2012

PuReMD-GPU: A Reactive Molecular Dynamic Simulation Package for GPUs

Sudhir B. Kylasa
Purdue University, skylasa@purdue.edu

Ananth Grama
Purdue University - Main Campus

Hasan Aktulga
Purdue University - Main Campus

Report Number:
13-005

PuReMD-GPU: A Reactive Molecular Dynamic Simulation Package for GPUs

S. B. Kylasa[a,1,*], H.M. Aktulga[c], A.Y. Grama[b,1,**]

[a] *Department of Elec. And Comp. Eng., Purdue University, West Lafayette, Indiana 47907 USA*
[b] *Department of Computer Science, Purdue University, West Lafayette, Indiana 47907 USA*
[c] *Lawrence Berkeley National Laboratory, 1 Cyclotron Rd, MS 50F-1650, Berkeley, CA 94720*

Abstract

We present an efficient and highly accurate GP-GPU implementation of our community code, PuReMD, for reactive molecular dynamics simulations using the ReaxFF force field. PuReMD and its incorporation into LAMMPS (Reax/C) is used by a large number of research groups world-wide for simulating diverse systems ranging from bio-membranes to explosives (RDX) at atomistic level of detail. The sub-femtosecond time-steps associated with ReaxFF strongly motivate significant improvements to per-timestep simulation time through effective use of GPUs. This paper presents, in detail, the design and implementation of PuReMD-GPU, which enables ReaxFF simulations on GPUs, as well as various performance optimization techniques we developed to obtain high performance on state-of-the-art hardware. Comprehensive experiments on model systems (bulk water and amorphous silica) are presented to quantify the performance improvements achieved by PuReMD-GPU and to verify its accuracy. In particular, our experiments show up to 16× improvement in runtime compared to our highly optimized CPU-only single-core ReaxFF implementation. PuReMD-GPU is a unique production code, and is currently available on request from the authors.

1. Introduction

There has been significant effort aimed at atomistic modeling of diverse systems – ranging from materials processes to biophysical phenomena. Parallel formulations of these methods have been shown to be among the most scalable applications. Classical MD approaches typically rely on static bonds and fixed partial charges associated with atoms. These constraints limit their applicability to non-reactive systems. A number of recent efforts have addressed these limitations [1–5]. Among these, ReaxFF, a novel reactive force field developed by van Duin et al.[6] bridges quantum-scale and classical MD approaches by explicitly modeling

*Corresponding author
**Principal corresponding author
Email addresses: skylasa@purdue.edu (S. B. Kylasa), hmaktulga@lbl.gov (H.M. Aktulga), ayg@cs.purdue.edu (A.Y. Grama)

[1] This work is supported by the US National Science Foundation and Department of Energy.

bond activity (reactions) and the distribution of partial charges. The flexibility and transferability of the force field allows ReaxFF to be easily extended to systems of interest. Indeed, ReaxFF has been successfully applied to diverse systems [6–9].

ReaxFF is a classical MD method in the sense that atomic nuclei, together with their electrons, are modeled as basis points. Interactions among atoms are modeled through suitably parameterized functions and atoms obey the laws of classical mechanics. Accurately modeling chemical reactions while avoiding discontinuities on the potential energy surface, however, requires more complex mathematical formulations than those in classical MD methods (bond, valence angle, dihedral, van der Waals potentials). In a reactive environment in which atoms often do not achieve their optimal coordination numbers, ReaxFF requires additional modeling abstractions such as lone pair, over/under-coordination, and three-body and four-body conjugation potentials, which increase its computational complexity. This increased computational cost of bonded interactions (reconstructing all bonds, three-body and four-body structures at each time-step) approaches the cost of nonbonded interactions for ReaxFF. In contrast, for typical MD codes, the time spent on bonded interactions is significantly lower than that spent on nonbonded interactions [10].

An important part of ReaxFF is the charge equilibration procedure. This procedure recomputes partial charges on atoms to minimize the electrostatic energy of the system. Charge equilibration is mathematically formulated as the solution of a large linear system of equations, where the underlying matrix is sparse and symmetric, with dimension equal to the number of atoms. We note that the number of atoms in a simulation may range from thousands to millions and due to the dynamic nature of the system, a different set of equations needs to be solved at each timestep. An accurate solution of the charge equilibration problem is highly desirable, as the partial charges on atoms significantly impact forces and the total energy of the system. Suitably accelerated Krylov subspace methods are used for this purpose [11]. Since the timestep for ReaxFF is typically an order of magnitude smaller than conventional MD (tenth of femtoseconds as opposed to femtoseconds), scaling the solve associated with charge equilibration is a primary design consideration for GPU/parallel formulations. Note that partial charges on atoms are fixed in most classical MD formulations. Consequently, this is not a consideration for conventional methods.

Our prior work in the area led to the development of the PuReMD (**Pu**rdue **Re**active **M**olecular **D**ynamics) code, along with a comprehensive evaluation of its performance and its application to diverse systems [12–14]. PuReMD incorporates several algorithmic and numerical innovations to address significant computational challenges posed by ReaxFF. It achieves excellent per timestep execution times, enabling nanosecond-scale simulations of large reactive systems. Using fully dynamic interaction lists that adapt to the specific needs of simulations, PuReMD achieves low memory footprint. Our tests demonstrate that PuReMD is up to 5× faster than competing implementations, while using significantly lower memory. PuReMD has also been integrated with the LAMMPS software package (available as the User-ReaxC package).

PuReMD and its LAMMPS version have been used extensively in the scientific community to simulate phenomena ranging from oxidative stress on biomembranes (lipid bilayers) to explosives (RDX) [15]. It has a large number of downloads and an active developer community. While the code was designed from the ground up to be parallel, a significant segment of the user community uses it on serial platforms. On state-of-the-art serial platforms (Intel i7, 6 GB), a single timestep for a typical system (6K water atoms) takes about a second of simulation time. Timesteps in ReaxFF are of the order of 0.2 fs and several important physical analyses require simulations spanning nanoseconds (5×10^6 timesteps). This implies roughly 1400 hours (58 days) of compute time using PuReMD on a serial platform. These considerations provide compelling motivation for GPU acceleration of PuReMD. An order of magnitude improvement in speed in a production code would present tremendous opportunities for a large scientific community.

The highly dynamic nature of interactions and memory footprint, diversity of kernels underlying non-bonded and bonded interactions, the charge equilibration procedure, the complexity of force functionals, and the high accuracy requirement pose significant challenges for efficient GPU implementations of ReaxFF. Effective use of shared memory to avoid frequent global memory accesses and configurable cache to exploit spatial locality during scattered memory operations are essential to the performance of various kernels. These kernels also need to be optimized to utilize GPUs' capability to spawn thousands of threads, and coalesced memory operations are necessary to enhance the performance of specific kernels. The high cost of double precision arithmetic on conventional GPUs must be effectively masked through these optimizations. These requirements may be traded-off with increased memory footprint to further enhance performance.

In this paper, we present in detail, the design and implementation of all phases of PuReMD on GPUs – including non-bonded and bonded interactions, as well as the charge equilibration procedure. We carefully analyze and trade-off various memory and computation costs to derive efficient implementations. Comprehensive experiments with model systems (bulk water and amorphous silica) on a state-of-the-art GPU hardware are presented to quantify the performance and accuracy of PuReMD-GPU. Our experiments show over 16× improvement in runtime compared to our highly optimized CPU-only single core PuReMD implementation on typical GPU-equipped desktop machines (Core i7, 24 GB RAM and Nvidia 2075 GPU). These speedups represent important scientific potential for diverse simulations.

2. Atomistic Simulations Using ReaxFF

Conventional molecular dynamics techniques cannot simulate chemical activity, since reactions correspond to bond breaking and formation. In ReaxFF, the bond structure of the system is updated at every timestep using a bond order potential. This dynamic bond structure, coupled with the recomputation of partial charges through charge equilibration represents significant added computational complexity for ReaxFF implementations. We summarize various aspects of ReaxFF here, referring the reader to [6, 16] for a detailed description.

In ReaxFF, the bond order between a pair of atoms, i and j reflects the strength of the bond between the two atoms. It is computed in two steps – in the first step, an uncorrected bond order is computed, and in the second step, it is corrected. The uncorrected bond order is given by Eq. 1, which uses the types of atoms and their distance:

$$BO_{ij}^{\alpha'}(r_{ij}) = exp\left[a_\alpha \left(\frac{r_{ij}}{r_{0\alpha}}\right)^{b_\alpha}\right] \tag{1}$$

In this equation, α' corresponds to uncorrected $\sigma-\sigma$, $\sigma-\pi$, or $\pi-\pi$ bonds, a_α and b_α are parameters associated with the bond type, and $r_{0\alpha}$ is the optimal length for the bond type. The total uncorrected bond order (BO'_{ij}) is computed by summing the $\sigma-\sigma$, $\sigma-\pi$, and $\pi-\pi$ bond orders, which may actually be different from its ideal coordination number (valence). For this reason, it is necessary to correct the bond orders as follows:

$$BO_{ij} = BO'_{ij} \cdot f_1(\Delta'_i, \Delta'_j) \cdot f_4(\Delta'_i, BO'_{ij}) \cdot f_5(\Delta'_j, BO'_{ij}) \tag{2}$$

Here, Δ'_i is the deviation of atom i from its optimal coordination number, $f_1(\Delta'_i, \Delta'_j)$ enforces over-coordination correction, and $f_4(\Delta'_i, BO'_{ij})$, together with $f_5(\Delta'_j, BO'_{ij})$ account for 1-3 bond order corrections. Only corrected bond orders are used in energy and force computations in ReaxFF. Once bond orders are computed, other potentials can be computed in a manner similar to traditional MD methods. However, due to the dynamic bonding scheme of ReaxFF, potentials must be modified to ensure smooth potential energy surfaces as bonds form or break.

