHAKE: GPU	1 + (3 water)	6.04	4.07	2.53	7.58
Bond-relx	1 + (3 water)	6.04	30.8	6.04	30.03

[a]The execution of the matrix SHAKE on the GPU + one CPU core to instruct/monitor the GPU allows concurrent execution of water matrix SHAKE on the remaining CPU cores (so at four CPU cores, three cores are shaking water molecules). Asynchronous computational gain for SHAKE per time step is illustrated. We also included the results for the bond relaxation approach, which is the most commonly used SHAKE variant. In all calculations, water molecules were constrained by a specialized matrix SHAKE algorithm, a calculation conducted solely on the CPU.

Table 4. Energy Conservation in the DHFR System[a]

cut	time step	SHAKE	DRIFT (% μs)
10.0/9.0	2.0 fs	light	0.4
10.0/9.0	2.0 fs	matrix	0.1

[a]The RESPA algorithm[28] was used with an inner time step of 2 fs and outer time step of 4 fs. Quadratic force interpolation for electrostatics with 256 knots in the lookup table is used as described in ref 5. Under "cut" we report the cutoff distance used to generate the non-bonded list (10.0 Å) and the actual cutoff used in the calculation of the force (9.0 Å). The tolerance of error is defined as $|\Delta b/b|$, where b is the ideal distance and Δb the deviation from ideal value and is set to 10^{-10}.

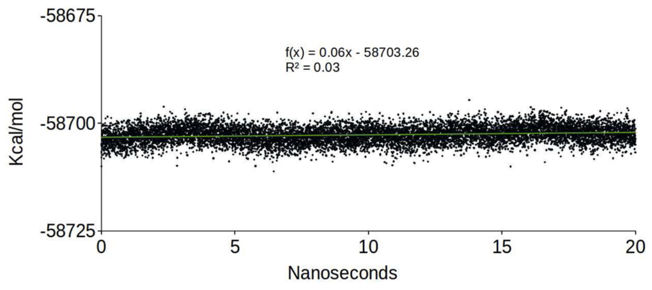

Figure 4. Energy conservation in the DHFR system. See legends of Table 1 and Table 4 for the simulation parameters. The total energy (in kcal/mol) is plotted as a function of time (in nanoseconds). A linear fit of the energy as a function of time suggests negligible energy drift, which we estimated as 0.1% relative error in a period of one microsecond.

energy conservation compared to SHAKE-L. The example below is for DHFR. We consider next the enhancement in performance that we obtained in the calculation of real space nonbonded interactions due to the introduction of the water kernel. In Figure 5, we compare the time of calculation in

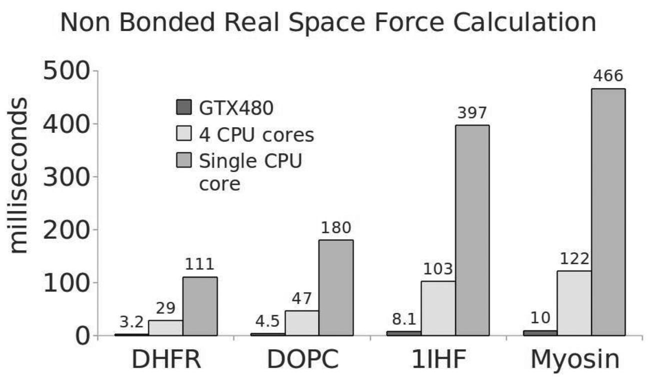

Figure 5. Calculation time of the nonbonded real space forces. Nonbonded real space force calculation on a single CPU core, four CPU cores, and a GTX480. GPU speedup factors versus a single CPU core are 37, 40, 49, and 47. For more details on the systems, see Table 1.

milliseconds on a system that contains a GPU card (GTX480), parallel calculations on all four cores of the shared-memory system, and calculations on a single core. We made a choice of computational setups that can be purchased cheaply and can be found in a usual laboratory setting. The speedup of the GPU system compared to a typical four core machine is roughly a factor of 10 for DHFR and DOPC and about a factor of 12 for 1IHF and myosin.

It is also useful to examine directly the benefit of the water kernel on the calculation of the nonbonded force on the GPU with a different measure. We consider the time (in nanoseconds) that is required per atom, and the comparison is provided in Figure 6 for the overall real-space nonbonded interaction using (black square) or not using (gray diamond) special water kernels. The speedup is consistently about a factor of 4. This is with exception of the two smallest systems for which overhead associated with the multiple kernels plays a role.

