Implementing Molecular Dynamics on Hybrid High Performance Computers - 
Particle-Particle Particle-Mesh

W. Michael Brown\textsuperscript{a,b,}, Axel Kohlmeyer\textsuperscript{c,}, Steven J. Plimpton\textsuperscript{d,}, Arnold N. Tharrington\textsuperscript{d}

\textsuperscript{a}National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA
\textsuperscript{b}Institute for Computational Molecular Science, Temple University, Philadelphia, PA, USA
\textsuperscript{c}Sandia National Laboratory, Albuquerque, New Mexico, USA
\textsuperscript{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 nodes containing more than one type of floating-point processor (e.g. CPU and GPU), are now becoming more prevalent due to these advantages. In this paper, we present a continuation of previous work implementing algorithms for using accelerators into the LAMMPS molecular dynamics software for distributed memory parallel hybrid machines. In our previous work, we focused on acceleration for short-range models with an approach intended to harness the processing power of both the accelerator and (multi-core) CPUs. To augment the existing implementations, we present an efficient implementation of long-range electrostatic force calculation for molecular dynamics. Specifically, we present an implementation of the particle-particle-particle-mesh method based on the work by Harvey and De Fabritiis. We present benchmark results on the Keeneland InfiniBand GPU cluster. We provide a performance comparison of the same kernels compiled with both CUDA and OpenCL. We discuss limitations to parallel efficiency and future directions for improving performance on hybrid or heterogeneous computers.

Keywords:
Molecular dynamics, electrostatics, particle mesh, 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. Use of accelerators such as GPUs are an important consideration for high performance computing (HPC) platforms due to potential benefits including lower cost, electrical power, space, cooling demands, and reduced operating system images [1]. To date already a number of the highest performing supercomputers listed in the Top500 list [2] utilize GPUs. In our previous work [3], we described an approach for accelerating molecular dynamics on hybrid high performance computers containing accelerators in addition to CPUs. The work was performed using the LAMMPS software package and the implementation was focused on acceleration for neighbor list builds and non-bonded short-range force calculation. The implementation allows for CPU/GPU concurrency and can be compiled using either CUDA or OpenCL. The approach is sufficient for many LAMMPS applications where electronic screening limits the range of interatomic forces. For simulations requiring consideration of long-range electrostatics, however, there is significant potential for performance improvement from the acceleration of long-range calculations. Although short-range force calculation typically dominates the computational workload on many parallel machines, GPU-acceleration of the short-range routines can result in the overall simulation time being dominated by long-range calculations.

The most common methods for calculation of long-range electrostatic forces in periodically closed systems are the standard Ewald summation [4] and related particle mesh methods that utilize fast Fourier transformations (FFTs) to speed up long-range calculations. Particle mesh Ewald (PME) [5], smooth particle mesh Ewald (SPME) [6], and particle-particle-particle-mesh (P\textsuperscript{3}M) [7] all use a grid representation of the charge density to allow FFT computation with a more favorable time complexity. Although there are many reports of GPU acceleration for short range forces in the literature, there are relatively few publications reporting speedups from acceleration of long-range calculations. Harvey and De Fabritiis [8] published an implementation of SPME for CUDA and an alternative approach, also for SPME, was recently presented [9]. Alternatives to the traditional Ewald and FFT-based particle mesh methods have also been published including GPU acceleration of the orientation-averaged Ewald sum [10] and GPU acceleration of multilevel summation [11].

Continued investigation into alternative algorithms for long-range force calculation on accelerators will likely be necessary
in order to best utilize the floating-point capabilities of accelerators in a distributed computing environment and this is an area of active research for our group. In this paper, however, we have chosen to focus on acceleration for the P$^3$M method because 1) an CPU implementation of P$^3$M has been available in LAMMPS for some time and this allows for a comparison between CPU and GPU calculation times, 2) the particle mesh methods are already accepted by most physicists as accurate and efficient methods for long-range force calculation, 3) an implementation of P$^3$M for accelerators gives a baseline for comparison of alternative accelerated algorithms, and 4) we believe that the porting of existing algorithms for use on accelerators is of general interest to the scientific computing community.

The algorithms we have used for P$^3$M acceleration follow those proposed by Harvey and De Fabritiis [8] for SPME. In this paper, however, we focus on parallel long-range force calculation on distributed systems with accelerators. We present several improvements that address some limitations to achieving good parallel efficiency. We compare results with the pre-existing CPU implementation in LAMMPS in order to assess the benefit from acceleration on GPUs. We describe an implementation that complies with both the CUDA and OpenCL APIs to allow for acceleration on a variety of platforms. We discuss approaches that minimize the amount of code that needs to be written for the accelerator with concurrent calculations performed on the CPU and the accelerator. We present benchmarks on an InfiniBand GPU cluster and discuss several important issues that limit strong scaling. Finally, we discuss future directions for improving performance on hybrid or heterogeneous clusters.

2. Methods

2.1. Ewald Summation and P$^3$M

Let $L = \text{diag}(l_x, l_y, l_z)$ be the diagonal matrix specifying the size of a periodic box with cell side lengths equal to $l_x$, $l_y$, and $l_z$. The total electrostatic energy resulting from pairwise interactions of $N$ Coulomb point charges ($q_i$) within the box is given by,

$$E = \frac{1}{2} \sum_{i,j=1}^{N} \sum_{\mathbf{m} \in \mathbb{Z}^3} \frac{q_i q_j}{|\mathbf{r}_{ij} + \mathbf{m} L|},$$

where $\mathbf{r}_{ij}$ is the distance between the point charges, $\mathbf{n}$ indexes all surrounding periodic cells for the simulation box and the $'$ is used to indicate that in the case $i = j$, the summation should not include the term where $\mathbf{n} = (0, 0, 0)$. Due to the slow decay of the Coulomb potential, a straightforward evaluation of equation 1 is impractical. Therefore, Ewald summation [4] is typically used to split the electrostatic energy into a summation over rapidly varying short-range interactions in real space ($E_r$) and a Poisson summation allowing the long-range interactions to be computed in reciprocal space ($E_k$). $E_r$ and $E_k$ are chosen such that $E_r$ is negligible beyond some cutoff distance, $E_k$ is a slowly varying function for all distances, and $E = E_r + E_k$. This allows for standard short-range cutoff evaluation for the real space and also for the Fourier transform to be represented with few $k$ vectors. The traditional Ewald summation is given by,

$$E = E_r + E_k + E_s,$$

$$E_r = \frac{1}{2} \sum_{i \neq j} \sum_{\mathbf{m} \in \mathbb{Z}^3} \frac{q_i q_j \text{erfc}(\alpha |\mathbf{r}_{ij} + \mathbf{m} L|)}{|\mathbf{r}_{ij} + \mathbf{m} L|},$$

$$E_k = \frac{1}{2} \sum_{i \neq j} \sum_{\mathbf{m} \in \mathbb{Z}^3} \frac{4\pi}{k^2} e^{-k^2/\alpha^2} |\tilde{\rho}(\mathbf{k})|^2.$$  

In order to correct the addition of erroneous self-interaction energies, $E_s$ is calculated as

$$E_s = -\frac{\alpha}{\sqrt{\pi}} \sum_i q_i^2.$$  

The Fourier transformed charge density is defined as

$$\tilde{\rho}(\mathbf{k}) = \int d^3 r \rho(r)e^{-i\mathbf{kr}} = \sum_{j=1}^{N} q_i e^{-i\mathbf{kr}}.$$  

The parameter $\alpha$ in the equations above tunes the relative weight of the real space and the reciprocal space contributions.