For the purpose of this discussion, we classify the potential terms into bonded and non-bonded terms. Bonded terms include two- (bond length), three- (bond angle), four-body (torsion), and hydrogen bond terms. Non-bonded terms include van der Waals and Coulomb terms. An important consideration here is that all terms, including the Coulomb term, are typically truncated at 10-12 Å. The total energy of the system is the sum of individual energy terms:

$$\begin{aligned} E_{system} = & E_{bond} + E_{lp} + E_{over} + E_{under} \\ & + E_{val} + E_{pen} + E_{3conj} \\ & + E_{tors} + E_{4conj} + E_{H-bond} \\ & + E_{pol} + E_{vdW} + E_{Coulomb} \end{aligned} \tag{3}$$

In terms of computational cost, E_{bond} is computed for each atom with respect to all the atoms it is bonded to. We maintain a separate list of all bonded atoms for each atom. Since bond interactions take

place within a distance of 3-5 Å, typically, the number of atoms in the bond list of an atom is small. E_{lp}, E_{over}, and E_{under} are correction terms that involve computations based on the actual coordination number of an atom with respect to its ideal coordination number. The computation of these terms requires a single pass over the atom list, which is not expensive. E_{val}, E_{pen}, and E_{3conj} are computed for pairs of bonds that form a valence angle for each atom. To determine the potential three-body interactions of the form (i,j,k), we iterate twice over the bond list of the central atom j. This computation may be expensive for highly coordinated systems. We maintain a list of valence angles for each bond, which is used later for computing E_{tors} and E_{4conj}. E_{tors} and E_{4conj} are computed for a pair of valence angles between two atoms j and k such that valence angles (i,j,k) and (j,k,l) exist. Computationally, this can be expensive as the number of valence angles increases. E_{H-bond} is computed between a covalently bonded H atom and certain types of atoms within a 6-7 Å cutoff radius. Due to the relatively large cutoff distance, hydrogen bonds can be expensive to compute, if present. E_{pol} is computed with a single pass over the atom list. E_{vdW} and $E_{Coulomb}$ are computed for each atom with respect to all its neighboring atoms in a 10-12 Å distance. Since the list of neighboring atoms within this cutoff distance is typically much larger compared to the number of bonds of an atom, these two terms are the most expensive to compute. An important and expensive precursor to the $E_{Coulomb}$ computation is the solution of the charge equilibration problem, which is discussed in detail in Section 4.3.

3. CUDA Overview and Performance Considerations

GPU architectures typically comprise of a set of multiprocessor units called streaming multiprocessors (SMs), each one containing a set of processor cores called streaming processors (SPs). There are various levels of memory available in GPUs, including: (i) off-chip global memory, (ii) off-chip local memory, (iii) off-chip constant memory with on-chip cache, (iv) off-chip texture memory with on-chip cache, (v) on-chip shared memory, and (vi) on-chip registers. Effectively handling data across the memory hierarchy is essential to application performance.

Global memory is typically large and has high latency, about 400-800 cycles [17]. Shared memory is present at each SM and has lower latency, 10-20 cycles per access. Each SM has a set of registers used for storing automatic (local) variables. The constant and texture memories reside in (off-chip) global memory and have an on-chip read-only cache.

Computational elements of algorithms in the CUDA programming model are called kernels. Kernels can be written in different programming languages. Once compiled, kernels consist of threads that execute the same instructions simultaneously – the Single Instruction Multiple Thread (SIMT) execution model. Multiple threads are grouped into thread blocks. All threads in a thread block are scheduled to run on a single SM. Threads within a block can cooperate using the available shared memory. Thread blocks are divided into warps of 32 threads. A warp is a fundamental unit of dispatch within a block. Thread blocks

are grouped into grids, each of which executes a unique kernel. Thread blocks and threads have identifiers (IDs) that specify their relationship to the kernel. These IDs are used within each thread as indices to their respective input and output data, shared memory locations, etc. [18]. Control instructions can significantly impact instruction throughput by causing threads of the same warp to diverge; that is, they follow different execution paths. If this happens, different execution paths must be serialized, increasing the total number of instructions executed for this warp. When all execution paths have completed, threads converge back to the same execution path [19].

Each SM executes multiple blocks simultaneously to hide the memory access latency. Our development platform, the Tesla C2075 SM can run up to 8 blocks simultaneously. Each SM has a finite number of registers and limited amount of shared memory. The availability of these resources dictate the total number of warps that can be scheduled on each SM. Increasing either the shared memory used by each thread block, or the number of registers, decreases the occupancy of the kernel and vice-versa. Occupancy of the kernel is defined as the ratio of number of active warps to maximum active warps. The maximum number of active warps varies depending on the GPU model.

Memory operations are handled per half-warp (16 threads). If all threads access non-consecutive memory locations, then multiple memory reads/writes are issued. In the worst case, up to 16 memory operations may be issued. If all threads access consecutive memory locations, then fewer memory operations are issued. In the best case, just one memory read/write is issued if all the data fits on the memory bus. This coalesced memory access plays a vital role in the performance of the kernel.

Each floating-point arithmetic operation involves rounding. Consequently, the order in which arithmetic operations are performed is important. During parallel execution of kernels, the order of operations is potentially changed. Therefore floating point operation results obtained on GPUs may not match those obtained from a sequential CPU run bit-by-bit [19, 20]. We refer readers to [20] for detailed discussion on floating-point arithmetic on NVIDIA GPUs.

4. PuReMD-GPU Design

We describe PuReMD-GPU in detail and discuss major design decisions in its implementation. PuReMD-GPU is derived from the serial version of PuReMD [12], which is targeted towards typical desktop environments. Figure 1 presents key functional blocks and control flow in the PuReMD-GPU implementation. We identify the following algorithmic components of PuReMD: initialization, generating neighbors, computing bond orders, energy, and forces, and recomputing partial charges on atoms. Each of these components is considerably more complex compared to a non-reactive classical MD implementation. It is this complexity that forms the focus of our GPU implementation.

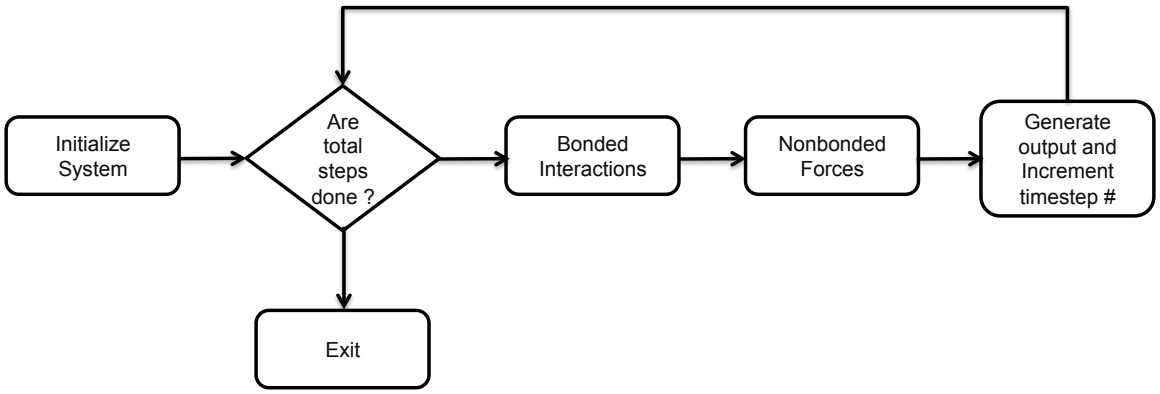


Figure 1: High level flow chart of the major steps involved in a typical ReaxFF algorithm

4.1. Initialization

During initialization, PuReMD and PuReMD-GPU construct a number of data structures used in subsequent processing. These include the neighbor list, the bond list, hydrogen bond list, and coefficients of charge equilibration (QEq) matrix. Note that the relations underlying the lists are themselves symmetric, i.e., if atom x is a neighbor of atom y, then atom y is also a neighbor of atom x. Likewise, if atom x has a bond with atom y, atom y also has a bond with atom x; and so on. This symmetric relation may be stored redundantly, i.e., both copies of the symmetric relation are stored for computational efficiency. Alternately, this data may be stored non-redundantly, i.e., only one copy of the symmetric relation is stored (by convention, the ordered pair from lower to higher indexed atom). The latter is done for memory efficiency, while the former is important for exposing more parallelism at the expense of additional memory. In PuReMD, where each process uses a strictly sequential execution model for its share of computations, the non-redundant storage option is preferred. However, PuReMD-GPU stores redundant versions of all lists. This greatly simplifies implementation while yielding better performance at the cost of increased memory footprint.

The first step in the initialization process for both PuReMD and PuReMD-GPU is to estimate the number of entries in the neighbor list, bond list, hydrogen bond list, and the QEq matrix. These estimates are used to dynamically allocate memory; the actual memory allocated is an overestimate to avoid repeated reallocations. In the course of the simulation, if the actual memory utilization hits a high-water mark, corresponding data structures are reallocated.

4.1.1. Computing Neighbor Lists

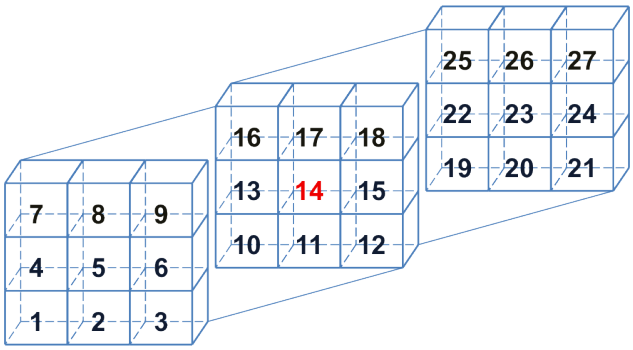
Both PuReMD and PuReMD-GPU first construct the list of neighbors for each atom in the input system. Other data structures are rebuilt at each step from the neighbor list. A typical atom may have several hundred atoms in its neighbor list.

