Computer Physics Communications 203 (2016) 45–52

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Efficient neighbor list calculation for molecular simulation of colloidal systems using graphics processing units

Michael P. Howard [a], Joshua A. Anderson [b], Arash Nikoubashman [a,1], Sharon C. Glotzer [b,c], Athanassios Z. Panagiotopoulos [a,*]

[a] Department of Chemical and Biological Engineering, Princeton University, Princeton, NJ 08544, USA

[b] Department of Chemical Engineering, University of Michigan, Ann Arbor, MI 48109, USA

[c] Department of Materials Science and Engineering, University of Michigan, Ann Arbor, MI 48109, USA

ARTICLE INFO

Article history:
Received 24 August 2015
Received in revised form
1 February 2016
Accepted 4 February 2016
Available online 3 March 2016

Keywords:
Molecular simulation
Colloid
Size disparity
Non-uniform
Neighbor list
Bounding volume hierarchy
GPU

ABSTRACT

We present an algorithm based on linear bounding volume hierarchies (LBVHs) for computing neighbor (Verlet) lists using graphics processing units (GPUs) for colloidal systems characterized by large size disparities. We compare this to a GPU implementation of the current state-of-the-art CPU algorithm based on stenciled cell lists. We report benchmarks for both neighbor list algorithms in a Lennard-Jones binary mixture with synthetic interaction range disparity and a realistic colloid solution. LBVHs outperformed the stenciled cell lists for systems with moderate or large size disparity and dilute or semidilute fractions of large particles, conditions typical of colloidal systems.

1. Introduction

There has been a steady growth of computational resources available to the scientific community [1]. The fastest supercomputers today offer petascale performance through hundreds of thousands of CPU cores with dedicated coprocessors or graphics processing units (GPUs) as accelerators. Whereas in the past most atomistic molecular simulations were restricted to no more than a few hundred particles over nanosecond time scales, modern computing architectures and simulation techniques have enabled simulations of millions of particles [2] and up to millisecond [3] time scales. In particular, molecular dynamics (MD) methods have emerged as powerful tools for large-scale molecular simulations for two important reasons: (1) the MD algorithm is highly parallel and so easily adapted for supercomputing, and (2) many highly optimized and flexible simulation packages are readily available to researchers. In the past two decades, significant time and resources have been devoted to the development of such MD packages, including GROMACS [4], LAMMPS [5], and NAMD [6], among many other commercial and open-source options. These packages provide robust MD implementations for massively parallel computers. A more recent addition is HOOMD-blue [7], which was developed and optimized for GPUs, and has a single GPU performance one order of magnitude faster than a single CPU [8].

Despite these advances in hardware and software, MD simulations of soft matter remain challenging because there is typically a large disparity in length and time scales between components. For example, colloidal particles (nanometers to micrometers in diameter) in solution are separated in size by several orders of magnitude from an atomistic description of the molecular solvent. MD simulations retaining full atomistic detail of the solvent can become intractable because many solvent atoms must be included to model only a few colloidal particles. Moreover, the time scales associated with the degrees of freedom of the solvent are generally much shorter than the relatively slow motion of the larger colloids. This means that these simulations require very short MD time steps to faithfully capture the dynamics of the solvent, and many such steps are required to observe any appreciable dynamics of the colloids.

The MD algorithm in its simplest form consists of two basic steps: (1) calculation of the forces on all particles and

* Corresponding author.
E-mail address: azp@princeton.edu (A.Z. Panagiotopoulos).

[1] Present address: Institute of Physics, Johannes Gutenberg University Mainz, Staudingerweg 7, 55128 Mainz, Germany.

http://dx.doi.org/10.1016/j.cpc.2016.02.003
0010-4655/

(2) integration of Newton's equations of motion. The force calculation is by far the most computationally expensive part of the MD algorithm. In particular, the calculation of nonbonded pair interactions between particles typically dominates the force calculation. In the simplest MD implementation, the forces between all possible pairs of the N total particles in the simulation are evaluated, leading to $O(N^2)$ scaling.

To reduce the number of force pairs evaluated, the interaction potential between particles of types i and j is typically truncated at a radial cutoff distance r_{ij} where the force has decayed sufficiently so that truncation does not significantly influence the properties of interest. A neighbor (Verlet) list storing a list of particles that are within the cutoff is then created for each particle [9]. The pair forces only need to be computed for the particles in the neighbor list, which is a small subset of N for each particle. The neighbor list can be rebuilt less frequently than every MD step if a small buffer width is added to r_{ij}, trading wasted pair force distance checks with the frequency of rebuilding the neighbor list, which accelerates the calculation compared to evaluating all possible force pairs at every step. However, the force calculation is ultimately still $O(N^2)$ if the neighbor list is built by simply checking the distances between all particle pairs.

Acceleration structures reduce the computational cost of building the neighbor list by restricting the neighbor search for each particle to a subset of the particles in the system. The most commonly employed acceleration structure in general-purpose MD codes is the cell list. A typical cell list spatially bins particles into uniformly sized cells in $O(N)$ [9]. Distance checks must only be performed for particles that are in neighboring cells, effectively reducing the cost of computing the neighbor list to $O(Nm)$, where m is the average number of particles in a cell (usually $m \ll N$). The cell width is typically determined by the largest cutoff radius between all pairs so that 27 cells must be checked for each particle in three-dimensional simulations.

The cell list is extremely efficient in simulations that have nearly equal pair force cutoffs and a uniform particle distribution between the cells. However, performance degrades significantly in colloidal systems due to the large disparity in interaction lengths. Many unnecessary distance checks are performed for particles with short interaction ranges when the cell width is based on the largest cutoff, schematically illustrated in Fig. 1. The cell width is based on the largest cutoff r_{BB}. The solvent particles with cutoff r_{AA} must check the same number of cells (particles) as the colloids with the larger cutoff r_{BB}. However, unlike the colloids, the solvent particles reject many of these particles from their neighbor lists.

