PG-PuReMD: A Parallel-GPU Reactive Molecular Dynamics Package

Sudhir B Kylasa
Elec. and Comp. Engg. Dept
Purdue University
West Lafayette, Indiana 47907
skylasa@purdue.edu

Hasan Metin Aktulga
Lawrence Berkeley National Laboratory
1 Cyclotron Rd, MS 50F-1650
Berkeley, CA 94720
hmaktulga@lbl.gov

Ananth Grama
Computer Science Dept
Purdue University
West Lafayette, Indiana 47907
ayg@cs.purdue.edu

***Abstract*—We present a parallel/GPU implementation of our open-source reactive molecular dynamics code, PG-PuReMD (Parallel GPU-Purdue Reactive Molecular Dynamics). Using a variety of innovative algorithms and optimizations, PG-PuReMD achieves over 350x speedup compared to a single CPU implementation on a cluster of 36 state of the art GPUs. This is a significant development, since it enables simulations of over 0.5M atoms in under 0.5 seconds per time-step of simulation time. We report on various design choices and implementation details of PG-PuReMD in this paper. PuReMD, on which this code is based, along with its integration into LAMMPS, is currently used by over 100 research groups worldwide and is an important community resource. PG-PuReMD is currently being independently validated at a small number of institutions and is in limited release.**

***Keywords*-Reactive Molecular Dynamics; Parallel GPU Implementations; Material Simulations;**

I. 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 molecular dynamics (MD) techniques typically rely on static bonds and fixed partial charges associated with atoms. These constraints limit their applicability to non-reactive systems. ReaxFF, which is an effort aimed at addressing this limitation, is a novel reactive force field developed by van Duin et al. [1]. ReaxFF bridges quantum-scale and classical MD approaches by explicitly modeling bond activity (reactions) and charge equilibration. The flexibility and transferability of the force field allows ReaxFF to be easily extended to systems of interest. ReaxFF has been successfully applied to diverse systems [1]–[4].

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. PuReMD incorporates several algorithmic and numerical innovations to address significant computational challenges posed by ReaxFF. It achieves excellent per time-step 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 five- to six-times faster than competing implementations, while using significantly lower memory. PuReMD has also been integrated with the LAMMPS (LAMMPS/User-ReaxC) software package for atomistic simulations. PuReMD and its LAMMPS version has been used by several research groups around the world on diverse systems, ranging from strain relaxation in Si–Ge nanobars [5] and water-silica systems [6] to explosives (RDX) and biomembranes (lipid bilayers). PuReMD has a large number of downloads and an active developer community.

Two important challenges in molecular simulation are the large number of time-steps required and the size of the systems that can be simulated. Time-steps in ReaxFF are of the order of tenths of femtoseconds, but several important physical analyses require simulation data spanning nanoseconds (millions of time-steps) and beyond. GPU clusters provide significant processing power in small, affordable hardware systems. These considerations provide compelling motivations for parallel/GPU acceleration of PuReMD. A production code with good scalability properties presents tremendous scientific opportunities for a large community.

The highly dynamic nature of interactions and the memory footprint, the diversity of kernels underlying non-bonded and bonded interactions, the complexity of functions describing the interactions, the charge equilibration procedure, which requires the solution of a large system of linear equations, and high numerical accuracy requirements pose significant challenges for parallel-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 on individual GPUs. These kernels are also optimized to utilize GPUs' capability to spawn thousands of threads, and coalesced memory operations are used to enhance performance of specific kernels. The high cost of double precision arithmetic on conventional GPUs must be effectively amortized/masked through these optimizations. These requirements are traded-off with increased memory footprint to further enhance performance. The significant increase in performance from use of GPUs puts tremendous pressure on parallel-GPU implementations, since faster computations without com-

mensurate reductions in communication costs result in lower efficiencies. We address these challenges through a sequence of design trade-offs of communication and redundant storage, along with alternate algorithmic choices for key kernels.

In this paper, we present in detail, the design and implementation of all phases of PG-PuReMD (Parallel GPU PuReMD). Comprehensive experiments on a state-of-the-art GPU cluster are presented to quantify accuracy as well as performance of PG-PuReMD. Our experiments show over 350x improvement in runtime on a cluster of 36 GPU-equipped nodes, compared to a highly optimized CPU-only PuReMD implementation on model systems (water). These speedups have tremendous scientific impact for diverse simulations. PG-PuReMD is the first production code of its kind. It is currently being validated, and in limited release.

The rest of the paper is organized as follows: Section II discusses the related work on parallel ReaxFF. Section III gives an overview of reactive potentials for atomistic simulations. In Section IV, we discuss critical design choices for parallelization of ReaxFF and discuss in detail the implementation of each phase on a GPU. We also outline numerical techniques used to achieve low computational times per simulation time-step. We comprehensively evaluate the performance of PG-PuReMD in Section V.

II. Related Efforts

The first-generation ReaxFF implementation of van Duin et al. [1] strongly established the utility of the force field in the context of various applications. This serial implementation was integrated into the parallel molecular dynamics (MD) package LAMMPS [7] by Thompson et al. [8]. Except for the charge equilibration part, this integration of ReaxFF into LAMMPS was based on the original Fortran code of van Duin [1]. In [9], [10], we describe PuReMD, which features novel algorithms and numerical techniques to achieve high runtime performance, and a dynamic memory management scheme to minimize memory footprint. PuReMD exhibits excellent scalability, and has been shown to achieve up to 5x speedup over the parallel ReaxFF code in LAMMPS on identical machine configurations of hundreds of processors and beyond.

