Implementing Molecular Dynamics on Hybrid High Performance Computers - Short Range Forces

W. Michael Brown[a,*], Peng Wang[b], Steven J. Plimpton[c], Arnold N. Tharrington[d]

[a]*National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA*
[b]*NVIDIA, Santa Clara, CA, USA*
[c]*Sandia National Laboratory, Albuquerque, New Mexico, USA*
[d]*National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA*

Abstract

The use of accelerators such as graphics processing units (GPUs) has become popular in scientific computing applications due to their low cost, impressive floating-point capabilities, high memory bandwidth, and low electrical power requirements. Hybrid high-performance computers, machines with more than one type of floating-point processor, are now becoming more prevalent due to these advantages. In this work, we discuss several important issues in porting a large molecular dynamics code for use on parallel hybrid machines - 1) choosing a hybrid parallel decomposition that works on central processing units (CPUs) with distributed memory and accelerator cores with shared memory, 2) minimizing the amount of code that must be ported for efficient acceleration, 3) utilizing the available processing power from both multi-core CPUs and accelerators, and 4) choosing a programming model for acceleration. We present our solution to each of these issues for short-range force calculation in the molecular dynamics package LAMMPS, however, the methods can be applied in many molecular dynamics codes. Specifically, we describe algorithms for efficient short range force calculation on hybrid high-performance machines. We describe an approach for dynamic load balancing of work between CPU and accelerator cores. We describe the Geryon library that allows a single code to compile with both CUDA and OpenCL for use on a variety of accelerators. Finally, we present results on a parallel test cluster containing 32 Fermi GPUs and 180 CPU cores.

Keywords:
Molecular dynamics, GPU, hybrid parallel computing

1. Introduction

Graphics processing units (GPUs) have become popular as accelerators for scientific computing applications due to their low cost, impressive floating-point capabilities, and high memory bandwidth. Numerous molecular dynamics codes have been described that utilize GPUs to obtain impressive speedups over a single CPU core [29, 33, 18, 1, 19, 4, 6, 13, 26, 10]. The incorporation of error-correcting codes and double-precision floating-point into GPU hardware now allows the accelerators to be used by production codes. These advances have made accelerators an important consideration in high-performance computing (HPC). Lower cost, electrical power, space, cooling demands, and reduced operating system images are all potential benefits from the use of accelerators in HPC [16]. Several hybrid platforms that include accelerators in addition to conventional CPUs have already been built and more are planned.

The trend toward hybrid HPC platforms that use accelerators in addition to multi-core CPUs has created a need for new MD algorithms that effectively utilize all of the floating-point capabilities of the hardware. This has created several complications for developers of parallel codes. First, in addition to a parallel decomposition that divides computations between nodes with distributed memory, the workload must be further divided for effective use of shared memory accelerators with many hundreds of cores. This can be accomplished, for example, by further dividing the simulation domain into smaller partitions for use by accelerator processors. Other options include decomposing the work by atom or using a force-decomposition that evenly divides force computation among accelerator cores.

In addition to partitioning the work among accelerator cores, it will likely be beneficial in many cases to also utilize the processing power of CPU cores. This is due to the fact that many GPU algorithms are currently unable to achieve peak floating-point rates due to poor arithmetic intensity. GPU speedups are often reported versus a single CPU core or compared to the number of CPU processors required to achieve the same wall time for a parallel job. For HPC, it is important to consider the speedup versus multi-core CPUs and take into account non-ideal strong scaling for parallel runs. Comparisons with a similar workload per node on clusters with multi-core nodes are more competitive. In addition to algorithms with poor arithmetic intensity, some algorithms might benefit from assigning

*Corresponding author.
Email addresses: `brownw@ornl.gov` (W. Michael Brown), `penwang@nvidia.com` (Peng Wang), `sjplimp@sandia.gov` (Steven J. Plimpton), `arnoldt@ornl.gov` (Arnold N. Tharrington)

different tasks to CPU and accelerator cores, either because one task might not be well-suited for processing on an accelerator or simply because the time to solution is faster. Additionally, utilizing CPU cores for some computations can reduce the amount of code that must be ported to accelerators.

Another complication that arises in software development for hybrid machines is the fact that the instruction sets differ for the CPU and accelerators. While software developers have had the benefit of "free" performance gains from continuously improving x86 performance, software for hybrid machines currently must be ported to a compiler suitable for the new architectures [8]. It is undesirable to port an entire legacy code for use on accelerators [22], and therefore minimizing the routines that must be ported for efficient acceleration is an important concern. This is further complicated by the fact that developers must choose a programming model from a choice of different compilers and/or libraries. Development tools are less mature for accelerators [6] and therefore might be more susceptible to performance sensitivities and bugs. Currently, the CUDA Runtime API is commonly used for GPU acceleration in scientific computing codes. For developers concerned with portability, OpenCL offers a library that targets CPUs in addition to GPUs and that has been adopted as an industry standard [28]. Our current concerns in adopting the OpenCL API include the immaturity of OpenCL drivers/compilers and the potential for lagging efficiency on NVIDIA hardware.

In this work, we present our solution to these issues in an implementation of MD for hybrid high-performance computers in the LAMMPS molecular dynamics package [23], however, the methods are applicable to many parallel MD implementations. We describe our algorithms for accelerating neighbor list builds and short-range force calculation. Our initial focus on short-range force calculation is because 1) short-range models are used extensively in MD simulations where electronic screening limits the range of interatomic forces and 2) short-range force calculation typically dominates the computational workload even in simulations that calculate long-range electrostatics. We evaluate two interatomic potentials for acceleration - the Lennard-Jones (LJ) potential for van der Waals interactions and the Gay-Berne potential for ellipsoidal mesogens. These were chosen in order to present results at extremes for low arithmetic intensity (LJ) and high arithmetic intensity (Gay-Berne) in LAMMPS. We describe an approach for utilizing multiple CPU cores per accelerator with dynamic load balancing of short-range force calculation between CPUs and accelerators. We describe the Geryon library that allows our code to compile with both CUDA and OpenCL for use on a variety of accelerators. Finally, we present results on a parallel test cluster containing 30 Fermi GPUs and 180 CPU cores.

2. Methods

2.1. LAMMPS

In this work, we are considering enhancements to the LAMMPS molecular dynamics package [23]. LAMMPS is parallelized via MPI, using spatial-decomposition techniques that partition the simulation domain into smaller subdomains, one per processor. It is a general purpose MD code capable of simulating biomolecules, polymers, materials, and mesoscale systems. It is also designed in a modular fashion with the goal of allowing additional functionality to be easily added. This is achieved via a variety of different *style* choices that are specified by the user in an input script and control the choice of force-field, constraints, time integration options, diagnostic computations, etc. At a high level, each style is implemented in the code as a C++ virtual base class with an appropriate interface to the rest of the code. For example, the choice of *pair* style (e.g. lj/cut for Lennard-Jones with a cutoff) selects a pairwise interaction model that is used for force, energy, and virial calculations. Individual pair styles are child classes that inherit the base class interface. Thus, adding a new pair style to the code (e.g. lj/cut/hybrid) is as conceptually simple as writing a new class with the appropriate handful of required methods or functions, some of which may be inherited from a related pair style (e.g. lj/cut). As described below, this design has allowed us to incorporate support for acceleration hardware into LAMMPS without significant modifications to the rest of the code. Ideally, only the computational kernel(s) of a pair style or other class need to be re-written to create the new derived class.

2.2. Accelerator Model

For this work, we consider accelerators that fit a model suited for OpenCL and CUDA. Because OpenCL and CUDA use different terminology, we have listed equivalent (in the context of this paper) terms in Table 1. Here, we will use OpenCL terminology. The *host* consists of CPU cores and associated addressable memory. The *device* is an accelerator consisting of 1 or more multiprocessors each with multiple cores (note that for OpenCL this device might be the CPU). The device has *global memory* that may or may not be addressable by the CPU, but is shared among all multiprocessors. Additionally, the device has *local memory* for each multiprocessor that is shared by the cores on the multiprocessor. Each core on the device executes instructions from a work-item (this concept is similar to a thread running on a CPU core). We assume that the multiprocessor might require SIMD instructions and branches that could result in divergence of the execution path for different work-items are a concern. In this paper, the problem is referred to as *work-item divergence*. We also assume that global memory latencies can be orders of magnitude higher when compared to local memory access.

We assume that access latencies for coalesced memory will be much smaller. Coalesced memory access refers to sequential memory access for data that is correctly aligned in memory. This will happen, for example, when data needed by individual accelerator cores on a multiprocessor can be "coalesced" into a larger sequential memory access given an appropriate byte alignment for the data. Consider a case where each accelerator core needs to access one element in the first row of a matrix with arbitrary size. If the matrix is row-major in memory, the accelerator can potentially use coalesced memory access; if the matrix is column-major, it cannot. The penalties for incorrect

Table 1: Equivalent OpenCL and CUDA terminology.

OpenCL	CUDA
Local memory	Shared memory
Work-item	Thread
Work-group	Thread Block
Command Queue	Stream

alignment or access of non-contiguous memory needed by accelerator cores will vary depending on the hardware.

A *kernel* is a routine compiled for execution on the device. The work for a kernel is decomposed into a specified number of *work-groups* each with a specified number of *work-items*. Each work-group executes on only one multiprocessor. The number of work-items in a work-group can exceed the number of cores on the multiprocessor, allowing more work-items to share local memory and the potential to hide memory access latencies. The number of registers available per work-item is limited. A device is associated with one or more *command queues*. A command queue stores a set of kernel calls and/or host-device memory transfers that can be executed asynchronously with host code.

2.3. Parallel Decomposition

The parallel decomposition for hybrid machines consists of a partitioning of work between distributed memory nodes along with a partitioning of work on each node between accelerator cores and possibly CPU cores. For LAMMPS, we have chosen to use the existing spatial decomposition [23] to partition work between MPI processes with each process responsible for further dividing the work for an accelerator. This is similar to the approach that is used in NAMD acceleration [29]. An alternative task-based approach has also been proposed [13]; however, this has been designed only to scale to multiple GPUs on a single desktop.

