Acceleration of Dynamic *n*-Tuple Computations in Many-Body Molecular Dynamics

Patrick E. Small
Collaboratory for Advanced Computing and Simulations, Dept. of Computer Science
University of Southern California
U.S.A
patrices@usc.edu

Kuang Liu
Collaboratory for Advanced Computing and Simulations, Dept. of Computer Science
University of Southern California
U.S.A
liukuang@usc.edu

Subodh Tiwari
Collaboratory for Advanced Computing and Simulations, Dept. of Materials Science
University of Southern California
U.S.A
sctiwari@usc.edu

Rajiv K. Kalia
Collaboratory for Advanced Computing and Simulations, Dept. of Computer Science, Dept. of Physics & Astronomy, Dept. of Chemical Engineering & Materials Science,
University of Southern California
U.S.A
rkalia@usc.edu

Aiichiro Nakano
Collaboratory for Advanced Computing and Simulations, Dept. of Computer Science, Dept. of Physics & Astronomy, Dept. of Chemical Engineering & Materials Science, Dept. of Biological Sciences
University of Southern California
U.S.A
anakano@usc.edu

Ken-ichi Nomura
Collaboratory for Advanced Computing and Simulations, Dept. of Materials Science
University of Southern California
U.S.A
knomura@usc.edu

Priya Vashishta
Collaboratory for Advanced Computing and Simulations, Dept. of Computer Science, Dept. of Physics & Astronomy, Dept. of Chemical Engineering & Materials Science,
University of Southern California
U.S.A
priyav@usc.edu

ABSTRACT

Computation on dynamic n-tuples of particles is ubiquitous in scientific computing, with an archetypal application in many-body molecular dynamics (MD) simulations. We propose a tuple-decomposition (TD) approach that reorders computations according to dynamically created lists of n-tuples. We analyze the performance characteristics of the TD approach on general purpose graphics processing units for MD simulations involving pair (n = 2) and triplet (n = 3) interactions. The results show superior performance of the TD approach over the conventional particle-decomposition (PD) approach. Detailed analyses reveal the register footprint as the key factor that dictates the performance. Furthermore, the TD approach is found to outperform PD for more intensive computations of quadruplet (n = 4) interactions in first principles-informed reactive MD simulations based on the reactive force-field (ReaxFF) method. This work thus demonstrates the viable performance portability of the TD approach across a wide range of applications.

HPC Asia 2018, January 28–31, 2018, Chiyoda, Tokyo, Japan

ACM ISBN 978-1-4503-5372-4/18/01…$15.00
https://doi.org/10.1145/3149457.3149463

CCS CONCEPTS

• Theory of computation → Massively parallel algorithms
• Applied computing → Physics

KEYWORDS

Applications/Computational materials science and engineering, Performance Measurement/Analysis, modeling or simulation methods.

1 INTRODUCTION

Computation on dynamic *n*-tuples of particles is ubiquitous in scientific computing, with an archetypal application in many-body molecular dynamics (MD) simulations. MD is the most widely used simulation method for studying structural and dynamic properties of material [1]. MD simulations follow the trajectories of all atoms, while computing the interatomic interaction as a function of atomic positions. In his pioneering MD

simulation in 1964, Aneesur Rahman used a pair-wise interatomic potential that only depended on relative positions of atomic pairs [2]. More complex interatomic potentials (or force fields) have been developed later to study a wide variety of materials. In MD simulations of biomolecules, for example, the connectivity of atoms is fixed throughout the simulation, and the interatomic potential is a function of the relative positions of fixed *n*-tuples (n = 2, 3, 4) [3]. To describe wider materials processes such as structural transformations [4, 5] and chemical reactions [6, 7], however, the connectivity of atoms needs be dynamically updated, hence resulting in many-body MD simulations based on dynamic *n*-tuple interactions. Such is the case for many-body MD simulations of inorganic materials, which typically involve pair (n = 2) and triplet (n = 3) interactions [8]. Higher-order *n*-tuple computations are used in first principles-informed reactive molecular dynamics (RMD) simulations based on the reactive force-field (ReaxFF) method [6, 9, 10]. ReaxFF describes the formation and breakage of chemical bonds based on a reactive bond-order concept, and its interatomic forces involve computations on up to quadruplets (n = 4) explicitly and sextuplets (n = 6) implicitly through the chain rule of differentiation.

One of the simplest ways to map MD simulations onto parallel computers is spatial decomposition [11]. Here, the three-dimensional space is subdivided into spatially localized domains, and the interatomic forces among *n*-tuples involving the atoms in each domain are computed by a dedicated processor in a parallel computer [12]. To achieve higher parallelism than this spatial-decomposition approach, interatomic forces are often decomposed in various force-decomposition approaches [13-15].

On high-end parallel supercomputers, each of networked computing nodes consists of many cores and often augmented with accelerators such as general-purpose graphics processing units (GPGPUs) [16]. On such platforms with deep hierarchical memory architectures, metascalable (or "design once, continue to scale on future architectures") parallel algorithms often employ globally scalable and locally fast solvers [17-19]. An example of such global-local separation is the computation of long-range electrostatic potentials, where highly scalable real-space multigrids for internode computations are combined with fast spectral methods for intranode computations [17-19]. For MD simulations, the global-local separation insulates the optimization of intranode computations of dynamic *n*-tuples from internode parallelization approaches described above. In this paper, we thus focus on GPGPU acceleration of local dynamic *n*-tuple computations. Various schemes have been proposed for the optimization of local MD computations for pair-wise [20] and more general dynamic *n*-tuple interactions [21].

Extensive research and prior work exist, which explored the problem of accelerating CPU-based MD simulation codes with the GPGPU architecture [22-26]. However, majority of these works focused on the mechanical aspects of accelerating and tuning a set of existing codes on GPGPU. These mechanical aspects include better memory organization to promote coalescing of global memory-access operations, tuning of register usage, exploiting the GPU shared/cache memory hierarchy, and minimization of communication between the host and device.

While all of these aspects are crucial for achieving large speedups of scientific codes on GPU, here, we instead propose an alternative approach — named tuple decomposition (TD) — to GPU acceleration that restructures the enumeration of interatomic interactions and the calculation of potential energies, so that they can be performed more efficiently on GPGPU. Our approach employs two techniques: (1) pipelining and (2) *in situ* construction of two-body and three-body atomic interaction lists on GPGPU. An existing MD codebase, which calculates two-body and three-body interatomic potentials based on the conventional particle decomposition (PD) approach, serves as a platform to demonstrate our approach.

Despite various GPGPU implementations of dynamic *n*-tuple computations, less studies have focused on the critical factors that dictate the performance of these approaches. In this work, we analyze the performance characteristics of the TD and PD approaches. Among more conventional factors such as thread divergence, we have found that the register footprint plays a critical role in controlling the GPGPU performance. In addition to a many-body (n = 2 and 3) MD simulation, this finding is shown to hold for higher-order *n*-tuple computations in ReaxFF-based RMD simulations.