Zheng et al. recently reported a single GPU implementation of ReaxFF, called GMD-Reax [11]. This is the closest effort in literature to the PG-PuReMD code presented in this paper. GMD-Reax is reported to be up to 6 times faster than the User-Reax/C package in LAMMPS. This speedup of GMD-Reax is in large part due to their use of single-precision arithmetic in the costly charge equilibration computations. Our experiments suggest that in real applications single precision arithmetic does not yield acceptable accuracy (gross energy drifts are observed at picosecond scales and beyond). Although direct comparisons are not possible (GMD-Reax is not available over the public domain, currently), our results indicate our PG-PuReMD is over 2X faster than GMD-Reax on a single GPU, in spite of its use of full double precision arithmetic. Further, in this paper, we show excellent scalability of PG-PuReMD to multiple GPUs.

III. Reactive Potentials for Atomistic Simulations

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 suitable parameterizations 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 [12].

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 matrix is a symmetric sparse matrix with dimension equal to the number of atoms. Note that the number of atoms in a simulation may range from thousands to millions. Due to the dynamic nature (unlike classical MD, bonds are not static in ReaxFF) of the system, a different (perturbed) set of equations needs to be solved at each time-step. An accurate solution of the charge equilibration problem is necessary, 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. Since the time-step 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 parallel/GPU formulations. Note that partial charges on atoms are fixed in typical classical MD formulations; therefore this is not a consideration for conventional methods.

In the interest of space, we refer readers to [1] for a detailed discussion on mathematical formulation of ReaxFF. In the remainder of this paper, we focus on the algorithmic issues regarding an efficient and scalable parallel GPU implementation of ReaxFF.

IV. Parallel GPU Implementation

We describe our parallel-GPU implementation in two steps – we first describe the parallelization strategy using MPI. We then describe how each MPI process is executed on a GPU. An important aspect of ReaxFF that significantly impacts design choices for a parallel GPU implementation is that it uses shielded electrostatics, modeled by range-limited pairwise interactions with Taper corrections. This obviates the need for computation of long-range electrostatic interactions.

A. Problem Decomposition and Interprocess Communication

There are two important aspects of our parallel-GPU implementation – problem decomposition and inter-process synchronization/communication. PG-PuReMD adopts a 3D domain decomposition technique with wrap-around links (a torus) for periodic boundary conditions. This domain decomposition also induces a partition of the degrees of freedom for parallel charge equilibration. We refer to the domain of simulation specified in the input files as the *simulation box*, and the part assigned to a process as the *sub-domain* of that process. Decomposition techniques for MD have been extensively studied. We discuss decomposition techniques for ReaxFF in parallel environments in [10]. In this section, we focus primarily on decomposition techniques as they relate to our parallel-GPU implementation.

1) Interactions Spanning Multiple Processes: We first describe our handling of interactions that span multiple processes. We specifically focus on bond-order potentials, and associated dynamic bonded interactions in ReaxFF. Carefully analyzing different ways of handling bonded interactions in ReaxFF that span multiple processes, we outline the scheme used in PG-PuReMD below:

- **Bond(i,j):** The process that owns the atom with the smaller index (indices are unique and are determined by a field in the input file) handles the bond.
- **LonePair(i):** This is a single body potential and the owner of atom i computes the energy and forces resulting from the unpaired electrons of atom i.
- **OverUnder-coordination(i):** These are multi-body interactions directly involving all bonded neighbors of atom i, computed by the owner of i.
- **Valence Angle(i,j,k):** This includes the valence angle, penalty, and three-body conjugation potentials, all of which are computed by the owner of middle atom j.
- **Dihedral Angle(i,j,k,l):** This includes the torsion and four-body conjugation potentials, both of which are handled by the owner of middle atom with the smaller index. Middle atoms here are j and k.
- **Hydrogen Bond(x,H,z):** The presence of a dynamic bond between atoms x and H implies that the owner of H atom computes this hydrogen bond interaction.
- **Nonbonded(i,j):** As in the bonded case, the owner of the atom with the smaller index computes the van der Waals and Coulomb interactions between atoms i and j.

Establishing this coordination mechanism enables us to avoid double (or multiple) computation of interactions straddling process boundaries. The ratio of such interactions to those entirely within process sub-domains can be significant, especially as sub-domain volumes decrease. The potential drawback of this approach is the reciprocal communication of forces required at the end of each time-step when processes need to compute the total forces on their assigned atoms. We adopt this approach in PG-PuReMD because force computations in ReaxFF are relatively expensive, compared to associated additional communication.

While we avoid double computations for expensive potential terms, we perform redundant computations in order to avoid the reverse communication during the matrix-vector multiplications associated with the charge-equilibration solve. This strategy results in slightly worse performance on small numbers of cores due to redundant computations; however, it delivers better performance by eliminating a costly communication step, as we scale to larger number of cores.