The partitioning of work for the accelerator can be achieved in several ways. For MD, we can divide the simulation into routines for neighbor calculation, force calculation, and time integration - all of which can potentially be ported for acceleration. Previous work has included running only the force calculation on the GPU [33], running the neighbor and force calculations on the GPU [29, 18, 26], and running the entire simulation on the GPU [1, 19, 4, 6, 13]. A breakdown of the time spent in each routine for a CPU simulation in LAMMPS for the LJ and Gay-Berne test cases is shown in Figure 1 up to 180 cores (12 cores per node). Because the time integration represents a small fraction of the work load, we have focused our initial work on porting neighbor and force routines for acceleration. The advantage of this approach is that the many auxiliary computations in LAMMPS, used for time integration or calculation of thermodynamic data, do not need to be ported or maintained for acceleration to be fully compatible with LAMMPS features. The disadvantage is that on current accelerators, all data must be transferred from the host to the device memory and vice-versa on each timestep, not just ghost particles for interprocess communication. Additionally, the integration time will become a larger fraction of the computational effort in the accelerated code.

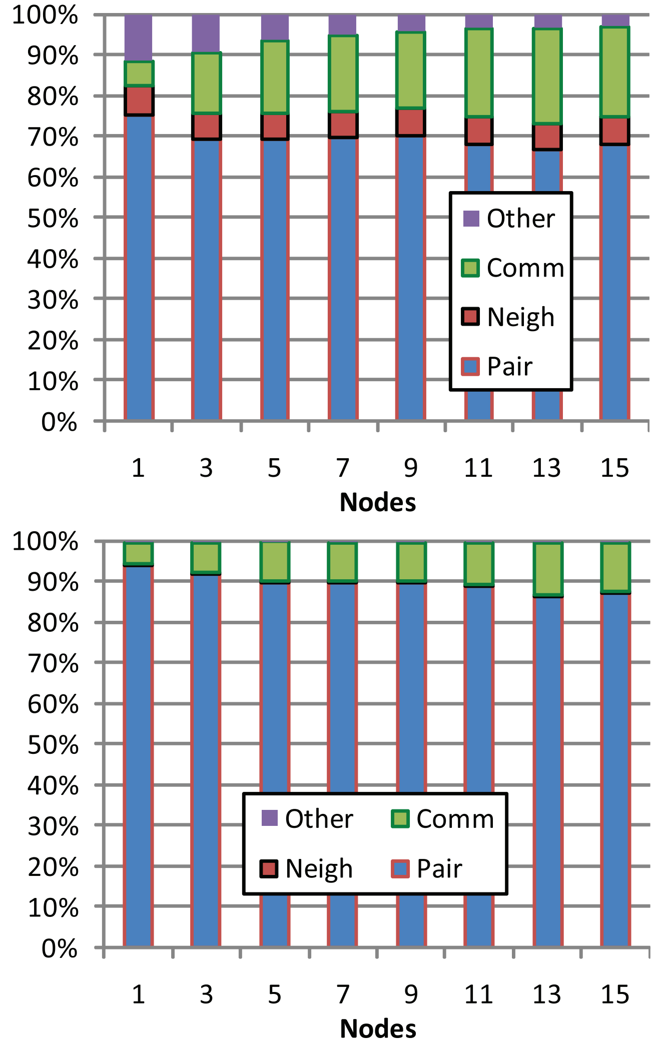

Figure 1: Percentage of loop time spent on pairwise forces, neighbor calculation, and MPI communications for LAMMPS on a conventional cluster (dual hex-core Opteron per node). Top: Breakdown for a strong scaling benchmark using the Lennard-Jones potential with a cutoff of 2.5 and 864000 particles. Bottom: Breakdown for a strong scaling benchmark using the Gay-Berne potential with a cutoff of 7 and 125000 particles.

Due to the modularity in LAMMPS, acceleration is achieved with no significant modification to the existing code. A new *pair style* using acceleration is derived from the non-accelerated parent class. The accelerated pair style does not request a neighbor list from the CPU, calculating the list on the accelerator instead. A user can switch from CPU calculation of the neighbors and forces to accelerator calculation by switching the pair style in the input script. Although it is not a focus of this work, the same procedure can be used to port additional functionality to the accelerator. By adding a new option to the pair style and deriving computes and fixes that utilize accelerators, the host-device communication can be reduced with data transfer for all particles only when necessary for I/O.

Options for parallel decomposition on the accelerator for neighbor, force, and time integration routines include spatial decomposition, atom decomposition, force decomposition or some combination of these approaches for different routines. The algorithms we chose, along with the strategy for partitioning work between the CPU and accelerator are discussed below.

2.4. Neighbor List Calculation

For short-range force calculations in MD, the force summations are restricted to atoms within some small region surrounding each particle. This is typically implemented using a cutoff distance r_c, outside of which particles are not used for force calculation. The work to compute forces now scales linearly with the number of particles. This approach requires knowing which particles are within the cutoff distance r_c at every timestep. The key is to minimize the number of neighboring atoms that must be checked for possible interactions. Traditionally, there are two basic techniques used to accomplish this. The first idea, that of neighbor lists, was originally proposed by Verlet [31]. For each atom, a list is maintained of nearby atoms. Typically, when the list is formed, all neighboring atoms within an extended cutoff distance $r_s = r_c + \gamma$ are stored. The list can be used for multiple timesteps until an atom has have moved from a distance $r > r_s$ to $r < r_c$. The optimal value for γ will depend on simulation parameters, but is typically small relative to r_c.

The second technique commonly used for speeding up MD calculations is known as the link-cell method [15]. At every timestep, all the atoms are binned into 3-D cells of side length d where $d = r_c$ or slightly larger. This reduces the task of finding neighbors of a given atom to checking in 27 bins. Since binning the atoms requires only $O(N)$ work, the extra overhead associated with it is acceptable for the savings of only having to check a local region for neighbors. The fastest approach will typically be a combination of neighbor lists and link-cell binning and this is the approach used in LAMMPS.

In the combined method, atoms are binned only once every few timesteps for the purpose of forming neighbor lists. In this case atoms are binned into cells of size d, and a stencil of bins that fully overlap a sphere of radius r_s is defined. For each particle in a central bin, the stencil of surrounding bins is searched to identify the particle's neighbor list. At intermediate timesteps the neighbor lists alone are used in the usual way to find neighbors within a distance r_c of each atom. The optimal choice of bin size is typically $d = 0.5r_s$. The combined method offers a significant savings over a conventional link-cell method since there are far fewer particles to check in a sphere of volume $4\pi {r_s}^3$ than in a cube of volume $27{r_c}^3$. For pairwise forces, additional savings can be gained due to Newton's third law by computing a force only once for each pair of particles. In the combined method this is done by searching only half the stencil of bins surrounding each atom to form its neighbor list. This has the effect of storing atom j in atom i's list, but not atom i in atom j's list thus halving the number of force computations to perform. Here, we refer to this as a *half neighbor list* as opposed to a *full neighbor list*, where the i, j pair is stored twice.

In GPU implementations, all three approaches have been used for calculating neighbors. Neighbor lists calculated with a brute force distance check of all pairs of particles have been used [6, 5, 26]. Although this approach has a $O(N^2)$ time complexity, it can be faster than other approaches for a smaller number of particles because coalesced memory access can be used to load particle positions. Link cell approaches have also been implemented [29, 19, 24, 13]. This approach is convenient for accelerated force calculations that use a spatial decomposition; particles in each cell can be stored in local memory for much faster access in cutoff and force evaluation. The combined approach has also been used [1, 4] and is our choice for most of the potential energy models used in LAMMPS. This is due to the favorable time complexity and the atom decomposition used for force calculations in accelerated potentials. The drawback is that the neighbor kernel can become memory bound due to the non-coalesced global memory fetches required to obtain each particle's position. In CPU implementations, sorting atoms by spatial position can be used to decrease memory latencies [20] and this approach is currently used in LAMMPS with a sort occurring at some specified frequency. This approach has also been shown to improve performance in GPU-accelerated neighbor calculations [1] and it is the approach used in the HOOMD simulation package. In our implementation we do not implement spatial sorting for the accelerator, but rely on the CPU sort to reduce cache miss counts.

In our implementation, two arrays are used for storing the particles in each cell, the *cell list*. Let *nlocal* be the number of particles in the subdomain for a process, *nghost* be the number of ghost particles for the process, and *nall* = *nlocal* + *nghost*. One array, *CellID*, of at least size *nall* lists the cell id of all local and ghost particles in a packed manner. A second array, *CellList*, of size *ncell*+1 lists the starting position of each cell in the first array. Here *ncell* is the total number of cells. One advantage of this storage scheme is that it can handle cases where a few cells have significantly more particles than most other cells with much more efficient memory utilization. This data structure may lead to misaligned access of the cell list which may lead to a performance drop. However, this is not a problem in practice for two reasons. First, the neighbor-list build kernel is compute-bound which means the memory load time is only a small fraction of the whole kernel time. Thus, even if the load time may increase due to the misaligned access, it will not increase the kernel time significantly. Second, the new L1 cache on NVIDIA's Fermi architecture significantly reduces the performance drop due to misaligned access.

The algorithm for building the cell list is divided into 4 steps.

1. Initialize CellList[i]=i.
2. Calculate CellID.
3. Sort using CellID and CellList as the key-value pair.
4. Calculate CellCounts from CellList.

Steps 1 and 2 are embarrassingly parallel and can be combined into a single kernel. In this kernel, work-item i is assigned to particle i and will calculate the ID of the cell particle i belongs to and store the result to CellID[i].

For step 3, we use the radix sort routine in the NVIDIA CUDPP library [25], with CellID as the key and CellList as the value. After the sort, particles belonging to each cell will

be ordered correctly. In step 4, the number of particles in each cell are counted. We launch at least *nall* work-items. Work-items 0 and $nall - 1$ first initialize boundary cases. Then each remaining work-item with ID less than *nall* checks the cell ID of the work-item to its left; if it is different from the current work-item, the current work-item corresponds to a cell boundary. In this case, the work-item will store the boundary position to the *CellCounts* array. The *CellList* array is allocated at the beginning of the simulation with storage for $nall \times 1.10$ particles. If the number of local and ghost particles grows past this value, it is reallocated, again allowing room for up to 10% more particles. The *CellCounts* array is allocated at each cell list build using the the current size of the simulation subdomain. In our implementation, the CUDA compilation resulted in a kernel that uses 12 registers and 68 bytes of local memory for the cell ID calculation and 6 registers and 32 bytes of local memory for the CellCounts kernel.