Finally, we provide an overall view of our code performance. We consider the same seven benchmark systems and compare a four core computer to a four core computer + GTX480. We measure the performance in nanoseconds per day emphasizing the optimization of different components in Figure 7. The best possible speedups are obtained for relatively small systems, and

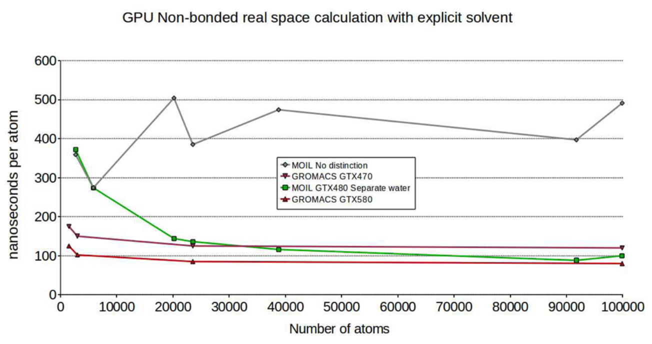

Figure 6. Time of nonbonded real space force calculation as a function of the number of atoms (time per atom shown) on the GTX480 GPU. The same seven test systems (see Table 1) are used. The GPU nonbonded real-space calculation is done with and without utilization of separate water force kernels. The GPU performs better with an increasing number of atoms in the system. The system at ∼39 000 atoms is a DOPC membrane with relatively less water molecules compared to other systems. Other fluctuations in separate water trends can be due to ionic concentration and ion distribution (more nonwater−water interactions). The additional overhead of separate water kernels causes slower results in small systems. Comparing our results to those of GROMACS (cores flyer: www.binarybio.com), we note that they use one faster and a second slower GPU (GTX470 and GTX580) in their benchmarks. We used an intermediate GPU (GTX480). For systems larger than 20 000 atoms, the results are comparable.

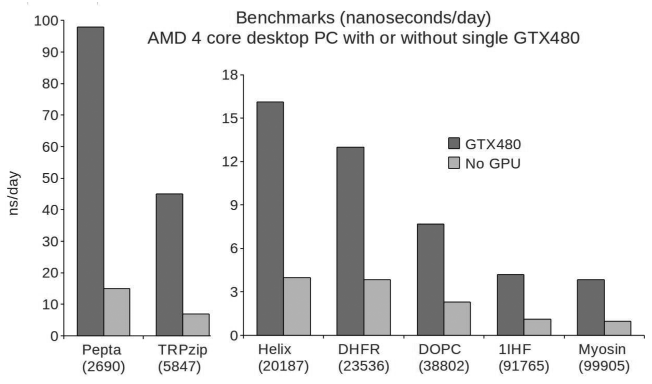

Figure 7. MOIL-opt with GPU overall performance. Although the speedup of real-space nonbonded force calculation does not degrade on the GPU with the number of atoms (see Figure 6), the overall speedup decreases with number of atoms due to newly exposed bottlenecks such as the PME (see Figure 8). The overall performance on DHFR (13 ns/day) is similar to results reported for the GROMACS 4.5 (http://www.gromacs.org/Documentation/Installation_Instructions/GPUs, see PME results).

the speedup is reduced for larger systems. The reason is the lack of successful optimization of the reciprocal summation of the nonbonded interactions (PME). In our earlier paper,[5] we noted that to conserve energy, the PME calculation is better conducted in double precision, and we therefore keep it on the CPU. While we parallelized the PME code on four cores, the performance is still lagging behind other speedup of the code. This optimization remains a topic for future work.

Figure 8 graphically illustrates the increased weight of the PME code and the necessity for future optimization to address this component or find an alternative to the calculation of long-range electrostatics. The figure is arranged such that the smallest system is at the center and the largest at the upper layer of the circle. It is evident that the yellow component of the calculation