2) Selection of the Outer-Shell Method: The range-limited nature of force fields, associated symmetries, and relative speed of computation and communication in a parallel platform determine the choice of full-shell, half-shell, midpoint-shell, or neutral territory (zonal) methods [13]. PG-PuReMD uses the full-shell method as the domain of each MPI process for reasons discussed below.

To motivate our choice, we illustrate in Fig. 1, position information of atoms at neighboring process (P2) required by a process (P1) to compute all ReaxFF interactions that it is responsible for, based on the conventions we have adopted in Section IV-A1. Taking the maximum span among all interactions, we determine the outer-shell width r_{shell} as:

$$r_{shell} = max(3 \times r_{bond}, r_{hbond}, r_{nonb}) \tag{1}$$

A careful inspection of Fig. 1 reveals that the nature of bonded interactions in ReaxFF does not allow the use of half-shell boundaries or zonal methods. Due to the over/under-coordination and valence angle interactions, even when the midpoint boundary method is used, r_{shell} does not shrink at all. Consequently, we use the full-shell scheme in spite of its higher communication cost.

3) Inter-process communication: With the choice of 3D domain decomposition and full-shell as the outer-shells of processors, inter-process communication can either be performed using direct messaging or a staged messaging scheme. In direct messaging, each process prepares a separate message for each of its neighbors containing the required data and sends these messages using point-to-point communications. In the staged messaging scheme, every

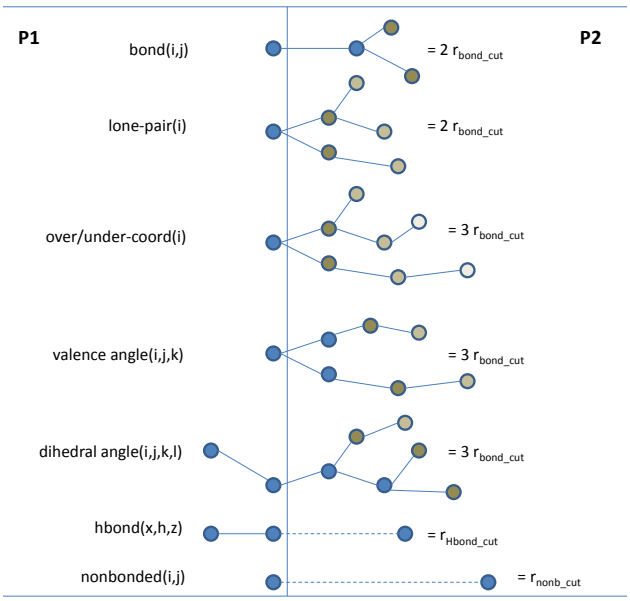


Figure 1. Handling of each interaction in ReaxFF when it spans multiple processes. Blue colored circles represent atoms that directly participate in the interaction. Gray colored circles represent atoms that directly or indirectly affect the interaction's potential and therefore experience some force due to it. We show only such atoms in the neighboring process for clarity. Lighter tones imply weaker interaction. Next to each interaction, we note its maximum span in terms of the cut-off distances in ReaxFF.

process sends/receives messages along a single dimension at each stage. In a three-stage communication scheme, for example, each process sends/receives atoms in -x, +x dimensions first, then in -y, +y dimensions and finally in -z, +z dimensions; at each stage augmenting its subsequent messages with the data it receives in previous stages. Our experiments show that staged communication scheme is most efficient as the number of processors increases. Therefore we use this scheme in our PG-PuReMD implementation.

B. CUDA Overview

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)*). Computational elements of algorithms 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 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.. Control instructions can significantly impact instruction throughput by causing threads of the same warp to diverge; that is, to 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.For a more detailed discussion on GPU architectures, we refer readers to [14].

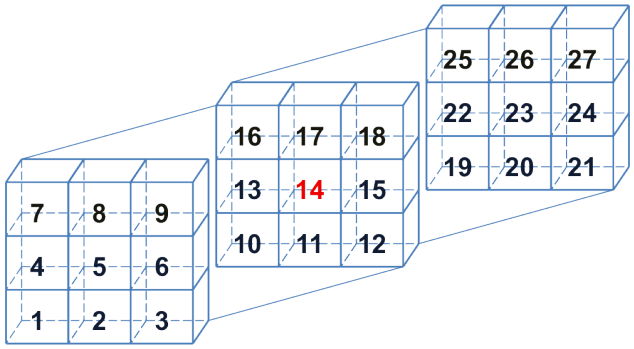
C. GPU Implementation

Once the input simulation box is decomposed into sub-domains, individual GPUs process each of these sub-domains. The major computational elements for each GPUs are: neighbor list generation, initialization, computing bonded and non-bonded interactions, and communication. Each GPU starts by identifying its own sub-domain and its outer-shell and iterates over the major computational elements for the specified number of time-steps.