After the cell list is built, the neighbor list kernel is executed for building the neighbor list. The neighbor list requires an array with storage for at least *nlocal* counts of the number of neighbors for each local particle. Additionally, a matrix with at least enough storage for $n_{max} \times nlocal$ neighbors is required where n_{max} is the current maximum number of neighbors for a particle. Initially, space is allocated for $nlocal \times 1.1$ counts in the array and $300 \times nlocal \times 1.1$ neighbors in the matrix. If the number of neighbors is found to be greater than the available storage space, reallocation is performed, again reserving room for 10% extra neighbors or local particles. Using a matrix to store neighbors is inefficient when the density of particles is not uniform throughout the simulation box. This approach is beneficial for GPU implementations, however, because it allows for coalesced memory access of neighboring particles during the force calculation [1]. For cases where a relatively small number of particles have a much greater neighbor count (e.g. colloidal particles in explicit solvent), we have shown that a tail list implementation can provide for efficient memory access [32]. In the tail list implementation, the last row of the neighbor matrix can be used to point to additional neighbors stored in a separate packed array.

The neighbor list kernel assigns one work-group to one cell. One work-item in the group calculates the neighbor list for one particle in the cell. If the particle number is larger than the block size, the block will iterate over the particles until all particles are processed. When calculating the neighbors for a particle, the work-item iterates through all the particles in its cell along with the 26 neighboring cells and calculates their distances to the particle. If the work-item finds a particle that is within r_s, it will increase the neighbor count and add the neighbor to the neighbor list. Note that we are evaluating 26 neighbor cells in the accelerated case to build a full neighbor list. This is because the cost of atomic operations to avoid memory collisions in the force update is currently generally greater than doubling the amount of force calculations that must be performed [1]. In our implementation, the CUDA compilation of the neighbor list kernel used 42 registers and 1360 bytes of local memory for single precision.

The kernels for building the cell and neighbor lists can be compiled to use particle positions in either single or double precision. Because the kernels do not have the full functionality of the LAMMPS neighbor list (e.g. simulation in a triclinic box), acceleration for the neighbor list build is an option that can be specified in the input script. In the case where acceleration is not used for neighbor builds, data must be copied to the device and organized into the same neighbor matrix and counts array format used in the accelerated build. Whenever a neighbor rebuild occurs on the CPU, this is accomplished using a packed neighbor array of size 131072 on the host. This size was chosen to reduce the footprint of write-combined memory on the host and to allow for packing in a loop concurrently with data transfer to the device. In this process, a double loop over all local particles and over all particle neighbors is used to pack the array until it is filled with 131072 neighbors. Then, an asynchronous copy of the packed array to the device is placed in the command queue. The process is repeated using a second array of 131072 elements. Once this has been filled, the host blocks to assure that the first data transfer has finished, starts another data transfer, and repeats the process. A second host array allocated with at least *nlocal* elements, stores the starting offset within the device packed array of each particles neighbors. After all neighbors have been copied to the device, this array is copied to a similarly sized array allocated on the device.

This is followed with execution of an *unpack* kernel on the device. In this kernel, each work-item is assigned an atom. The work-item loops over all neighbors in the packed array, placing them in the neighbor matrix for coalesced access in the force evaluation. It should be noted that the time for neighbor list build for pair-potentials will be longer than that typically performed on the CPU because a full neighbor list must be built for accelerator force calculation instead of a half list. The unpack kernel requires 5 registers and 28 bytes of shared memory for the CUDA compilation.

For both neighbor calculation and force evaluation, atom positions must be copied to the device. In LAMMPS, positions are stored in a $3 \times nall$ double precision matrix. Because length 3 vectors are inefficient (and in fact not currently defined) for OpenCL, we use length 4 vectors to store each atom's position. The extra element is used to store the particle type. This allows the position and type to be obtained in a single fetch, but requires repacking the atom positions at each timestep. Because single precision is much more efficient on many accelerators, this step can include type-casting to single precision. The array for repacking positions is stored in write-combined memory on the host. Using the same technique as all other allocations, the array initially allows room for 10% more particles and ghosts than currently in the subdomain with reallocation as necessary.

2.5. Force Kernels

Force calculation has typically been implemented in GPU codes using atom decompositions. A notable exception is the NAMD code [24]. In their approach, termed small-bin cutoff summation, each work-group selects which bin it will traverse. The atoms in the bin are loaded into local memory to allow very fast access to particle positions. The drawback of this approach is that the maximum number of atoms that can be evaluated in

a work-group is dictated by the amount of local memory on the device. This will change depending on the device and precision used to store the positions. In NAMD, when atom positions will not fit in local memory, "bin overflow" is calculated concurrently on the CPU. Because LAMMPS is used for a wide variety of simulations, some with very large neighbor lists [30], we have chosen to use an atom decomposition as the general framework for the accelerator implementations. In this case we rely on the use of spatial sorting and improving cache size and performance on accelerators to reduce memory access latencies.

Most implementations use analytic expressions for force calculation but some have used force interpolation [29, 13]. The drawback for force interpolation for potential models with low arithmetic intensity is that the number of memory fetches is increased in kernels that are already memory-bound [1]. For spatial decompositions that effectively use local memory, this might not impact performance significantly, especially when graphics texture memory can be used for fast interpolation. For the atom decomposition used here, we have chosen to use analytic expressions in order to minimize memory access and maintain consistency with the CPU LAMMPS calculations. We note, however, that any given pair style can choose to use interpolation for acceleration and some will require it.

The general steps in the force kernel are:

1. Load particle type data into local array position *work_item_id*
2. Block until data in local memory loaded by all work-items in group
3. Load particle position and type i = *global_id*
4. Load extra particle data for i
5. Load neighbor count n for particle i
6. for ($jj = 0$; $jj < n$; jj++) {
7. Load neighbor jj to obtain index j
8. Load position and type for j
9. Calculate distance between i and j
10. if (distance<cutoff) {
11. Load extra particle data for j
12. Accumulate force and possibly torque
13. if (thermo_energy)
14. Calculate and accumulate energy
15. if (thermo_virial)
16. Calculate and accumulate virial
17. }
18. }
19. Store particle force and possibly torque
20. if (thermo_energy)
21. Store particle energy
22. if (thermo_virial)
23. Store particle virial

Extra particle data in the listing includes data other than the position and type that are needed for force calculation. Because the Gay-Berne potential is anisotropic, a quaternion representing particle orientation is loaded in addition to the particle position. Another example is particle charge. The global memory access in listing numbers 1, 3, 4, 5, 19, 21, and 23 will be coalesced. The access in numbers 8 and 11 will not necessarily be coalesced. For simple potential models, the loop over non-coalesced memory access will cause the kernel to be memory-bound. For more complicated models, there is another source of inefficiency. This is due to work-item divergence resulting from some particles pairs with separation distances greater than the force cutoff. To address this issue, we have implemented a *cutoff* kernel for potentials with high arithmetic intensity that will evaluate and pack only the neighbors within the cutoff at each time step. The force kernel can then be called without a branch for checking the cutoff. This requires twice the memory for neighbor storage but can be more efficient for complicated models such as the Gay-Berne potential.

As with the particle positions, forces and torques are stored in length 4 vectors in order to maintain alignment that is efficient for accelerators. The extra position is unused. This currently results in some penalty for repacking the length 4 vectors into length 3 vectors for the CPU. Instructions to copy force/torque and possibly energy/virial terms are placed in the command queue after the force kernel call. Energy and virial terms are accumulated on the host to allow compatibility with statistics and I/O that need per-particle energies or virials.

Force kernels can be compiled to use single, double, or mixed precision. The drawback of double precision for memory-bound kernels is that twice as many bytes must be fetched for cutoff evaluation. A potential solution is to use mixed precision. In this case, the positions are stored in single precision, but accumulation and storage of forces, torques, energies, and virials is performed in double precision. Because this memory access occurs outside the loop, the performance penalty for mixed precision is very small.

2.6. Load Balancing

Because most machines will have multi-core CPUs in addition to accelerators, algorithms that take advantage of both resources will likely have the best performance for many problems. This idea has already proven beneficial in several codes. One approach is to overlap short-range calculations with parts of the long-range electrostatics calculation [22] and this approach has shown impressive speed-ups for protein simulations in LAMMPS [10]. In another approach, different parts of the long range electrostatics calculation in multilevel summation are run concurrently on the CPU and GPU [11]. As already discussed, concurrent CPU execution of bin overflow in NAMD is used to handle cases where there is insufficient local memory to store all particles [24].

In all of these approaches, the partitioning of work between the CPU and GPU is fixed and therefore the algorithms cannot make optimal use of hybrid resources. Since short range force calculations typically dominate the computational work for most MD simulations, dynamic partitioning of short-range force calculation between the accelerator(s) and CPU cores is an attractive possibility. In our approach, time integration and possibly neighbor list builds are performed on the CPU and therefore dividing this work between all available CPU cores would be beneficial. Therefore, we have implemented a load

balancing capability that allows all CPU cores on a node to perform calculations, regardless of the number of accelerators.

In LAMMPS, a natural approach to achieve this is to run an MPI process for every core on a node and allow multiple MPI processes to share the same accelerator. Host-device data transfers and force calculation from multiple processes can be placed in the queue for execution on the same device. This can improve performance for several reasons. First, the calculations for routines that have not been ported for accelerators can be split between multiple cores. Second, the work will be partitioned spatially, and therefore memory latencies for accelerated routines will possibly be improved with better data locality. Finally, force calculation can be divided between CPUs and accelerators in order to utilize all floating-point processors on a node. The disadvantage of this approach arises when the number of particles per core becomes so small that each process does not have sufficient work to utilize the accelerator efficiently.

In our implementation, fixed load balancing can be achieved by setting the CPU core to accelerator ratio and by setting the fraction of particles that will have forces calculated by the accelerator. For example, consider a job run with 4 MPI processes on a node with 2 accelerator devices and the fraction set to 0.7. Each accelerator will be shared by 2 MPI processes. At each timestep, each MPI process will place data transfer of positions, kernel execution of forces, and data transfer of forces into the device queue for 70 percent of the particles. At the same time data is being transferred and forces are being calculated on the device, the MPI process will perform force calculation on the CPU. For this case, the ideal fraction would result in a CPU time for each process that is equal to the device time for data transfer and kernel execution for both processes sharing the device. Because this is difficult to know in advance, we have implemented an approach for dynamic balancing with calculation of the optimal fraction based on CPU and device timings at some timestep interval.