In ReaxFF, both bonded and non-bonded interactions are truncated after a cut-off distance (which is

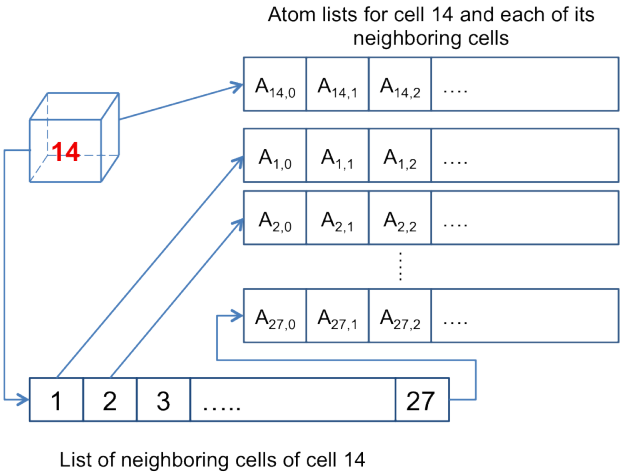
typically 4-5 Å for bonded interactions and 10-12 Å for non-bonded interactions). Given this truncated nature of interactions, we use a procedure based on "binning" (or link-cell method) [21]. First, a 3D grid structure is built by dividing the simulation domain into small cells. Atoms are then binned into these cells based on their spatial coordinates. It is easy to see that potential neighbors of an atom are either in the same cell or in neighboring cells that are within the neighbor cut-off distance r_{nbrs} of its own cell. The neighbor list for each atom is populated by iterating over cells in its neighborhood and testing whether individual atoms in neighboring cells are within the cutoff distance. Using this binning method, we can realize O(k) neighbor generation complexity for each atom, where k is the average number of neighbors of any atom. It is important to note that the cell size must be carefully selected to optimize performance.

In PuReMD-GPU the processing of each atom may be performed by one or more threads. The case of a single thread per atom is relatively straightforward. Each thread runs through all neighboring cells of the given atom and identifies neighbors. These neighbors are inserted into the neighbor list of the atom. Since shared data structures (cells and their atoms) are only read and there are no shared writes in this case, no synchronization is required. The case of multiple threads per atom, on the other hand requires suitable partitioning of the computation, as well as synchronization for writes into the neighbor list. This process is illustrated in Figure 2 for the case of four threads per atom. Each thread takes as argument the atom-id for which it computes neighbors. This atom-id is used by all threads to concurrently identify neighbor cells. Each of these neighbor cells is processed by the threads in a lock-step fashion. In the case of four threads, every fourth atom in a neighbor cell is tested by the same thread; i.e., thread 0 is responsible for testing whether atoms 0, 4, 8, and so on are in the neighbor list, thread 1 is responsible for testing atoms 1, 5, 9, .. In this scenario, one may note that atoms 0, 1, 2, and 3 of the first neighbor cell will be processed concurrently by threads with ids 0, 1, 2, and 3 in the first step; atoms 4, 5, 6, and 7 are processed concurrently by threads 0, 1, 2, and 3, in the second step respectively, and so on.

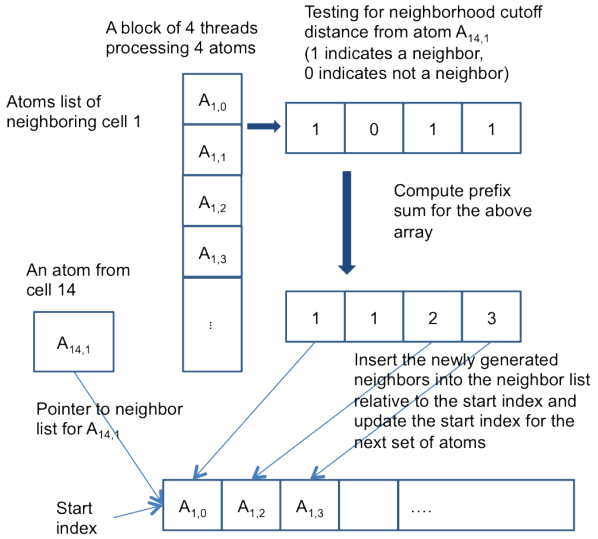
Since the threads themselves are executed in a synchronized SIMT fashion, a single SIMT execution step of four threads results in an integer-vector of length 4. A one in this vector corresponds to the fact that the atom in the cell belongs to the neighbor list. At this point, all atoms that have 1's in this vector must be inserted into the neighbor list concurrently, which requires synchronization. This (potential) concurrent write is handled by computing a prefix sum of the integer vector. The prefix sum vector gives each thread the offset for inserting into the neighbor list. Once this offset is computed, all writes are to independent locations in the shared memory. The writes are performed and the next block of atoms in the same neighbor cell is processed. In this way, computation proceeds until all atoms in all neighbor cells are processed. The pseudocode for this computation is described in Algorithm 1. CUDA function `sync_threads`, which is a thread barrier for all threads in a thread-block, is used to synchronize threads working on the same atom. Lines 2, 3 and 4 compute `my_atom`, corresponding cell `my_cell`, and `lane_id`, which is a number between `0` and `threads_per_atom`. All threads working on an atom loop through the neighboring cells of



(a) Input system divided into subdomains, cell 14 and its neighboring cells



(b) Data structures for cells, atoms-list and neighbor-list of each atom



(c) Block of 4 threads processing an atoms-list of a neighboring cell to build the neighbor-list of an atom from cell 14

Figure 2: Neighbor list generation in PuReMD-GPU

Algorithm 1: Multiple threads per atom kernel for neighbor list generation

```
   Data: atoms
   Result: neighbor list's of all the atoms
 1 thread_id = blockIdx.x * blockDim.x + threadIdx.x;
 2 my_atom = GetAtomId (blockIdx.x, threadIdx.x, threads_per_atom);
 3 lane_id = thread_id & (threads_per_atom - 1);
 4 my_cell = get_cell (my_atom);
 5 end_index = get_end_index (my_atom, neighbors_list);
 6 sync_threads ();
 7 foreach cell in the neighboring cells of my_cell do
 8 |   if cell is within cutoff distance of my_cell then
 9 |   |   total_iterations = get_iterations(# of atoms in cell, threads_per_atom);
10 |   |   iterations = 0;
11 |   |   nbr_gen = false;
12 |   |   start = starting atom of cell;
13 |   |   atom = start + lane_id;
14 |   |   while iterations less than total_iterations do
15 |   |   |   if atom and my_atom are far neighbors then
16 |   |   |   |   nbr_gen = true;
   |   |   |   end
17 |   |   |   if nbr_gen then
18 |   |   |   |   designated thread computes the offset (relative to end_index) in the neighbor list for
   |   |   |   |   each thread of this atom;
19 |   |   |   |   insert atom in the neighbor list of my_atom ;
20 |   |   |   |   update end_index based on # of neighbors generated by all the threads for atom;
   |   |   |   end
21 |   |   |   nbr_gen = false;
22 |   |   |   iterations ++;
23 |   |   |   atom += threads_per_atom;
   |   |   end
   |   end
   end
24 sync_threads ();
25 if lane_id == 0 then
26 |   set_end_index (my_atom, end_index, neighbor_list);
   end
```

`my_cell` checking for the cutoff distance. If this condition is satisfied, each thread picks an atom from `cell`'s atom list and checks whether `my_atom` and `atom` are neighbors on line 15. Threads that generate a neighbor elect a designated thread (thread with least thread-id) to compute the offset within the neighbor list relative to the `end_index` (where the previous sets of writes into the neighbor list ended) on line 18. Newly generated neighbors are inserted into the `neighbor_list` using the offset and a designated thread updates the `end_index` on line 20. Each thread increments the iteration count and picks up the next atom from `cell` at lines 22 and 23. Single Instruction Multiple Thread execution model of CUDA runtime guarantees that all threads execute the same control instruction (lines 7 through 23) within the current block. At the end of the main loop on lines 25 and 26 `end_index` is updated in the neighbor list for each atom.

4.1.2. Computing Bond List, Hydrogen-bond List and QEq Matrix

Both PuReMD and PuReMD-GPU maintain redundant bond lists, i.e., bond information is maintained by both atoms at either end of a bond. PuReMD iterates over the neighbor list of each atom and generates the bond lists of both the current atom and neighboring atom at the same time, since neighbor lists are non-redundant. In the serial PuReMD code, this is relatively easy, since we can safely modify the bond lists of both atoms at the same time without having to worry about synchronization. In PuReMD-GPU, since neighbor lists are redundantly stored, computation at each atom is only responsible for inserting into the lists of its own atom. The concurrency in constructing bond lists, hydrogen bond lists, and the QEq matrix can be viewed along two dimensions: (i) the three tasks (bond lists, hydrogen bond lists and one QEq entry) associated with each atom pair can be performed independently; (ii) the processing of each atom pair (the source atom and the atom in the neighbor list) can be performed independently. Indeed, both of these elements of concurrency can also be used. The base implementation of the first form creates three kernels for each atom – one for bond list, one for hydrogen bond list, and one for QEq. However, this implementation does not yield good performance because the neighbor list is traversed multiple times, leading to poor cache performance. For this reason, we roll all three kernels into a single kernel which handles all three lists. With this kernel, one may still partition the neighbor list, as was done in our implementation of neighbor list construction. However, there are several key differences here – the quantum of computation associated with an atom pair is larger here, since the combined kernel is more sophisticated. On the other hand, the number of atom pairs is smaller, since the neighbor list is smaller than the potential list of neighbors in cells examined in the previous case. Finally, there are subtle differences in synchronizations. For instance the number of synchronized insertions into the lists is much smaller (the number of bonds is relatively small). For these reasons, PuReMD-GPU relies on a single kernel for each atom and all processing associated with an atom. Traversal through the entire neighbor list and insertions are handled by a single thread.

4.2. Implementation of Bonded Interactions

Bonded interactions in ReaxFF consist of bond order, bond energy, lone pair, over- and under-coordination, three-body, four-body and hydrogen bond interactions. All these interactions, except hydrogen bond, iterate over the bond list of each atom to compute the effective force and energy due to respective interactions. In addition to iterating over the bond list, the three-body interactions kernel generates the three-body list, which is used by the four-body interactions kernel during its execution. Hydrogen bond interaction iterates over the hydrogen bond list to compute effective force and energy E_{H-bond}, if hydrogen bonds are present in the system.

The number of entries in the bond list for each atom is small compared to the number of entries in the neighbor list and hydrogen bond list. Allocating multiple threads per atom (to iterate over the bond list of each atom) would result in a very few coalesced memory operations. This would imply creation of a large number of thread blocks, where each thread block performs the computations of only a few atoms, resulting in poor overall performance of the kernel. For this reason, we use a single thread per atom for bond order, bond energy, lone pair, over- and under-coordination, three-body, and four-body interaction kernels.