Although the Ewald approach facilitates an accurate method for calculation of electrostatic interaction energies in a periodic box, the best implementations have a time complexity of $O(N^{3/2})$. For this reason, several methods have been proposed for representing the charge density for reciprocal space calculations on a mesh. This allows for a discretized solution to Poisson’s equation using FFTs and an overall time complexity of $O(M \log M)$, where $M$ is the number of $k$-space mesh points. For typical accuracies of interest in MD calculations, a mesh spacing that yields $M \approx N$ is typical. The most popular variants of these particle mesh methods include particle mesh Ewald (PME) [5], smooth particle mesh Ewald (SPME) [6], and particle-particle particle-mesh (P$^3$M) [7].

In all three approaches, the reciprocal space calculation can be broken down into 3 steps. In the first step, charge assignment is performed to calculate charge densities on a mesh. For a given mesh point $\tilde{r}_p$, the mesh-based charge density due to a charge at position $\mathbf{r}$ is obtained from the charge density of the system $\rho$ by the following convolution,

$$\rho_M(\tilde{r}_p) = \frac{1}{h_x h_y h_z} \int d^3W(\tilde{r}_p - r)\rho(r)$$  

$$= \frac{1}{h_x h_y h_z} \sum_{i=1}^{N} q_i W(\tilde{r}_p - \mathbf{r}_i).$$

where $h$ is the mesh spacing and $W$ is the charge assignment function. $W$ is chosen such that it is a product of an assignment function that is independent in each dimension,

$$W(\mathbf{r}) = w(r_x)w(r_y)w(r_z).$$
with \( w \) chosen to have a finite and small support in order to decrease computational cost. For \( P^3M \), it is a spline of order \( P \).

In the second step, a field solve is performed to solve Poisson’s equation on the mesh in order to obtain the mesh-based electrostatic potential. This consists of (1) calculation of the finite Fourier transform of the mesh-based charge density, (2) multiplication with a precomputed influence function defined to give the Fourier space contribution to the electrostatic potential, and (3) calculation of an inverse finite Fourier transform. In the third step, interpolation is performed in order to obtain the force on each particle due to \( Ei \). For \( P^3M \), this is performed using the same function \( W \) used for charge assignment. The error from discretization, typically defined as the root mean square (RMS) error in the force, will depend on the values chosen for \( P \) and \( h \); the loss of accuracy from a decrease in the spline order \( P \) can be compensated for with a decrease in \( h \) (and vice versa).

At some point in the calculation, differentiation must be performed in order to obtain the particle forces due to the electrostatic potential. Several approaches have been applied for differentiation in \( P^3M \) [12]. Originally, the gradients were obtained in real space using finite-difference operators on the mesh. Because the splines for charge assignment are continuously differentiable to order \( P-1 \), another option is to obtain the gradients analytically with additional calculations during the charge assignment and interpolation steps. Calculation of the gradients in reciprocal space, referred to as ik-differentiation, is another option that results in more accurate results.

For brevity, we have left out many details of the Ewald method and \( P^3M \); a thorough presentation including a comparison of particle mesh methods is available in an excellent paper by Deserno and Holm [13].

### 2.2. LAMMPS

In this work, we are considering enhancements to the LAMMPS molecular dynamics package [14]. LAMMPS is parallelized via MPI, using spatial-decomposition techniques that partition the 3d simulation domain into a regular grid of smaller 3d subdomains, one per processor. LAMMPS is 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 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. \( ij/cut \) 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. \( ij/cut/gpu \) or \( ij/cut/hybrid \)) is as conceptually simple as writing a new class with the appropriate handful of required methods, some of which may be inherited from a related pair style (e.g. \( ij/cut \)). This design has allowed us to incorporate support for acceleration hardware into LAMMPS without significant modifications to the rest of the code. Specifically, accelerating long-range electrostatics involves adding new pair styles for the short-range real-space calculation and new k-space styles for the reciprocal space calculations.

Discussion of the parallel implementation of \( P^3M \) in LAMMPS has been published previously [15]; here we provide a brief summary of how the three steps of the previous section operate in parallel. Conceptually, a 3d grid is overlayed on the simulation domain. For the first and third steps (charge assignment and interpolation), each processor owns the subset of grid points that reside in its 3d sub-domain. For the second step (field solve), when FFTs are performed on the grid, each processor owns a subset of 1d columns of the grid in successive dimensions. For example, the projection of the grid into the \( xy \) plane is partitioned into 2d squares (one per processor), and each processor owns the entire \( z \)-dimension of grid points (columns) in its square. Thus MPI communication is required to move grid point data between processors at the various steps, as detailed below.

The \( P^3M \) parameters can be completely specified in LAMMPS, however, the default behavior is to use \( P = 5 \) with calculation of \( h \) and \( a \) such that the RMS error (see [16]) does not exceed a user-specified value.

In the first step, each processor maps the charge on particles it owns to its 3d sub-section of the grid. This affects grid points within a \( P \)-width stencil centered on the grid point nearest the particle. \( 3P \) fractional weights are calculated for the charge assignment function \( w \), and are then applied to the \( P^3 \) surrounding grid points to accumulate charge density on the mesh. Since this operation assigns charge to ghost grid points owned by other processors, communication of ghost values is required at the end of this step. Each processor exchanges grid point data with 6 neighboring processors, in the same manner that particle coordinates and forces are exchanged every timestep to enable computation of short-range interactions [14]. Unlike with particles, however, the data at a grid point is never needed by a diagonal neighbor process. Therefore, communication of ghost values at the corners of the grid subdomain is not necessary.