A general solution to this problem has been successfully deployed in LAMMPS [10]. The standard cell list is extended so that the cell width is based on the shortest cutoff and each particle type searches a different "stencil" of adjacent cells based on the largest cutoff radius. Distances to each cell in the stencil are precomputed so that a particle distance check can be skipped for many of the searched particles. This stenciled cell list method was reported to give speedups of nearly 100× for a colloidal solution with a 20:1 ratio in diameter compared to a standard cell list in LAMMPS. However, simulations of colloidal systems with such large size disparity are still extremely computationally intensive, requiring hundreds of CPU cores to obtain reasonable performance [10].

In this article, we explore two parallel algorithms for efficiently building neighbor lists in colloidal systems on the GPU: one based on stenciled cell lists and one based on a hierarchical tree acceleration structure. To our knowledge, the stenciled cell list algorithm [10] has not been previously implemented and tested on the GPU. In graphics processing, hierarchical tree data structures are used for performance-critical neighbor searches [11]. One such tree structure, the bounding volume hierarchy (BVH), has previously been used to generate neighbor lists between large macromolecules on the CPU [12,13]. However, these prior studies did not extend their approach to general-purpose molecular simulations, did not address size disparity between different particle types, and did not discuss implementation of the algorithm on GPUs.

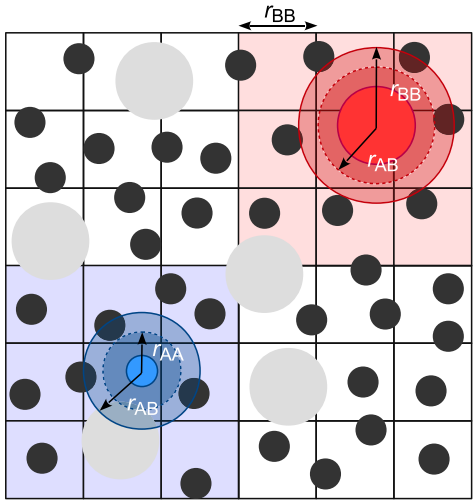

Fig. 1. Cell list needed to determine the neighbors of solvent (A) and colloid (B) particles when the bin size is based on the largest cutoff length r_{BB}. The pair interaction ranges are illustrated for each particle type. The shaded areas indicate the cells that each particle must search.

In Section 2, we present and compare parallel algorithms for the stenciled cell list and BVH. Technical details of the implementation of these algorithms within the HOOMD-blue simulation package are described in Section 3. Systematic performance benchmarks for the algorithms are reported in Section 4.

2. Algorithms

2.1. Stenciled cell list

The stenciled cell list [10] is a straightforward extension of the standard cell list, and is illustrated in Fig. 2. All particles are binned into a cell list of nominal cell width Δ_{bin}. The maximum cutoff radius is determined for each particle type, and a stencil is computed from the list of offsets to neighboring cells that have a nearest separation distance within that cutoff. For example, the solvent particle has a stencil radius corresponding to r_{AB}, and the list of offsets in 2D is $(0, 0), (+1, +2), (+2, -1), \ldots$. The colloid has a stencil radius r_{BB}. All the cells included in the stencils are shaded and outlined with a solid line.

Because the stencil size is set by the maximum cutoff radius per type, many particles that will not be included in the neighbor list must still be iterated over for shorter r_{ij}. In Fig. 2, both the solvent particle and colloid have the same effective stencil size for the given cutoffs. However, extra distance checks can be eliminated by precomputing the minimum distance to each cell in the stencil. For a particle of a given type, if the minimum distance to the nearest cell is greater than the pairwise cutoff, that particle can be skipped without distance checking or reading its position. The solvent particle only needs to distance check particles of type A inside the cells indicated by the dashed line corresponding to r_{AA}, and all particles of type A can be skipped in the other cells in the stencil.

The neighbor list is then built as described in Algorithm 1. A given particle looks up the appropriate stencil of cells based on its particle type (line 4). Each member of the stencil is iterated over (line 5), and each offset from the stencil is converted into a neighbor cell based on the current cell of the particle, including

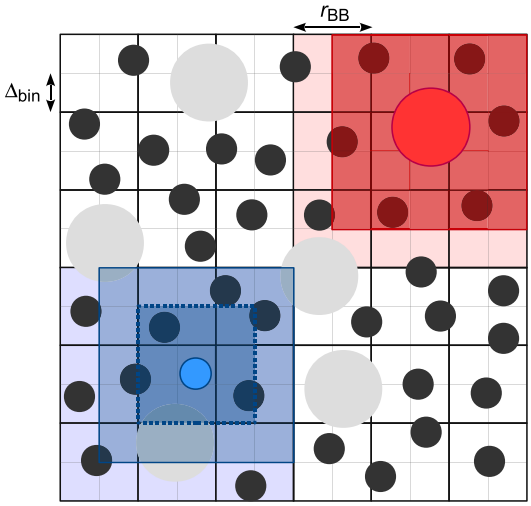

Fig. 2. Schematic illustration of a stenciled cell list with $\Delta_{bin} = r_{AA}$. Solid outlines of shaded areas indicate cells included in the stencil. The area marked by the dashed line around the solvent particle marks the cells in that stencil that are inside r_{AA}.

wrapping through the periodic boundaries (line 6). Particle i then iterates through all potential neighboring particles in the cell. Initially, only the type of neighbor particle j and its actual cutoff (for example, r_{AA} or r_{AB}) are read into memory (line 9). If the minimum distance to the nearest cell is greater than the cutoff, particle j can be skipped without a distance check (line 10). Otherwise, a distance check is performed, and the particle is saved if it is a neighbor (lines 11–13).

Algorithm 1 Stenciled cell neighbor list

```
1:  for each particle 0 ≤ i < N in parallel
2:      x_i ← particle position
3:      c_i ← particle cell
4:      S ← stencil for type of i
5:      for each s in S do
6:          c ← WRAP(c_i + s)
7:          d ← minimum distance to c
8:          for each particle j in c do
9:              r_ij ← cutoff between types of i and j
10:             if d > r_ij then continue
11:             x_j ← position of j
12:             if |WRAP(x_j − x_i)| ≤ r_ij then
13:                 ADDNEIGHBOR(j)
```

Although reducing the cell width can significantly reduce the number of distance checks performed, there is an associated penalty due to the additional data that is accessed for each cell. As the cell width shrinks, more cells must be accessed, and each cell contains fewer particles on average so that some cells may even become empty. The number of particles in these cells must still be read regardless of occupancy. In the original CPU implementation in LAMMPS [10], the optimal Δ_{bin} was found to be half the minimum cutoff distance, or $\Delta_{bin} = r_{AA}/2$. It is unclear if this will be the optimal value for GPUs, and we will discuss the effect of Δ_{bin} in Section 4.