The three-body interaction kernel is complex. It requires the use of a large number of registers and contains several branch instructions which can potentially cause thread-divergence. The available set of hardware registers in a GPU is shared by the entire SM. Consequently, if a kernel uses a large number of registers, the number of active warps (groups of schedulable threads in a block) is limited. Our experiments with this kernel showed that in typical cases, we could only achieve an effective occupancy (ratio of active to maximum schedulable warps) of 25.3% out of the maximum possible occupancy of 33%. The limiting factor for the performance of this kernel is the number of registers used per thread (64 registers/thread). Allocating multiple threads per atom increases the total number of blocks for this kernel, and, as a consequence, decreases the kernel's performance as the system size increases.

For three-body interactions, a thread assigned to the central atom j iterates twice over the bond list of j to generate the three-body list for it. Using multiple threads per atom increases the number of thread blocks to be executed and in each of these thread blocks, each thread spends a number of cycles to fetch spilled variables from global memory, which decreases the kernel's performance. Due to occupancy limitations discussed earlier, larger number of thread blocks can not be scheduled onto SMs. Similar issues arise for the four-body interactions kernel as well. For these reasons, PuReMD-GPU creates one thread per atom kernels for both the three-body and four-body interactions. We present detailed results in Section 5 demonstrating these overheads and motivating our eventual design choices.

Hydrogen bond interaction, if present, is the most expensive of all the bonded interactions. This kernel iterates over the hydrogen bond list. Since the cutoff distance for hydrogen bond interactions is typically 6-7.5 Å, the number of hydrogen bond entries per atom can be quite large. This indicates that multiple threads per atom would yield better performance because of coalesced global memory accesses. The pseudocode for

hydrogen bonds computation is described in Algorithm 2. Lines 1, 2 declare shared memory arrays used in the reduce operation to compute the force on each atom and hydrogen bond energy contributed by each atom. Lines 8 and 9 iterate over `bond_list` and build `my_bond_list`, which is a list of bonds between a hydrogen atom and a non-hydrogen atom. All threads assigned to an atom start at the main loop on line 10. This loop iterates over the `my_bond_list`. Each thread picks up a unique hydrogen bond (`my_hbond`) and computes hydrogen bond energy and force on `my_atom`, which is stored in shared memory. Each thread then picks up the next hydrogen bond (`my_hbond`) on line 22. SIMT semantics of the CUDA runtime ensure that all the threads in a thread block execute the same control instruction simultaneously within the loop between lines 10 and 22. Line 23 performs a parallel reduce operation in shared memory to compute the effective force on each atom.

Algorithm 2: Multiple threads per atom kernel for hydrogen-bonds interaction

```
   Data: atoms, hydrogen-bond list, bond list
   Result: updated hydrogen-bond list and forces on atoms and hydrogen-bond energy of the system
 1 shared-memory sh_atom_force[];
 2 shared-memory sh_hbond_energy[];
 3 thread_id = blockIdx.x * blockDim.x + threadIdx.x;
 4 my_atom = GetAtomId (blockIdx.x, threadIdx.x, threads_per_atom);
 5 lane_id = thread_id & (threads_per_atom - 1);
 6 if my_atom is an hydrogen atom then
 7 |   my_bond_list = empty list;
 8 |   foreach bond in bond_list of my_atom do
 9 |   |   if the other atom in bond is not hydrogen-atom append it to the my_bond_list;
   |   end
10 |   foreach my_bond in my_bond_list do
11 |   |   max_iterations = get_max_iter(threads_per_atom, # of hydrogen_bonds for my_atom);
12 |   |   iterations = 0;
13 |   |   my_hbond = start_index (my_atom, hydrogen_bond_list) + lane_id;
14 |   |   my_bond_neighbor = my_bond->neighbor;
15 |   |   while iterations < max_iterations do
16 |   |   |   my_hbond_neighbor = my_hbond->neighbor;
17 |   |   |   if my_bond_neighbor != my_hbond_neighbor then
18 |   |   |   |   compute derivatives and update hydrogen bond accordingly;
19 |   |   |   |   sh_hbond_energy[threadIdx.x] = compute hydrogen-bond energy;
20 |   |   |   |   sh_atom_force[threadIdx.x] = compute force on my_atom;
   |   |   |   end
21 |   |   |   iterations ++;
22 |   |   |   my_hbond += threads_per_atom;
   |   |   end
   |   end
23 |   perform parallel reduction on sh_atom_force and sh_hbond_energy arrays;
24 |   and compute resultant force on my_atom and hydrogen_bond energy contributed by my_atom;
   end
```

Eliminating bond order derivative lists. All bonded potentials (including the hydrogen bond potential) depend primarily on the strength of the bonds between the atoms involved. Therefore, all forces arising from bonded interactions depend on the derivative of the bond order terms. A close examination of Eq. 2 suggests that BO_{ij} depends on all the uncorrected bond orders of both atoms i and j, which could be as many as 20-25 in a typical system. This also means that when we compute the force due to the i-j bond, the expression dBO_{ij}/dr_k evaluates to a non-zero value for all atoms k that share a bond with either i or j. Considering the fact that a single bond takes part in various bonded interactions, the same dBO_{ij}/dr_k expression may need to be evaluated several times over a single timestep. One approach to efficiently computing forces due to bond order derivatives is to evaluate the bond order derivative expressions at the start of a timestep and then use them repeatedly as necessary. Besides the large amount of memory required to store the bond order derivative list, this approach also has implications for costly memory lookups during the time-critical force computation routines.

We eliminate the need for storing the bond order derivatives and frequent look-ups to the physical memory by delaying the computation of the derivative of bond orders until the end of a timestep. During the computation of bonded potentials, coefficients for the corresponding bond order derivative terms arising from various interactions are accumulated into a scalar variable $dEdBO_{ij}$. After all bonded interactions are computed, we evaluate the expression dBO_{ij}/dr_k and add the force $dEdBO_{ij} \times \frac{dBO_{ij}}{dr_k}$ to the net force on atom k directly. This technique enables us to work with much larger systems by saving considerable memory and computational cost.

Atomic operations to resolve the dependencies. CUDA runtime does not support built-in atomic operations on double-precision variables. Atomic operations on double precision variables are slow because 64-bit floating point numbers require two single-precision atomic operations (one operation for each of the 32-bit words). Atomic operations are performed in memory controllers and hardware caches; the associated hardware is shared by each SM, and if all the threads access the same memory location on a multiprocessor they are effectively serialized [22, 23]. This may significantly impact the performance of a kernel that makes heavy use of atomic operations with double-precision variables. Therefore we use additional memory, which we discuss next, in resolving all dependencies and race conditions in PuReMD-GPU.

Using additional memory to resolve the dependencies. To resolve concurrent updates to energies, forces, and intermediate derivatives by multiple threads, PuReMD-GPU uses additional memory to avoid costly atomic operations. For this purpose, bond list and hydrogen bond list structures have been augmented with temporary variables. During bonded interactions, temporary results are stored in these augmented variables. After the interaction is processed, a separate kernel is launched that iterates over the bond list and hydrogen bond list to compute the final result. Multiple threads per atom kernels employ shared memory to store intermediate results and a designated thread for each atom updates the global memory, where the final

result is stored. All algorithms discussed in this section use the additional memory approach over atomic operations. All the experiments reported in Section 5 are based on the additional memory implementation, unless otherwise mentioned for that specific experiment.

4.3. Charge Equilibration (QEq) Solve

Unlike conventional MD techniques, where charges on atoms are static, in ReaxFF, charges are redistributed periodically (most commonly at each timestep). This is done through a process called charge equilibration, which assigns charges to atoms so as to minimize electrostatic energy of the system, while keeping net charge constant. The mathematical formulation of this process can be written as:

$$\begin{aligned} \text{Minimize } E(q_1 \dots q_N) = & \sum_i \quad (E_{i0} + \chi_i^0 q_i + \frac{1}{2} H_{ii}^0 q_i^2) \\ & + \sum_{i<j} \quad (H_{ij} q_i q_j) \end{aligned} \tag{4}$$

$$\text{subject to } q_{net} = \sum_{i=1}^{N} \quad q_i \tag{5}$$

Here, χ_i^0 is the *electronegativity* and H_{ii}^0 the *idempotential* or *self-Coulomb* of i. H_{ij} corresponds to the Coulomb interaction between atoms i and j.

To solve these equations, we extend the mathematical formulation of the charge equilibration problem by Nakano *et al.* [24]. Using the method of Lagrange multipliers to solve the electrostatic energy minimization problem, we obtain the following linear systems:

$$-\chi_k = \sum_i H_{ik} s_i \tag{6}$$

$$-1 = \sum_i H_{ik} t_i \tag{7}$$

Here, H denotes the QEq coefficient matrix, which is an N by N sparse matrix, N being the number of atoms in the simulation. The diagonal elements of H correspond to the polarization energies of atoms, and off-diagonal elements contain the electrostatic interaction coefficients between atom pairs. χ is a vector of size N, comprised of parameters determined by the types of atoms in the system. s and t are fictitious charge vectors of size N, which arise in the solution of the minimization problem. Finally, partial charges, q_i's, are derived from the fictitious charges based on the following formula:

$$q_i = s_i - \frac{\sum_i s_i}{\sum_i t_i} t_i \tag{8}$$

$$\text{subject to } q_{net} = 0 \tag{9}$$

We solve the linear systems in Eq. 6 and Eq. 7 using an iterative solver based on the GMRES algorithm [25, 26] with a diagonal preconditioner. PuReMD-GPU uses CUBLAS library from NVIDIA for various vector

operations to realized a high performance GMRES implementation. The implementation of the sparse matrix-vector multiplication (SpMV), which is the most time consuming kernel in GMRES, follows the compressed sparse row (CSR) storage based algorithm presented in [27]. Each row of the sparse matrix uses multiple threads to compute the product of the row with the vector, and temporary sums are stored in shared memory. Note that this corresponds to a highly optimized 2-D partitioning of the sparse matrix.

Individual timesteps in ReaxFF must be much shorter than those in classical MD methods (less than a femtosecond). Therefore the geometry of the system does not change significantly between timesteps. This observation implies that solutions to Eq. 6 and Eq. 7 in one timestep yield good initial guesses for the solutions in the next timestep. Indeed, by making linear, quadratic or higher order extrapolations on the solutions from previous steps, we obtain better initial guesses for solving the QEq problem. The combination of an efficient GMRES implementation, an effective diagonal preconditioner as well as good initial guesses yield a highly optimized solver for the charge equilibration problem. We present detailed performance results in Section 5.