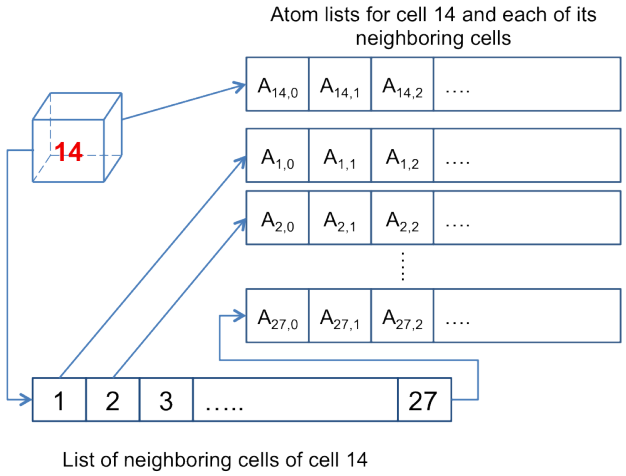
1) Computing Neighbor Lists: A significant fraction of the computation associated with an atom in ReaxFF involves other atoms within a prescribed distance from the source atom. To facilitate these computations, we construct a list of neighbors for each atom. These neighbor lists are generated by embedding a 3D grid within each process' sub-domain. Partitions induced by this 3D grid are called *cells* or *grid cells*. Using the binning method, we can realize O(k) neighbor generation complexity for each atom, where k is the average number of neighbors of any atom, by simply examining neighbor cells for identifying neighboring atoms. A typical atom may have several hundreds of atoms in its neighbor list.

During initialization, PG-PuReMD constructs neighbor-lists for each atom in its own sub-domain. Other data structures are built from the neighbor list, since the neighborhood cutoff, r_{nbr} is larger than that for other interactions.

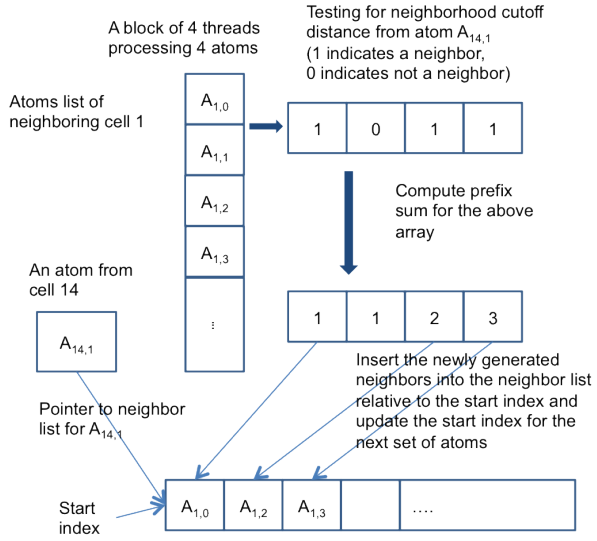
In neighbor generation, the processing of each atom may be performed by one or more threads. The case of a single thread per atom is relatively straight-forward. Each thread runs through all the neighboring cells of the given atom and identifies neighbor atoms. These neighbors are inserted into the neighbor list of the atom. Since shared data structures (cells and their atoms) are only read, and there are no concurrent writes, 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 four threads per atom. Each thread takes as argument the *atom-id* for which they compute 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



(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 a atoms-list of neighboring cell to build the neighbor-list of an atom from cell 14

Figure 2. Neighbor-list generation in PuReMD-GPU

fashion. In the illustrated case of four threads, every fourth atom in a neighbor cell is tested by the same thread.

2) Computing Bond List, Hydrogen-bond List and QEq Matrix: PG-PuReMD maintains redundant representations for bonds and the QEq matrix. For bonds (including hydrogen bonds), information is maintained at both atoms on either end of the bond. For the QEq matrix, both upper and lower triangular parts of the symmetric matrix are stored. To generate these lists, we iterate over the *neighbor-list* of each atom in its outer shell and generate *bond-list*s, hydrogen-bond list, and QEq matrix entries for both atoms. 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, combinations of these two 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 (i.e., a single function that handles all three lists). With this kernel, one may still partition the neighbor list, as 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, PG-PuReMD relies on a single kernel for each atom and all processing associated with an atom (i.e., traversal through the entire neighbor list and insertions) is handled by a single thread.

3) Implementation of Bonded Interactions: Bonded interactions in ReaxFF consist of *bond, over/under-coord, lone-pair, valence-angles, dihedral-angles 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 *valence-angles* interactions kernel generates the *three-body-list*, which is used by the *dihedral-angles* 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 input 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 (even though the occupancy is high, because of the large number of blocks created, they must wait, since CUDA limits the total number of active blocks to eight). This impacts the performance of the kernel. For this reason, we use a single thread per atom for bond-order, over/under-coord, lone-pair, valence-angles, and dihedral-angles kernels.

The *valence-angles* interaction kernel is complex, with several branch instructions (thread-divergence) and uses a large number of registers. The available set of hardware registers is shared by the entire streaming multiprocessor in the GPU. 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 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 *valence-angles* interactions, each thread iterates over *bond-list* and each bond in-turn iterates over the entire bond-list to generate the three-body list for later use. Using multiple threads per atom increases the number of CUDA thread blocks to be executed, and in each of these thread blocks each thread spends a significant number of cycles to fetch spilled variables from global memory, decreasing the kernel's performance. And because of occupancy limitations discussed earlier more thread blocks cannot be scheduled onto *Streaming Multiprocessors*. Similar arguments can be made about the *dihedral-angle* interactions kernel as well. For these reasons, PG-PuReMD creates one thread per atom for both the *valence-angles* and *dihedral-angles* kernels.