The approach requires some knowledge of how the MPI processes are mapped to nodes in a given parallel job. The user selects the accelerator resources that will be utilized by specifying in the input script an ID for the first and last device to be used on each node. The IDs must be the same for every node. At initialization, the MPI_COMM_WORLD communicator is split into per-node communicators according to the host-names for each node. The processes on each node are then assigned to one accelerator device. If the number of processes per node is greater than the number of devices on the node, multiple processes are assigned to the same device. The number of processes per device should be constant for efficient utilization because the subdomain size does not currently vary between different processes in LAMMPS. In order to perform device timings necessary for dynamic load balancing, the per-node communicators are further split into per-device communicators.

When neighbor list calculation is performed on the accelerator, the dynamic load balancing of force calculation is performed as follows (where *device_comm* is the per-device communicator, p_d is the current fraction of particles to be calculated on the device, and p_{new} is the most recent calculation of the optimal fraction from previous host and device timings):

1. $n_d = p_d * nlocal$
2. If a rebuild is required, build a full neighbor list for particles $i < n_d$ and a half neighbor list for particles $i \geq n_d$ on the device, copy the half neighbor list to the host, and set $p_d = p_{new}$
3. Cast/pack atom data
4. if (load_balance_this_step) {
5. Block for device completion
6. MPI_Barrier(device_comm)
7. Start device timer
8. Block for device completion
9. MPI_Barrier(device_comm)
10. Start CPU timer
11. }
12. Enqueue asynchronous transfer of atom data to device
13. Enqueue asynchronous force calculation on device
14. Enqueue asynchronous transfer of force/energy/virial data to host
15. Begin force calculation on host
16. if (load_balance_this_step) {
17. Stop device timer
18. Stop CPU timer
19. Block for device completion
20. $cpu_time / = nlocal - n_d$
21. $device_time / = n_d$
22. MPI_Allreduce for maximum cpu (c_{max}) and device (d_{max}) times over device_comm
23. $p_{new} = 0.5 * c_{max}/(c_{max} + d_{max}) + 0.5 * p_{new}$
24. }
25. Block for device completion
26. Cast/pack forces/energies/virials into LAMMPS data structures

Because it is desirable to implement a portable method for timing device data transfers and kernels from multiple processes, barriers are used to ensure that all timers are started before any data transfers and/or force calculations have begun. Then, the maximum time recorded on the device represents the total time required for execution of all data transfers and kernels from all processes using the device. Ideally, this should be equal to the time required for force calculation on the CPU for each process. Currently, the timings for load balancing are performed for the first 10 timesteps and then every 25 timesteps. We set $p_d = p_{new} = 0.9$ at the beginning of a simulation run. Because full neighbor lists are used on the device and half neighbor lists are used on the host, p_d is only changed when a neighbor rebuild occurs.

When neighbor list calculations are performed on the host, a slightly different procedure is used. In this case, a full neighbor list is used for both host and device calculations. Additionally, p_d can be decreased on any timestep, not just when neighbor rebuilds occur.

2.7. *Geryon Library*

Currently, there are 3 prevalent low-level APIs for programming accelerators - CUDA-Driver, CUDA-Runtime, and OpenCL. CUDA has been the most popular choice for programming GPUs due to its maturity and optimized performance for NVIDIA hardware. For CUDA programming, CUDA-Runtime is the most commonly used API because it allows for more succinct code at a slightly higher level than CUDA-Driver. There are some advantages to the CUDA-Driver API, however. For HPC, one notable advantage is that there is more freedom in the selection of the compiler for host code. Only kernels that are run on the device need to be compiled with the NVIDIA compiler and all host code can be compiled with other compilers optimized for the machine. Additionally, one can perform more advanced context management with the CUDA-Driver API - an important consideration when multiple processes or threads are utilizing GPUs. As of CUDA version 3.1, both APIs can be used in a single code. For portability, OpenCL is an attractive alternative with an API that is very similar to the CUDA-Driver API. OpenCL has been adopted as an industry standard and allows OpenCL kernels to run on CPUs. For us, the main concern with adopting the OpenCL API as the sole programming model is the relative immaturity of the compilers and the potential for lagging efficiency on current NVIDIA hardware.

The OpenCL and CUDA-Driver APIs are more tedious and less succinct than the CUDA-Runtime API. The solution is to write a library that provides a more succinct interface by abstracting away low-level code. Because we want both portability and fast code for NVIDIA hardware, our solution was to write a library that provides a succinct interface, but also allows the same code to compile with CUDA-Runtime, CUDA-Driver, or OpenCL. The software, called Geryon, is intended to be a simple library for managing all three APIs with a consistent interface. This is performed with classes for 1) device management, 2) data storage, 3) command queue management, 4) kernel management, and 5) device timing. Commands for data copying and host-device transfer, type-casting, and I/O are provided. The library is written such that the same set of commands can be used with any of the APIs - to switch from one compiler to another, only the namespace must be changed. Templates are used such that there is little or no overhead for using the library and the memory management and I/O routines are greatly simplified.

Geryon also handles the case where the device memory is addressable by the host in an efficient manner. Currently this occurs when the device is the CPU in OpenCL, but future accelerators might also have this advantage. In this case, host-device transfers are an unnecessary expense. This functionality is provided with an option for a data object to "view" existing memory rather than allocate new memory. In this case, host-device data transfers will be ignored.

The Geryon library allows acceleration in LAMMPS with both CUDA and OpenCL. It is important to note that many common routines such as data sorts, BLAS, LAPACK, etc. are provided in API-specific libraries and it would be undesirable to rewrite these routines in Geryon. Indeed, neighbor list builds on the device are not currently supported in LAMMPS for OpenCL due to the use of the CUDPP library. The sort and scan routines have been released for OpenCL, however, and we are working on a version that is fully functional with both CUDA and OpenCL. Although API-specific libraries complicate the use of Geryon for writing portable codes, the library is useful for our purposes because it allows use of the CUDA-Driver and OpenCL APIs with a simpler and more succinct interface that is intended to make the transition between current and future accelerator APIs much simpler and more efficient. For example, the data types have changed for some routines in newer versions of the CUDA API; in these cases we have only had to modify a few underlying routines in the library to allow support for new CUDA versions in the codes that use Geryon. While we expect similar API-specific libraries to be available for both CUDA and OpenCL, hopefully future efforts in programming hybrid machines will converge on a single programming model that is efficient and portable.

The Geryon library is available under the Free-BSD license from http://users.nccs.gov/˜wb8/geryon/index.htm.

2.8. *Lennard-Jones Potential*

The Lennard-Jones potential [17] is widely used for modeling van der Waals forces in MD simulations,

$$U = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right], \tag{1}$$

where r is the interparticle separation, σ parameterizes the optimal interparticle separation, and ϵ is used to parameterize the well depth for the interaction energy. We have chosen the LJ potential for benchmarking in this work because it is a very common potential with very low arithmetic intensity.

For our implementation, the CUDA compilation of the LJ kernel used 29 registers and 2128 bytes of local memory for single precision.

2.9. *Gay-Berne Potential*

The Gay-Berne potential is a single-site interaction potential for rigid molecules derived from heuristic modifications to a Gaussian overlap potential [7]. The potential, which can be considered as an anisotropic and shifted Lennard-Jones (LJ) 6-12 interaction, has been extensively used for the modeling of mesogenic systems. Although it was originally presented as a soft potential for ellipsoidal particles of equivalent size, it has since been generalized for dissimilar biaxial ellipsoids [2]. The potential can be written as a product of 3 terms,

$$U = U_r \cdot \eta \cdot \chi, \tag{2}$$

parameterized by the ellipsoid shapes and relative interaction energies. For shape, the ellipsoid semiaxes a_i, b_i, and c_i for each particle i are specified to form the diagonal elements of a 'shape' matrix, $\mathbf{S}_i = \mathrm{diag}(a_i, b_i, c_i)$. Likewise, the relative well depths ϵ_{ai}, ϵ_{bi}, and ϵ_{ci} for particles interacting along the corresponding semiaxes (side-to-side, face-to-face, and end-to-end interactions) give the matrix $\mathbf{E}_i = \mathrm{diag}(\epsilon_{ai}, \epsilon_{bi}, \epsilon_{ci})$. The

orientation of each particle is given here by the rotation matrix $\mathbf{A}_i$ representing the transformation from the lab frame to the body frame.

In Eq. 2, U_r represents the shifted LJ interaction given by the interparticle distance h, the atomic interaction radius σ, and the shift factor γ,

$$U_r = 4\epsilon(\varrho^{12} - \varrho^6), \tag{3}$$

$$\varrho = \frac{\sigma}{h + \gamma\sigma}. \tag{4}$$

Because the particles are aspherical, the interparticle distance h is not between particle centers but rather represents the distance of closest approach between particles. While an exact calculation of h is non-trivial[34], an approximation has been given by Perram *et al.*[21] that is commonly used in Gay-Berne calculations,

$$h = r - [\frac{1}{2}\hat{\mathbf{r}}^T\mathbf{G}^{-1}\hat{\mathbf{r}}]^{-1/2}, \tag{5}$$

where $\mathbf{r} = \mathbf{r_2} - \mathbf{r_1}$ is the particle center separation, $r = |\mathbf{r}|$ is the center-to-center distance, $\hat{\mathbf{r}} = \mathbf{r}/r$, and

$$\mathbf{G} = \mathbf{A}_1^T\mathbf{S}_1^2\mathbf{A}_1 + \mathbf{A}_2^T\mathbf{S}_2^2\mathbf{A}_2. \tag{6}$$

In addition to the distance of closest approach, the interaction anisotropy is characterized by the distance-independent terms η and χ that control interaction strength based on the particle shapes and relative well depths respectively,

$$\eta = \left[\frac{2s_1s_2}{\det(\mathbf{G})}\right]^{\upsilon/2}, \tag{7}$$

$$s_i = [a_ib_i + c_ic_i][a_ib_i]^{1/2}, \tag{8}$$

and

$$\chi = [2\hat{\mathbf{r}}^T\mathbf{B}^{-1}\hat{\mathbf{r}}]^{\mu}, \tag{9}$$

$$\mathbf{B} = \mathbf{A}_1^T\mathbf{E}_1^2\mathbf{A}_1 + \mathbf{A}_2^T\mathbf{E}_2^2\mathbf{A}_2. \tag{10}$$

The parameters μ and υ in equations 7 and 9 are empirically determined exponents that can be tuned to adjust the potential.

The analytic expressions for the forces and torques as well as details of the parallel implementation of the Gay-Berne potential for biaxial ellipsoidal particles in the LAMMPS MD code have been described previously [3].