2 PHYSICAL MODEL

2.1 Interatomic Potential

The MD simulation software used as a basis for this research is described in Ref. [7]. To make the discussion specific, we first consider an interatomic potential proposed in Ref. [8] to study structural and dynamic correlations in silica (SiO_2) material. This implementation includes an interatomic potential combining pair and triplet interactions and the linked-list cell method for reducing the computational complexity of the force calculation to $O(N)$ (where N is the number of atoms), while the standard velocity Verlet algorithm is used for time-stepping [7, 12, 21, 27].

The potential energy of the system is a sum of atomic pair (or two-body) and triplet (or three-body) contributions [8]:

$$V = \sum_{i<j} u_{ij}^{(2)}(r_{ij}) + \sum_{i,j<k} u_{ijk}^{(3)}(\mathbf{r}_{ij}, \mathbf{r}_{ik}), \quad (1)$$

where $r_{ij} = |\mathbf{r}_{ij}|$, $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$, and $\mathbf{r}_i$ is the three-dimensional vector to represent the position of the i-th atom. In Eq. (1), the pair potential is given by

$$u_{ij}^{(2)}(r) = A\left(\frac{s_i + s_j}{r}\right)^{h_{ij}} + \frac{Z_i Z_j}{r} e^{-r/l} - \frac{a_i Z_j^2 + a_j Z_i^2}{2r^4} e^{-r/z}. \quad (2)$$

Here, the three terms represent the steric repulsion, the Coulombic interaction due to charge transfer, and an induced dipole-charge interaction caused by electronic polarizabilities of atoms, respectively. The pair potential is truncated at a cutoff distance, $r_{ij} = r_{c2}$. The triplet potential in Eq. (1) is expressed as

$$u_{ijk}^{(3)}(\mathbf{r}_{ij}, \mathbf{r}_{ik}) = B_{ijk} \exp\left(\frac{\mu}{r_{ij} - r_{c3}} + \frac{\mu}{r_{ik} - r_{c3}}\right) \times \frac{\left(\frac{\mathbf{r}_{ij} \cdot \mathbf{r}_{ik}}{r_{ij} r_{ik}} - \cos\bar{\theta}_{jik}\right)^2}{1 + C_{jik}\left(\frac{\mathbf{r}_{ij} \cdot \mathbf{r}_{ik}}{r_{ij} r_{ik}} - \cos\bar{\theta}_{jik}\right)^2} \times \Theta(r_{c3} - r_{ij})\Theta(r_{c3} - r_{ik}), \quad (3)$$

where $\Theta(x)$ is the step function. In Eq. (3), the cutoff radius, r_{c3}, for triplet interactions is much less than that for pair interactions, r_{c2}. The parameters in Eqs. (2) and (3) are found in Ref. [8]. These parameters are fitted to reproduce the experimentally measured structural and mechanical properties of the normal-density silica glass.

2.2 MD Simulation

The trajectories of atoms are discretized with a time discretization unit of Δt. In the widely used velocity Verlet algorithm, the positions $\mathbf{r}_i(t)$ and velocities $\mathbf{v}_i(t)$ of atoms (i = 1,..., N) at time t are updated as

$$\mathbf{r}_i(t + \Delta t) = \mathbf{r}_i(t) + \mathbf{v}_i(t)\Delta t + \frac{1}{2}\mathbf{a}_i(t)\Delta t^2 + O(\Delta t^4), \quad (4)$$

$$\mathbf{v}_i(t + \Delta t) = \mathbf{v}_i(t) + \frac{\mathbf{a}_i(t) + \mathbf{a}_i(t + \Delta t)}{2}\Delta t + O(\Delta t^3), \quad (5)$$

where

$$\mathbf{a}_i = -\frac{\mathbf{F}_i}{m_i} = -\frac{1}{m_i}\frac{\partial V}{\partial \mathbf{r}_i} \quad (6)$$

is the acceleration of the i-th atom. In Eq. (6), Fi is the force acting on the i-th atom and mi is its mass. The velocity Verlet algorithm repeats the body of the main MD simulation loop as shown in Algorithm 1. Note that the acceleration at time t, ai(t), has already been computed in the previous simulation step or before the main MD simulation loop is entered for the first simulation step.

Algorithm 1: Body of the main MD simulation loop based on the velocity-Verlet algorithm.

1. $\mathbf{v}_i(t + \frac{\Delta t}{2}) \leftarrow \mathbf{v}_i(t) + \frac{\Delta t}{2}\mathbf{a}_i(t)$ for all i
2. $\mathbf{r}_i(t + \Delta t) \leftarrow \mathbf{r}_i(t) + \mathbf{v}_i(t + \frac{\Delta t}{2})\Delta t$ for all i
3. Compute $\mathbf{a}_i(t+\Delta t)$ as a function of the set of atomic positions $\{\mathrm{r}_i(t + \Delta t) | \ i = 1, \ldots, N\}$, according to Eq. (6) for all i
4. $\mathbf{v}_i(t + \Delta t) \leftarrow \mathbf{v}_i(t + \frac{\Delta t}{2}) + \frac{\Delta t}{2}\mathbf{a}_i(t + \Delta t)$ for all i

On parallel computers, we employ a spatial decomposition approach. The simulated system is decomposed into spatially localized subsystems, and each processor is assigned the computation of forces on the atoms with one subsystem [7, 12, 21, 27]. Message passing is used to exchange necessary data for the computations. Specifically, before computing the forces on atoms in a subsystem (step 3 in Algorithm 1), atomic positions within the interaction cutoff radius r_{c2} within the boundaries of the neighboring subsystems are cached from the corresponding processors.

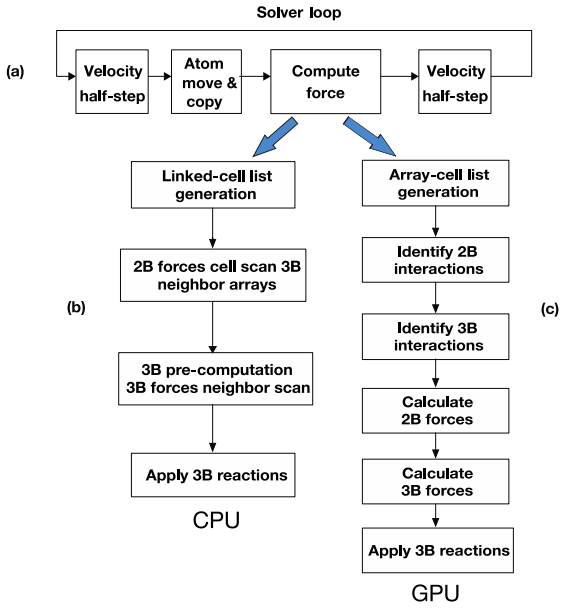

Figure 1: The workflow of the main MD simulation loop: (a) main solver loop, (b) original pipelining of atomic force computation on CPU, (c) restructured pipeline on GPU.

Fig. 1 (a) shows the workflow of the main MD simulation loop on a parallel computer, where "Atom

copy" denotes this interprocessor atom caching. After updating the atomic positions according to the time stepping procedure (step 2 in Algorithm 1), some atoms may have moved out of its subsystem. These moved-out atoms are migrated to the proper neighbor processors. "Atom move" in Fig. 1 (a) represents this interprocessor atom migration along with the atomic-position update.