Hydrogen-bond interaction, if present, is the most expensive of all the bonded interactions. This kernel iterates over the *hydrogen-bond-list*. The number of *hydrogen-bond* entries per atom can be large (up to few hundreds per atom). The cutoff distance for hydrogen-bond terms is larger than bond cutoff distance, indicating that multiple threads per atom would yield better performance. This is because of coalesced global memory accesses. Each group of threads working on the same atom, iterates over the *hydrogen-bond* list in a strided manner (similar to neighbor-list generation), computing the hydrogen-bond energy and force on its respective atoms. A final reduction is performed, in shared memory, to compute the final force on each atom.

Require: atoms list, neighbors list
Ensure: 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, $neighbors\_list$);
8: end = end_index (my_atom, $neighbors\_list$);
9: my_index = $start$ + $lane\_id$;
10: **while** $my\_index < end$ **do**
11: **if** $neighbor\_list[my\_index]$ 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;
15: **end if**
16: my_index += $threads\_per\_atom$;
17: **end while**
18: perform parallel reduce in shared memory for coulombs/van der waals forces;
19: update force on my_atom;

Figure 3. Multiple threads per atom kernel for Coulombs and van der Waals forces

4) 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. Typically the uncorrected bond orders between atoms could be as many as 20-25 in a typical systems. This also means that when we compute the force due to the i-j bond, the bond order derivative evaluates to non-zero values 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 derivative needs to be evaluated several times over a single time-step. Storing these derivatives in memory results in costly lookups during time critical force computations. We eliminate the frequent re-computations and memory overheads of these derivatives by delaying the computation of the derivative of bond orders until the end of a time-step. During the computation of bonded potentials, coefficients for the corresponding bond order derivative terms arising from various interactions are accumulated into a scalar variable $CdBO_{ij}$. After all the bonded interactions are computed, we evaluate the derivative term and add the force to the net force on atom k directly.

5) Implementation of Non-Bonded Interactions: Non-bonded interactions consist of charge equilibration and Coulombs/van-der-Waals force computation. *Coulombs* and *van der Waals* forces 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 for this kernel as well. The process is summarized in Figure 3.

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 while loop between lines 10 and 17 simultaneously. Line 18 performs a parallel reduce operation in the shared memory to compute the final coulombs/van der waals forces of the system and net force on the atoms.

Charge equilibration corresponds to the problem of assigning partial charges to atoms with a view to minimizing electrostatic energy under constraints of charge neutrality. In the absence of electronic degrees of freedom, we do this using the QEq method of Rappe and Goddard [15]. We rely on well-known Krylov subspace methods, the preconditioned conjugate gradient method (PCG) for our purpose. Our sequential implementation [9] relies on an ILUT preconditioned GMRES method [16], [17]. However, in a parallel context our tests indicate better scalability for our diagonally scaled parallel PCG implementation. Diagonal scaling works as a cheap and effective preconditioner for the QEq problem because the coefficient matrix H carries a heavy diagonal. Consequently, all results reported in this paper use a diagonally scaled parallel CG solver for charge equilibration,

It is important to solve the QEq problem to high accuracy (low residual), otherwise the energy of the system shows unacceptable drifts as the simulation progresses in time. PCG involves a matrix-vector product and two dot products in each iteration. In a sequential context, the matrix-vector product dominates the QEq solve time. However, in a parallel context, a significant portion of the QEq solve time is spent in communications: two local communications (one for sharing the updated vector contents, and another for communicating back the partial results from matrix-vector multiplication) and two global communications (two all-reduce operations for dot products).

We use the CUBLAS library from NVIDIA for various vector operations. The sparse matrix-vector product implementation is similar to the one in [18]. 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 an optimized 2-D partitioning of the sparse matrix.

6) Data structures and memory management: In a reactive force field, the dynamic nature of bonds, valence-angles and dihedral-angles interactions, together with the significant amount of book-keeping required for these interactions require large memory and sophisticated procedures for managing allocated memory. We store all the data structures – neighbor lists, bond lists, hydrogen bond lists, and QEq matrix, in a redundant fashion (which helps to exploit the coalesced global memory operations on these lists). Before allocating memory to any of these lists we estimate the number of entries for each atom in the corresponding list at the beginning of the simulation.

For example, let eb_i denote the number of estimated bonds for atom i. We compute $max(eb_1, eb_2, \ldots, eb_n)$, and increment by certain percentage to provide additional buffer space for growth during the course of the simulation, as the estimated size of bond list of each atom. Since the average number of bonds per atom changes gradually, the number of bonds per atom for the entire system deviates only slightly from the median number of bonds per atom, resulting in a very good estimate. A similar technique is also used for neighbor list, hydrogen-bond list and QEq matrix as well. This technique has the added advantage that threads inserting entries into these data structures can compute their relative index based on the atom for which they are computing. Since atoms are exchanged between MPI processes after each time-step the starting and ending indices do not have to be recomputed at the beginning of each time-step. For the three-body list, we estimate the number of three-body interactions per bond during each time-step and compute the beginning and ending indices for each bond (three-body list is indexed by bonds per atom). This estimation kernel takes a fraction of the total time per time-step of the simulation. Since not all bonds of an atom participate in three body interaction during every time-step this method yields tremendous savings in memory compared to the estimation of this list in the PuReMD's serial implementation. At the end of each time-step, all data structures are validated for any memory overflows and usage. If any of these lists hit a prescribed high water mark then it is reallocated.