In the second step, performing 3d FFTs in parallel requires each processor own entire columns of the grid in successive dimensions, first \( x \), then \( y \), finally \( z \), so that 1d FFTs can be performed on-processor in each dimension using a fast numeric library such as FFTW. Thus the forward FFT requires 3 communication stages to re-partition the grid points across processors: (a) 3d decomposition to 2d columnar (with \( x \)-dimension columns), (b) 2d \( x \) to 2d \( y \), and (c) 2d \( y \) to 2d \( z \). The convolution calculation is performed in the final \( z \)-column decomposition. The inverse FFTs then perform the same 3 communication stages in reverse order to restore a 3d decomposition of the grid points. LAMMPS uses ik-differentiation and therefore the gradients are calculated as part of the field solve with 3 inverse 3d FFTs to obtain the force in each dimension. We note that each communication stage is effectively a one-to-one mapping of each grid point in the global grid from a source processor to a destination processor. Thus the entire grid is communicated at each stage and each processor exchanges grid point data with roughly \( \sqrt{N_p} \) other processors within the 2d decompositions,
where $N_p$ is the total number of processors. This is a large volume of communication with many messages, relative to the computational cost of the 1d FFTs, and is a commonly encountered bottleneck in the scalability of parallel 3d FFTs in general and particle-mesh methods in MD in particular.

In the third step, interpolation of electric field vectors to each of a processor’s particles requires values from the $P^3$ grid points surrounding each particle, using the same fractional weights that were used for charge assignment. As in the first step, this requires vector values for ghost grid points, which are acquired in a similar 6-way communication exchange with neighboring processors.

2.3. Benchmarks

For this work, we have used an all-atom protein benchmark supplied with LAMMPS to assess the performance of GPU acceleration. The benchmark is for the rhodopsin protein in a solvated lipid bilayer using the CHARMM force field, $P^3M$ long-range coulombics, and SHAKE constraints. The model contains counter-ions and a reduced amount of water to make a 32000 atom system. An inner cutoff of 8Å and an outer cutoff of 10Å are used for short-range force calculations. The simulations are performed using the isothermal-isobaric ensemble with a timestep of 2.0 femtoseconds. The benchmark was run for 400 timesteps on 1 to 32 GPUs or CPU cores or for 1000 timesteps on 3 to 96 GPUs (12 - 384 CPU cores). In order to show results for a larger simulation with similar properties, benchmarks were also performed using a 256000 atom initial configuration obtained from replicating the rhodopsin simulation box 8 times in order to double the length in each dimension. These simulations were also performed for 1000 timesteps.

2.4. Accelerating $P^3M$

A breakdown of the simulation loop times for the rhodopsin benchmark is given in Figure 1 for an approximate RMS precision of $7.57 \cdot 10^{-5}$. In this case, the spline order is 5 and the reciprocal space calculations are performed on a 25 x 25 x 32 mesh. In serial, charge assignment to the mesh (step 1) accounts for 47.1% of the long-range calculation, with 41% of the time spent on force interpolation (step 3) and the remaining 11.9% on the field solve (step 2). For parallel simulations, the percent time spent on the field solve increases, however, this is due to the MPI communication outlined above, and also the increase in the surface to volume ratio for each processor’s 3d sub-domain. Because acceleration can provide little, if any, benefit to MPI communication between processors and because the computational portion of the field solve (1d FFTs) is relatively small, we have focused our efforts on acceleration for the first and third steps: the charge spreading and force interpolation routines. Because LAMMPS already supports a choice of on-processor libraries (like FFTW) for 1d FFTs, adding the appropriate wrappers for GPU FFT computation, e.g. using the CUFFT library, would be a trivial extension. For most parallel simulations, however, it seems unlikely that this would lead to a significant increase in performance on current hardware.

![Figure 1: Percentage of loop time spent on short range non-bonded force calculation (Short Range), bond, angle, dihedral, and improper forces (Bond), neighbor list builds (Neigh), MPI communication excluding Poisson solves (Comm), $P^3M$ charge assignment (Charge Assign), $P^3M$ field solves (Field solve), $P^3M$ force interpolation (Force Interp), time integration, statistics, and other calculations (Other). Percentages are taken from a strong scaling benchmark (1 to 32 cores) with the 32000 atom rhodopsin simulation.](image-url)
alignment for the data. Consider a case where each processing element 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 alignment or access of non-contiguous memory needed by processing elements 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 compute unit. The number of work-items in a work-group can exceed the number of physical processing elements on the compute unit, 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.

2.6. Short-range Forces

Here, short-range force calculation refers to the computations performed by the set of routines responsible for van der Waals force computations cutoff at a user-specified distance and the real-space contribution to electrostatic forces. In order to avoid a $O(N^2)$ time complexity, neighbor list builds are required for short range force calculations. Our methods for using accelerators for short-range force calculation have been described previously [3]. Neighbor list builds are performed on the accelerator by first constructing a cell list that is utilized to build a verlet list. In order to assert that the short range force calculation is deterministic, we sort the data structure storing the cell list such that the order of neighbors for each particle will always be the same. Additionally, full neighbor lists are used such that the force for each pair of particles is calculated twice, doubling the number of force calculations to avoid memory collisions. A single neighbor list is built using the larger of the cutoff distances for van der Waals and electrostatic calculations extended by the user-specified skin.

The van der Waals and electrostatic forces are computed in a separate kernel. In our previous work, this was performed by assigning one particle to each work-item with a single loop over all neighbors to compute the particle force. This is the most common approach used for GPU-acceleration of short-range force calculation. The parallel efficiency can potentially be improved, however, by assigning multiple work-items to each atom. In this case, multiple work-items accumulate the force over a subset of neighbors with a final reduction to store the total force.

This additional reduction can add significant storage and computational overhead, however. First, additional local memory must be allocated for reduction to accumulate the total force, virial, and energy terms. For some accelerators, this can impact performance by limiting the number of work-items that can run on each compute unit and thus the ability to hide memory access latencies. Second, additional operations are required to perform the reduction when compared to the case of a single work-item per atom. Finally, synchronization barriers can be necessary to assert that each thread has finished the force calculation before performing a reduction. Because simulations that require consideration of long-range electrostatics typically have larger cutoff distances, however, we felt that it was important to evaluate this approach for improving parallel efficiency.

The reduction for the force, $f$, and energy terms is performed as follows. Let $\text{BLOCK\_PAIR}$ be the number of work-items per work-group, $\text{BLOCK\_PAIR} > tid >= 0$ be the index for the work-item within the work-group, $\text{red\_acc}$ be the local memory allocated for performing the reduction, and $\text{t\_per\_atom} < \text{BLOCK\_PAIR}$ be an arbitrary number of work-items assigned per atom under the restraint that $\text{BLOCK\_PAIR}$ is divisible by $\text{t\_per\_atom}$. Then, the reduction can be performed with the following code:

```c
__local float red_acc[6][BLOCK_PAIR];
red_acc[0][tid]=f.x;
red_acc[1][tid]=f.y;
red_acc[2][tid]=f.z;
red_acc[3][tid]=energy;

decide_sync_barrier();
int offset=tid%t_per_atom;
for (unsigned int s=t_per_atom/2; s>0; s>>=1) {
    if (offset < s)
    for (int r=0; r<s; r++)
        red_acc[r][tid] += red_acc[r][tid+s];
    decide_sync_barrier();
}

f.x=red_acc[0][tid];
f.y=red_acc[1][tid];
f.z=red_acc[2][tid];
energy=red_acc[3][tid];
```