2.2. Bounding volume hierarchy

Although the stenciled cell list significantly improves upon the standard cell list by skipping many distance checks, it still requires iteration over many particles that are ultimately unnecessary because all particles are read out of a *single* cell list. One possible way to alleviate this would be to construct one cell list *per type*,

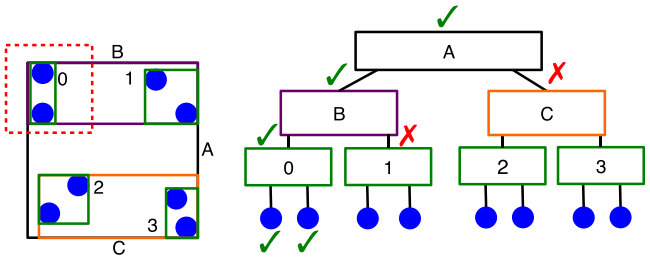

Fig. 3. Schematic illustration of a bounding volume hierarchy. *left:* Two particles are enclosed into each leaf node, outlined in green AABBs and labeled from 0 to 3. The internal nodes are labeled A to C. A query AABB is shown as a dashed box. *right:* Hierarchical representation of the tree nodes. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

and to construct different stencils for each particle type on each cell list so that only those particles that need to be checked are actually included. However, there are two complications to doing this. First, the construction of one cell list per type increases memory demands, since memory must be allocated for each cell list that spatially covers the entire simulation box. Perhaps more significantly, separating cell lists by type will lead to lower occupancy of cells, which increases the overhead associated with searching the cell lists. This is especially problematic if a certain particle type is dilute, as is often the case for colloidal systems, or if there are many types of particles.

To address these issues in a general way, we can draw inspiration from graphics rendering where an analogous neighbor search problem occurs. Realistic rendering of scenes on the computer requires the accurate projection of light. One way to do this is to cast and trace light rays from a source onto a scene, detect the objects that the rays reflect from, and appropriately render the reflections. Acceleration structures are used to efficiently determine the objects that the rays reflect from. For scenes with non-uniform object density, tree data structures such as k-d trees, octrees, and bounding volume hierarchies are generally favored over cell-based structures. In this article, we have chosen to focus on bounding volume hierarchies (BVHs). BVHs have been demonstrated as useful accelerators for ray tracing and collision detection in real-time rendering engines for computer gaming [11], in part because they can be reconstructed very quickly for scenes undergoing dynamic changes. Although BVHs do not always give the fastest *traversal* performance compared to other types of trees [14,15], their *total* performance, including both construction and traversal time, is very competitive.

BVHs partition a system based on objects rather than space. An object or multiple nearby objects are enclosed to form a leaf node. Leaf nodes are then enclosed by larger bounding ("parent") internal nodes. In an axis-aligned bounding box (AABB) BVH, the volume of the nodes is chosen so that all "child" objects are enclosed in an orthorhombic box aligned to the Cartesian axes, illustrated in the left panel of Fig. 3. Together, the nodes form a tree hierarchy that can be traversed using a binary search algorithm that tests for volume overlap between a query AABB and the AABBs of the tree.

The BVH neighbor list algorithm is outlined in Algorithm 2. One BVH is first built per particle type, and each particle is considered a point object. The height of the BVH (and so traversal time per particle) is $O(\log N)$, so multiple points that are nearby in space are merged into a leaf node for more efficient traversal. Each particle queries each tree (line 3) with a cutoff length specified per type (line 4). Periodic boundary conditions are implemented by translating the query AABB by the appropriate combinations of the box dimensions, leading to a maximum of 27 queries per tree in three dimensions (lines 5–6). However, many of these queries are

Algorithm 2 BVH neighbor list

```
1:  for each particle 0 ≤ i < N in parallel
2:      x_i ← particle position
3:      for each BVH tree t do
4:          r_ij ← cutoff between types of i and t
5:          for each image v do
6:              a ← AABB(x_i + v, r_ij)
7:              n ← root node of t
8:              while untested nodes remain do
9:                  if a overlaps n then
10:                     if n is not a leaf node then
11:                         n ← left child of n
12:                     else
13:                         for each particle j in n do
14:                             x_j ← position of j
15:                             if |x_j − x_i| ≤ r_ij then
16:                                 AddNeighbor(j)
17:                         n ← next node to test
18:                 else
19:                     Mark branch inactive
20:                     n ← next node to test
```

trivially rejected when the AABB is translated to a position where it does not overlap the simulation box.

A binary search is performed on each tree using the query AABB (lines 8–20). The search is illustrated in the right panel of Fig. 3 for the query AABB from the left panel. The query AABB overlaps the root node A (line 9), and so proceeds to test against internal node B (line 11). It intersects with B, and so tests against leaf node 0, which it overlaps. A direct distance check is performed against objects contained within leaf node 0 for inclusion as neighbors (lines 13–16). Note here that no periodic wrapping of the distance between particles is required because all periodic images of i are considered. Traversal then proceeds to the remaining untested nodes (line 17). The query AABB does not overlap 1, so it does not need to distance check those objects, and proceeds to test against C (lines 19–20). It also does not overlap C, so none of the nodes in that branch need to be checked either. Traversal of the tree for this image is then complete, and the neighbor search returns to line 5.

Although traversing the BVH is a relatively straightforward parallel algorithm, efficiently generating the BVH in parallel is much more difficult. Several parallel algorithms have been proposed to build BVHs, differing mainly in the quality of the tree that is constructed and the speed of the algorithm. Lauterbach et al. introduced one parallel method to construct "linear" BVHs (LBVHs) by ordering objects along a Z-order curve, which groups spatially close objects near each other in the tree [16]. The sorted objects can then be processed in parallel to determine the hierarchy of the tree. This algorithm has been further refined [17,18] to build "hierarchical" LBVHs that provide better construction speeds and lower memory footprints.