4.4. Implementation of Nonbonded Forces

Nonbonded forces include Coulomb and van der Waals force computations which are computed by iterating over the neighbor list of each atom. Each atom may have several hundred neighbors in its neighbor list. In order to exploit the spatial locality of the data and coalesced reads/writes on global memory, multiple threads per atom are used in this kernel. Algorithm 3 summarizes our implementation of nonbonded forces.

Algorithm 3: Multiple threads per atom kernel for Coulombs and van der Waals forces

```
   Data: atoms, far neighbors list
   Result: Coulombs and van der Waals forces
 1 shared-memory sh_atom_force[];
 2 shared-memory sh_coulombs[];
 3 shared-memory sh_vdw[];
 4 thread_id = blockIdx.x * blockDim.x + threadIdx.x;
 5 my_atom = GetAtomId (blockIdx.x, threadIdx.x, threads_per_atom);
 6 lane_id = thread_id & (threads_per_atom - 1);
 7 start = start_index (my_atom, far_neighbors_list);
 8 end = end_index (my_atom, far_neighbors_list);
 9 my_index = start + lane_id;
10 while my_index < end do
11 |   if far_neighbor_list[my_index] is within cutoff distance then
12 |   |   sh_vdw[threadIdx.x] = compute van der Waals force;
13 |   |   sh_coulombs[threadIdx.x] = compute coulombs force;
14 |   |   sh_atom_force[threadIdx.x] = compute force on my_atom;
   |   end
15 |   my_index += threads_per_atom;
   end
16 perform parallel reduce operation in shared memory to compute;
17 final coulombs and van der Waals forces and update force on my_atom;
```

Lines 1, 2 and 3 declare shared memory to store intermediate force values. Lines 7 and 8 mark the beginning and end of `my_atom`'s neighbor list. The while loop at line 10 performs the force computations for the atom. Each thread operating on the neighbor list of an atom works on distinct neighbors indicated by the variable `my_index`. SIMT execution model of CUDA runtime ensures that all the threads in a thread block execute the same control instruction simultaneously within the while loop between lines 10 and 15. Line 16 performs a parallel reduction in the shared memory to compute the final forces for each atom.

Use of Lookup Tables. PuReMD uses lookup tables for computing nonbonded interactions due to the larger number of interactions within the cutoff radius, r_{nonb}. These tables are used to approximate the complex expressions through cubic spline interpolation. This is a common optimization technique used by many MD codes, which yields significant performance improvement with relatively little impact on accuracy[12]. When using GPUs, however, global memory latency is several hundreds of cycles, compared to a few cycles for local memory access. The larger memory footprint of lookup tables precludes their storage in local memory. Even when stored in global memory, lookup tables limit memory available to other kernels and data structures. For this reason, lookup tables do not yield significant performance improvements on GPUs. On the contrary, PuReMD's performance deteriorates significantly if lookup tables are *not* used as discussed in Section 5.

5. Experimental Results

In this section, we report on our comprehensive evaluation of the performance, stability, and accuracy of PuReMD-GPU. All simulations are performed on a testbed consisting of two Intel Xeon CPU E5606 processors (four cores per processor) operating at 2.13 GHz with 24GB of memory, running the Linux OS, and equipped with a Tesla C2075 GPU card. All arithmetic in PuReMD-GPU is double precision. The peak double precision floating point performance of the C2075 GPU is 515 Gflops [28]. The PuReMD code is compiled using GCC 4.6.3 compiler with the following options - "-funroll-loops -fstrict-aliasing -O3". PuReMD-GPU is compiled in the CUDA 5.0 environment with the following compiler options "-arch=sm_20 -funroll-loops -O3". Fused Multiplication Addition (fmad) operations are used in PuReMD-GPU implementation.

We use bulk water (H_2O) and amorphous silica (SiO_2) model systems for in-depth analysis of accuracy and performance, since they represent diverse stress points for the code. Bulk water systems of various sizes containing 6,540 atoms, 10,008 atoms, 25,050 atoms, 36,045 atoms and 50,097 atoms have been used. Amorphous silica systems used has consisted of 6,000 atoms, 12,000 atoms, 24,000 atoms, 36,000 atoms and 48,000 atoms. We use a timestep of 0.1 femtoseconds, a tolerance of 10^{-10} for the QEq solver and the NVT ensemble at 300 K in all our simulations.

The key result of our development effort is the significant speedups achieved on a single GPU card. Our detailed simulations demonstrate speedups of up to 16× over a highly optimized single core CPU run by leveraging the algorithms and optimizations described in Section 4.

5.1. Performance and Scalability of PuReMD-GPU

We use bulk water systems of different sizes to demonstrate the performance improvements and scalability achieved by using PuReMD-GPU in ReaxFF simulations. In Figure 3, we show the total time taken per step for our model water systems, and Table 1 gives the corresponding speedups. We achieve speedups of up to 16× compared to the PuReMD code running on a single CPU core and almost 3× speedup compared to execution on all 8 cores available on the testbed system. We also observe that as the system size increases, the effective speedup increases too, which points out to better scaling properties of our GPU implementation compared to the CPU versions. In Figure 3 and Table 1, we also present the performance data from

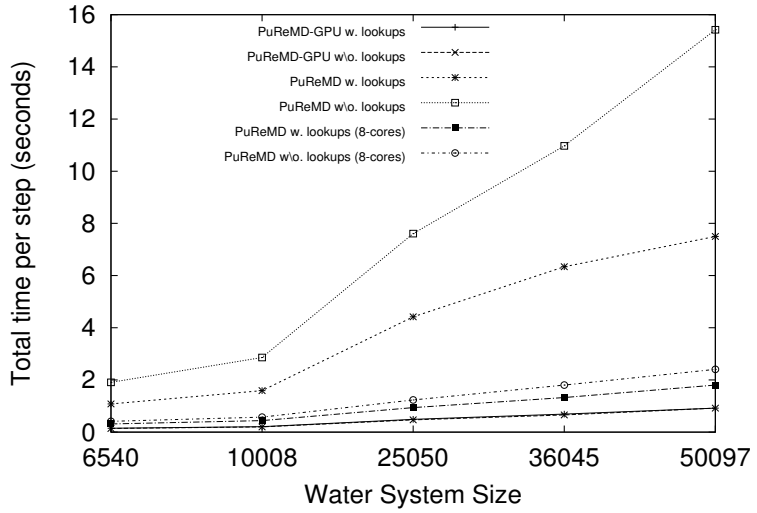


Figure 3: Time per timestep for the Water systems.

	One core		8 cores	
System size	Look-up tables	No look-up tables	Look-up tables	No look-up tables
6540	7.1	13.6	2.1	2.8
10008	7.6	13.0	2.2	2.9
25050	9.1	16.0	2.0	2.8
36045	9.2	16.4	2.1	2.9
50097	8.2	16.6	2.1	2.9

Table 1: Speedups achieved with bulk water systems of various sizes. Speedups of PuReMD-GPU over the PuReMD code running on a single CPU core and all 8 cores available on the testbed are shown.

PuReMD and PuReMD-GPU with and without the use of lookup tables. We observe in our performance

tests that PuReMD's performance is significantly degraded if lookup tables are not used, whereas PuReMD-GPU's performance remains inert to this change due to the reasons discussed in Section 4.4. In particular, the relative ratios of computation to memory access favor increased computation on GPUs as compared to CPU-only implementations that can benefit from fewer computations resulting from the use of lookup tables.

We note that factors such as the number of threads used per atom and the thread block size are important for obtaining the best performance with PuReMD-GPU. For our tests with the bulk water systems and amorphous silica systems, multiple threads per atom are used for kernels with high degree of available parallelism such as neighbor generation, hydrogen bonds, the SpMV in QEq, Coulomb and van der Waals interaction kernels, while a single thread per atom was used for all other kernels. Specifically, neighbor generation kernel uses 16 threads per atom; hydrogen bonds, Coulomb interactions, van der Waals interactions and SpMV kernels use 32 threads per atom. Thread block size for all kernels is set to 256 except for the SpMV kernel, which uses a thread block size of 512. The effect of the number of threads and thread block size on performance are discussed in detail in Sections 5.4 and 5.5.

5.2. Performance and Scalability across Systems

Apart from the bulk water systems, we have also used amorphous silica systems of different sizes to test the performance of PuReMD-GPU. Amorphous silica which is in solid phase at 300 K is chemically very different than a bulk water system which is a liquid. In silica (SiO_2), each Si atom makes 4 bonds and each O atom makes 2 bonds. However, in bulk water (H_2O), while O still makes 2 bonds, H makes a single bond only. Therefore bonded interactions, especially the three-body and four-body interactions, become more expensive in the amorphous silica system. Also the number of the atoms in the nonbonded interaction cutoff radius is different between the two systems. Finally, there are no hydrogen bonds in the silica system. As a result, we hope that the comparison of performance results obtained from these two different systems can provide clues to the reader about the performance portability of PuReMD-GPU across diverse systems.

As can be seen in Figure 4 and Table 2, compared to the baseline PuReMD results on CPUs, PuReMD-GPU yields performance improvements and scalability similar to those observed with bulk water systems. However, speedups achieved with silica systems are lower than the water systems (11× vs. 16× and 2× vs. 3×). The main reason for the lower speedups with silica systems is the four-body interactions. Due to the high number of bonds associated with each atom in a silica system, four-body interactions constitute a significant fraction of the total running time in these systems (see Table 3). Even with the use of auxiliary memory in the bond list, atomic operations can not be avoided in this kernel. This is because of the following reason: assuming that atoms (i, j, k, l) participate in a four-body interaction, we store three-body interactions among atoms (i, j, k) and atoms (j, k, l) in the three-body list where atoms j and k are central atoms. The thread processing the four-body interactions for atom j needs to update the force on atom l

while processing this interaction. We use an atomic operation in this case to avoid maintaining another data structure for resolving this dependency. Avoiding this atomic operation may improve the performance of this kernel at the expense of additional memory usage for systems where four-body interactions are dominant. However, this may significantly increase the complexity of the kernel too. Because overheads due to the use of additional local variables and increased use of shared memory can be computationally expensive. For these reasons, we have decided to use atomic operations for the force updates in four-body interactions.