The Gay-Berne potential was chosen for benchmarking due to the very high arithmetic intensity. For typical problems, it is approximately 15 times more expensive than the LJ calculation per particle pair. It does require additional memory access when compared to LJ, however. In addition to particle positions, quaternions representing the orientation of each particle are passed into the force kernels. In addition to particle forces, torques must be copied back to the host. Because the force calculation for each pair is computationally intensive, the cutoffs are evaluated in a separate kernel from the forces (as described above) in order to eliminate work-item divergence resulting from the cutoff check. For our implementation, the CUDA compilation resulted in a kernel that uses 119 registers and 104 bytes of shared memory for single precision.

2.10. Yona Test Platform

Benchmarks were performed on a test cluster with 15 nodes and a Mellanox MT26428 QDR InfiniBand interconnect. Each node had two six-core AMD Opteron 2435 processors running at 2.6GHz and two Tesla C2050 GPUs each with 3GB GDDR5 memory and 448 cores running at 1.15GHz with a memory bandwidth of 144 GB/s. GPUs were connected on PCIx16 gen 2.0 slots. Tests were run with ECC support enabled. The bandwidth reported by the CUDA 3.1 SDK for 32MB host-to-device data transfers was 2.4GB/s for pageable memory and 3.9GB/s for page-locked memory. For 32 MB device-to-host data transfers, the reported bandwidth was 1.4GB/s for pageable memory and 4.0GB/s for page-locked memory. For the CUDA molecular dynamics tests, device code was compiled with the CUDA toolkit 3.1. Host code was compiled using OpenMPI 1.7 with the Intel 11.1 C++ compilers. Host code was compiled with O2 optimization for an SSE2 target. Device driver version was 256.35. For the OpenCL tests, code was compiled using the GNU 4.3.2 compilers with OpenMPI 1.7.

2.11. Test cases

For the Lennard-Jones simulations, the LAMMPS LJ benchmark was used as available in the source distribution. Parameters are described in dimensionless units. Initial configurations consisted of 256000 or 864000 atoms. Benchmark simulations were performed using the microcanonical (NVE) ensemble with a cutoff of 2.5σ for 5000 timesteps for liquid simulations with a reduced density of 0.8442. In the CPU-only simulations, forces for ghost atoms are communicated in order to save computational time (this is the default setting in LAMMPS). For accelerated simulations, if two interacting particles are on different processors, both processors compute their interaction and the resulting force information is not communicated. This allows the use of full neighbor lists without special treatment for ghost atoms.

For the Gay-Berne simulations, we have used the same model parameters as our previous work [3]. These parameters are described in dimensionless units in terms of the characteristic length σ_0, energy ϵ_0, and mass m_0. The mesogen is modeled as a uniaxial prolate ellipsoid with a mass $1.5m_0$, an aspect ratio of 3, and Gay-Berne parameters $\epsilon^{meso} = \epsilon_0$, $\sigma^{meso} = \sigma_0$, $a_i^{meso} = b_i^{meso} = \sigma_0$, $c_i^{meso} = 3\sigma_0$, $\epsilon_a^{meso} = \epsilon_b^{meso} = \epsilon_0$, and $\epsilon_c^{meso} = 0.2\epsilon_0$. The Gay-Berne model parameters have been set as $\gamma = 1$, $\mu = 1$, and $\upsilon = 3$. A cutoff radius for the potential of $r_c = 7\sigma_0$ and a neighbor list radius of $7.8\sigma_0$ was used.

The starting configuration for the Gay-Berne benchmark was generated using equilibrium molecular dynamics simulations carried out in an isothermal-isobaric (NPT) ensemble with a time step of 0.002τ ($\tau = \sigma_0[m_0/\epsilon_0]^{\frac{1}{2}}$). Starting with a dilute lattice of 125000 particles, the pressure was increased from $P = 0\epsilon_0/\sigma_0^3$ to $P = 8.0\epsilon_0/\sigma_0^3$ over 5000 time steps. The damping parameters for the thermostat and barostat were both set to 0.5τ. The temperature $T^* = k_BT/\epsilon_0$ of the simulations was 2.4. This was followed by equilibration for an additional 5000 timesteps at a pressure of $P = 8.0\epsilon_0/\sigma_0^3$ to generate the starting configuration for the benchmark.

The benchmark simulations for the ellipsoidal particles were carried out using the microcanonical (NVE) ensemble with a timestep of 0.002τ and a cutoff of $r_c = 7\sigma_0$ for 1000 timesteps. In the CPU-only simulations, forces for ghost atoms are communicated. For accelerated simulations, if two interacting particles are on different processors, both processors compute their interaction and the resulting force information is not communicated.

3. Results

3.1. Single Node Results

The timings for the LJ and Gay-Berne benchmarks on a single node are shown in Figure 2 for single, mixed, and double precision. The single precision LJ case with a 2.5σ cutoff is intended to be a worst case for LAMMPS acceleration due to the low arithmetic intensity. As shown in Figure 2, the speedup for the LJ potential over a dual hex-core Opteron is only 0.73; it is slower than the CPU-only calculation. In this simulation, the CPU work from neighbor list builds and time integration dominate the calculation time. Additionally, these tasks are performed on only 2 cores instead of the 12 used for the CPU-only benchmark. In this case, the CPU calculations represented 83.1% of the loop time, the wall time required to complete the entire simulation loop. The atom copy represented 6.3%, the neighbor copy was 3.5%, the unpack kernel was 1%, the force kernel was 4.4%, and the force copy was 1.7% of the loop time.

Performing neighbor list builds on the GPU improves this speedup to 1.7. In this case, the CPU calculations were reduced to 62.9% of the loop time. The atom copy is 14.6%, the neighbor kernel is 8.2%, the force kernel is 10.4%, and the force copy is 3.9% of the total loop time. For mixed precision, the results were very similar to the single precision case. The speedup with neighboring on the CPU was 0.71 and the speedup with neighboring on the GPU was 1.69. For double precision the performance was substantially different; with CPU neighboring the speedup was 0.61 and with GPU neighboring, the speedup was 1.2. This is primarily due to the increase in the force kernel time which is memory-bound.

The Gay-Berne potential is at the opposite end of the spectrum. For the CUDA compilation, 119 registers per work-item are required for force and torque calculation. For the Tesla C1060, this is not a significant issue since 127 registers can be used per work-item. For the C2050, this decreases to 63 registers per work-item. For single precision, we still obtain impressive results, however. With CPU neighboring, the speedup is 6.3 versus a dual hex-core Opteron with 47% of the computation time performed on the CPU. With GPU neighboring, the speedup is 10.4 with 13.1% of the calculation performed on the CPU. The GPU loop time breakdown for the CPU neighboring case was 2.1% atom copy, 2.2% neighbor copy, 12.9% neighbor kernels, 35.5% force/torque kernels, and 3.9% force/torque copy. For GPU neighboring, the breakdown was 3.6% atom copy, 23.8% neighbor kernels, 59% force/torque kernels, and 0.5% force/torque copy. Again, for mixed precision, the results were similar to single precision - a speedup of 5.9 with CPU

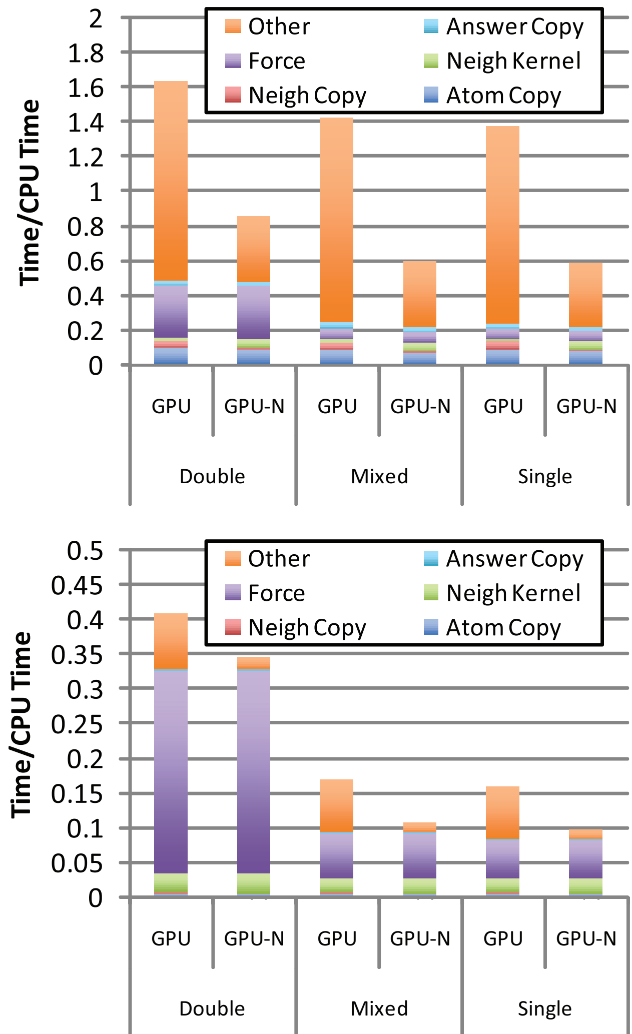

Figure 2: Performance of accelerated simulations on a single node (2 GPUs). Top: Results for the Lennard-Jones potential. Bottom: Results for the Gay-Berne potential. GPU-N is a test case with neighboring performed on the GPU. GPU is a test case with neighboring performed on the CPU. Tests are performed for single, mixed, and double precision. Time in orange is computed on the CPU for time integration, etc. Other colors are for data transfer and kernel execution on the GPU. Time is normalized by the time required to complete the simulation loop on 12 CPU cores. Simulations contained 256000 particles for LJ and 125000 particles for Gay-Berne.

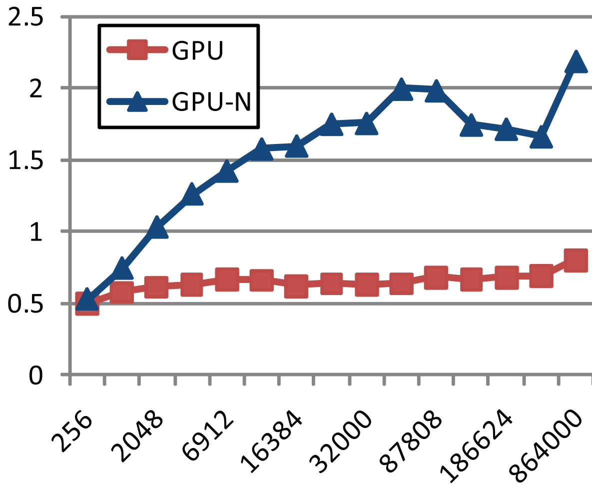

Figure 3: Speedup of accelerated code on a single node vs 12 CPU cores as a function of number of particles. GPU-N is performed with neighboring on the GPU. Tests were performed for Lennard-Jones with a cutoff of 2.5 using single precision.