2.3 Linked-List Cell Method

A naive method of computing the forces between atoms in MD simulations is to first consider an atom *i* and then loop over all other *j* atoms to calculate their separations. This approach imposes an $O(N^2)$ computational complexity for pair interactions, and an even worse $O(N^3)$ complexity for triplet interactions. Thus, the naive method becomes untenable when the atom count, *N*, becomes large.

The linked-cell list method [28] reduces this computational complexity to $O(N)$ for both cases by dividing the simulation region into cells whose width is slightly greater than the pair interaction cutoff distance r_{c2}. At each time step, each atom is classified by cell, and then only interactions between atoms in the same or adjacent cells are considered in the force calculation. For dynamic pair and triplet computations, the conventional linked-list cell method works as follows [12, 27]. On each processor, the spatial subsystem containing both the resident and cached atoms is divided into cells of equal volume whose edge is slightly larger than r_{c2}. Lists of atoms residing in these cells is constructed by the linked-list method [28]. Pair forces on the resident atoms are computed by traversing atomic pairs using the linked lists. An atom in a cell interacts only with the atoms within the cell and its 26 neighbor cells. At the same time as the computation of pair forces, a list of primary pair atoms, *lspr*, is constructed. Here, *lspr*[*i*][*k*] stores the identifier of the *k*-th neighbor atom, which is within the shorter cutoff distance, r_{c3}, of the *i*-th atom. Triplet forces are computed using the primary pair list *lspr*, which has been constructed during pair-force computations. In Eq. (1), only the resident atoms within the processor are included in the summation over index *i* to avoid over-counting. On the other hand, indices *j* and *k* are summed over both the resident and copied atoms. Partial derivatives of the potential energy with respect to the positions of *j* and *k* atoms therefore produce forces on the cached atoms as well as on the resident atoms. The reaction terms on the cached atoms are sent back to the processors in charge of the neighbor spatial subsystems and are added to the forces there.

3 GPGPU IMPLEMENTATION

Our focus in accelerating these codes was finding an efficient method of porting the main MD simulation loop to a graphics processing unit (GPU). To guide that process, we follow two fundamental principles: (1) minimization of control divergence (conditional branching, non-uniform iteration counts) within threads of a warp, and (2) minimization of synchronization events between threads. The first principle is important on a single instruction, multiple threads (SIMT) architecture such as GPU, since the kernel scheduler launches threads in groups of 32 (called a warp). Each thread within a warp executes the instructions of a kernel in lockstep. Divergent execution from conditional branches is allowed, but the threads in a warp suffer large performance penalties when this occurs since threads will pause and resynchronize at the end of the conditional branch. A similar situation occurs when a load imbalance is present. For example, if one or more threads in a warp execute many more iterations of a loop, warp resources will sit idle while the overloaded threads execute.

With these principles in mind, we now turn to the computations performed within the solver loop as outlined in Fig. 1 (a). The velocity half-step updates, atomic position updates, and atom copies (for enforcement of the periodic boundary conditions, *etc*.) are all directly translated to GPU kernels. These operations are trivially parallelized by unrolling their loops by atom. Thus, each thread of the corresponding GPU kernels is responsible for updating the state of a single atom.

However, the vast majority of work performed by the solver occurs within the acceleration-computation step ("Compute force" in Fig. 1 (a)). This is much more difficult to parallelize, and here, we are forced to dramatically restructure the algorithm for efficient execution on GPU. This restructuring involves application of a pipelining technique to decompose the computation into a longer sequence of simple steps (implied, but not specifically discussed in Ref. [24]). Fig. 1 (b) and (c) show the original acceleration computation juxtaposed with the new pipelined approach.

The original MD algorithm fuses the identification of interactions between atoms *i, j* (pair or two-body) and *i, j, k* (triplet or three-body) with the calculation of the potentials from Eqs. (2) and (3), respectively. This organization is appropriate for the CPU since having spent the time searching for the neighbors of a particular atom *i*, no other overheard is incurred to compute the potential other than clock cycles in the arithmetic logic unit. On GPU, however, this is suboptimal as a particular atom *i* will have a variable number of neighbors in its vicinity. If each thread of a kernel is assigned to find the neighboring interactions for a specific atom, it is very likely that the threads in a warp will be executing differing amounts of work, thus violating one of the optimizations principles we committed to follow. In addition, determining the validity of an interaction between atoms involves numerous checks, such as a comparison of atom types, resident atom within this spatial subsystem versus cached atom, and the distance cutoff

between the atoms. These checks are conditional branches, which cause control divergence as well.

The GPU implementation alternatively factors out the computation of two-body and three-body potentials from the identification of the participating interactions (Fig. 1 (c)). The calculation of potentials is an embarrassingly parallel operation, with little control divergence and no synchronization required other than an *atomicAdd*() to sum the potential contributions for an atom *i* across many threads. Thus, calculating two-body potentials is now a two-step process: construct a list (or array) of valid interactions within the cutoff distance r_{c2} for all atom pairs *i*, *j*, and then compute the potential contribution from each interaction in parallel. The same process can be used for three-body potentials. In other words, GPU parallelizes the potential computation by interaction rather than atom.

With this factorization, we achieve a significant speedup in performance over a baseline GPU implementation that mirrored the CPU approach with parallelization by atom (Fig. 1 (c)). Having separated the easily parallelizable computation from the tricky interaction determination, we can then focus on ways to efficiently accelerate the construction of two-body and three-body interaction lists on GPU. The approaches we employed to accomplish this are explained in the following section

3.1 Two-Body Interaction List Generation

The two-body interaction list generator used in our application draws inspiration from the Verlet neighbor-list algorithm by Lipscomb *et al*. [24], but has significant differences. The Lipscomb method is elegant and simple, yet it does exhibit a drawback. The method relies on the complete enumeration of all possible two-body atomic interactions at the outset (the master list), before the sorted member list can be generated. With the cell structure of MD, the number of such interactions is bounded by $O(N)$, however, the hidden constant factor is very large. In addition, Ref. [24] does not address how to generate the master list; the implication is that it may have been generated on CPU and transferred to GPU. We instead would like the list generation to occur entirely on GPU.

To address this problem, we extend the interaction metaphor to its logical conclusion — we not only consider atomic interactions, but also cell interactions. At the outset, during application initialization, once the problem space has been decomposed into cells, the program catalogs all combinations of pairs of cells *u*, *v* that are adjacent to one another within the lattice (including interaction with self, and interaction with the boundary cells). This cell interaction list $I_{u,v}$ is saved for future use, and is immutable for the runtime of the application. Within the source code, the cell interaction list is represented by two arrays, *linterci* and *lintercj*.

More concretely, as the solver executes on GPU, the kernel responsible for the array-cell list phase scans the atoms in the lattice in parallel, classifies each by the cell they are located within, and counts the number of atoms in each cell. The identifier of each atom is stored in an array A_u, where *u* is the scalar-index of the cell that the atom is located within. Thus, there is one such array for each scalar cell in the problem domain. The atom counts per cell is kept in C_u, again where *u* is the scalar cell index. Within the source code, A_u is implemented as the two-dimensional array *cellatomT*, while the cell counts are represented in the array *cellcount*.