We also augment bond lists and hydrogen bond list with additional variables that are used to resolve write-write dependencies between competing threads trying to update the same memory location (forces on each atom and various energies of the system). Since atomic operations on double precision numbers in CUDA is very expensive, using augmented data structures yields tremendous performance benefits at the expense of additional memory.

V. Experimental Results

We report on our comprehensive evaluation of the performance of PG-PuReMD. All the simulations are performed on Lawrence Berkeley National Laboratory's GPU Cluster (DIRAC). This is a 50 GPU node cluster inter-connected with Quad Data Rate (QDR) InfinitiBand switch. Each GPU node contains 2 Intel 5530 2.4 GHz, 8MB cache, 5.86GT/sec QPI Quad core Nehalem processors (8 cores per node)

and 24GB DDR3-1066 Reg ECC memory. Out of the 50 nodes, 44 nodes have NVIDIA Tesla C2050 Fermi GPUs with 3GB of memory (this pool was used extensively for the evaluation of PG-PuReMD). PG-PuReMD is compiled in CUDA 5.0 environment with the following compiler options "-arch=sm_20 -funroll-loops -O3" and MPI is used to link all the object files to produce the final executable. All the arithmetic operations are double precision. Fused Multiplication Addition (*fmad*) operations are not used in the PG-PuReMD implementation so that the results perfectly match the serial implementation. Thread block size for all the kernels in this section is 256 except matrix-vector dot product which uses a block size of 512 threads. Neighbor-list kernel uses 16 threads per atom, while hydrogen-bonds, coulombs/van der waals force and matrix-vector dot product kernels uses 32 threads per atom. The model systems in all these simulations use a time-step of 0.25 *femtoseconds*, a tolerance of 10^{-6} for the QEq solver, and NVE ensemble.

We used water systems of various sizes for in-depth analysis of performance, since it represents diverse stress points for the code. For the weak scaling tests, we use water systems with 10,000 atoms per GPU and 16,000 atoms per GPU, and for strong scaling analysis, we use water systems with 80,000 and 200,000 atoms.

To better understand the results of our experiments, we identify six key parts of PG-PuReMD:

- **comm:** initial communications step with neighboring processors for atom migration and boundary atom information exchange.
- **nbrs:** neighbor generation step, where all atom pairs falling within the interaction cut-off distance r_{nbrs} are identified.
- **init_forces:** generation of the charge equilibration (QEq) matrix, bond list, and H-bond list based on the neighbors list.
- **QEq:** is the charge equilibration part that solves a large sparse linear system using Conjugate Gradients with a diagonal preconditioner. This involves costly matrix-vector multiplications and both local and global communications.
- **bonded:** is the part that includes computation of forces due to all interactions involving bonds (hydrogen bond interactions are included here as well). This part also includes identification of 3-body and 4-body structures in the system.
- **nonb:** is the part that computes nonbonded interactions (van der Waals and Coulomb).

Each of these parts has different characteristics: some are compute-bound, some are memory-bound while others are interprocess communication-bound. Together they comprise almost 99% of the total computation time for typical systems. We perform detailed analysis of these major components to better understand how PG-PuReMD responds to increasing system sizes and increasing number of processors. We also use these results to infer the impact of various machine parameters on performance.

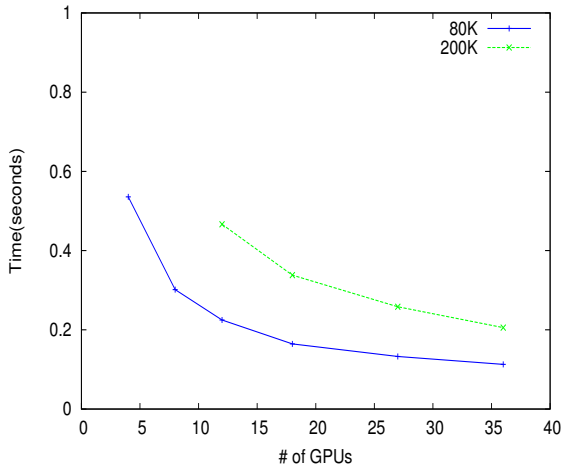


Figure 4. Strong scaling results for Water systems (Water-80K and Water-200K).

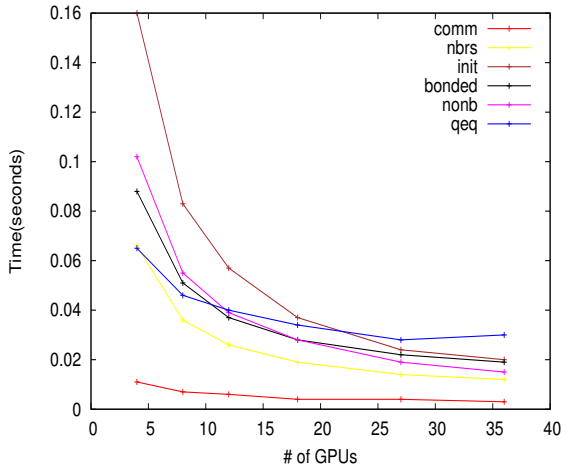


Figure 5. Strong scaling results for key components of ReaxFF for Water-80K system.