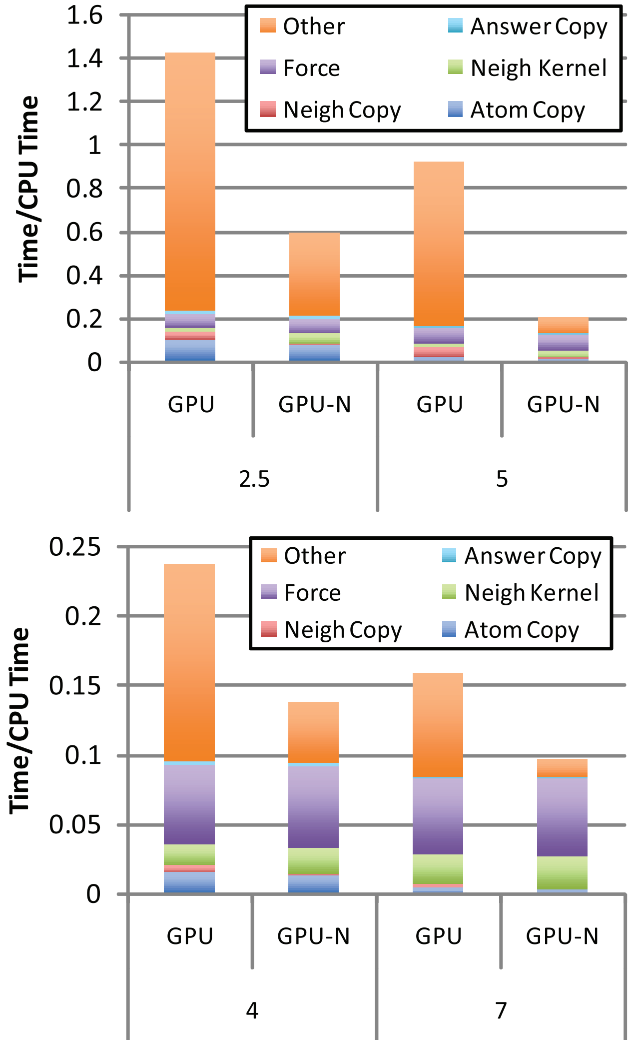

Figure 4: Performance of accelerated simulations on a single node for different cutoffs (2 GPUs). GPU-N is a test case with neighboring performed on the GPU. GPU is a test case with neighboring performed on the CPU. Top: Performance for Lennard-Jones with a cutoff of 2.5 and 5. Bottom: Performance for Gay-Berne with a cutoff of 4 and 7. Time is normalized by the time required to complete the simulation loop on 12 CPU cores. Simulations contained 256000 particles for LJ and 125000 particles for Gay-Berne.

neighboring and a speedup of 9.4 with GPU neighboring. For double precision, the speedups were reduced to 2.4 and 2.9 respectively. Because the number of available registers is reduced for double precision, memory latencies for variables private to each work-item are increased and the performance is impacted.

The relative performance of the accelerated code will depend on the number of particles per node and the cutoff. The impact of problem size on the speedup is shown in Figure 3 for the LJ benchmark. When neighbor list builds are performed on the CPU, the speedup is relatively insensitive to the number of particles. For GPU neighbor builds, this is not the case and there is more variance in the relative performance. At around 5488 particles per GPU, a speedup of greater than 1.5 is achieved. The performance for different cutoffs is shown in Figure 4 for the LJ and Gay-Berne benchmarks. Increasing the cutoff from 2.5σ to 5σ increases the speedup from 0.7 to 1.1 for CPU neighboring and from 1.7 to 5.1 for GPU neighboring. This results from the change from 37 neighbors per particle on average for the 2.5σ cutoff to 264 neighbors per particle in the 5σ case. For Gay-Berne, decreasing the cutoff from $7\sigma_0$ to $4\sigma_0$ decreases the speedup from 6.3 to 4.2 for the CPU neighboring case and from 10.4 to 7.3 in the GPU neighboring case.

Use of the Geryon library allows the same code to compile with both CUDA and OpenCL. Although OpenCL performance is not of immediate concern in our efforts, we have compared the loop times for code compiled with CUDA and NVIDIA's OpenCL implementation. These results are shown in Figure 5. In the LJ case, the OpenCL code takes 1.9 times longer than the CUDA code to complete the LJ benchmark. For the Gay-Berne case, the OpenCL code takes 1.5 times longer to complete when compared with the CUDA code. Profiling shows that the slowdown occurs due to a greatly increased instruction count for the OpenCL kernels. We have not investigated the cause of this, but because the same kernels are being compiled for the same hard-

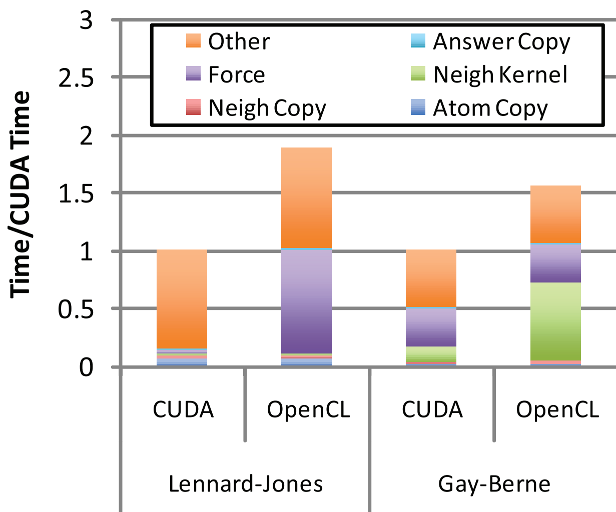

Figure 5: Performance of accelerated simulations on a single node for code compiled with CUDA and OpenCL. Tests were performed with 256000 particles with neighboring performed on the GPU. Time is normalized by the time required to complete the simulation loop with CUDA.

ware with the same instruction set, we expect this difference to improve as the OpenCL compiler matures.

3.2. Multi-Node Results

Results for strong scaling on the test cluster for the LJ and Gay-Berne test cases are shown in Figure 6. For the LJ case, timings were made for 864000 particles with 1 process per device (2 processes per node (ppn)) used for the accelerated benchmarks. For single precision, the speedups of the cluster with GPUs versus the cluster without GPUs ranged from 0.68 to 0.76 with the same number of particles per node. When neighbor builds are performed on the GPU, this speedup range is increased to between 1.9 and 2.1. For double precision, the ranges were 0.54 - 0.63 for CPU neighboring and 1.09 - 1.30 for GPU neighboring. For Gay-Berne, 125000 particles were used for strong scaling tests due to the better parallel scaling efficiency for the computationally intensive force calculation. The single precision ranges were between 5.6 and 6.3 for the CPU neighboring case and between 9.4 and 10.5 for the GPU neighboring case. For double precision, the speedup ranges were 2.2 - 2.4 for CPU neighboring and 2.7 - 2.9 for GPU neighboring due to the high number of private variables per work-item.

A break down of the simulation loop times per routine is given in Figures 6 and 7. The "Other" time in these plots represents the time spent on the CPU by tasks such as time integration. The neighbor calculation dominates the loop time for the LJ calculation and is a significant fraction of the Gay-Berne calculation when neighbor list builds are performed on the CPU. When the neighbor list builds are performed on the GPU, the integration time becomes more significant for the LJ benchmark, but not the Gay-Berne. This is due to the smaller problem size for the Gay-Berne benchmark and the computationally intensive force calculation.

3.3. Load Balancing

With part of the code running on the CPU and part on the GPU, a significant fraction of the hybrid resources are wasted when the code is run with one MPI process per device. Using the host/device load balancing approach described above with 12 ppn, we can improve the results with better utilization of the machine (Figure 6). In these cases, the speedups for the single precision LJ case are improved to 2.2 - 2.5 versus the machine without acceleration. With neighbor builds performed on the GPU, the speedups ranged from 2.9 and 3.7. For a cutoff of 5σ (data not shown), the speedups ranged from 5.9 to 7.8. As shown in Figure 6, the fraction of particles handled by the GPU decreases as the number of particles per process decreases. This might be counterintuitive, however, as shown in Figure 3, the relative performance of the GPU decreases with problem size and therefore the CPU calculation rates become more competitive.

The improvements from this approach will be sensitive to the problem size and relative performance of the force kernel. Once the number of particles per process decreases below some threshold, there will not be enough work to efficiently utilize the GPU with each kernel call. If the GPU performance for the force evaluation has a high speedup versus the CPU code, there might be little to gain from CPU evaluation of forces. Both of these issues arise in the Gay-Berne benchmark. In this case, splitting the neighbor and time integration calculations does not impact the performance as significantly because these calculations represent a smaller fraction of the loop time. Additionally, as the number of particles per process decreases, there is less work for the GPU and as shown in Figure 3, relative GPU performance will be worse. For these reasons, running on 12ppn results in decreased performance for the Gay-Berne benchmark as the number of particles per process decreases below 2000. This also occurs in the LJ benchmark at a similar number of particles per node (data not shown). At some point, it becomes more efficient to run the simulation with a smaller number of processes per device.

The performance impact resulting from splitting the force calculation between the host and device will depend on the CPU core to device ratio and the relative rates of force calculation on the host and device. As shown in Figures 6 and 7, the calculated fraction of particles on the host is less than 12 percent for both the LJ and Gay-Berne single precision test cases. For these cases, no improvement is seen with dynamic load balancing of force calculation. For the double precision cases, the impact is more significant, however. For the LJ cases, the loop times were between 5.8 and 22.8 percent slower without force load balancing. For the Gay-Berne cases, the loop times were between 5 and 12.6 percent slower without force load balancing. The best relative speedups for the test runs are summarized in Table 2.