Using the CUDA Thrust software development kit (SDK) [29], the set of cell counts C_u is scanned to find the maximum value $m = \max_u(C_u)$, representing the cell with the most atoms. At this point, the application has sufficient information to explore the entire space of possible interactions in parallel. Any pair of interacting cells $I_{u,v}$ will have at most m^2 combinations of two-body interactions since the maximum number of atoms in any given cell is *m*. And since there are $\|I_{u,v}\|$ possible cell interactions, an upper bound on the number of two-body atomic interactions in the problem domain is

$$t = m^2 \left\| I_{u,v} \right\|. \tag{7}$$

A kernel is then launched with *t* threads to execute the "identify 2B interactions" phase in Fig. 1 (c). Inside the kernel, each thread t_{id} extracts the identifiers of one pair of atoms *i*, *j* to test for two-body interaction via the following multi-step process. First, the index of the cell interaction, i_{C}, and the atomic interaction combination within that cell, a_{C}, are computed from the thread identifier t_{id} by

$$i_{\text{c}} = t_{\text{id}} \bmod \left\| I_{u,v} \right\|, \tag{8}$$

$$a_{\text{c}} = t_{\text{id}} / \left\| I_{u,v} \right\|. \tag{9}$$

Then, the indices of the interacting cells, *u* and *v*, are determined by looking up the i_{c} entry in $I_{u,v}$:

$$\{u, v\} = I_{u,v}(i_{\text{c}}). \tag{10}$$

The index positions of the (possibly) interacting atoms in A_u and A_v are computed by

$$i_{\text{idx}} = a_{\text{c}} / C_v, \tag{11}$$

$$j_{\text{idx}} = a_{\text{c}} \bmod C_v. \tag{12}$$

If iidx ≥ Cu, the atomic interaction combination ac is not in the set of possible interactions for cells u and v, and that thread terminates. Otherwise, the atom identifiers i and j are finally retrieved from Au and Av:

$$i = A_u\left(i_{\text{idx}}\right), \tag{13}$$

$$j = A_v\left(i_{\text{idx}}\right). \tag{14}$$

At this point, a final check is performed to remove duplicate i, j interactions when considering cell interactions with u = v (again, by termination). The remaining running threads then each submits its *i*, *j* pair of atoms to the same cutoff threshold testing as the original MD code, and appends *i*, *j* to the two-body interaction list if it passes the threshold tests. In this manner, the entire set of possible two-body atomic interactions are tested in parallel, and *in situ*, on GPU. A single contiguous list of interactions is produced, which can then be evaluated in parallel within the "calculate 2B forces" phase in Fig. 1 (c). Within the code, the list is represented by the arrays *linteri* and *linterj*. Fig. 2 illustrates the procedure of two-body list construction on GPU. Each thread examines an interaction between atom *i* and *j*, and transfers the index pair to *linteri* and *linterj* only if it passes the threshold test.

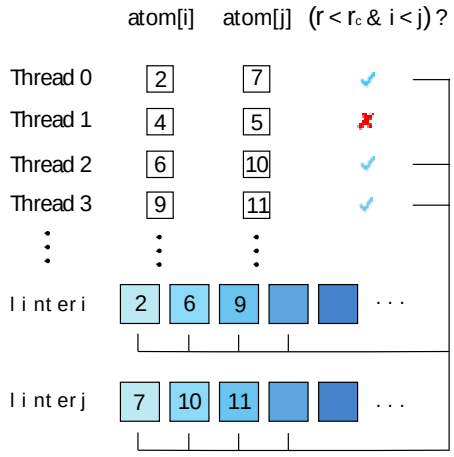

Figure 2: Concurrent construction of valid interactions.

At first glance, this method may seem wasteful of processor resources. It is true that perhaps even one-half of the threads will terminate without ever testing two atoms for an interaction, due to imbalances in the distribution of atoms among the cells. However, the terminating threads are often grouped contiguously in large swathes within the thread space, allowing the GPU warp scheduler to immediately marshal those released resources for other warps waiting in the run queue. Another disadvantage that should be noted is this method's reliance on a global counter, incremented with *atomicAdd*() when new *i*, *j* interactions are appended to the interaction list. Despite testing various alternative solutions, which replace that counter with other data structures, the current implementation provides the best overall runtime performance. We discuss a possible solution to this problem in the concluding remarks.

3.2 Three-Body Interaction List Generation

The three-body interaction list generator utilizes a different algorithm than its two-body counterpart. This method examines the two-body interaction list produced in the previous section, and searches for those interactions that fall within the more restrictive three-body distance threshold, r_{c3}. Those interactions that fall within this cutoff are saved in the *lspr* array in much the same way as is done in the original MD code explained in Section 2.3. However, in the GPU implementation, this array is populated by a kernel that inspects each two-body interaction in parallel (one thread per interaction). As shown in Algorithm 2, kernel *gen_lspr* produces array *lspr* from the existing two-body interaction lists *linteri* and *linterj*.

Algorithm 2: GPU implementation of the generation of three-body lists.

CUDA kernel, *gen_lspr*

```
// tid is thread ID
// lnum3b[i] is the number of atoms within rc3 of atom i
// linteri and linterj are two-body interaction pairs
i ← linteri[tid]
j ← linterj[tid]
if rij < rc3
    lspr[i][lnum3b[i]] ← j
    lspr[j][lnum3b[j]] ← i
    lnum3b[i]++
    lnum3b[j]++
end if
```

CUDA kernel, *gen_3body_list*

```
// offset is the prefix-sum from Thrust operation
i ← tid
for j from 1 to lnum3b[i]
    for k form j + 1 to lnum3b[i]
        linter3i[offset] ← i
        linter3j[offset] ← lspr[i][j]
        linter3k[offset] ← lspr[i][k]
        update offset by computing prefix-sum of
        combinations
    end for
end for
```

Once the *lspr* array is populated, Thrust is used to scan the neighbor counts for each atom to generate an array of prefix sums, with the sums produced after applying a function to the neighbor counts. More specifically, the number of three-body interactions, n_i, associated with atom *i* is computed with the formula,

$$n_i = \frac{c_i(c_i-1)}{2} \quad (15)$$

where c_i is the number of neighboring atoms for atom *i* in *lspr*. Then the array of n_i is used as input into the Thrust prefix-sum operation to produce a new array of values p_i. Thrust is also employed to compute the total number of

three-body interactions by simply performing a reduction on n_i.

The prefix-sum array p_i is then used as input by a follow-on kernel that populates the actual three-body interaction list. Each thread in the kernel is assigned to process an atom *i*, and it iterates through the n_i 3-tuple combinations for that atom in *lspr*. The threads save the resulting *i*, *j*, *k* interactions into the three-body interaction list starting at offset p_i. The result of this complicated process is a fully populated *i*, *j*, *k* interaction list, entirely produced on GPU, which can be used in the "calculate 3B forces" phase in Fig. 1 (c). In the source code, this interaction list is represented by the trio of arrays *linter3i*, *linter3j*, and *linter3k*. In Algorithm 2, kernel *gen_3body_list* is implemented to store the three-body interactions into these arrays.