A. Strong Scaling Results

Figure 4 presents timings of PG-PuReMD for two water systems. It can be seen that as the number of GPUs increases, the time per time-step for the water systems decreases consistently, suggesting that PG-PuReMD scales well when the system size is constant while increasing the number of GPUs, at least to moderate configurations. Because of limited available memory on GPUs (3GB of global memory per GPU), the water-200K system can only be run on 12 GPUs and beyond.

Figure 5 presents the timings of major components (per time-step) for the water 80K system. *comm*, *nbrs*, *init* and

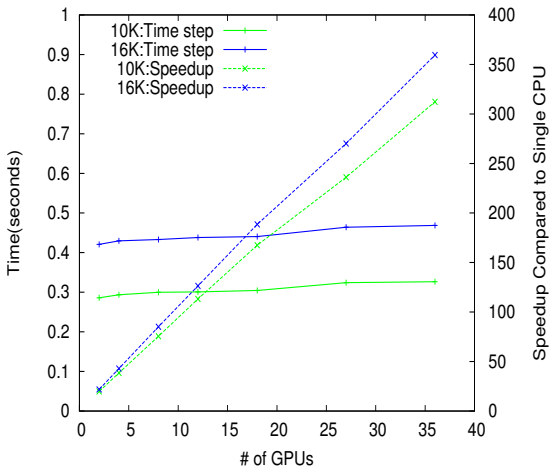


Figure 6. Weak scaling results for Water systems (10K/GPU and 16K/GPU).

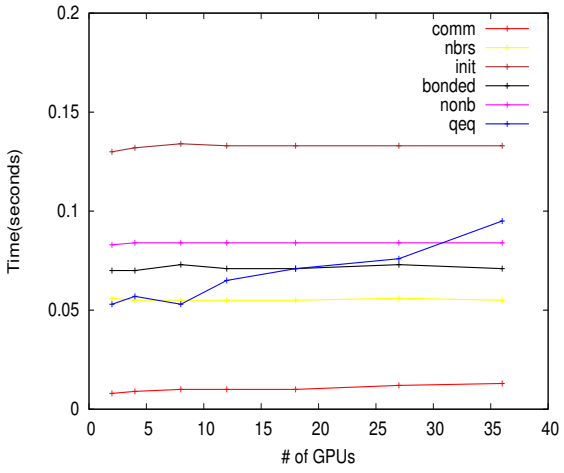


Figure 7. Weak Scaling results for key components of ReaxFF for Water system (16K/GPU).

nonb scale well as the number of GPU's is increasing for this system. The time for *comm* is almost constant after 4 GPUs. *nbrs* and *nonb* use multiple threads per atom, when using 36 GPUs, each GPU is processing about 2222 atoms. With 16 threads per atom and a block size of 256 *nbrs* kernel has about 139 blocks and *nonb* kernel (32 threads/atom) has about 278 blocks yielding enough thread blocks to keep the *SM*'s saturated because of which we see the curves for these two kernels trending downwards consistently as the GPUs are increasing for this system. Because of multiple threads per atom, the number of thread blocks decreases sharply for these two kernels when increasing the number of GPUs, which is the reason why a steep drop is noticed for these two curves for small number of GPUs. The time for the *bonded* kernel is almost constant beyond 27 GPUs. This is because all the bonded interactions, except hydrogen-bond interaction, uses single thread per atom implementation. With 27 GPUs, each GPU processes about 2962 atoms, resulting in 12 blocks each of 256 threads. From this point onwards we have less number of thread blocks than *SM*'s indicating that it reached the lower bound on its timing per time-step. Since all the bonded interactions, except hydrogen-bond interactions, uses single thread per atom implementation, we see consistent drop when using small number of GPUs and it tends to become flat after 18 GPUs, after this point the number of thread blocks created for these kernels become less than the number of *SM*'s on the GPUs. The *init* kernel, which builds bond-list, hydrogen-bond list and QEq matrix, drops sharply compared to other kernels when small number of GPUs are used, because each GPU is processing large number of atoms (quantum of work per each thread is also large) and we have large number of thread blocks. As we increase the number of GPUs, this curve tends to become flatter since the number of thread blocks becomes less than the number of *SM*'s on the GPU.

The communication bound parts of PG-PuReMD do not scale as well. *QEq* is one of the most expensive parts of PG-PuReMD. Since QEq involves four communication operations (two message exchanges and two global reductions) in every iteration of the linear solver, as the number of GPUs increases the communication overhead increases as well. Moreover, as the sub-domain size decreases, the amount of computation decreases and the communication overhead starts to dominate.

B. Weak Scaling Results

We used water systems with 10,000 atoms per GPU and 16,000 atoms per GPU for benchmarking the PG-PuReMD application under weak scaling scenarios. For water systems with 16,000 atoms per GPU, we achieve a speedup of 359x on 36 GPUs when compared to single CPU time for the same water system. Figure 6 illustrates the weak scaling performance and speedup's achieved.