As shown in Figures 6 and 7, the benefits from porting additional routines characterized by the "Other" time will vary depending on problem size. For the LJ benchmark, this varied from 31% of the loop time for 1 node to less than 5% for 15 nodes. For the Gay-Berne, the "Other" time was less than 4%

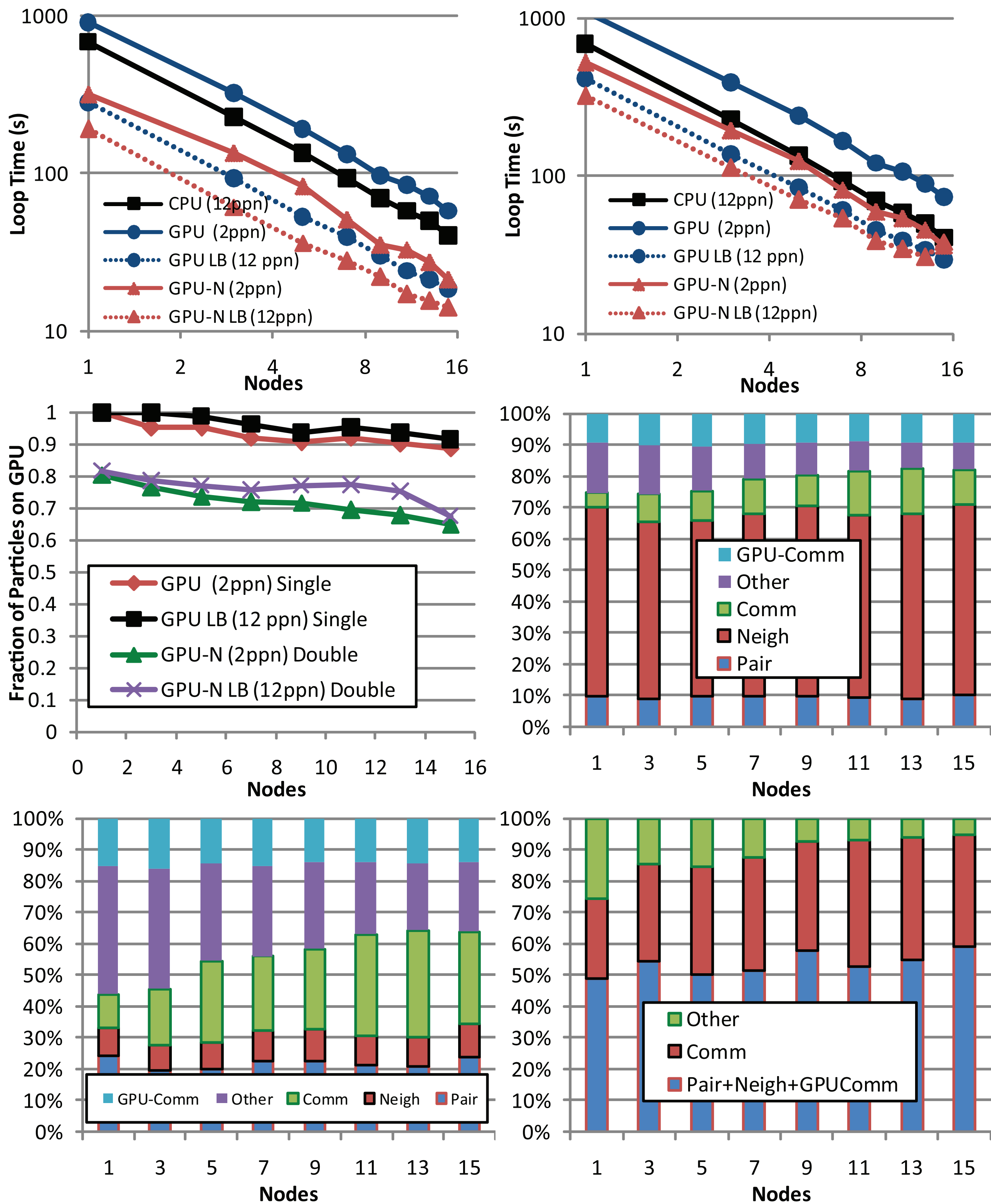

Figure 6: Strong-scaling for the Lennard-Jones test case with and without acceleration. Top Left: Comparison of loop time without acceleration (CPU), acceleration with 1 process per GPU (2ppn), and load balancing (LB) with 6 processes per GPU (12 ppn) for single precision. Neighboring is performed on the GPU for the GPU-N cases. Top Right: Results for double precision. Middle Left: Fraction of particles handled by the GPU for the LB test cases. Middle Right: Loop time breakdown for the single precision GPU 2ppn case. Bottom left: Loop time breakdown for the single precision GPU-N 2ppn case. Bottom Right: Loop time breakdown for the single precision GPU-N LB case. Loop times are the wall time required to complete the entire simulation loop.

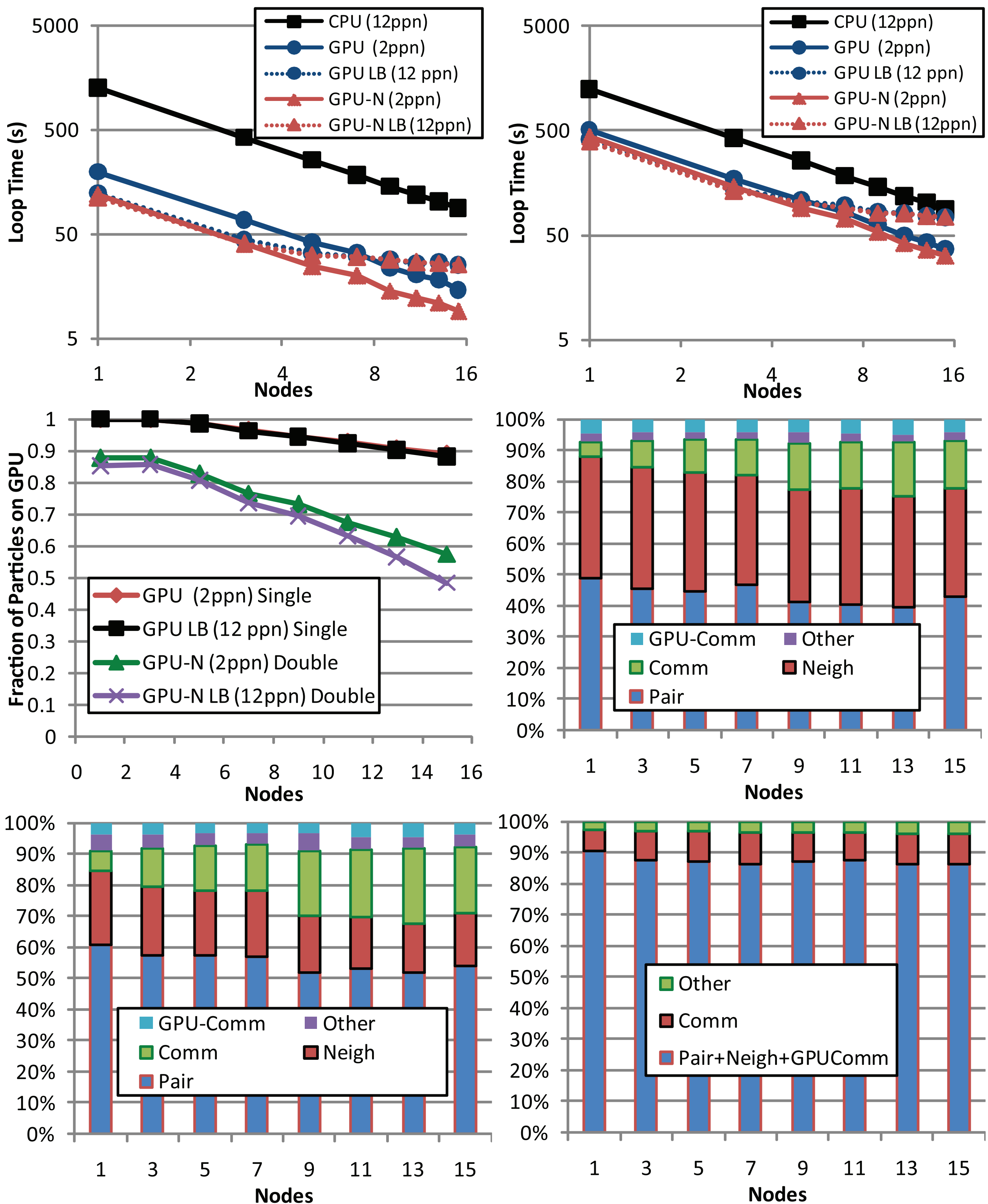

Figure 7: Strong-scaling for the Gay-Berne test case with and without acceleration. Top Left: Comparison of loop time without acceleration (CPU), acceleration with 1 process per GPU (2ppn), and load balancing (LB) with 6 processes per GPU (12 ppn) for single precision. Neighboring is performed on the GPU for the GPU-N cases. Top Right: Results for double precision. Middle Left: Fraction of particles handled by the GPU for the LB test cases. Middle Right: Loop time breakdown for the single precision GPU 2ppn case. Bottom left: Loop time breakdown for the single precision GPU-N 2ppn case. Bottom Right: Loop time breakdown for the single precision GPU-N LB case. Loop times are the wall time required to complete the entire simulation loop.

Table 2: Summary of best speedups versus a single CPU core for CPU-only and accelerated runs. GPU-N cases used the GPU for neighbor list calculation. The speedups were calculated from single core loop times of 6605 seconds for the LJ CPU case and 16018 seconds for the GB CPU case. Note that superlinear speedups are also seen in the CPU-only tests for the GB case.

	1 node		15 nodes	
Test case	Cores	Speedup	Cores	Speedup
LJ CPU	12	9.6	180	162.5
LJ GPU Single	12	23.4	180	356.4
LJ GPU-N Single	12	34.4	180	467.1
LJ GPU Double	12	16.0	180	224.1
LJ GPU-N Double	12	20.4	180	172.6
GB CPU	12	12.8	180	182.5
GB GPU Single	12	146	30	1747.4
GB GPU-N Single	12	144.1	30	1541.7
GB GPU Double	12	37.2	30	511.4
GB GPU-N Double	12	40.9	30	503.7

of the total calculation. Because the host/device communication time is over 10% of the loop time for LJ calculations, additional savings from reducing the amount of host/device communication at each step can potentially be gained by porting additional routines to the accelerator.