This interaction list generator is remarkably different from its counterpart. Its primary advantage is the elimination of a global counter for insertion into the three-body interaction list. The purpose of the prefix-sum array is to assign a dedicated range of entries within the final interaction list to each thread (atom) so that no explicit coordination is required between threads.

This is a significant advantage, but it comes with a cost. This method does not scale when there are large numbers of three-body interactions per atom since the memory write into the final interaction list is strided by the number of combinations. Thus, as the interaction count per atom increases, the stride correspondingly increases, and memory coalescing worsens. We have found in practice that the combination counts are typically very small, as well as the total number of three-body interactions, so the performance degradation from the poor coalescing strategy is minimal. It should be noted here that $r_{c3} << r_{c2}$, as stated before. This comes from a physical principle that higher-order *n*-tuple interactions ($n \geq 3$) are short-ranged and only a few neighbors per atom contribute to them, compared to hundreds of neighbors for pair interactions.

4 BENCHMARKING AND VALIDATION

To evaluate the runtime performance of the GPU implementations, we run a series of benchmarking tests at a center for high performance computing that operates both a large, heterogeneous CPU cluster and a GPU-accelerated cluster. The GPU-accelerated cluster is comprised of 264 compute nodes, each with two Nvidia Kepler K20m GPU accelerators. We demonstrate the performance improvement of the new TD approach on MD simulations by analyzing two comparisons: TD's GPU implementation versus (1) the original MD code and (2) a baseline GPU implementation that directly mirrors the CPU implementation.

4.1 GPU Implementation versus the Original MD Code

For this timing experiment, we have selected three reference CPU chipsets from the collection of available compute nodes on the CPU cluster at HPC. We compare the original MD runtime performance on those chipsets versus the GPU implementation executing on the Kepler K20m for a series of MD simulations. The lattice sizes in the simulations varies from 4×4×4 to 16×16×16 crystalline unit cells (where each unit cell contains 24 atoms), and each simulation runs for 1,000 time steps. The CPU chipsets used in the comparison are AMD Opteron 2356 with a clock speed of 2.3 GHz and 2 MB cache, AMD Opteron 2376 (2.3 GHz, 6 MB cache), and Intel Xeon E5-2665 (2.4 GHz, 20 MB cache). The purpose behind the multi-chipset comparison is to give a comprehensive view of the GPU implementation's performance characteristics versus a range of common CPUs.

Fig. 3 plots the wall-clock time versus simulation size (*i.e.*, the number of atoms) for the three CPU chipsets and the GPU. The corresponding speedups of the GPU implementation over the three CPU implementations are plotted as a function of the simulation size in Fig. 4. As can be seen in the graph, the speedup varies greatly with the power of the reference CPU, with the newer Intel Xeon faring better than its counterparts. However, even against the Xeon, the GPU implementation demonstrates a significant speedup of up to 6-fold, which improves as the lattice size is increased. This overall trend is expected as the overhead costs of device initialization, PCI bus communications, and kernel launches are amortized over a larger set of atoms.

An interesting observation noticeable in those charts is the large increase in speedup that the GPU application experiences for the largest simulation versus the two AMD Opteron CPUs. This is because the Opteron cache size (6 MB) is not large enough to accommodate the entire 98,304 atomic states along with other necessary data, and the cache-miss rate increases dramatically at that point. If we were to scale up the tests further, we would likely continue to see their runtime performance drop in comparison to the GPU.

To validate the simulation results generated by the GPU, we have compared the final atomic positions and velocities with those of reference datasets produced by the original MD code. We have examined simulations with lattice sizes 4×4×4, 8×8×8, 10×10×10, 12×12×12, and 16×16×16, all executed over a span of 1,000 time steps. Across all configurations, these output parameters agreed within a tolerance of 10^{-9}. However, we did discover a problem with accumulating numerical error that caused the GPU results to drift from that of the CPU while testing extended

simulations of more than 4,000 time steps. This problem is analyzed further in the concluding remarks.

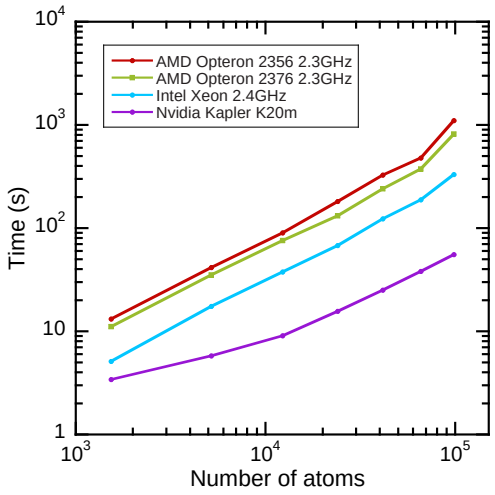

Figure 3: Concurrent construction of valid interactions.

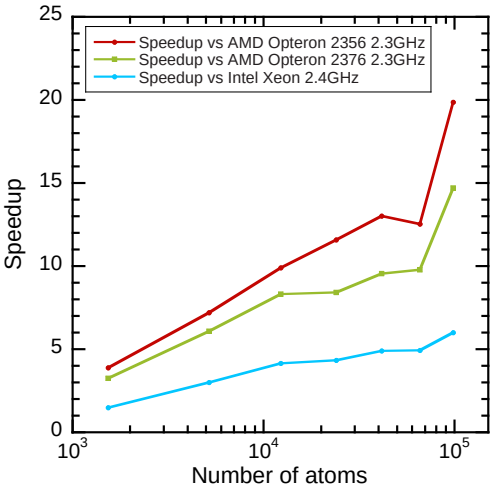

Figure 4: Speedup of GPU MD over reference CPU MD for varying numbers of atoms.

4.2 GPU Implementation versus the Baseline

The TD approach is doubtlessly designed to outperform its sequential origin since parallelization is employed aggressively. The advantage of the tuple-based restructuring of computations, however, is still nontrivial and must be investigated. To address this problem, we introduced a straightforward GPU implementation of the original MD code, which is based on particle decomposition (PD), as a baseline for comparison. As illustrated in Fig. 1, the TD approach separates the computation and identification of 2-body and 3-body interactions into different kernels. The baseline approach, on the contrary, fuses the two kernels back into a single one with each GPU thread scanning the interatomic potentials followed by calculation of the potentials. Thus, this code is simply a GPU-accelerated implementation of the original MD code on CPU.

The results are collected on Tesla K20m with simulations ran in the lattice sizes ranging from 4×4×4 to 32×32×32, or equivalently the number of atoms from 1,536 to 786,430. As shown in Fig. 5, our TD approach achieves an approximately 20% performance improvement over the baseline (PD). We observe that the wall-clock time for TD is higher than that of the baseline upon the first two lattice sizes. The likely reason is that our TD implementation launches two extra CUDA kernels than the baseline in each time step, and the overhead of initializing these kernels is relatively high on a small number of atoms, which needs to be large enough to leverage the computational capability of GPU.