Asserting that $\text{t\_per\_atom} < \text{BLOCK\_PAIR}$ allows the reduction to be performed in local memory on a single compute unit. If there is a possibility that the work-items assigned to a particular particle can be executing different instructions, the `decide_sync_barrier` routine in the listing above must execute a synchronization barrier instruction so that the data from all work-items is guaranteed to be available in local memory for summation. These barriers within the reduction loop can
hinder performance for the kernel. For GPUs (NVIDIA and AMD), these synchronization barriers can be removed under certain conditions because of the restriction that the processing elements on the compute unit are not free to execute different instructions. For all current NVIDIA GPUs, groups of 32 work-items with sequential IDs are guaranteed to be on the same instruction. Therefore, a significant performance improvement is attained by removing the barriers under the restriction that \( t_{\text{per atom}} \leq 32 \) and \( t_{\text{per atom}} \) is 0; this is the case for all tests performed here. In cases where the virtual tensor must be computed, a second reduction loop is performed to accumulate the 6 per-particle virial terms (not shown); therefore, \( \text{red}_\text{acc} \) is allocated with 6 elements. Two loops are used in order to reduce the local memory requirements of the kernel.

The neighbor list is stored on the GPU as a dense row-major matrix with each column containing the neighbor indices for one particle. In the case where \( t_{\text{per atom}} \) is 1, this allows for coalesced memory access when fetching the neighbors for the work-items on the compute unit. This will not necessarily be the case when particles are shared by multiple work-items, however. In the case where \( t_{\text{per atom}} = \text{BLOCK}_\text{PAIR} \), a column-major matrix can be used for coalesced access. Because better performance can be achieved on many accelerators by setting \( \text{BLOCK}_\text{PAIR} \) to be much larger than the number of physical processing elements on the compute unit and synchronization barriers are required when \( t_{\text{per atom}} \) is sufficiently large, this restriction is unlikely to improve performance. Alternative solutions are not straightforward and also impact the performance of the neighbor build kernels. For the work performed here, we use the same neighbor storage (row-major/column per particle) regardless of the value for \( t_{\text{per atom}} \), however, we note that because neighbor builds are not performed every timestep, there is potential for improvement with a data packing kernel that preserves coalesced access.

Three precision modes for short-range force calculation are supported. \textit{Single precision} uses single precision for all floating point storage and calculations. \textit{Double precision} uses double precision for all floating point storage and calculations. \textit{Mixed precision} uses double precision only for storage and accumulation of forces, energies, and virials. The number of work-items per work-group is decided at compile time based on the hardware. Here, 128 is used for all cases.

2.7. Accelerating Charge Assignment

We consider charge assignment using splines of arbitrary order, \( P \). The naive implementation, assigning one particle to each work-item, can be used to minimize the number of calculations performed with each work-item looping over the particle’s surrounding \( P^3 \) grid points to assign the charge density. For shared memory parallelism, however, this becomes complicated by the possibility of memory collisions when multiple work-items simultaneously update the same grid point. Although single precision atomic operations are now supported on NVIDIA GPUs, we have been unable to achieve efficient charge assignment with the use of floating point atomic operations.

A more efficient method has been proposed by Harvey and De Fabritiis [8] that assigns work-items to each grid point, rather than each particle. Although this approach was implemented for SPME, the charge assignment for \( P^3M \) is very similar. Their approach is a pencil decomposition that assigns all grid points at fixed \( y \) and \( z \) position to a work-group. The \( g \) work-items within the group loop over the grid points in the pencil with a stride of \( g \). At each loop iteration, the fractional charges from any particles within the support of the charge assignment function are accumulated for the grid point. In order to avoid an \( O(N) \) search for particles within the support, a “placement” kernel is executed before the charge assignment to map each particle to a grid point. The placement kernel stores the position and charge of each mapped particle in a 4-dimensional array. In their approach, at most one particle is mapped to a grid point. Any additional particles are placed into an “overflow list” that is subsequently processed by a separate kernel that assigns charge using the naïve approach.

This approach has some advantages for use on accelerators. First, floating-point atomic operations are not required for charge density accumulation. Second, assuming that the grid points along \( x \) are in the fastest changing dimension of the density array and the particle map array, memory access to retrieve particle charge and positions along each pencil can be coalesced, as can the storage of the charge density for each grid point. The tradeoff for these advantages, however, is the requirement for recomputation of spline terms. Recomputation along the \( x \) dimension can be eliminated by using local memory for temporary accumulation of the \( g \) grid points processed per loop iteration along with the surrounding \( P \) points within the support of the charge assignment function. In this approach, each work-item only loads the positions/charges mapped to the surrounding \( P^2 \) grid points that are at the same position along the \( x \) dimension. This can reduce the redundant computations by a factor of \( P \).

In order to achieve efficient charge assignment within LAMMPS, we have made several modifications to the algorithm in order to improve efficiency. First, we allow an arbitrary number of particles to be mapped to each grid point. In the work by Harvey and De Fabritiis [8], they reported that it was more efficient to use the naïve approach to handle any additional particles than to use subsequent iterations of their charge gathering algorithm. However, for the simulations and associated SPME parameterizations of interest, the particle mapping arrays were typically 90% sparse. The relative efficiency of the two algorithms will depend on both the particle density and the mesh spacing. For example, for the 32000 atom rhodopsin benchmark using \( P^3M \) in Figure 1, 20000 grid points are used \( (P = 5) \) for an approximate RMS precision of \( 7.57 \cdot 10^{-5} \). Therefore, the naïve approach is not used here. Instead, an arbitrary number of particles can be mapped to each point with dynamic resizing of the map array as necessary. As an additional optimization, we do not store the particle location, but rather, a function of the particle distance from the grid point needed for the charge assignment and back-interpolation spline calculations.

The kernel for the mapping assigns each particle to a work-item. The work-item updates the particle count for the grid point and stores the relative position and charge at the appropriate location within the array. Therefore, this particle count up-
date must be an atomic operation to avoid collisions from multiple work-items mapping particles to the same grid point. In LAMMPS, the particle indices are periodically sorted according to spatial location in order to improve cache performance. Therefore, in-order access of particle locations for mappings have a much higher probability for collisions during the atomic update because the probability that two particles are simultaneously mapped to the same point is much higher. Although this does not affect correctness, it can significantly impact performance because the latencies for simultaneous atomic updates to the same memory location can be much higher. We therefore reindex the order in which particles are mapped using this kernel. For \( n \) particles processed by the kernel, index \( i \) will access the \( i \cdot g-(i \cdot g)/n \cdot (n-1) \) particle within the array. This reindexing can reduce the kernel time by up to 75% in our tests.