4. Discussion

We have described a framework for implementing molecular dynamics for hybrid high-performance computers in LAMMPS. For some hybrid machines, we can expect that significant computational resources will be available in the form of multi-core CPUs in addition to accelerators. In order to make efficient use of hybrid resources, we have described a method for utilizing all CPU and accelerator cores on each node. Because our approach currently uses only accelerator devices for neighbor list builds and force calculation, additional performance is gained by splitting the other calculations performed on the CPU between multiple cores. In LAMMPS this can be done straightforwardly by assigning one MPI process per core at run time, with each process able to share accelerators on the node. For large particle counts, the approach has the potential to decrease memory latencies for accelerated kernels by further dividing the work spatially to improve data locality. As the number of particles per process becomes smaller, a point will be reached where it is more efficient to run on fewer cores in order to efficiently utilize the accelerator. For the test cases presented here, six cores per device became less efficient at around 2000 particles per process. Additional performance can be gained with dynamic load balancing of force calculation between CPU cores and accelerators on each node. This will depend on the relative rates of force calculation on the CPU and accelerator and the ratio of CPU cores to accelerator devices. For the test cluster used here, up to a 20 percent reduction in loop time was achieved with dynamic load balancing of forces; however, there was little change for single precision calculations.

Due to the sensitivities of accelerator speedups to particle counts, cutoff, density, and host and device specifications, it is difficult to provide a comparison between different approaches or to give a simple estimate of the speedup for a given simulation. For this work, we have chosen to evaluate performance on an accelerated cluster with comparison to the same cluster without acceleration. For the LJ case with a low cutoff of 2.5σ, running the simulations with accelerators was between 2.9 and 3.7 times faster for between 12 and 180 CPU cores (2-30 accelerators). For a cutoff of 5σ, more similar to cutoffs used in protein simulations, the speedups ranged from 5.9 to 7.8. For the Gay-Berne case with a high cutoff, the speedups ranged from 9.4 to 11.2.

These results are all for single precision calculation on the GPU. The results for mixed precision will be very similar. The Fermi GPU offers improved performance for double precision with 515 Gflops on the Tesla C2050. For our test cases, double precision performance was still significantly worse than single or mixed precision. For the LJ case, the memory-bound kernel requires twice as many bytes for atom positions in double precision. For the Gay-Berne case, the lack of available registers per work-item limited performance. The use of single and mixed precision for MD simulations has been analyzed by many including evaluation of error in force and energy calculations, energy conservation, trajectory divergence, temperature changes due to numerical error, and comparison of ensemble-averaged quantities [1, 14, 6, 13, 10, 26, 24]. For current accelerators, single and mixed precision might be the best choice for many models.

The performance benefit from porting additional routines for acceleration will depend on the hardware configuration and simulation. For the 180-core accelerated simulations performed here, less than 5 percent of the loop time was spent on time integration and statistics for the LJ case. For the Gay-Berne, the time was less than 1 percent. This will not be the case for all jobs, however, and porting additional calculations for acceleration can decrease the times required for host/device data transfer because data for all particles does not need to be transferred for inter-process communications at every timestep. For future hybrid machines, memory might be available that is efficiently addressable by the host and the device. Currently, however, our approach is to overlap host/device communications with force calculations on the CPU. This has the advantage that the code porting and maintenance burden is not as substantial. For some common statistics and time integration approaches, however, it might prove beneficial to port additional routines in order to achieve efficient acceleration that is more general to a variety of host and device configurations.

Additional issues in hybrid high-performance computing have been discussed elsewhere [22, 16]. One important issue discussed by the authors concerns the use of direct memory access (DMA) and non-uniform memory access (NUMA) on current hybrid machines. Incorrect mapping of process/device pairs or thread/device pairs given the topology of the PCI express buses can have a significant performance cost. Addition-

ally, developers must address host-device data transfer times in addition to message-passing times between nodes when developing a scalable code. Therefore, hardware and/or software that allow memory allocations to be shared for DMA for both MPI and accelerator data transfers can improve parallel efficiency. Although future accelerators might allow the host to address device memory efficiently or support device-to-device communication directly, on current accelerators efforts aimed at hiding the host-device data transfer latencies might be necessary in scaling molecular dynamics codes for large hybrid machines.

Calculation of force contributions from long range electrostatics are necessary for many simulation models in MD. Implementations of particle-mesh Ewald (PME) and multilevel summation have already been described for GPUs [12, 11]. Another approach is to overlap the PME calculation performed on the CPU with short-range calculations on the GPU [22] and this type of approach has been shown to give impressive speedups in LAMMPS [10]. The host/device load balancing presented here could be used to further optimize the utilization of hybrid resources in long-range models. Because the time complexities and collective communications in many long-range models limit scaling for MD simulations, fast multipole [9] and multigrid-based approaches [27, 11] will likely result in the best scaling for large hybrid machines.

Current efforts at utilizing hybrid machines have focused on porting current physics models for acceleration. Many of these models are in use because they have provided excellent performance on computer hardware. As we begin to see significant changes in the hardware used for computational science, the development of new physics models with improved accuracy might prove more beneficial than porting existing force-fields. For example, the use of complicated pairwise potentials, 3-body models, and aspherical coarse graining can be much more competitive on current hybrid machines due to their high arithmetic intensity.

5. Acknowledgements

This research was conducted in part under the auspices of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC. This research used resources of the Leadership Computing Facility at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes. Sandia is a multipurpose laboratory operated by Sandia Corporation, a Lockheed-Martin Co., for the U.S. Department of Energy under Contract No. DE-AC04-94AL85000. Support for this work was provided by the CSRF program at Sandia National Laboratories.

References

[1] J.A. Anderson, C.D. Lorenz, A. Travesset, Journal of Computational Physics 227 (2008) 5342–5359.
[2] R. Berardi, C. Fava, C. Zannoni, Chem. Phys. Lett. 236 (1995) 462–468.
[3] W.M. Brown, M.K. Petersen, S.J. Plimpton, G.S. Grest, Journal of Chemical Phyics 130 (2009) 044901.
[4] J.E. Davis, A. Ozsoy, S. Patel, M. Taufer, in: S. Rajasekaran (Ed.), Bioinformatics and Computational Biology, Proceedings, volume 5462 of *Lecture Notes in Bioinformatics*, pp. 176–186. 1st International Conference on Bioinformatics and Computational Biology APR 08-10, 2009 New Orleans, LA.
[5] P. Eastman, V.S. Pande, Journal of Computational Chemistry 31 (2010) 1268–1272.
[6] M.S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A.L. Beberg, D.L. Ensign, C.M. Bruns, V.S. Pande, Journal of Computational Chemistry 30 (2009) 864–872.
[7] J.G. Gay, B.J. Berne, J. Chem. Phys. 74 (1981) 3316–3319.
[8] G. Giupponi, M.J. Harvey, G. De Fabritiis, Drug Discovery Today 13 (2008) 1052–1058.
[9] L.F. Greengard, J.F. Huang, Journal of Computational Physics 180 (2002) 642–658.
[10] S. Hampton, S.R. Alam, P.S. Crozier, P.K. Agarwal, In proceedings of The 2010 International Conference on High Performance Computing and Simulation (HPCS 2010) (2010).
[11] D.J. Hardy, J.E. Stone, K. Schulten, Parallel Computing 35 (2009) 164–177. Sp. Iss. SI.
[12] M.J. Harvey, G. De Fabritiis, Journal of Chemical Theory and Computation 5 (2009) 2371–2377.
[13] M.J. Harvey, G. Giupponi, G. De Fabritiis, Journal of Chemical Theory and Computation 5 (2009) 1632–1639.
[14] B. Hess, C. Kutzner, D. van der Spoel, E. Lindahl, Journal of Chemical Theory and Computation 4 (2008) 435–447.
[15] R.W. Hockney, S.P. Goel, J.W. Eastwood, Journal of Computational Physics 14 (1974) 148–158.
[16] V.V. Kindratenko, J.J. Enos, G.C. Shi, M.T. Showerman, G.W. Arnold, J.E. Stone, J.C. Phillips, W.M. Hwu, in: 2009 Ieee International Conference on Cluster Computing and Workshops, pp. 638–645. IEEE International Conference on Cluster Computing (Cluster 2009) AUG 31-SEP 04, 2009 New Orleans, LA.
[17] J.E. Lennard-Jones, Proceedings of the Physical Society 43 (1931) 461.
[18] W.G. Liu, B. Schmidt, G. Voss, W. Muller-Wittig, Computer Physics Communications 179 (2008) 634–641.
[19] J.A. van Meel, A. Arnold, D. Frenkel, S.F.P. Zwart, Molecular Simulation 34 (2008) 259–266.
[20] S. Meloni, M. Rosati, L. Colombo, Journal of Chemical Physics 126 (2007).
[21] J.W. Perram, J. Rasmussen, E. Praestgaard, J.L. Lebowitz, Phys. Rev. E 54 (1996) 6565–6572.
[22] J.C. Phillips, J.E. Stone, K. Schultent, in: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 444–452. International Conference for High Performance Computing, Networking, Storage and Analysis NOV 15-21, 2008 Austin, TX.
[23] S. Plimpton, Journal of Computational Physics 117 (1995) 1–19.
[24] C.I. Rodrigues, D.J. Hardy, J.E. Stone, K. Schulten, W.M. Hwu, CF'08: Proceedings of the 2008 conference on Computing Frontiers (2008) 273–282.
[25] N. Satish, M. Harris, M. Garland, in: 2009 Ieee International Symposium on Parallel and Distributed Processing, Vols 1-5, International Parallel and Distributed Processing Symposium (IPDPS), pp. 1–10.
[26] N. Schmid, M. Botschi, W.F. Van Gunsteren, Journal of Computational Chemistry 31 (2010) 1636–1643.
[27] R.D. Skeel, I. Tezcan, D.J. Hardy, Journal of Computational Chemistry 23 (2002) 673–684.
[28] J.E. Stone, D. Gohara, G.C. Shi, Computing in Science and Engineering 12 (2010) 66–72.
[29] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco, K. Schulten, Journal of Computational Chemistry 28 (2007) 2618–2640.
[30] P.J. in't Veld, S.J. Plimpton, G.S. Grest, Computer Physics Communications 179 (2008) 320–329.
[31] L. Verlet, Physical Review 159 (1967) 98.
[32] P. Wang, W.M. Brown, P.S. Crozier, Submitted (2010).

[33] J.K. Yang, Y.J. Wang, Y.F. Chen, Journal of Computational Physics 221 (2007) 799–804.
[34] X.Y. Zheng, P. Palffy-Muhoray, Phys. Rev. E 75 (2007).