These algorithms process the tree from the *top down* (starting from the tree root); typically one level of the tree is processed in parallel at a time. However, this significantly restricts parallelism at top levels of the tree because these levels contain very few nodes [16]. Karras developed an alternative LBVH construction algorithm that proceeds from the *bottom up* and fully exploits the parallelism of the GPU by processing all internal nodes concurrently [19]. The quality of these LBVHs have been refined for higher ray casting performance [20]. The additional cost of refining the LBVH required casting $O(10^7)$ rays to see a significant speedup in the overall performance. However, typical molecular dynamics simulations contain $O(10^5)$ particles, and so a full neighbor list build requires only $O(10^6)$ traversals. Accordingly, we have chosen to implement the unrefined LBVH algorithm [19].

3. Implementation

Here we describe the implementation of the algorithms described in Section 2 in the HOOMD-blue simulation package [7,8,21], designed to run on NVIDIA GPUs using the CUDA programming language. In particular, we target modern NVIDIA GPU architectures ("Kepler" and newer) for optimization of our design. The code for both algorithms is available in HOOMD-blue v. 1.3 [8].

3.1. Stenciled cell list

The stenciled cell list algorithm consists of two steps: building the cell list and stencils, followed by neighbor list generation from the cell list. A cell list is constructed on the GPU every time the neighbor list is built using the existing methods in HOOMD-blue. The cell list stencil is computed on the CPU. Because it depends only on the maximum cutoff radius of each particle and the size of the cells, the cell stencil only needs to be recomputed when one of these changes. Each member of the stencil is saved in a four element structure that holds the x, y, and z offsets from a reference cell and the minimum distance to that cell. Care is taken during the stencil construction so that no cells are "double counted" due to periodic boundaries when the stencil covers the entire simulation box.

In HOOMD-blue, the standard cell list is processed to build the neighbor list using multiple threads per particle to increase parallelism [21]. The stenciled cell list similarly benefits from this optimization, and so Algorithm 1 is slightly modified in our implementation. We assign n threads per particle. Each thread follows Algorithm 1 in a strided manner so that it only processes every nth particle read from the cell list, effectively parallelizing lines 8–13 of Algorithm 1. The optimal value of n is automatically tuned at run time because the performance gained by using multiple threads per particle varies depending on the number of particles in the system, the number of particles per cell, and the GPU architecture. We restrict n to be a power of 2 smaller than the CUDA warp size so that the full neighbor list can be aggregated efficiently using intra-warp stream compaction. The original Algorithm 1 is recovered when $n = 1$.

Significant divergence can occur in the inner loop of Algorithm 1 because randomly ordered particle types can have very different stencil sizes to search or can search different areas of space. In HOOMD-blue, all particle data is periodically sorted along a space-filling curve to improve spatial locality of the data. We then additionally sort the particle indexes by type before the neighbor list is calculated. By applying a stable sorting algorithm, we still preserve most of the benefits of the space-filling curve sort. During the neighbor list build, threads operate on the particle data in this sorted order.

3.2. Building LBVHs

We build LBVHs using the highly parallel algorithm of Karras [19], which is outlined in Algorithm 3 for a single particle type. All particle indexes are first sorted along a Z-order curve using 30-bit Morton codes (lines 1–4) [22]. The Morton codes are generated by placing each particle into one of 2^{10} bins along each coordinate axis in the simulation box (line 2) and interleaving the three 10-bit integers corresponding to these bins (line 3). In order to construct one LBVH per particle type, we also prepend the bit string representing the particle types to the Morton codes to effectively sort first by particle type and then along the Z-order curve. Fig. 4(a) shows a schematic of this process, where 4-bit Morton codes are assigned in two dimensions for two particle types. Reading in lexicographic order, the first bit corresponds

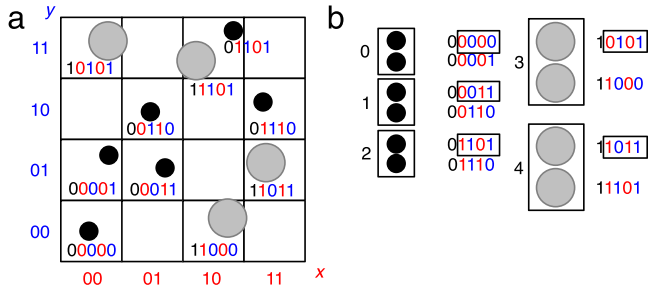

Fig. 4. 2D example of (a) assigning 4-bit type-Morton codes and (b) sorting and merging particles into primitive AABBs.

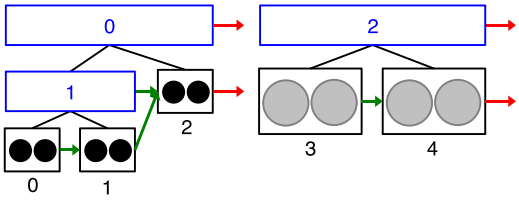

Fig. 5. Schematic LBVH tree constructed on GPU, including bubbled AABBs and skip ropes. Internal nodes are shaded in blue, leaf nodes are in black. A green arrow indicates a skip rope, while a red arrow indicates termination of traversal. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

to the particle type, while the next 4 bits are the Morton code representation of the bin. The type-Morton codes are efficiently sorted using a parallel radix sort (line 4). Because radix sort is $O(Nk)$ where k is the number of bits to compare in the key, we restrict the sort to the lowest $30+b$ bits where b is the number of bits necessary to represent the largest type index. Fig. 4(b) shows the particles in sorted order.