Figure 7 presents weak scaling results of water system with 16K atoms/GPU. For all the components, we observe that the time per time-step is almost constant, yielding excellent scaling over the range of GPUs used. An interesting observation can be made about the *QEq* computation – this computation is heavily dependent on the communication. Furthermore, it is executed in a lockstep fashion for every iteration of the linear solve. The *QEq* time almost doubles from 4 GPUs case to 36 GPUs. This can be attributed to the amount of communication and execution of each iteration in a lock step fashion.

Table I presents efficiency results for water systems under weak scaling. We achieve a speedup of about 10x per MPI process when compared to PuReMD implementation. This results in decreased parallel efficiency as we increase the number of GPUs. PuReMD [10] achieves an efficiency of 99% with 32 MPI processes; PuReMD runs on CPUs and

	10K per GPU		16K per GPU	
GPU's	$\frac{QEq}{TotalTime}\%$	Efficiency	$\frac{QEq}{TotalTime}\%$	Efficiency
1	11.80	100.00	10.99	100.00
2	13.11	95.97	11.66	99.17
4	14.61	93.46	12.33	97.10
8	15.45	91.52	12.93	96.33
12	16.10	91.21	13.61	95.27
18	16.47	90.17	13.70	94.74
27	20.53	84.68	17.78	89.92
36	21.04	84.02	18.25	89.04

Table I
EFFICIENCY RESULTS OF WATER SYSTEM FOR WEAK SCALING SIMULATIONS

it takes 16 MPI processes to completely use a single CPU. In comparison PG-PuReMD achieves about 84% efficiency for comparable number of MPI processes. The reduction in efficiency is a consequence of the faster serial execution by the GPUs.

VI. CONCLUSION

In this paper, we presented an efficient and scalable parallel implementation of ReaxFF using MPI on CUDA platforms. Our open-source implementation is shown to achieve a 350x speed up compared to single CPU implementation under weak-scaling scenarios. PG-PuReMD's accuracy has been verified against the benchmark production PuReMD code by comparing various energy and force terms for large numbers of time-steps under diverse application scenarios and systems.

ACKNOWLEDGMENT

We thank Adri van-Duin for significant help in validating our software on a variety of systems. We also thank Joe Fogarty at the University of Southern Florida for constructing model systems for testing and validation and Lawrence Berkeley National Laboratories for providing us access to their GPU cluster for benchmarking our application.

REFERENCES

[1] A. C. T. van Duin, S. Dasgupta, F. Lorant, and W. A. G. III, "Reaxff: A reactive force field for hydrocarbons," in *J Phys Chem A*, vol. 105, 2001, pp. 9396–9409.

[2] K. D. Nielson, A. C. T. van Duin, J. Oxgaard, W.-Q. Deng, and 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.

[3] K. Chenoweth, S. Cheung, A. C. T. van Duin, W. A. G. III, and 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.

[4] 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.

[5] Y. Park, H. M. Aktulga, A. Y. Grama, and A. Strachan, "Strain relaxation in si/ge/si nanoscale bars from md simulations," in *J Appl Phys*, vol. 106, 2009, p. 034304.

[6] J. C. Fogarty, H. M. Aktulga, A. C. T. van Duin, A. Y. Grama, and S. A. Pandit, "A reactive simulation of the silica-water interface," in *J Chem Phys*, vol. 132, no. 174704, 2010.

[7] S. J. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," in *J Comp Phys*, vol. 117, 1995, pp. 1–19.

[8] A. Thompson and H. Cho, "Lammps/reaxff potential," April 2010. [Online]. Available: http://lammps.sandia.gov/doc/pair_reax.html

[9] H. M. Aktulga, S. Pandit, A. C. T. van Duin, and A. Grama, "Reactive molecular dynamics: Numerical methods and algorithmic techniques," in *SIAM J. Sci. Comput*, vol. 34(1), pp. C1–C23.

[10] H. M. Aktulga, J. C. Fogarty, S. A. Pandit, and A. Y. Grama, "Parallel reactive molecular dynamics: Numerical methods and algorithmic techniques," in *Parallel Computing*, vol. 38(4-5), pp. 245–259.

[11] M. Zheng, X. Li, and L. Guo, "Algorithms of gpu-enabled reactive force field (reaxff) molecular dynamics," in *J Mol Graph Model*, vol. 41, 2013, pp. 1–11.

[12] 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, and S. C. Wang, "Anton: A special-purpose machine for molecular dynamics simulation," in *ISCA*, June 2007.

[13] K. J. Bowers, R. O. Dror, and D. E. Shaw, "Zonal methods for the parallel execution of range-limited n-body simulations," in *J Comp Phys*, vol. 221, 2007, pp. 303–329.

[14] "Nvidia white paper on fermi architecture." [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[15] A. K. Rappe and W. A. G. III, "Charge equilibration for molecular dynamics simulations," in *J Phys Chem*, vol. 95, 1991, pp. 3358–3363.

[16] Y. Saad and 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.

[17] Y. Saad, "Iterative methods for sparse linear systems," in *SIAM*. SIAM, 2003.

[18] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," December 2008.