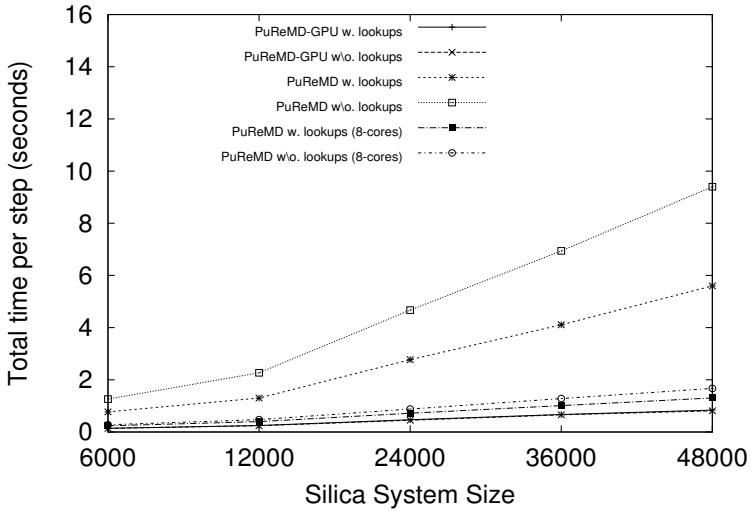


Figure 4: Time per timestep for Silica systems.

	One core		8 cores	
System size	Look-up tables	No look-up tables	Look-up tables	No look-up tables
6000	5.8	9.6	1.7	2.0
12000	5.9	10.1	1.7	2.0
24000	6.5	10.9	1.6	2.1
36000	6.6	11.4	1.5	2.0
48000	6.6	11.4	1.6	2.0

Table 2: Speedups achieved with amorphous silica systems of various sizes. Speedups of PuReMD-GPU over the PuReMD code running on a single CPU core and all 8 cores available on the testbed are shown.

5.3. Detailed Performance Analysis

Table 3 presents a detailed performance comparison of PuReMD and PuReMD-GPU implementations. In all cases, kernels associated with nonbonded interactions, which include Coulomb and van der Waals forces, charge equilibration (QEq) procedure and initialization (contains the QEq matrix construction), are the most expensive ones. This is expected because nonbonded interactions are typically effective at a larger distance than bonded interactions (10-12 Å vs 4-5 Å). However, these kernels contain a higher level of parallelism

	Water-6540			Silica-6000		
Computation	T_{cpu}	T_{gpu}	T_{ov}	T_{cpu}	T_{gpu}	T_{ov}
Neighbor Generation	117	14	-	85.3	11.9	-
Initialization	232.8	31.1	12	163.0	10.8	0.6
Bond Orders	4.1	0.5	-	10.9	0.7	-
Bond Energies	2.4	0.2	0.03	4.5	0.3	0.03
Lone Pair Intr.	2.7	0.8	0.1	4.8	1.2	0.1
Three-Body Intr.	37.4	5.9	0.4	113.5	13.2	0.6
Four-Body Intr.	34.7	8.1	0.5	252.8	58.9	0.9
Hydrogen Bonds	133.1	6.2	1.3	0.05	0.9	0.5
Charge Equilibration	589.9	43.0	-	97.5	21.9	-
Coulomb/van der Waals	1001.9	24.2	0.1	587.2	14.9	0.1

Table 3: Timings for various computational components of a single time-step for the Water-6540 system and Silica-6000 systems. All the timings are in milliseconds. T_{cpu} is the time required for the single core CPU run with PuReMD. T_{gpu} is the time taken by PuReMD-GPU. T_{ov} shows the overhead incurred on the GPU, and is already included in T_{gpu}.

that can be exploited by GPUs compared to bonded interactions. Therefore much of the overall speedup we observe with PuReMD-GPU comes from the acceleration of nonbonded interaction computations. For example, we obtain more than 40× speedup for Coulomb and van der Waals force computations in both of our model systems. Of all the bonded interactions, hydrogen bond interaction is typically the most expensive one, if present. This is because it contains a bonded H atom interacting with nonbonded atoms as far as 6-7 Å away. Hydrogen bond computations are also easily parallelizable and we see good speedups with the use of GPUs in this kernel. Hydrogen bond computations are followed by three-body and four-body interactions in terms of computational complexity. For reasons discussed above, four-body interactions and to some degree the three-body interactions dominate the total running time for the bonded interactions in silica systems. Bond-order, bond-energy and lone-pair interactions are simpler kernels, costing only a fraction of the total running time. However, the speedups obtained for these kernels are not as impressive as the nonbonded interaction kernels. As the GPU technology becomes more advanced and supports even higher degrees of concurrency, we anticipate that the bonded interaction kernels may become bottlenecks which may require further performance optimization work for bonded interaction kernels.

In Table 3, we also give the overheads associated with various computations in the GPU implementation. As described in Section 4, some overheads are incurred in certain kernels of PuReMD-GPU in order to expose more parallelism in those kernels. It is evident from the detailed performance data that the overhead incurred by most kernels is much less than the cost of (essential) computations in those kernel, thus resulting in good speedups. The exceptions are the initialization and hydrogen bond kernels which require relatively high computational overheads.

Neighbor generation does not require any additional computations over the PuReMD implementation,

since it is implemented as a single kernel. The initialization step, during which bond list, QEq matrix and hydrogen bonds list are generated, uses separate post processing kernels to make the bond and the hydrogen bond lists symmetric. Bond list is made symmetric in PuReMD-GPU in two stages. First, the bond list is generated during the initialization step and then a separate kernel iterates over it to update pointers so that bond (i, j) and bond (j, i) point to each other. A similar approach is taken for hydrogen bond list construction as well.

All force computation kernels require the use of a separate kernel, which causes a small overhead, to compute the overall energy contribution of that interaction. For example, bond energy requires a separate kernel to compute E_{bond} of the system; lone-pair interaction needs 3 such kernels to compute E_{lp}, E_{over}, and E_{under}; three-body interactions kernel needs to perform a reduction to compute E_{ang}, E_{pen}, and E_{3coa}; four-body interactions kernels needs to do the same for computing E_{tor} and E_{4con}; hydrogen-bond kernel has the overhead of computing E_{H-bond} and finally nonbonded forces kernel incurs the same overhead for computing $E_{Coulomb}$ and E_{vdW}. An additional post-processing step is used for the lone-pair forces to compute the net force on each atom due to 3 different kinds of interactions involved. Similarly, during the processing of both the three-body and four-body interactions, a post-processing step is used for computing the net force on each atom, as well as the intermediate derivatives, due to different kind of interactions involved. Finally, two separate kernels are used for post-processing in the hydrogen bond computations, one to iterate over the hydrogen bond list and another one to iterate over the bond list to compute the net force on each atom due to this interaction.

5.4. The Effect of Thread Block Size on Performance

GPU kernel performance is a function of block size, as well as resources (shared memory and registers) available at each SM. The Tesla C2075 GPU has a 32K register file and 48KB of shared memory per block. To hide memory access latency, CUDA switches between blocks, which means that a single SM can run up to 8 blocks of threads (this may vary depending on the GPU used), provided that it has the needed resources. Each resident or active block is assigned its own set of resources, facilitating fast switch among the resident blocks on the SM. Each block is bound to the multiprocessor on which it is scheduled to execute until its completion. Recall that the occupancy of a CUDA kernel is defined as the ratio of active warps to maximum number of active warps. Occupancy is limited by the resource constraints on the SM (shared memory usage per block, number of registers used per block, and block-size). Kernels with low occupancy have a harder time hiding latency. Consequently, it is desirable to have high occupancy in order to achieve better performance, especially for memory-bound applications. Kernels with fewer threads in a block may have high occupancy, but the performance degrades because the SM cannot schedule enough instructions to hide latency (only 8 blocks can be scheduled at any point of time). As a result, the SM waits on the instructions to complete before scheduling the rest of the blocks. Larger number of threads limits the occupancy because

of resource constraints, decreasing the performance of the kernel.

For the Water-6540 system, Figure 5 shows the timings of each of the kernels in PuReMD-GPU (timing for bonded interactions are accumulated into a single number) with different thread-block sizes. As discussed in Section 3, each thread-block may contain a maximum of 1024 threads (may vary with the compute compatibility of the GPU) and should always be a multiple of 32, because kernels always issue instructions in warps (32 threads). For example, if the block size is 50 threads, the GPU will still issue instructions to 64 threads resulting in under utilization of the GPU cores. We observe that the performance of kernels can vary significantly with block size. As explained above, the thread-block sizes for our model systems have been hand-tuned for best performance. In general, the optimal thread-block size will change depending on the system being studied and the specifications of the underlying GPU card. An automatic tuning method to ensure ideal performance in general is among the future work planned for PuReMD-GPU.

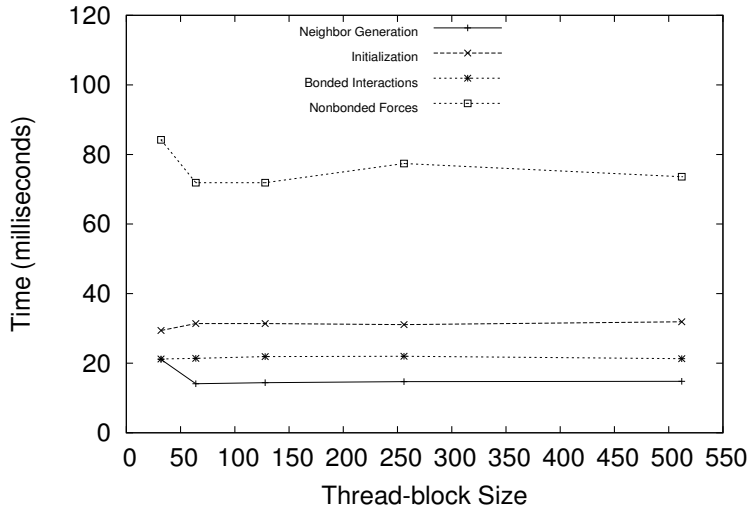


Figure 5: Effect of block size on the performance of kernels in computations with the Water-6540 system.