Algorithm 3 LBVH build

```
1:  for each particle 0 ≤ i < N in parallel
2:      (x, y, z) ← 3D bin of i
3:      m_0(i) ← MortonCode(x, y, z)
4:  SortMortonCodes() in parallel
5:  N_leaf ← ⌈N/w⌉
6:  for each leaf node 0 ≤ i < N_leaf in parallel
7:      {p} ← set of w particles in i
8:      a ← merge({p})
9:      m(i) ← m_0(p_0)
10: GenerateHierarchy(m) in parallel
11: for each leaf node 0 ≤ i < N_leaf in parallel
12:     n ← i
13:     do
14:         p ← parent of n
15:         s ← sibling of n
16:         atomic
17:             v ← times p has been visited
18:             v ← v + 1
19:         if v = 0 then return
20:         p ← merge(n, s)
21:         n ← p
```

The number of leaf nodes in each tree is computed assuming a constant number w particles per leaf filled in order along the Z-order curve (line 5). Using one thread per leaf, we merge successive sets of w particles into an AABB enclosing all the particles, and take the Morton code of the first particle as an approximation of the Morton code for that leaf node (lines 7–9). Schematically, this is indicated by the indexes and boxes shown in Fig. 4(b). Although this is not necessarily the optimal way to group particles into leaves or to represent the spatial position of the merged AABB, it is inexpensive and guarantees that the Morton codes remain in lexicographic order without an additional sorting call. We found that employing $w = 4$ particles per leaf gave the best traversal times in our benchmarks (see Section 4 for benchmark details).

The tree hierarchy, defining the parent–child edges between the nodes, is then processed using one thread per internal node (line 10). In essence, each thread processes the Morton codes to determine a "split position" in the leaf nodes for the current internal node. This split identifies both the indexes of the children of the internal node as well as whether those children are leaf nodes or internal nodes. The reader is referred to Ref. [19] for a detailed presentation of this algorithm.

Finally, the AABBs of the internal nodes are determined by processing up from the leaf nodes using one thread per leaf (lines 11–21). Each internal node is processed by the second thread to arrive at the node, with the arrival order determined by an atomic counter (lines 16–19), which guarantees that the parent internal node is processed only after its children have been processed. The AABB of each internal node is determined by merging the AABB of the active thread with the AABB of the active thread's sibling, so that the AABB of the parent node encloses both child AABBs (line 20). The active thread then iterates to the next level in the BVH (line 21).

3.3. Traversing LBVHs

Once the LBVH is built, traversing it is a trivially parallel process, since each particle may independently query the tree and record its neighbor list. The most challenging part of Algorithm 2 is to determine the next nodes to traverse or skip. One way to perform this traversal is to explicitly manage a stack of node pointers to test per thread. However, this can lead to large memory demands per thread, which can be undesirable if it reduces the device occupancy. We instead implemented a stackless traversal method based on "ropes" [23–25], in which each AABB stores the indices of its left child and a "rope" that is the index of the node that traversal should proceed to if the current node is a leaf node or it is not intersected (lines 17 and 20 in Algorithm 2). In practice, we generate the skip ropes during the hierarchy generation and AABB determination.

The ropes are shown schematically in Fig. 5, where green arrows indicate ropes to other nodes within the tree, while red arrows indicate that traversal should be terminated. If the query AABB overlaps an internal node, the traversal proceeds to the left child. If it does not overlap, then traversal proceeds to the next node along the rope. The rope is always followed for a leaf node regardless of overlap. For example, if the query AABB overlaps internal node 0, traversal proceeds to internal node 1. If it does not overlap internal node 1, then leaf nodes 0 and 1 can be skipped, and traversal proceeds along the rope to leaf node 2. After leaf node 2 is processed, traversal of this tree is complete.

We found that memory access patterns significantly affected the traversal speed. The AABB data was aligned in two four element structures containing the lower and upper bound coordinates of the AABB, the traversal rope, and either the left child for an internal node or the number of particles in a leaf node. With this layout, our implemented rope algorithm is advantageous compared to other proposed stackless traversal algorithms [26–28] because its memory accesses are minimal and fully coalesced. Similarly to the stenciled cell list, the neighbor list is constructed from the sorted particle order so that threads within a warp traverse similar parts of the tree. We have not implemented the LBVH traversal using multiple threads per particle because Algorithm 2 does not lend itself as naturally to strided access as Algorithm 1. Such an optimization is left as possible future work.

3.4. Memory management

The neighbor list memory layout affects the speed of the subsequent pair force calculation. In previous versions of HOOMD-blue, the neighbor list was stored in an effectively two-dimensional row-major matrix with the particle indexes as columns and the neighbors as rows [7]. This memory layout demands that the number of rows be equal to the maximum number of neighbors for any particle. Although this works well when all particles have roughly the same number of neighbors, it presents problems in size-asymmetric systems when a minority of particles has many more neighbors than the majority. The number of neighbors for a particle scales as r_{ij}^3. In a system with a cutoff radius disparity of 10:1, the large particles have roughly 1000 times more neighbors than the small particles. This in turn leads to intense memory demands that can easily exceed the typical capacities of GPUs, even if there is only a single large particle.

To remedy this problem, we manage the neighbor list memory in an alternative fashion that significantly reduces memory requirements. We track the maximum number of neighbors per particle type, and allocate the memory for a one-dimensional array holding the maximum total number of neighbors for the system based on the number of particles per type. We round the maximum number of neighbors per type to the next-highest multiple of 8 for improved memory alignment and less frequent resizing. Because particles are randomly ordered in memory by type, we maintain a second array that gives each particle an index into its memory in the one-dimensional neighbor list array. We compute this list of indexes in parallel using a device-wide exclusive prefix sum on an array initially filled with the number of neighbors reserved for each particle.

4. Performance

All benchmarks were performed on XSEDE [29] Comet, hosted by the San Diego Supercomputer Center, on an NVIDIA Tesla K80 GPU. HOOMD-blue was compiled using CUDA 7.0 in single precision without MPI support because this is a typical configuration for many users. The open-source CUB library [30] (version 1.4.1) is embedded in HOOMD-blue for the radix sort.

4.1. Lennard-Jones binary mixture

We first performed a synthetic benchmark of a Lennard-Jones binary mixture of particles that differ solely in their cutoff radius. Because the Lennard-Jones potential decays quickly with radial distance, the local density of the bulk liquid is essentially unchanged by expanding the cutoff beyond a sufficiently long length. This means that varying the cutoff systematically varies the number of neighbors to compute while the particle configurations are mostly unperturbed. This benchmark can then systematically probe the performance of the neighbor list algorithms as a function of both size disparity and composition.