The charge assignment is performed in parallel by dividing the mesh into “bricks” assigned to each MPI process according to the spatial decomposition in LAMMPS. The grid points in each brick are then divided into pencils for charge assignment on the accelerator. Achieving good parallel efficiency with this type of decomposition is difficult, however. For the rhodopsin benchmark in Figure 1, the number of grid points per pencil is limited to 32 for a serial run without any spatial decomposition. This is already equal to the number of cores on a single multiprocessor for a M2070 accelerator. Although a parameterization with a finer mesh spacing can potentially be more efficient when using acceleration for \( P^3M \), clearly, a solution with better parallel efficiency is desirable.

A solution to this problem is not trivial, however, when restricted to the standard particle mesh algorithms. Assigning multiple work-items to perform the accumulation for a single grid point, for example, requires (in addition to the overhead for the reduction), an increase in the number of repeated calculations associated with an increase in work-item divergence. The solution to the problem, proposed here, is to assign an arbitrary number of pencils, \( p \), to each work-group. This allows for a finer degree of parallelism that retains the desirable features of the algorithm: 1) no atomic floating point operations are required, 2) local memory can be used to remove redundant spline computations in the \( x \) dimension, 3) memory access along the \( x \) dimension can potentially be coalesced into \( g/p \) reads per compute unit, and 4) the algorithm is general for a range of different spline orders \( P \).

The potential gains from this approach will depend on the accelerator. On current NVIDIA hardware, the charge assignment kernel is memory bound. On these chips, memory access is grouped according to \( t \) work-items. In the best case, this can result in a single memory access for the \( t \) work-items on the compute unit when the data can be coalesced. In the worst case, \( t \) accesses are required. This is the restriction that limits the performance gains from an increase in \( p \) in our benchmarks. Therefore, we cannot improve parallel efficiency beyond the case where \( p = g/t \). For the GF100 chips used here, \( t = 32 \). In earlier models, \( t = 16 \). The work-group sizes for the particle map and charge assignment kernels (\( g \)) are determined at compile time based on the accelerator. For the benchmarks presented here, 64 is used for both kernels (and therefore, 2 pencils are processed in each work-group on a compute unit).

Because we do not consider acceleration for the field solve to compute the potential gradient on the mesh, the data within the density bricks must be transferred back to the host. Because the size of the brick array is fixed throughout the simulation loop, this can be performed using a single page-locked memory allocation on the host along with a single allocation on the device. In this case, the data transfer turns out to be a small fraction of the kernel time for mapping and interpolation in our tests.

2.8. Accelerating Force Interpolation

In order to compute per-particle forces, the mesh is transferred back from the host to the device. Because the computed data is now stored per particle, each work-item in the interpolation kernel can be assigned to a particle. This allows for a trivial implementation for force interpolation with no redundant computation. In order to reduce work-item register usage, our implementation uses local memory to store the weight coefficients computed from the splines in the \( y \) and \( z \) dimensions. Once the interpolation kernel has completed, the per-particle forces are transferred back to the host. As with the other kernels, the work-group size is hardware dependent with 128 used for the benchmarks here.

The charge assignment and interpolation calculations for \( P^3M \) can be performed in both single and double precision.

2.9. CPU/GPU Concurrency

The routines for host-device data transfer, device memory initialization, short-range non-bonded force calculation, and \( P^3M \) charge assignment are all run with asynchronous launches that allow for concurrent calculation of short-range non-bonded forces, and bond, angle, dihedral, and improper forces on the CPU. In the case of non-bonded forces, calculations can be split between CPUs and accelerators either by specifying a fixed split of particles or with dynamic calculation of the split to balance the calculation times. Multiple MPI processes can share an accelerator (details of this approach have been described previously [3]), but there is no current support for using multiple GPUs from a single MPI process.

Poisson’s equation is solved on the mesh following completion of all short-range, bond, angle, dihedral, and improper forces. If the charge assignment routine and data transfer of the density brick has not completed on the accelerator, the host will block until calculations can begin. Because there is no current method for asynchronous memory allocation on the device, a flag is set by the charge assignment kernel if there is insufficient storage available for mapping all particles to the mesh. In this case, the memory allocation is performed and the charge assignment is repeated before proceeding. While back-interpolation of forces is performed on the accelerator, the CPU remains idle until final time integration can be performed. (The same is true for the neighbor list kernel on timesteps where the list must be rebuilt).
2.10. Geryon Library

Currently, there are 3 prevalent low-level APIs for programming accelerators - CUDA-Driver, CUDA-Runtime, and OpenCL. For our LAMMPS implementation, we have used the Geryon library that provides a succinct API allowing a single code to compile with both CUDA and OpenCL [3]. The Geryon library is available under the Free-BSD license from http://users.nccs.gov/~wb8/geryon/index.htm.

2.11. Keeneland IDS

 Benchmarks were performed on the Keeneland initial delivery system. The system is an experimental HP SL-390 (Ariston) cluster with 120 nodes. Each node contains two 2.8 GHz Intel Westmere hex-core CPUs and 3 Tesla M2070 GPUs. Each GPU contains 6GB GDDR5 memory. Nodes are networked with a Qlogic QDR InfiniBand interconnect. For the CUDA molecular dynamics tests, device code was compiled with the CUDA toolkit 3.2. Host code was compiled using OpenMPI 1.5.1 with the Intel 11.1.073 C++ compilers. Host code was compiled with “-O2 -xHost -ipo -fno-alias” optimization. Device driver version was 270.41.49. For the OpenCL comparisons, device code for both CUDA and OpenCL was compiled with version 4.0 of the CUDA toolkit. All tests were run with ECC support enabled.

3. Results

For our initial analysis of acceleration for simulations considering long-range, we have used the rhodopsin benchmark with a 1:1 CPU core to GPU ratio. Although most hardware platforms currently have a higher ratio of CPU cores to GPUs, the 1:1 ratio is convenient in that it allows for 1) a comparison where the times for non-accelerated routines are similar and 2) accurate timings for individual device kernels and host-device data transfer times (not currently possible with GPU sharing in LAMMPS).

The timings for short-range non-bonded van der Waals and electrostatic force calculations are shown in Figure 2. When using 1 thread per atom, the calculation is 33.4 times faster with acceleration for 1 CPU core/GPU and 5 times faster for 32 CPU cores/GPUs (based on CPU-only times of 120 s and 3.85 s respectively). For 4 GPUs, approximately 8000 local atoms are processed on each GPU. Each GPU has 448 cores, however, little improvement in the computational time is obtained using 8 or more GPUs. The times for GPU acceleration do not include data transfer or latencies from GPU drivers. The primary reason for the relative decrease in performance is the decrease in the number of work-items performing simultaneous force calculations on each compute unit; the force calculations are scheduled with 128 work-items per work-group in order to allow overlap of memory access and computations.