5.5. Single vs. Multiple Threads per Atom

Another important factor that affects performance is the use of single vs. multiple threads per atom in the implementation of GPU kernels. Table 4 presents the timings associated with using different number of threads per atom for the most expensive kernels in PuReMD-GPU. Overall, we observe that using multiple threads per atom yields significant performance gains (ranging from about 20% to 6×) for kernels which contain several interactions per atom such as neighbor generation, hydrogen bond computations, charge equilibration (QEq) and nonbonded force computations. The performance gains can be attributed to better leveraging of coalesced memory operations and effective use of shared memory to process intermediate results. For the larger Water-36045 system, the performance gains in these kernels are higher, especially for the QEq computations.

However, in bonded force computations where there are few number of interactions per atom, using multiple threads can actually lead to worse performance. This can be seen from the performance data for three-body and four-body interactions in Table 4. In these cases, if multiple threads per atom are used, a large number of thread blocks are created. This situation adversely affects the performance of these kernels due to low kernel occupancy. As noted above, performance results presented in this paper have been obtained

		Threads Per Atom					
	Kernel	1	2	4	8	16	32
Water 6540 Atoms	Neighbor Gen.	23.1	16.2	13.9	13.7	14.7	22.2
	Three-Body Intr.	6.1	6.4	5.7	6.3	9.3	13.9
	Four-Body Intr.	8.1	11.2	11.7	13.3	17.8	24.9
	Hydrogen Bonds	7.6	6.6	6.1	5.7	5.9	6.2
	Charge Eq.	193.0	100.0	74.5	58.9	53.9	54.7
	Nonbonded Forces	30.4	28.5	26.8	24.5	23.9	24.3
Water 36045 Atoms	Neighbor Gen.	127.2	85.6	73.2	73.7	80.1	120.5
	Three-Body Intr.	23.0	25.3	25.5	31.9	47.3	71.9
	Four-Body Intr.	39.9	55.7	59.7	69.7	98.4	138.9
	Hydrogen Bonds	38.0	32.8	31.1	30.4	31.6	33.1
	Charge Eq.	1132.9	562.8	329.5	220.0	183.8	178.5
	Nonbonded Forces	162.4	148.6	136.9	132.1	128.7	132.5

Table 4: Effect of multiple threads per atom on the kernel performance for Water-6540 system. All the timings are in miliseconds.

by using 16 threads per atom during neighbor generation, 32 threads per atom in hydrogen bond, charge equilibration and nonbonded force computations, while using a single thread per atom in all other kernels.

5.6. Use of Atomic Operations vs. Additional Memory

In Table 5, we compare the timings for each of the bonded and nonbonded interaction kernels when atomic operations vs. additional memory approaches are used for concurrent updates to global memory locations. We observe from this table that the performance of all of the kernels is significantly improved by the additional memory approach for reasons discussed in Section 4.2. As we show in Section 5.7, the use of additional memory causes the memory footprint of PuReMD-GPU to be larger than that of PuReMD, but the performance gains are significant enough to justify the increase in memory footprint.

5.7. Memory Footprint of PuReMD and PuReMD-GPU

Table 6 presents memory usage of various data structures used in PuReMD and PuReMD-GPU. Neighbor list memory usage in PuReMD-GPU is doubled, compared to PuReMD. This is because PuReMD-GPU maintains redundant neighbors in this list as described in Section 4.1.1. Bond list and hydrogen bond list data structures (in PuReMD-GPU) are augmented with additional memory to store temporary results during various computations to help resolve race conditions, hence the increased memory usage by these lists

Computation	Atomic Ops.	Additional Memory
Bond Orders	0.5	0.5
Bond Energies	7.6	0.2
Lone Pair Intr.	122.1	0.8
Three-Body Intr.	265.3	6.0
Four-Body Intr.	55.9	8.1
Hydrogen Bonds	157.5	6.2
Nonbonded Forces	24.2	24.2

Table 5: Timings for each interaction when using atomic operations vs additional memory to resolve dependencies for the Water-6540 system. All the timings are in miliseconds.

compared to PuReMD. Separate kernels process this auxiliary memory, utilizing coalesced global memory access and shared memory resulting in better performance as shown in Section 5.6. QEq matrix storage is also doubled in PuReMD-GPU, because both upper and lower triangular parts of the symmetric matrix are stored which is not the case in PuReMD. PuReMD-GPU computes the three-body list size in each iteration of the simulation, while PuReMD estimates its size at the beginning of the simulation. Not all bonds participate in three-body interactions, and computing the offsets, start and end indices of each bond in the three-body list gives a tighter bound on the memory allocated to this list allowing us to limit the memory used in three-body list for PuReMD-GPU at the expense of a slight increase in computational load.

	Water-6540		Silica-6000	
Data-structure	PuReMD	PuReMD-GPU	PuReMD	PuReMD-GPU
Neighbor list	75.1	150.1	43.3	86.5
Hydrogen bond list	7.1	32.8	4.5	13.7
QEq Matrix	25.0	54.4	14.0	32.8
Bond list	27.7	42.6	26.0	40.1
Three-body list	9.7	5.2	27.0	20.0
Total Memory	144.6	285.1	114.8	193.2

Table 6: Memory footprint for the Water-6540 and Silica-6000 systems. All memory footprints are given in MBs at system initialization. They may grow as the simulation progresses.

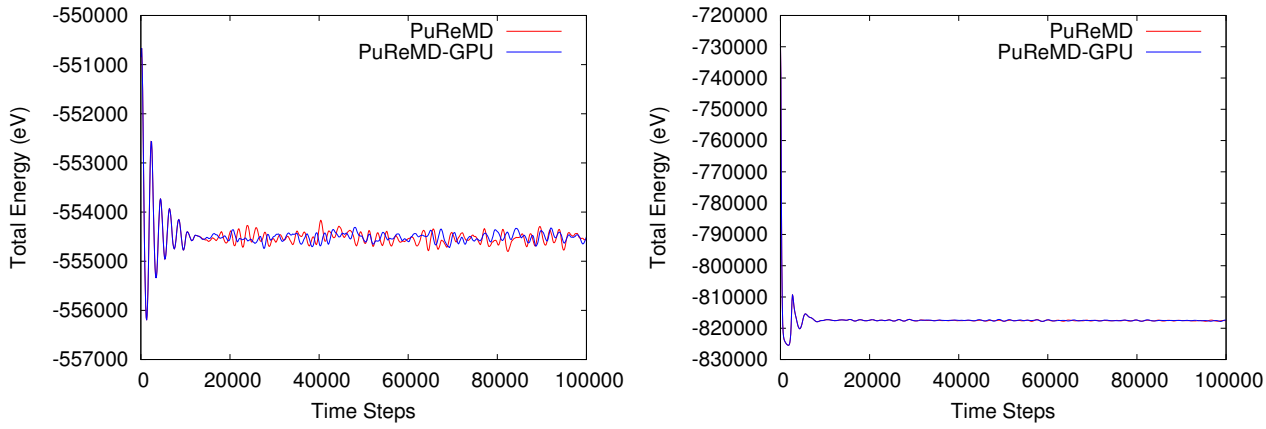
5.8. Accuracy of PuReMD-GPU

We perform extensive validation tests of PuReMD-GPU's accuracy compared to PuReMD. Table 7 presents the deviation in various energies between PuReMD and PuReMD-GPU at start and after 10,000 and 100,000 steps. The minor deviations observed in the table are attributed to the fact that arithmetic operations on the GPU are not performed in the same order as the CPU. Figures 6(a) and 6(b) provide a comparison of total energy of silica-6000 and water-6540 systems between PuReMD and PuReMD-GPU implementations. From these figures, it is evident that (i) there are no systematic total energy drifts for

either system in either implementation; and (ii) the total energy for the two systems starts to converge as the simulation progresses.

	After 0 steps		After 10K steps		After 100K steps	
Energy	water-6540	silica-6000	water-6540	silica-6000	water-6540	silica-6000
E_{be}	$< 10^{-16}$	$< 10^{-16}$	6.2×10^{-6}	1.1×10^{-4}	2.3×10^{-2}	1.5×10^{-2}
$E_{ov} + E_{un}$	$< 10^{-16}$	$< 10^{-16}$	1.4×10^{-6}	6.9×10^{-6}	4.8×10^{-2}	1.2×10^{-2}
E_{lp}	$< 10^{-16}$	$< 10^{-16}$	5.4×10^{-8}	4.3×10^{-4}	1.3×10^{-3}	2.8×10^{-3}
$E_{ang} + E_{coa}$	$< 10^{-16}$	$< 10^{-16}$	1.5×10^{-5}	1.9×10^{-5}	1.8×10^{-2}	4.4×10^{-3}
E_{H-bond}	$< 10^{-16}$	–	3.6×10^{-6}	–	1.2×10^{-1}	–
$E_{Tor} + E_{conj}$	$< 10^{-16}$	$< 10^{-16}$	1.6×10^{-5}	2.3×10^{-4}	1.2×10^{-2}	1.8×10^{-2}
E_{vdw}	$< 10^{-16}$	$< 10^{-16}$	3.4×10^{-6}	2.1×10^{-4}	5.0×10^{-2}	3.4×10^{-3}
E_{coul}	$< 10^{-16}$	$< 10^{-16}$	2.4×10^{-5}	7.8×10^{-5}	4.7×10^{-2}	4.6×10^{-3}
E_{pol}	$< 10^{-16}$	$< 10^{-16}$	2.4×10^{-5}	7.8×10^{-5}	4.6×10^{-2}	4.7×10^{-3}

Table 7: Absolute values of the difference in various energies between PuReMD and PuReMD-GPU after zero, ten thousand and hundred thousand steps of simulation on water-6540 and silica-6000 systems. All values are scaled by the observed variation in the corresponding energy term, i.e. $[< (E- < E >)^2 >]^{0.5}$ where $< E >$ means the time average of the energy term E. At the beginning of the simulation the energies between CPU and GPU match perfectly to the machine precision and due to numerical errors they start drifting away slowly as the simulation progresses.