We first equilibrated a Lennard-Jones liquid of 192,000 particles (all marked type "A") at $T = 1.5$ in a cubic box of edge length $L = 72$ with all quantities defined in the standard reduced Lennard-Jones units. We used a cutoff of $r_{AA} = 3.0$ between all particles during the equilibration. Integration was performed with particles coupled to a Langevin heat bath using a simulation timestep $\Delta t = 0.005$ and damping factor $\gamma = 1.0$. After equilibration, we marked a certain number fraction x_B as "B" particles with a large cutoff radius r_{BB} between them, using arithmetic mixing rules for the cross interactions.

We then performed short benchmark runs in the NVE ensemble using the standard cell list, stenciled cell list, and LBVHs for building the neighbor list. The neighbor list buffer width was set to 0.0 to force a rebuild at every simulation step. For the stenciled cell list, we performed most benchmarks with $\Delta_{bin} = r_{AA}$. We performed 1000 warm-up steps to determine optimal kernel launch parameters for each algorithm, and then profiled the time required to construct the neighbor list during 10,000 simulation steps. We varied r_{BB} from 3.0 to 18.0, choosing values so that an integer number of cells covered the simulation box. We then varied x_B from 10% to 50%. The benchmarked absolute neighbor list build times are reported as supplementary data (see Appendix A).

We expected and confirmed that the stenciled cell list performance was essentially indistinguishable from the standard cell list for the single component Lennard-Jones liquid ($x_B = 0.0$) with $\Delta_{bin} = r_{AA}$. For other values of x_B and r_{BB}, we scanned over a range of nominal cell widths Δ_{bin}, and found that $\Delta_{bin} = r_{AA}$ gave the optimal performance in almost all systems. This bin width gave neighbor list build times that were between 50% and 100% faster than with $\Delta_{bin} = r_{AA}/2$. For the particles of type A, setting $\Delta_{bin} = r_{AA}/2$ reduced the number of distance checks by about 40% from $\Delta_{bin} = r_{AA}$; however, nearly five times more cells were accessed. The overhead of these additional cell accesses evidently outweighs the benefits of choosing a smaller Δ_{bin} in our GPU implementation. Accordingly, we set $\Delta_{bin} = r_{AA}$ for all subsequent results.

The authors of Ref. [10] describe an additional optimization for the stenciled cell list to further reduce the number of distance checks. In their CPU algorithm, the maximum distance between cells in the stencil is also computed in addition to the minimum distance. If the maximum distance is shorter than the cutoff, the particle can be included without distance check. We implemented this on the GPU with a single thread and multiple threads per particle, but found that it led to comparable or increased neighbor list build times in our benchmarks, and so chose not to include it in our final implementation.

For this synthetic benchmark, the most important measurement is the relative speed of the three neighbor list algorithms. Fig. 6 compares the speedup of the stenciled cell list and the LBVH neighbor list build times versus the standard cell list for different concentrations and cutoffs. Here, a speedup greater than one indicates that the stenciled cell list or LBVH is faster than the standard cell list. For $r_{BB} < 6.0$, the stenciled cell list performs comparably to the standard cell list, while the LBVH is slower. This behavior is expected because the A and B interaction lengths are nearly the same, and it is faster to build and search the cell list than to build and traverse the LBVH. As r_{BB} is increased, both the stenciled cell list and the LBVHs considerably outperform the simple cell list. We note that this speedup is more pronounced for systems with smaller x_B for both algorithms.

Fig. 7 compares the speedup in the neighbor list build time between the LBVHs and the stenciled cell list. Here, a speedup greater than one indicates that LBVH is the faster algorithm. For $r_{BB} < 6.0$, the stenciled cell list typically performs better than the LBVH, as does the simple cell list. The LBVH outperforms the stenciled cell list for larger asymmetries $r_{BB} > 6.0$ and low concentrations $x_B < 0.3$. This behavior is in agreement with our expectation that trees outperform cell lists for non-uniform or sparse systems. For many colloidal systems, the number density of large particles is typically quite low, so in these systems we might expect LBVHs to outperform the stenciled cell list.

4.2. Colloid solution

We benchmarked the stenciled cell list and LBVHs for a representative colloidal system, the colloid solution described in Ref. [10]. We fixed the solvent density at $\rho = 0.6$ and varied the colloid diameter a and the colloid volume fraction $\phi = \pi a^3 \rho_C/6$ where ρ_C is the colloid number density. The solvent–solvent interactions were modeled with a standard Lennard-Jones potential

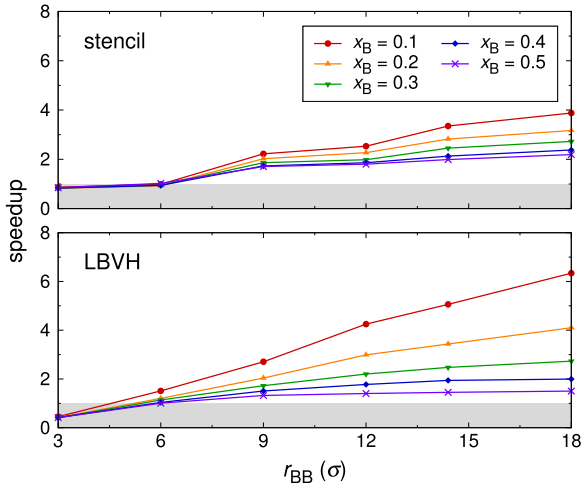

Fig. 6. Speedup in the stenciled cell list (*top*) and LBVH (*bottom*) neighbor list build time compared to a standard cell list for the Lennard-Jones binary mixture with varying largest cutoff r_{BB} and composition x_B.