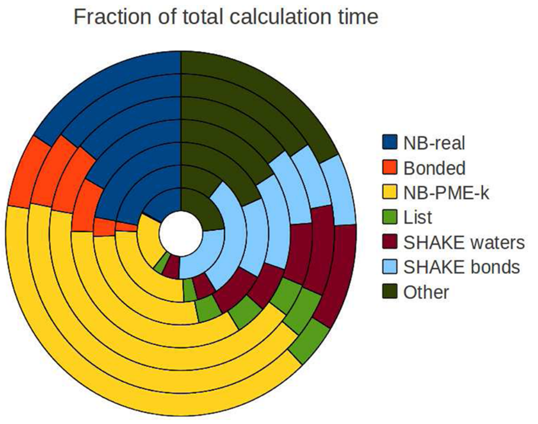

Figure 8. Fraction of computation cost of different parts of the molecular dynamics algorithm on a four core desktop PC equipped with a single GTX480 GPU. All four CPU cores and GPU are used here. The inner time of RESPA is 2 fs and the outer time 4 fs. In accord with our previous publication, the PME reciprocal calculation is not repeated every step.[5] The seven test systems are shown as different layers in the pie chart. The outermost layer is the largest system (myosin) and innermost pepta (see Table 1 for more details). Note that positive times per step are shown, while actual time per step includes a "negative" pie slice contribution because some time is gained from asynchronous execution on the GPU. This gain is not shown here. It is coming from SHAKE (see Table 3). Another gain is the simultaneous calculation of bonded interactions and a part of the reciprocal sum calculation on the CPU while the GPU computes the real nonbonded interactions.

dominates for the largest system and is significantly less important for the smallest system. Future direction may consider parallelization on more cores, or doing parts of the PME calculations in single precision (like we are doing for the SHAKE algorithm, the Lagrange multipliers are computed in single precision and the coordinates in double).

■ CONCLUSIONS

In the present manuscript, we revisited a molecular dynamics code (MOIL) that was ported to the heterogeneous environment of the CPU and GPU and further enhance the efficiency and versatility of the program. We have shown that a special kernel for water molecules speeds up the GPU calculations of real-space nonbonded forces by a factor of 4 for systems larger than 20 000 atoms. We have also illustrated that the usually difficult-to-parallelize SHAKE algorithm can run efficiently when the matrix form is used both on multiple CPUs or entirely on the GPU. The speedup is evident even on a single core compared to the bond relaxation approach. The GPU SHAKE implementation performance is similar to a calculation conducted on five cores. In our first CPU/GPU implementation of MD reported in ref 5, we emphasized the use of double precision coordinates and velocities to ensure energy conservation and correct dynamics in the microcanonical ensemble. Therefore, we split the SHAKE algorithm such that the coordinate adjustments are done in double precision on both the GPU or CPU. We were further able to take advantage of asynchronous calculations in which the SHAKE of water (that is done in separate routines in matrix form) is conducted simultaneously with the GPU SHAKE of the macromolecular coupled bond system. We anticipate that the gain from the new version of SHAKE will be more significant as the number of coupled bonds increases. Future directions may include the study of a more complex network of bonds (e.g., the inclusion of bond angle as Urey Bradely terms) and the expansion of a special kernel to other molecular fragments.

■ ASSOCIATED CONTENT

Ⓢ Supporting Information

A code of the preconditioner conjugate gradient algorithm on the GPU (Appendix A), code for "in kernel parallel reduction" (Appendix B), and code for GPU water−water interactions (Appendix C) are provided. This information is available free of charge via the Internet at http://pubs.acs.org.

■ AUTHOR INFORMATION

Corresponding Author

*E-mail ron@ices.utexas.edu. Phone: 512-232-5415. Fax: 512-471-8694.

Notes

The authors declare no competing financial interest.

■ ACKNOWLEDGMENTS

This research was supported by NIH grant GM05976, NSF grant CCF-0833162, and by Welch grant F-1783. Helpful discussions and the advice of Professor Martin Burtscher are gratefully acknowledged.