The multi-GPU parallel efficiency for the short-range calculation can be improved by assigning multiple threads to each particle. The optimal number of threads will depend on the hardware, the force-field, and the number of particles per kernel. We have found that using 8 threads per particle is generally efficient at both large and small particle counts on the GF100 chips. Therefore, we have set this as the default parameter when using long-range models on “Fermi” cards. For the rhodopsin benchmark here, the single GPU performance improves to be 36 times faster than the CPU; more importantly, the 32 GPU case improves to be 32.3 times faster. The performance at small particle counts can be improved by increasing the number of threads per particle above 8, however, for simplicity, we use 8 for all tests presented here.

We have considered splines with order 4, 5, or 6 for P^3M acceleration parameterized such that the approximate RMS error is below 1 \cdot 10^{-4}. For P = 4, this results in a 30 x 40 x 36 mesh. For P = 5, the mesh is 25 x 32 x 32 and for P = 6, it is 24 x 32 x 30. The relative performance for the charge assignment acceleration is shown in Figure 3 which compares the time required for the CPU routines with the cumulative time for mesh initialization, particle mapping, charge assignment, and host-device data transfer on the accelerator. The relative performance with a spline order of 5 is generally best, with a speedup of 7.77 on a single GPU and 1.61 on 32 GPUs (based on CPU-only times of 7.46 s and 0.24 s respectively).

The charge assignment on the GPU is bound by memory access latencies. This impacts the results in two important ways. First, the double precision performance is similar to the single precision performance with a speedup of 6.63 on a single GPU and 1.5 on 32 GPUs despite the decrease in available double precision ALUs on the M2070. Second, the relative performance of the accelerated simulations decreases much more rapidly with an increase in GPUs when compared to the short range force calculation. We have improved the parallel efficiency of the charge assignment significantly with the use of multiple pencils on each microprocessor; the charge assignment kernel is 46% faster on a single GPU for order 5 splines and
Figure 3: Speedups for the charge assignment routines (Upper) and force interpolation (Lower) with use of GPU acceleration. Speedups are shown for order 4, 5, and 6 splines in single (SP) and double (DP) precision. The numbers are from strong scaling timings using the 32000 particle rhodopsin benchmark and compare times with 1 CPU core per GPU. Host-device data transfer is included in the timings. Speedup for the charge assignment includes mesh initialization, particle mapping, and charge assignment kernels.

Figure 4: Strong scaling timings for long-range force calculations representing the sum of the execution times for charge assignment and force interpolation on the GPU and field solves performed on the CPU. Timings are for order 4 (30 x 40 x 36 mesh), 5 (25 x 32 x 32 mesh), and 6 (24 x 32 x 30 mesh) splines using single (SP) and double (DP) precision GPU calculations. The timings for CPU-only long-range calculations with an order 5 spline are shown for reference. Comparisons use 1 CPU core per GPU.

17% faster on 32 GPUs. However, due to the relatively small number of grid points on each brick and the memory access requirements on the GF100, it is difficult to achieve significant arithmetic intensity for problems of this size.

The relative performance for the force interpolation is shown in Figure 3. Because this kernel can efficiently use a particle decomposition without atomic operations, the speedups are more significant. The speedups for splines with order 5 or 6 are most significant. In single precision, the cumulative time for the host-device data transfer and force-interpolation kernel is 14.7 times faster on a single GPU and 5.5 times faster on 32 GPUs (based on CPU-only times of 6.48 s and 0.22 s respectively). In double precision, these numbers are 8.67 and 3.89 respectively.

The relative performance of the entire long-range calculation, including all host-device data transfers, mesh initialization, computational kernels, the Poisson solve, and MPI communications is shown in Figure 4. The best results are achieved using order-5 splines. In single precision, the calculation is 5.19 times faster on a single GPU and 2.28 times faster on 32 GPUs (based on CPU-only times of 17.04 s and 1.61 s respectively). The double precision results are very similar - 4.54 times faster on a single GPU and 2.19 times faster on 32 GPUs. The breakdown of the total time on the GPU for these order-5 long-range calculations is shown in Figure 5. Although our approach involves data transfer of the entire brick from the host to the accelerator (and back) each timestep, this represents a small fraction of the total long-range time (at most 13% of the time). The majority of the time is spent in the charge assignment routine.

Use of the Geryon library allows LAMMPS to compile with both CUDA and OpenCL. This allows LAMMPS to run on
any accelerator or multi-/many-core CPU with an available OpenCL driver. A performance comparison on NVIDIA hardware using OpenCL is shown in Figure 6 for the rhodopsin benchmark. The accumulated time for host-device data transfer, short range force calculation, P3M particle mapping, charge assignment, and force interpolation is approximately 24% larger for the OpenCL executable. The majority of the difference results from decreased performance in the short range force kernels. The OpenCL timings in this case were approximately 34% slower.

Relative performance for the entire simulation loop is shown in Figure 7. In single precision, the simulation is 8.61 times faster when using a single GPU and 3 times faster on 32 GPUs. Mixed precision results are very similar, 8.45 times and 2.89 times respectively. The double precision results are 5.46 times faster on a single GPU and 2.55 times faster when using 32 GPUs.

The breakdown of these simulation times on the GPU and the CPU for the accelerated simulations are shown in Figure 8 for mixed precision. Host-device data transfers (including data casting and packing) are between 2% and 4% of the total simulation time. The percentage of loop time that the CPU cores are idle is smaller than the percentage of time spent performing computations on the GPU due to overlap of the short range force and charge assignment routines on the GPU with force calculations for bonded atoms (bond, angle, dihedral, and improper terms) on the CPU cores. Although the neighbor list builds are not performed every timestep, they constitute a significant fraction of the simulation time. This fraction grows with the number of GPUs, limiting CPU/GPU concurrency and
Figure 8: Breakdown of the mixed precision loop times in Fig. 7 into time spent for various routines on the CPU (top) and on the GPU (bottom). In both cases, percentages represent the fraction of the wall time required to complete the simulation loop. "Idle" timings represent the percent time that the CPU or accelerator are not performing computations while waiting for data. Certain host-device data transfers, short range pairwise force calculations, and charge assignment can be performed concurrently with bond, angle, dihedral, and improper force calculations.

significantly impacting the parallel efficiency for the accelerated simulations.

The GPUs are idle for greater than 50% of the time for all benchmarks. A significant fraction of this is unavoidable due to MPI communications and the field solve. However, a significant fraction of this time is spent on force calculations for bonded atoms and “other” calculations such as time integration. For these reasons, performance can often be improved by running on more CPU cores than GPUs with MPI processes sharing available GPUs. This shortens the computational times for routines that are not ported for use on the accelerator, decreases idle times on the GPU and improves CPU/GPU concurrency.