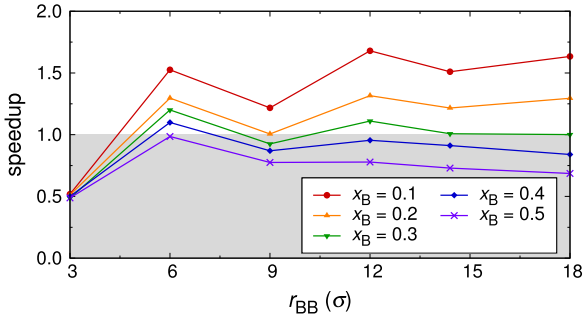

Fig. 7. Speedup in the LBVH neighbor list build time compared to a stenciled cell list for the Lennard-Jones binary mixture of Fig. 6.

with cutoff $r_{SS} = 3.0$. The colloid–colloid and colloid–solvent interactions were modeled through integrated Lennard-Jones potentials [31] using the same parameterization as Ref. [10] with the colloid–colloid interactions truncated at $r_{CC} = 5a/2$, and the colloid–solvent interactions at $r_{CS} = a/2 + 4.0$. The maximum colloid diameter explored was $a = 20$, which led to a maximum interaction range asymmetry of 50:3.

We initially equilibrated only the Lennard-Jones solvent coupled to a Langevin heat bath in a cubic simulation box at $T = 1.0$ (simulation timestep $\Delta t = 0.005$, damping factor $\gamma = 1.0$). We then randomly dispersed N_C colloids into the simulation box, and deleted solvent particles within 5% of the colloid–solvent contact distance $(a + 1.0)/2$. The final system parameters are listed in Table 1, where N_S is the number of solvent particles after deletion. The resulting dispersion was then equilibrated using the same integration scheme and a neighbor list buffer radius of 1.0. For profiling, we performed 25,000 NVE simulation steps to determine the optimal kernel launch parameters and an additional 25,000 simulation steps for data collection using the stenciled cell list and the LBVHs. During the profiling time, the neighbor list was rebuilt between roughly 1200 and 1400 times.

Fig. 8 shows the profiling results for three colloid diameters $a = 5$, 10, and 20 at volume fractions $\phi = 0.1$ and 0.2. Here, the solid shaded areas indicate the time required to build the neighbor list while the cross-hatched areas indicate the total simulation time to complete 25,000 steps. In all cases, the total run time is shorter for $\phi = 0.2$ than $\phi = 0.1$. This is due to the larger amount of colloid excluded volume, which decreases the number of solvent particles in the simulation.

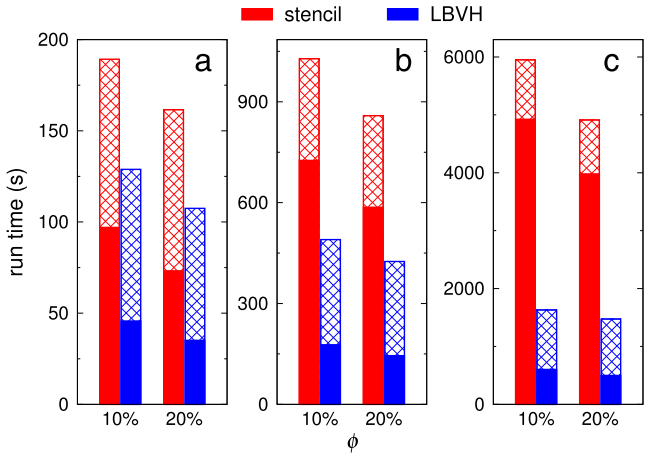

Fig. 8. Colloid solution neighbor list build time (solid) and total simulation time (cross-hatched) for 25,000 simulation steps using stenciled cell list ($\Delta_{bin} = r_{SS}$) and LBVHs at varied colloid diameters (a) $a = 5$, (b) $a = 10$, and (c) $a = 20$ and volume fractions ϕ. See Table 1 for benchmark parameters and relative speedups.

Table 1 reports the speedups in the neighbor list build time and total run time. We observe that for all systems the LBVHs outperform the stenciled cell list, with a maximum speedup in total run time of 3.6×. It is clear that as the colloid diameter increases, the speedup of the LBVHs versus the stenciled cell list increases, which is consistent with our expectation that LBVHs are more favorable for decreasing number density from the synthetic Lennard-Jones benchmark. For the stenciled cell list, the neighbor list calculation typically accounted for roughly 50% of the simulation time, so the actual speedup in the neighbor list calculation time for the LBVHs was between 2.1× and 8.2×. Although explicit timings are not reported here, we found that both algorithms clearly outperformed the standard cell list as in Ref. [10].

5. Conclusions

We developed a parallel algorithm for computing neighbor lists based on linear bounding volume hierarchies on GPUs. We compared this algorithm to a GPU implementation of an established CPU algorithm based on stenciled cell lists. We found that both the stenciled cell list and LBVH algorithms outperform a standard cell list for a synthetic benchmark with asymmetric interaction ranges, and that the highest speedups relative to a standard cell list are obtained when the fraction of particles with large cutoffs is small. These benchmarks also revealed that LBVHs outperform the stenciled cell list for large asymmetries and low number fractions of large particles. We confirmed this in a realistic colloidal system where the LBVHs consistently outperformed the stenciled cell list.

We have focused our discussion on the GPU implementation of the LBVH algorithm, but we emphasize that a BVH neighbor-list algorithm is also applicable to CPU codes. We observed sizable speedups in the neighbor list build time using CPU BVHs compared to a CPU standard cell list in HOOMD-blue as size asymmetry increased (not reported in the present article). Both the CPU and GPU BVH neighbor list implementations are available as open-source software in HOOMD-blue v. 1.3, and can be downloaded from the HOOMD-blue webpage [8].

The LBVH neighbor-list algorithm described here has numerous applications in soft matter simulations, especially when the system of interest has components with disparate length scales. These length scales may be geometric constraints, as is the case for colloidal particles dispersed in an atomistic solvent, or due to disparate interaction length scales. As in graphics processing,

Table 1
Colloid solution benchmark parameters and speedups for LBVH versus stenciled cell list.

a	ϕ	N_C	N_S	L	neighbor list speedup	total speedup
5	0.1	500	158,517	68.9	2.1×	1.5×
	0.2	1000	123,267		2.1×	1.5×
10	0.1	250	665,540	109.4	4.1×	2.1×
	0.2	500	548,740		4.1×	2.0×
20	0.1	100	2,176,957	161.2	8.2×	3.6×
	0.2	200	1,841,931		8.0×	3.3×

LBVH neighbor lists may also find useful applications in systems with uniform length scales but spatially non-uniform particle distributions, as is often the case for studying systems with interfaces or particle clustering.