■ REFERENCES

(1) Hess, B.; Kutzner, C.; van der Spoel, D.; Lindahl, E. *J. Chem. Theory Comput.* **2008**, *4*, 435−447.
(2) Phillips, J. C.; Braun, R.; Wang, W.; Gumbart, J.; Tajkhorshid, E.; Villa, E.; Chipot, C.; Skeel, R. D.; Kale, L.; Schulten, K. *J. Comput. Chem.* **2005**, *26*, 1781−1802.
(3) Harvey, M. J.; Giupponi, G.; De Fabritiis, G. *J. Chem. Theory Comput.* **2009**, *5*, 1632−1639.
(4) Shaw, D. E.; Deneroff, M. M.; Dror, R. O.; Kuskin, J. S.; Larson, R. H.; Salmon, J. K.; Young, C.; Batson, B.; Bowers, K. J.; Chao, J. C.; Eastwood, M. P.; Gagliardo, J.; Grossman, J. P.; Ho, C. R.; Ierardi, D. J.; Kolossvary, I.; Klepeis, J. L.; Layman, T.; McLeavey, C.; Moraes, M. A.; Mueller, R.; Priest, E. C.; Shan, Y. B.; Spengler, J.; Theobald, M.; Towles, B.; Wang, S. C. *Commun. ACM* **2008**, *51*, 91−97.
(5) Ruymgaart, A. P.; Cardenas, A. E.; Elber, R. *J. Chem. Theory Comput.* **2011**, *7*, 3072−3082.
(6) Elber, R.; Roitberg, A.; Simmerling, C.; Goldstein, R.; Li, H. Y.; Verkhivker, G.; Keasar, C.; Zhang, J.; Ulitsky, A. *Comput. Phys. Commun.* **1995**, *91*, 159−189.
(7) Kirmizialtin, S.; Nguyen, V.; Johnson, K. A.; Elber, R. *Structure* **2012**, *20*, 618−627.
(8) Kirmizialtin, S.; Elber, R. *J. Phys. Chem. A* **2011**, *115*, 6137−6148.
(9) Majek, P.; Elber, R. *J. Chem. Theory Comput.* **2010**, *6*, 1805−1817.
(10) West, A. M. A.; Elber, R.; Shalloway, D. *J. Chem. Phys.* **2007**, *126*, 145104.
(11) Faradjian, A. K.; Elber, R. *J. Chem. Phys.* **2004**, *120*, 10880−10889.
(12) Ryckaert, J. P.; Ciccotti, G.; Berendsen, H. J. C. *J. Comput. Phys.* **1977**, *23*, 327−341.
(13) Olender, R.; Elber, R. *J. Chem. Phys.* **1996**, *105*, 9299−9315.
(14) Weinbach, Y.; Elber, R. *J. Comput. Phys.* **2005**, *209*, 193−206.
(15) Hess, B. *J. Chem. Theory Comput.* **2008**, *4*, 116−122.
(16) Elber, R.; Ruymgaart, A. P.; Hess, B. *Eur. Phys. J. Spec. Top.* **2011**, *200*, 211−223.
(17) Verlet, L. *Phys. Rev.* **1967**, *159*, 98−103.
(18) Barth, E.; Kuczera, K.; Leimkuhler, B.; Skeel, R. D. *J. Comput. Chem.* **1995**, *16*, 1192−1209.
(19) Eastman, P.; Pande, V. S. *J. Chem. Theory Comput.* **2010**, *6*, 434−437.
(20) Miyamoto, S.; Kollman, P. A. *J. Comput. Chem.* **1992**, *13*, 952−962.
(21) Hess, B.; Bekker, H.; Berendsen, H. J. C.; Fraaije, J. *J. Comput. Chem.* **1997**, *18*, 1463−1472.

(22) Barrett, R.; Berry, M.; Chan, T. F.; Demmel, J.; Donato, J.; Dongarra, J.; Eijkhout, V.; Pozo, R.; Romine, C.; der Vorst, H. V. *Templates for the solution of Linear Systems: Building Blocks for Iterative Methods*; SIAM: Philadelphia, PA, 1994.
(23) Buatois, L.; Caumon, G.; Levy, B. In *High Performance Computing and Communications, Proceedings*; Perrott, R., Chapman, B. M., Subhlok, J., DeMello, R. F., Yang, L. T., Eds.; Springer: New York, 2007; Vol. *4782*, p 358.
(24) van Meel, J. A.; Arnold, A.; Frenkel, D.; Zwart, S. F. P.; Belleman, R. G. *Mol. Simul.* **2008**, *34*, 259−266.
(25) Stone, J. E.; Phillips, J. C.; Freddolino, P. L.; Hardy, D. J.; Trabuco, L. G.; Schulten, K. *J. Comput. Chem.* **2007**, *28*, 2618−2640.
(26) Eastman, P.; Pande, V. S. *J. Comput. Chem.* **2010**, *31*, 1268−1272.
(27) Anderson, J. A.; Lorenz, C. D.; Travesset, A. *J. Comput. Phys.* **2008**, *227*, 5342−5359.
(28) Tuckerman, M.; Berne, B. J.; Martyna, G. J. *J. Chem. Phys.* **1992**, *97*, 1990−2001.