Timings for the 32000 particle rhodopsin benchmark are shown in Figure 9 when using 1, 2, and 4 MPI processes for each GPU on the node. Timings are also shown using a 256000 particle simulation obtained from replicating the rhodopsin simulation box 8 times in order to double the length in each dimension. For these benchmarks, all CPU cores are used for CPU-only simulations and therefore the results are a fair comparison of expected LAMMPS rhodopsin simulations times that can be achieved with and without GPU acceleration on the Keeneland cluster. The simulations were run using 1 to 32 nodes (3 to 96 GPUs, 3 to 384) cores.

For the 32K simulation, the simulation rate is between 4% and 30% faster when using 2 MPI processes per GPU than for a single process. Using 4 processes per GPU (not shown) was slower in all timings. For the 256K simulation, the performance using 4 MPI processes per GPU is 40% faster than a single MPI process on a single node. On 32 nodes, the timings were similar for 1 and 2 MPI processes and slower for 4 MPI processes. A breakdown of the timings when using 2 MPI processes per GPU is shown in Figure 9. Using multiple MPI processes sharing GPUs decreases the wall times for routines not ported for acceleration, increases GPU utilization (lower idle times on the GPU), and improves GPU/CPU concurrency. In many of the timings, for example, there are no potential gains from acceleration of the calculation of force terms between bonded atoms; these calculations run entirely on the CPU while the GPU performs other computations.

For the 32K simulation, the simulation rate is 3.1 times faster with GPU acceleration on a single node and 1.36 times faster on 8 nodes. For the 256K simulation, the rate is 3.9 times faster on a single node and 2 times faster on 32 nodes. If P3M calculations are instead performed on the CPU, they can also run concurrently with other GPU calculations. Although this can decrease the potential for performance improvements, using P3M acceleration is faster for all tests described here. For the 32K simulation, the best overall simulation rates on 1 to 4 nodes are 14% to 20% faster with P3M acceleration. On 8 nodes, however, it is only 3% faster. For the 256K simulation, the overall rates are 8% to 19% faster.

The results for the 32000 and 256000 rhodopsin benchmarks are summarized in Table 2.
4. Discussion

Algorithms for efficient acceleration of P^3M are not straightforward on current hardware due to the high latencies for atomic floating point operations and non-contiguous memory access, the relatively small mesh sizes for P^3M, and the difficulty in achieving fine-grained parallelism for spline computations without redundant computations and work-item divergence. The relative performance of the charge assignment and force-interpolation kernels on NVIDIA GPUs is much less impressive when compared to short-range force calculations. Nonetheless, use of GPU acceleration for P^3M results in faster simulations on the Keeneland GPU cluster than would be possible without. Performing charge assignment calculations on the accelerator allows for concurrent computations with CPU routines and allows for efficient long-range calculations with fewer MPI processes. The latter is significant due to the MPI communications bottleneck for Poisson solves on many nodes. For these reasons, along with the impressive speedups for short-range force calculation, the overall relative performance of the simulations is much more impressive. The simulation times presented here with GPU acceleration on the Keeneland cluster can be faster than those achieved using 4 times as many nodes without GPU acceleration. For simulations in the microcanonical (NVE) ensemble, the speedups are more significant (data not shown).

Host-device data transfer is often cited as a bottleneck to attaining efficient GPU acceleration. In our approach, we transfer all data positions, types, forces, mesh charge densities, and mesh energy gradient terms at every timestep. When necessary, per-particle energies and virial terms are transferred, as are neighbor lists for concurrent CPU force calculation on certain timesteps. However, the data transfer times are typically under 2% of the total simulation time for the tests performed here (data packing and casting on the host can require up to an additional 1.4%). For our work, data localization is a more precise description of the bottleneck. That is, the latencies for getting data into registers for computation are an important concern. We have presented approaches for increasing the number of active work-items performing short-range force calculation and charge assignment for P^3M for fixed problem sizes. These approaches significantly improve the parallel efficiency for accelerated simulations. However, the efficient increase in the number of work-items is limited by a tradeoff requiring additional computations and synchronization barriers. As a result, the parallel efficiency of CPU-only runs will typically be much better beyond some threshold.

LAMMPS was not initially designed for biomolecular simulation. It supports a wide range of simulations including polymers, materials, biomolecules, and mesoscale systems, many of which can require consideration of long range electrostatics. Therefore, porting the code for efficient GPU acceleration can potentially require a significant effort. Therefore, we have chosen an approach that minimizes the amount of code that must be ported for acceleration while keeping full compatibility with LAMMPS wide range of features. The profile for many simulations has flattened with GPU acceleration and therefore ap-

---

Table 2: Summary of best speedups versus a single CPU core for CPU-only and accelerated runs. The speedups were calculated from single core loop times of 436 seconds for the 32K rhodo CPU case and 3510 seconds for the 256K rhodo CPU case. Mixed precision accelerated runs were performed using mixed precision for short-range force calculation and double precision for long-range P^3M calculation.

<table>
<thead>
<tr>
<th>Test case</th>
<th>Nodes (Cores)</th>
<th>Speedup</th>
<th>Nodes (Cores)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>32K CPU</td>
<td>1 (12)</td>
<td>9.6</td>
<td>8 (96)</td>
<td>47.8</td>
</tr>
<tr>
<td>32K CPU</td>
<td>1 (6)</td>
<td>29.3</td>
<td>8 (48)</td>
<td>64.8</td>
</tr>
<tr>
<td>256K CPU</td>
<td>1 (12)</td>
<td>9.5</td>
<td>32 (384)</td>
<td>173.68</td>
</tr>
<tr>
<td>256K GPU</td>
<td>1 (12)</td>
<td>37.5</td>
<td>32 (192)</td>
<td>344.12</td>
</tr>
</tbody>
</table>

---

Figure 9: Simulation results for the rhodopsin benchmark on the Keeneland GPU cluster. Simulations are run using 1 to 32 nodes using 3 GPUs on each node. Simulations are run using 3, 6, or 12 MPI processes on each node. Results for the rhodopsin benchmark scaled to 256000 particles are also shown. (Upper) Wall clock times for the simulation loops. (Lower) Percent time required for the rhodopsin benchmark scaled to 256000 particles are also shown. (Upper) Wall clock times for the simulation loops. (Lower) Percent time required for various routines as measured on the host for the GPU 6PPN simulations. Percent times for concurrent GPU calculations are not shown.
In our approach, the calculation of force terms for bonded atoms, time integration, SHAKE constraints, and the various statistical calculations are all performed on the CPU and not the accelerator. It is tempting to argue that the most significant performance improvements can be attained by porting these routines for GPU acceleration; after all, for the rhodopsin benchmark run on 8 GPUs and 8 CPU cores (Figure 8), the GPU is idle for 56% of the computation. This is not necessarily the case, however. First, there is significant overlap of calculations performed on the CPU and GPU. If we eliminate the times for data casting, packing, and transfer and obtain an infinite speedup for the remaining CPU routines on the accelerator, a 35% performance improvement is the best that can be obtained. The Nose-Hoover style thermostat and barostat calculations constitute approximately 80% of the SHAKE-constrained time-integration in the isothermal-isobaric ensemble used here. Therefore, if we instead sample from the microcanonical (NVE) ensemble, this upper bound is reduced to 17%.