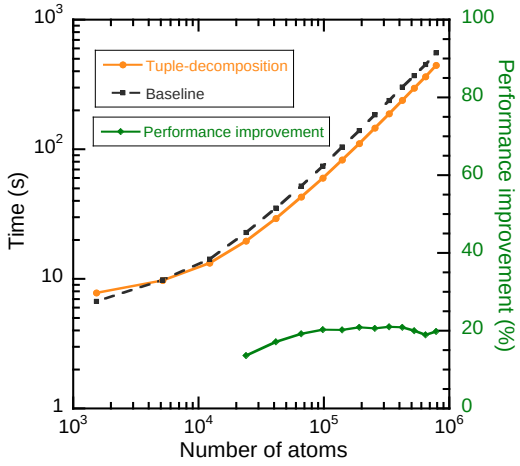

Figure 5: Wall-clock time of the TD approach (orange circles) and that of baseline (PD, black squares) as a function of the number of atoms. The figure also shows the performance improvement of the TD approach over the baseline as a function of the number of atom.

To better understand the performance characteristics of the TD and PD approaches, we next carry out performance profiling on the most compute-intensive CUDA kernels that account for approximately 90% of the total running time for both implementations, *i.e.*, *2body_inter* in baseline, and *gen_lists_2body_inter* and *compute_accel_2body_inter* in TD. We use the NVIDIA profiling tool, nvprof, for the measurements.

Fig. 6 compares the wall-clock time of the TD approach with that of the baseline for these most compute-intensive kernel executions. The total number of atoms is 786,432. As their names suggest, the test case consists of computing 2-body interactions in a unit time step. Here, the running time of each kernel is measured on average throughout the

entire CUDA threads, because a single thread may not execute the kernel when the interaction it carries falls out of cutoff radius. The TD approach (right column) is approximately 23% faster than the baseline (left column), which in turn echoes the global performance improvement of the former over the latter in Fig. 5. Within the TD approach, generation of the 2-body interaction list and actual computation of 2-body interactions using the list account for approximately identical computing times.

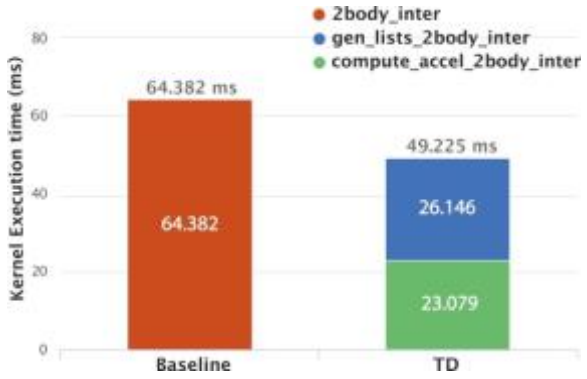

Figure 6: Wall-clock time on kernels for 2-body potential computation per time step for the baseline (left) and TD (right) approaches.

In addition, we investigate a major concern of our design mentioned in Section 3, i.e., unbalanced workloads over threads. We introduce "achieved occupancy" into our analysis as a metric to quantify the workload balance on GPU. More precisely, it is defined as the ratio of active warps on a streaming multiprocessor (SM) to the maximum number of active warps supported by the SM [30]. Table 1shows the achieved occupancy of kernels measured by nvprof. The compound achieved occupancy of TD, though with statistical variation, is apparently higher than the baseline, indicating that the procedure of fetching the interatomic pairs from the restructured tuple-list followed by calculating the interactions produces more balanced workload distributions over time compared with the baseline approach.

Table 1: Achieved occupancy of kernel execution.

Baseline		TD	
2body_inter	0.386	gen_lists_2body_inter	0.916
		compute_accel_2body_inter	0.461

We next perform a hardware-level analysis to unveil the reason behind the achieved occupancy for the three kernels in Table . Occupancy is limited by multiple factors such as shared memory usage. In this application, register usage per SM is found to be the determining factor. As Fig. 7 shows, the kernel *gen_list_2body_inter* uses 32 registers per thread, thereby maximizing the capacity of K20m to hold 64 warps active (2 blocks) per SM, while the other two kernels take up 41 registers per thread (41,984 for 1 block). The testing device, Tesla K20m, provides up to 65,536 registers for each block with up to 2 blocks supported on each SM. However, these two kernels use 41,984 registers for one block, thereby each SM can only have 1 block (32 warps as the red dot suggests) run in parallel. Thus, the achieved occupancy is limited to the upper bound of 0.5, which prevents it from fully utilizing the GPU. This analysis thus demonstrates that the register footprint plays a key role in dictating the GPU performance.

4.3 Dynamic Quadruplet Computation in ReaxFF

To test the performance portability of the TD approach to more general dynamic n-tuple computations, we consider first principles-informed RMD simulations based on the ReaxFF method [6, 9, 10]. The ReaxFF approach significantly reduces the computational cost of simulating chemical reactions, while reproducing the energy surfaces and barriers as well as charge distributions of quantum-mechanical calculations. RMD simulations describe formation and breakage of chemical bonds using reactive bond orders [6, 31, 32], while a charge-equilibration (QEq) scheme [33-35] is used to describe charge transfer between atoms. The ReaxFF interatomic forces involve up to quadruplets (n = 4) explicitly and sextuplets (n = 6) implicitly through the chain rule of differentiation. In this work, we test the TD approach in the quadruplet-interaction computations in a production RMD simulation program named XRMD [10].

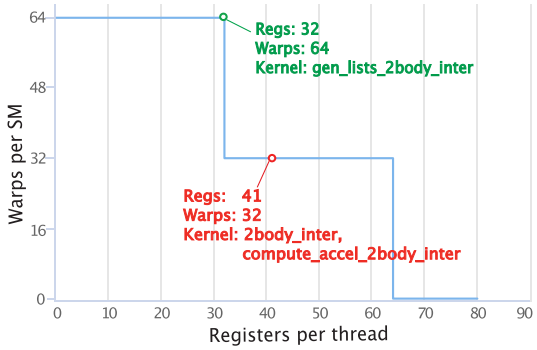

Figure 7: Relationship between the number of warps per SM and the number of registers per thread for kernels.

The quadruplet-interaction is one of the most computationally expensive functions in the ReaxFF method. It requires traversing deeply nested neighbor-list loops, within which the value of bond-order of a covalent bond or the combination of those is checked to be greater than predefined cutoff values, which can create significant

code branching. Our TD approach performs the neighbor-list traversal and the branch-condition evaluation on the CPU and the numerically intensive computations on the GPU, enabling us to take advantage of both host and GPU architectures. The GPU implementation of the TD approach for the computation of quadruplet interactions is shown in Algorithm 3.

Algorithm 3: GPU implementation of the generation of 4-body lists and computation of quadruplet forces in XRMD.