(a) Water-6540 system

(b) Silica 6000 system

Figure 6: Comparison of total energy of water-6540 and silica-6000 systems between PuReMD and PuReMD-GPU implementations upto 100K time steps

6. Related Efforts

The first-generation ReaxFF implementation of van Duin et al. [6] demonstrated the validity of the method in the context of various applications. Thompson et al. [29] successfully ported this initial implementation, which was not developed for a parallel environment, into the parallel MD simulation package

LAMMPS [21]. Except for the charge equilibration part, the REAX package in LAMMPS is based on the original FORTRAN code of van Duin [6]. In [12], we describe the serial version of PuReMD (sPuReMD), which features novel algorithms and numerical techniques to achieve excellent performance, and a dynamic memory management scheme to minimize the memory footprint in reactive molecular dynamics simulations. We demonstrate that sPuReMD is 5-7× faster than the REAX package on diverse systems such as bulk water, hexane, and PETN crystal, while using a fraction of the memory used by REAX [12]. The parallel PuReMD code extends the capabilities of sPuReMD to massively parallel distributed memory architectures. PuReMD exhibits excellent scalability, and it has been shown to achieve 3-5× speed-up over the parallel REAX code on hundreds of processors [13]. PuReMD has been used by us and by other research groups on diverse systems, ranging from strain relaxation in Si–Ge nanobars [30] to water-silica systems [31]. Recently, PuReMD has been ported to LAMMPS as the User-Reax/C package, where it is used by hundreds of users worldwide under the GPL license [32]. The above efforts summarize the ReaxFF implementations available publicly as open source code. In [33], Nomuro et al. report on a parallel ReaxFF implementation in Fortran 90.

Zheng et al. have recently reported a GPU-enabled implementation of ReaxFF, GMD-Reax [34]. This is the closest effort in literature to the PuReMD-GPU code presented in this paper. GMD-Reax is reported to be up to 6× faster than the User-Reax/C package in LAMMPS. GMD-Reax is not available over the public domain, so a direct comparison is not possible. However, it is noteworthy that GMD-Reax uses single-precision arithmetic, in the costly charge equilibration computations. Single precision arithmetic is considerably faster than corresponding double precision operations on GPUs. The use of single precision floating point arithmetic, however, significantly degrades the accuracy of typical MD trajectories and for this reason most widely-used MD codes forego the increased speed of single precision in factor of the increased accuracy of double precision. Our code has been verified through extensive tests, as well as through independent tests performed by Prof. van Duin's group. Currently, PuReMD-GPU represents the only publicly available GPU-enabled implementation of ReaxFF.

7. Concluding Remarks and Ongoing Work

In this paper, we presented the design, implementation, and comprehensive benchmarking of PuReMD-GPU, a publicly available code for reactive molecular dynamics simulations using ReaxFF. Using a variety of data structures, algorithms, and optimizations, our code achieves over an order of magnitude improvement in performance using a state-of-the-art GPU hardware. We validate the stability and accuracy of our code on diverse systems. Our code provides the community with a unique resource that enables long time simulations (up to nanoseconds) of small- to medium-sized systems (10K atoms) on workstation class machines equipped with GPUs.

8. Acknowledgements

We thank Adri van-Duin for significant help in validating our software on a variety of systems. We also thank Joe Fogarty at University of South Florida for constructing model systems for testing and validation.

References

[1] T. A. Halgren, W. Damm, Polarizable force fields, in: Current Opinion in Structural Biology, Vol. 11, 2001, pp. 236–242.
[2] J. E. Davis, G. L. Warren, S. Patel, Revised charge equilibration potential for liquid alkanes, in: J Phys Chem B, Vol. 12, 2008, pp. 8298–8310.
[3] D. W. Brenner, Empirical potential for hydrocarbons for use in simulating the chemical vapor deposition of diamond films, in: Phys Rev B, Vol. 42, 1990, pp. 9458–9471.
[4] D. W. Brenner, O. A. Shenderova, J. A. Harrison, S. J. Stuart, S. B. Sinnott, A second-generation reactive empirical bond order (rebo) potential energy expression for hydrocarbons, in: J Phys Condens Matter, Vol. 14, 2002, pp. 783–802.
[5] S. J. Stuart, A. B. Tutein, J. A. Harrison, A reactive potential for hydrocarbons with intermolecular interactions, in: J Chem Phys, Vol. 112, 2000, p. 6472.
[6] A. C. T. van Duin, S. Dasgupta, F. Lorant, W. A. G. III, Reaxff: A reactive force field for hydrocarbons, in: J Phys Chem A, Vol. 105, 2001, pp. 9396–9409.
[7] K. D. Nielson, A. C. T. van Duin, J. Oxgaard, W.-Q. Deng, W. A. G. III, Development of the reaxff reactive force field for describing transition metal catalyzed reactions, with application to the initial stages of the catalytic formation of carbon nanotubes, in: J Phys Chem A, Vol. 109, 2005, pp. 493–499.
[8] K. Chenoweth, S. Cheung, A. C. T. van Duin, W. A. G. III, E. M. Kober, Simulations on the thermal decomposition of a poly(dimethylsiloxane) polymer using the reaxff reactive force field, in: J Am Chem Soc, Vol. 127, 2005, pp. 7192–7202.
[9] M. J. Buehler, Hierarchical chemo-nanomechanics of proteins: Entropic elasticity, protein unfolding and molecular fracture, in: Mech Material Struct, Vol. 2(6), 2007, pp. 1019–1057.
[10] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao, M. P. Eastwood, J. Gagliardo, J. P. Grossman, C. R. Ho, D. J. Ierardi, I. Kolossvry, J. L. Klepeis, T. Layman, C. McLeavey, M. A. Moraes, R. Mueller, E. C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, S. C. Wang, Anton: A special-purpose machine for molecular dynamics simulation, in: ISCA, 2007.
[11] H. M. Aktulga, A. Y. Grama, S. Plimpton, A. Thompson, A fast ilu preconditioning-based solver for the charge equilibration problem, CSRI Summer Proceedings 2009 (2010) 50.
[12] H. Aktulga, S. Pandit, A. van Duin, A. Grama, Reactive molecular dynamics: Numerical methods and algorithmic techniques, SIAM Journal on Scientific Computing 34 (1) (2012) C1–C23. arXiv:http://epubs.siam.org/doi/pdf/10.1137/100808599, doi:10.1137/100808599.
URL http://epubs.siam.org/doi/abs/10.1137/100808599
[13] H. M. Aktulga, J. C. Fogarty, S. A. Pandit, A. Y. Grama, Parallel reactive molecular dynamics: Numerical methods and algorithmic techniques, Parallel Comput. 38 (4-5) (2012) 245–259. doi:10.1016/j.parco.2011.08.005.
URL http://dx.doi.org/10.1016/j.parco.2011.08.005
[14] H. M. Aktulga, Algorithmic and numerical techniques for atomistic modeling, Ph.D. thesis, West Lafayette, IN, USA, aAI3444466 (2010).
[15] T.-R. Shan, R. R. Wixom, A. E. Mattsson, , A. P. Thompson, Atomistic simulation of orientation dependence in shock-induced initiation of pentaerythritol tetranitrate, in: J. Phys. Chem, 2013, pp. 928–936.
[16] A. C. T. van Duin, A. Strachan, S. Stewman, Q. Zhang, X. Xu, W. A. G. III, Reaxffsio reactive force field for silicon and silicon oxide systems, in: J Phys Chem A, Vol. 107, 2003, pp. 3803–3811.
[17] D. J. Luitjens, Cuda memory hierarchy.
URL http://developer.download.nvidia.com/CUDA/training/cuda_webinars_GlobalMemory.pdf
[18] NVIDIA, Nvidia white paper on fermi architecture.
URL http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
[19] NVIDIA, Cuda c best practices guide.
URL http://docs.nvidia.com/cuda/cuda-c-best-practices-guide
[20] N. Wihtehead, A. Fit-Florea, Precision and performance: Floating point and ieee 754 compliance for nvidia gpus.
URL http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf
[21] S. J. Plimpton, Fast parallel algorithms for short-range molecular dynamics, in: J Comp Phys, Vol. 117, 1995, pp. 1–19.
[22] Nvidia developer forum.
URL https://devtalk.nvidia.com/default/topic/477967/?comment=3416415
[23] NVIDIA, Cuda tutorial.
URL http://supercomputingblog.com/cuda/cuda-tutorial-4-atomic-operations/
[24] A. Nakano, Parallel multilevel preconditioned conjugate-gradient approach to variable-charge molecular dynamics, in: Comp Phys Comm, Vol. 104, 1997, pp. 59–69.
[25] Y. Saad, M. H. Schultz, Gmres: A generalized minimal residual method for solving nonsymmetric linear systems, in: SIAM J Sci Stat Comput, Vol. 7, 1986, pp. 856–869.
[26] Y. Saad, Iterative methods for sparse linear systems, in: SIAM, SIAM, 2003.

[27] N. Bell, M. Garland, Efficient sparse matrix-vector multiplication on cuda.
URL http://www.nvidia.com/docs/IO/66889/nvr-2008-004.pdf
[28] NVIDIA, Nvidia c2075 specifications.
URL http://www.nvidia.com/docs/IO/43395/NV-DS-Tesla-C2075.pdf
[29] A. Thompson, H. Cho, Lammps/reaxff potential (April 2010).
URL http://lammps.sandia.gov/doc/pair_reax.html
[30] Y. Park, H. M. Aktulga, A. Grama, A. Strachan, Strain relaxation in si/ge/si nanoscale bars from molecular dynamics simulations, Journal of Applied Physics 106 (3) (2009) 034304–034304.
[31] J. C. Fogarty, H. M. Aktulga, A. Y. Grama, A. C. Van Duin, S. A. Pandit, A reactive molecular dynamics simulation of the silica-water interface, The Journal of Chemical Physics 132 (2010) 174704.
[32] H. M. Aktulga, S. J. Plimpton, A. Thompson, Lammps/user-reax/c.
URL http://lammps.sandia.gov/doc/pair_reax_c.html
[33] 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, L. H. Yang, De novo ultrascale atomistic simulations on high-end, in: Intl J High Perf Comp Apps, Vol. 22(1), 2008, pp. 113–128.
[34] M. Zheng, X. Li, L. Guo, Algorithms of gpu-enabled reactive force field (reaxff) molecular dynamics, in: J Mol Graph Model, Vol. 41, 2013, pp. 1–11.