Achieving performance near this upper bound on current hardware would be very difficult due to the low arithmetic intensity, the short computational times required for many of the routines on the CPU, and the desire for double precision accumulation. Additionally, some data transfer off of the accelerator is necessary for interprocess communication of ghost particles and mesh points. Because these routines scale very well in CPU-only calculations and because the ratio of CPU cores to GPUs is typically greater than 1, multiple MPI processes sharing a single GPU can be used to decrease computational times and reduce the potential for gains even further. In the benchmarks presented here with GPU sharing, there is no benefit possible from porting force calculations between bonded atoms in most cases. This approach is not without drawbacks, however. Increasing the number of MPI processes can impact interprocess communications performance and also decreases the amount of work in each accelerator kernel. Although the GPU accelerators used here allow for multiple kernels to be run simultaneously, our experience is that this is typically not as efficient as running a single kernel processing all of the data.

Improving MPI communication limitations or the performance of GPU-accelerated routines relative to the CPU is necessary to improve this upper bound. With regard to the latter, the simulation times presented here were performed with error checking and correction enabled on the GPU. On the M2070s this can result in performance penalties of anywhere between 0% and 30% for kernels in the SHOC benchmark suite [17]. For molecular dynamics, it might be sufficient to disable ECC and carefully monitor conserved thermodynamic quantities to decrease the probability of erroneous results. The peak calculation rates on GPUs have seen significant improvements relative to CPUs; a continuation of this trend could shift the balance of computational times. It is important to note, however, that improvements in peak performance do not necessarily translate into improved performance with acceleration. This is particularly true for kernels bound by memory access latencies or kernels with significant work-item divergence; increasing the number of processing elements on a compute unit without improvement in memory access times can result in little or no improvement in execution times. These kernels can limit performance improvement for the overall application (Amdahl’s law) and this is, in part, the reason for the distinction between a CPU and an accelerator.

Improving parallel efficiency for a code designed for distributed machines can be challenging when using accelerators optimized for massive data parallelism and contiguous memory access. The target is improving performance for latency-bound kernels performing calculations on small amounts of data. This can be achieved by improving memory access patterns for the algorithm or improving the fine-grained parallelism to overlap memory access with computation. We have presented improvements in LAMMPS targeted at the latter in order to maintain consistency with CPU algorithms. However, there are also options for the former for P3M algorithms. For example, there is an upper bound for Poisson solvers in LAMMPS targeted at the latter in order to maintain consistency with CPU algorithms. However, there are also options for the former for P3M algorithms.

Achieving performance near this upper bound on current hardware would be very difficult due to the low arithmetic intensity, the short computational times required for many of the routines on the CPU, and the desire for double precision accumulation. Additionally, some data transfer off of the accelerator is necessary for interprocess communication of ghost particles and mesh points. Because these routines scale very well in CPU-only calculations and because the ratio of CPU cores to GPUs is typically greater than 1, multiple MPI processes sharing a single GPU can be used to decrease computational times and reduce the potential for gains even further. In the benchmarks presented here with GPU sharing, there is no benefit possible from porting force calculations between bonded atoms in most cases. This approach is not without drawbacks, however. Increasing the number of MPI processes can impact interprocess communications performance and also decreases the amount of work in each accelerator kernel. Although the GPU accelerators used here allow for multiple kernels to be run simultaneously, our experience is that this is typically not as efficient as running a single kernel processing all of the data.

Improving MPI communication limitations or the performance of GPU-accelerated routines relative to the CPU is necessary to improve this upper bound. With regard to the latter, the simulation times presented here were performed with error checking and correction enabled on the GPU. On the M2070s this can result in performance penalties of anywhere between 0% and 30% for kernels in the SHOC benchmark suite [17]. For molecular dynamics, it might be sufficient to disable ECC and carefully monitor conserved thermodynamic quantities to decrease the probability of erroneous results. The peak calculation rates on GPUs have seen significant improvements relative to CPUs; a continuation of this trend could shift the balance of computational times. It is important to note, however, that improvements in peak performance do not necessarily translate into improved performance with acceleration. This is particularly true for kernels bound by memory access latencies or kernels with significant work-item divergence; increasing the number of processing elements on a compute unit without improvement in memory access times can result in little or no improvement in execution times. These kernels can limit performance improvement for the overall application (Amdahl’s law) and this is, in part, the reason for the distinction between a CPU and an accelerator.

Improving parallel efficiency for a code designed for distributed machines can be challenging when using accelerators optimized for massive data parallelism and contiguous memory access. The target is improving performance for latency-bound kernels performing calculations on small amounts of data. This can be achieved by improving memory access patterns for the algorithm or improving the fine-grained parallelism to overlap memory access with computation. We have presented improvements in LAMMPS targeted at the latter in order to maintain consistency with CPU algorithms. However, there are also options for the former for P3M algorithms. For example, there is an upper bound for Poisson solvers in LAMMPS targeted at the latter in order to maintain consistency with CPU algorithms. However, there are also options for the former for P3M algorithms.

Alternative algorithms for long-range electrostatics are an important consideration. For large simulations, Poisson solvers using multigrid [19] or similar approaches offer the potential for better parallel efficiency and significant speedups have been attained in serial with GPU acceleration for the multilevel summation approach [11]. The same or similar charge assignment and force interpolation routines can be used with different Pois-
son solvers and therefore much of the work presented here is still applicable. Pairwise $O(N)$ alternatives to Ewald summation for long-range electrostatics [20] are a very interesting direction for achieving high speedups and parallel efficiency with GPU acceleration.

Our implementation for using LAMMPS with accelerators allows for host-device load balancing of the non-bonded short-range force calculations to allow for concurrent calculations on the CPU and the accelerator. As can be seen in Figure 9, the CPU is idle for a significant fraction of the simulation time while waiting for GPU calculations to complete. Because the majority of this time is spent in neighbor list builds for many of the timings and because these calculations cannot be currently overlapped with CPU work, there is little improvement in performance for mixed precision simulations for the benchmarks and hardware configuration used here. Therefore, we have left this approach for discussion elsewhere.

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. The work was supported in part by the National Science Foundation through grant number CHE-09-46358 and computer time on the Keeneland initial delivery system hosted at the National Institute for Computational Science under grant number UT-NTNL0039. All of the code described in this paper is available in the open-source LAMMPS software package, available at http://lammps.sandia.gov/.

References