CUDA kernel, *gen_4body_list*

```
// tid is thread ID
// i, j, k, l are atom IDs and BO_ij, BO_ik, BO_lk are bond orders
   between pairs (i j), (i k) and (k l)
// ie4b is the list for four body and ne4b is interaction
   number for 4 body
   j ← tid
   k ← nbrlist[j,k1]
   if BO_jk > BO_cutoff
      i ← nbrlist[j,i1]
      if i ≠ k
         if BO_ij > BO_cutoff && BO_ij ×BO_jk > BO_cutoff
            l ← nbrlist[k, l1]
            if i ≠ l && j ≠ l
               if BO_kl > BO_cutoff && BO_jk×BO_kl > BO_cutoff
                  if BO_ij ×BO_jk^2 ×BO_kl > BO_cutoff2
                     ne4b++
                     ie4b[1:4][ne4b] ← (j, i1, k1, l1)
```

CUDA kernel, *gen_4body_list compute_4body_force*

```
   ne4b ← tid
      i ← ie4b[1][ne4b]
      j ← ie4b[2][ne4b]
      k ← ie4b[3][ne4b]
      l ← ie4b[4][ne4b]
      call E4b[i,j,k,l]
```

Fig. 8 compares the wall-clock time of the quadruplet interaction calculation using the TD and baseline (PD) approaches for three different problem sizes, i.e., the number of atoms N = 1,344, 4,536 and 10,752. For this kernel invocation, we employ one-dimensional thread assignment with the number of threads per grid, Nthreads, to be 32 and the number of grids to be the number of atomic quadruplets divided by Nthreads + 1. The figure shows that the TD approach consistently outperforms the baseline, providing nearly 1.3× speedup for the three system sizes.

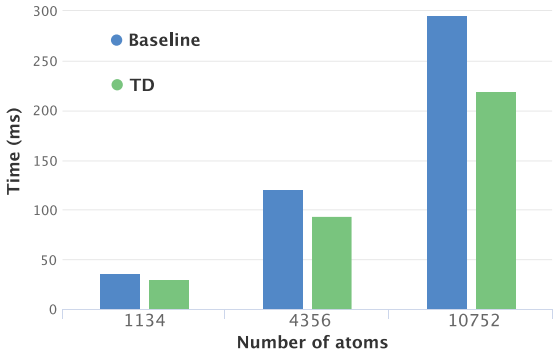

Figure 8: Timing comparison of the TD approach and baseline (PD) for the computation of quadruplet interactions in ReaxFF. The histogram shows wall-clock times per MD step in milliseconds averaged over 100 time steps. We use RDX material for this benchmark.

As stated in the introduction, our global-local separation scheme completely insulates the optimization of intranode computations of dynamic *n*-tuples from internode parallelization. In order to test the internode scalability of our parallel ReaxFF MD code, Fig. 9 shows the computing time per MD step as a function of the number of IBM Blue Gene/Q cores in both weak- and strong-scaling settings. The weak-scaling test simulates 86,016*P*-atom $C_3H_6N_6O_6$ molecular-crystal system on *P* cores, while the strong-scaling test simulates *N* = 4,227,858,432 atoms independent of *P*. The weak-scaling parallel efficiency is 0.977 for *P* = 786,432 for *N* = 67,645,734,912. The strong-scaling parallel efficiency is 0.886 for *P* = 786,432 for a much smaller system of *N* = 4,227,858,432.

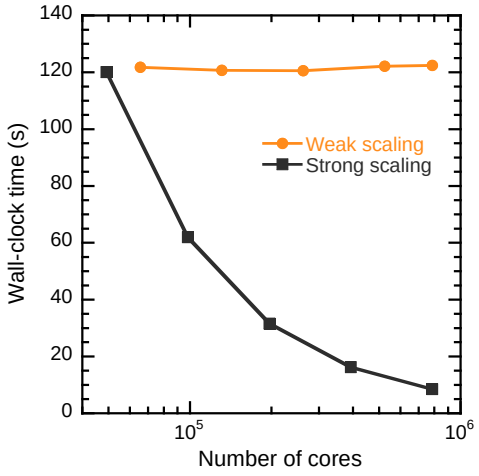

Figure 9: Strong and weak scaling of the parallel ReaxFF MD code on Blue Gene/Q.

5 CONCLUSIONS

In this paper, we have presented a new computational approach for performing many-body MD simulations on GPGPU. Our tuple-decomposition approach utilizes pipelining and efficient *in situ* generation of pair and triplet interaction lists to accelerate the application. Through a wide-ranging set of benchmarking tests, we have demonstrated that these relatively simple algorithmic changes provide significant performance speedups for varying simulation sizes and applications.

The algorithms we described for producing lists of pair and triplet interactions can be improved with further research. One possibility is combining our candidate two-body

interaction generator with the Lipscomb algorithm [24] for Verlet neighbor-list generation. Our approach addresses the major weakness in their algorithm — the generation of the master list — while their approach eliminates a need for synchronization through a global counter. We will synthesize these two methods to yield an exceptional hybrid approach.

We have also examined various techniques for fusing the pair and triplet interaction list generation into the same overall algorithm, but this proved to be too difficult as their requirements are slightly different. Further research could yield a way to integrate them in an elegant manner. This would simplify our computation model immensely.

Although the focus of this work is primarily on MD algorithm development on GPU, there are various optimizations that could be made to boost the performance even further. Memory coalescing could be improved in many places by restructuring the atomic state information stored in device global memory, and utilizing shared memory to coordinate reads among threads. With the former technique, the goal is to have adjacent threads in a thread block read data from adjacent addresses in global memory. The latter technique allows the efficient loading of strided array structures into local memory where the access times are much lower.

An intriguing idea that we plan to explore is imposing a sort order on the atoms, so that adjacent threads in the kernel tend to access adjacent atoms in the atom list. Theoretically, this could improve` memory coalescing. Determining a proper sort order is difficult, however, and enforcing it is expensive because it implies that a parallel sort be performed each time step since the atoms move throughout the simulation. A compromise may be to periodically sort the atoms and hope that that the benefits of temporarily improved coalescing outweigh the sorting cost.

ACKNOWLEDGMENTS

This research was supported as part of the Computational Materials Sciences Program funded by the U.S. Department of Energy (DOE), Office of Science, Basic Energy Sciences, under Award Number DE-SC00014607. Computation was performed at the Center for High Performance Computing of the University of Southern California.

REFERENCES

[1] D. Frenkel and B. Smit. 2001. Understanding Molecular Simulation. Academic Press, San Diego, CA.

[2] A. Rahman. 1964. Correlations in the motion of atoms in liquid argon. Physical Review, 136 (2A). A405-A411. https://doi.org/10.1103/PhysRev.136.A405

[3] M. Levitt. 2014. Birth and future of multiscale modeling for macromolecular systems (Nobel lecture). Angewandte Chemie International Edition, 53 (38). 10006-10018. 10.1002/anie.201403691

[4] S. Tsuneyuki, Y. Matsui, H. Aoki and M. Tsukada. 1989. New pressure-induced structural transformations in silica obtained by computer-simulation. Nature, 339 (6221). 209-211. 10.1038/339209a0

[5] F. Shimojo, I. Ebbsjo, R. K. Kalia, A. Nakano, J. P. Rino and P. Vashishta. 2000. Molecular dynamics simulation of structural transformation in silicon carbide under pressure. Physical Review Letters, 84 (15). 3338-3341.

[6] A. C. T. van Duin, S. Dasgupta, F. Lorant and W. A. Goddard. 2001. ReaxFF: a reactive force field for hydrocarbons. Journal of Physical Chemistry A, 105 (41). 9396-9409. 10.1021/jp004368u

[7] A. Nakano, R. K. Kalia, P. Vashishta, T. J. Campbell, S. Ogata, F. Shimojo and S. Saini. ACM/IEEE, 2001. Scalable atomistic simulation algorithms for materials research. Proceedings of Supercomputing, SC01. 10.1145/582034.582035

[8] P. Vashishta, R. K. Kalia, J. P. Rino and I. Ebbsjo. 1990. Interaction potential for SiO2 - a molecular-dynamics study of structural correlations. Physical Review B, 41 (17). 12197-12209. 10.1103/PhysRevB.41.12197

[9] A. Nakano, R. K. Kalia, K. Nomura, A. Sharma, P. Vashishta, F. Shimojo, A. C. T. van Duin, W. A. Goddard, R. Biswas, D. Srivastava and L. H. Yang. 2008. De novo ultrascale atomistic simulations on high-end parallel supercomputers. International Journal of High Performance Computing Applications, 22 (1). 113-128. 10.1177/1094342007085015

[10] K. Nomura, P. E. Small, R. K. Kalia, A. Nakano and P. Vashishta. 2015. An extended-Lagrangian scheme for charge equilibration in reactive molecular dynamics simulations. Computer Physics Communications, 192. 91-96. 10.1016/j.cpc.2015.02.023

[11] D. C. Rapaport. 1988. Large-scale molecular-dynamics simulation using vector and parallel computers. Computer Physics Reports, 9 (1). 1-53. 10.1016/0167-7977(88)90014-7

[12] A. Nakano, P. Vashishta and R. K. Kalia. 1993. Parallel multiple-time-step molecular-dynamics with 3-body interaction. Computer Physics Communications, 77 (3). 303-312. 10.1016/0010-4655(93)90178-F

[13] S. Plimpton. 1995. Fast parallel algorithms for short-range molecular dynamics. Journal of Computational Physics, 117 (1). 1-19. 10.1006/jcph.1995.1039

[14] L. Kale, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan and K. Schulten. 1999. NAMD2: greater scalability for parallel molecular dynamics. Journal of Computational Physics, 151 (1). 283-312. 10.1006/jcph.1999.6201

[15] D. E. Shaw. 2005. A fast, scalable method for the parallel evaluation of distance-limited pairwise particle interactions. Journal of Computational Chemistry, 26 (13). 1318-1328. 10.1002/jcc.20267

[16] D. A. Reed and J. Dongarra. 2015. Exascale computing and big data. Communications of the ACM, 58 (7). 56-68. 10.1145/2699414

[17] K. Nomura, R. K. Kalia, A. Nakano, P. Vashishta, K. Shimamura, F. Shimojo, M. Kunaseth, P. C. Messina and N. A. Romero. IEEE/ACM, 2014. Metascalable quantum molecular dynamics

simulations of hydrogen-on-demand. Proceedings of Supercomputing, SC14. 661-673. 10.1109/SC.2014.59
[18] F. Shimojo, R. K. Kalia, M. Kunaseth, A. Nakano, K. Nomura, S. Ohmura, K. Shimamura and P. Vashishta. 2014. A divide-conquer-recombine algorithmic paradigm for multiscale materials modeling. Journal of Chemical Physics, 140 (18). 18A529. 10.1063/1.4869342
[19] N. A. Romero, A. Nakano, K. Riley, F. Shimojo, R. K. Kalia, P. Vashishta and P. C. Messina. 2015. Quantum molecular dynamics in the post-petaflops era. IEEE Computer, 48 (11). 33-41.
[20] J. Mellor-Crummey, D. Whalley and K. Kennedy. 2001. Improving memory hierarchy performance for irregular applications using data and computation reorderings. International Journal of Parallel Programming, 29 (3). 217-247. 10.1023/A:1011119519789
[21] M. Kunaseth, R. K. Kalia, A. Nakano, K. Nomura and P. Vashishta. ACM/IEEE, 2013. A scalable parallel algorithm for dynamic range-limited n-tuple computation in many-body molecular dynamics simulation. Proceedings of Supercomputing, SC13.
[22] J. P. Walters, V. Balu, V. Chaudhary, D. Kofke and A. Schultz. ISCA, 2008. Accelerating molecular dynamics simulations with GPUs. Proceedings of International Conference on Parallel and Distributed Computing and Communication Systems (PDCCS 2008). 44-49.
[23] W. M. Brown, P. Wang, S. J. Plimpton and A. N. Tharrington. 2011. Implementing molecular dynamics on hybrid high performance computers - short range forces. Computer Physics Communications, 182 (4). 898-911. 10.1016/j.cpc.2010.12.021
[24] T. J. Lipscomb, A. Zou and S. S. Cho. ACM, 2012. Parallel Verlet neighbor list algorithm for GPU-optimized MD simulations. Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine, (BCB '12). 321-328. 10.1145/2382936.2382977
[25] W. M. Brown and M. Yamada. 2013. Implementing molecular dynamics on hybrid high performance computers-three-body potentials. Computer Physics Communications, 184 (12). 2785-2793. 10.1016/j.cpc.2013.08.002
[26] S. B. Kylasa, H. M. Aktulga and A. Y. Grama. 2014. PuReMD-GPU: a reactive molecular dynamics simulation package for GPUs. Journal of Computational Physics, 272. 343-359. 10.1016/j.jcp.2014.04.035
[27] A. Nakano, R. K. Kalia and P. Vashishta. 1994. Multiresolution molecular-dynamics algorithm for realistic materials modeling on parallel computers. Computer Physics Communications, 83 (2-3). 197-214.
[28] M. P. Allen and D. J. Tildesley. 1987. Computer Simulation of Liquids. Oxford University Press, Oxford, UK.
[29] W.-m. W. Hwu. 2011. GPU Computing Gems Jade Edition. Morgan Kaufmann, Waltham, MA.
[30] http://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm
[31] J. Tersoff. 1989. Modeling solid-state chemistry: interatomic potentials for multicomponent systems. Physical Review B, 39 (8). 5566-5568. 10.1103/PhysRevB.41.3248.2
[32] D. W. Brenner. 1990. Empirical potential for hydrocarbons for use in simulating the chemical vapor-deposition of diamond films. Physical Review B, 42 (15). 9458-9471.
[33] A. K. Rappe and W. A. Goddard. 1991. Charge equilibration for molecular-dynamics simulations. Journal of Physical Chemistry, 95 (8). 3358-3363. 10.1021/j100161a070
[34] F. H. Streitz and J. W. Mintmire. 1994. Electrostatic potentials for metal-oxide surfaces and interfaces. Physical Review B, 50 (16). 11996-12003. 10.1103/PhysRevB.50.11996
[35] S. W. Rick, S. J. Stuart and B. J. Berne. 1994. Dynamical fluctuating charge force-fields - application to liquid water. Journal of Chemical Physics, 101 (7). 6141-6156. 10.1063/1.468398