Although this work has focused on the application of LBVHs to general-purpose molecular simulations, other tree data structures, including octrees and k-d trees, are also likely excellent candidates for neighbor list construction. Higher quality LBVHs may also provide better performance for certain problems [20]. Moreover, hybrid approaches that combine cell lists and hierarchical trees are also known to significantly accelerate graphics processing [11]. These algorithms incur different costs for tree construction and traversal [14], and should be carefully compared to each other in simulation applications. Such alternative hierarchical algorithms are promising avenues for future work.

Acknowledgments

MPH, AN, and AZP acknowledge financial support from the Princeton Center for Complex Materials (PCCM), a U.S. National Science Foundation Materials Research Science and Engineering Center (awards DMR-0819860 and DMR-1420541). MPH received Government support under contract FA9550-11-C-0028 awarded by the Department of Defense, Air Force Office of Scientific Research, National Defense Science and Engineering Graduate (NDSEG) Fellowship, 32 CFR 168a. JAA and SCG acknowledge support by the National Science Foundation, Division of Materials Research, award DMR-1409620 and by NVIDIA Corp. under their NVIDIA GPU Research Center Awards program. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation Grant Number ACI-1053575.

Author contributions. MPH, JAA, and AN developed the algorithm. MPH implemented the algorithms and ran performance benchmarks. SCG advised research at the University of Michigan, and AZP advised research at Princeton University. All authors contributed to writing the article.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at http://dx.doi.org/10.1016/j.cpc.2016.02.003.

References

[1] TOP500 Supercomputing Sites, http://www.top500.org (Nov. 2014).
[2] D.C. Rapaport, Multi-million particle molecular dynamics I. Design considerations for vector processing, Comput. Phys. Comm. 62 (1991) 198–216.
[3] D.E. Shaw, R.O. Dror, J.K. Salmon, J.P. Grossman, K.M. Mackenzie, J.A. Bank, C. Young, M.M. Deneroff, B. Batson, K.J. Bowers, E. Chow, M.P. Eastwood, D.J. Ierardi, J.L. Klepeis, J.S. Kuskin, R.H. Larson, K. Lindorff-Larsen, P. Maragakis, M.A. Moraes, S. Piana, Y. Shan, B. Towles, Millisecond-Scale molecular dynamics simulations on anton, in: Proceedings of Conference on High Performance Computing Networking, Storage, Analysis, vol 39, 2009, pp. 1–11.
[4] B. Hess, C. Kutzner, D. van der Spoel, E. Lindahl, GROMACS 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation, J. Chem. Theory Comput. 4 (3) (2008) 435–447.
[5] S. Plimpton, Fast parallel algorithms for short-range molecular dynamics, J. Comput. Phys. 117 (1995) 1–19.
[6] J.C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R.D. Skeel, L. Kale, K. Schulten, Scalable molecular dynamics with NAMD, J. Comput. Chem. 26 (16) (2005) 1781–1802.
[7] J.A. Anderson, C.D. Lorenz, A. Travesset, General purpose molecular dynamics simulations fully implemented on graphics processing units, J. Comput. Phys. 227 (10) (2008) 5342–5359.
[8] http://codeblue.umich.edu/hoomd-blue.
[9] D. Frenkel, B. Smit, Understanding Molecular Simulation, second ed., Academic Press, San Diego, 2002.
[10] P.J. in't Veld, S.J. Plimpton, G.S. Grest, Accurate and efficient methods for modeling colloidal mixtures in an explicit solvent using molecular dynamics, Comput. Phys. Comm. 179 (5) (2008) 320–329.
[11] C. Ericson, Real-Time Collision Detection, Elsevier Science, New York, 2004.
[12] S. Artemova, S. Grudinin, S. Redon, A comparison of neighbor search algorithms for large rigid molecules, J. Comput. Chem. 32 (13) (2011) 2865–2877.
[13] S. Grudinin, S. Redon, Practical modeling of molecular systems with symmetries, J. Comput. Chem. 31 (9) (2010) 1799–1814.
[14] A.L. dos Santos, V. Teichrieb, J. Lindoso, Review and comparative study of ray traversal algorithms on a modern GPU architecture, in: 22nd International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision, 2014, pp. 203–212.
[15] M. Vinkler, V. Havran, J. Bittner, Bounding volume hierarchies versus Kd-trees on contemporary many-core architectures, in: Proceedings of the 30th Spring Conference on Computer Graphics, 2014, pp. 29–36.
[16] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, D. Manocha, Fast BVH Construction on GPUs, Comput. Graphics Forum 28 (2) (2009) 375–384.
[17] J. Pantaleoni, D. Luebke, HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry, in: Proceedings of the Conference on High Performance Graphics, 2010, pp. 87–95.
[18] K. Garanzha, J. Pantaleoni, D. McAllister, Simpler and faster HLBVH with work queues, in: Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, 2011, pp. 59–64.
[19] T. Karras, Maximizing parallelism in the construction of BVHs, octrees, and k-d trees, in: Proceedings of the Fourth ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics, 2012, pp. 33–37.
[20] T. Karras, T. Aila, Fast parallel construction of high-quality bounding volume hierarchies, in: Proceedings of the 5th High-Performance Graphics Conference, 2013, pp. 89–99.
[21] J. Glaser, T.D. Nguyen, J.A. Anderson, P. Lui, F. Spiga, J.A. Millan, D.C. Morse, S.C. Glotzer, Strong scaling of general-purpose molecular dynamics simulations on GPUs, Comput. Phys. Comm. 192 (2015) 97–107.
[22] G.M. Morton, A Computer Oriented Geodetic Data Base; and a New Technique in File Sequencing, Tech. Rep, IBM, 1966.
[23] J.D. MacDonald, K.S. Booth, Heuristics for ray tracing using space subdivision, Visual Comput. 6 (1990) 153–166.
[24] B. Smits, Efficiency issues for ray tracing, J. Graph. Tools 3 (2) (1998) 1–14.
[25] R. Torres, P.J. Martín, A. Gavilanes, Ray casting using a roped BVH with CUDA, in: Proceedings of the 25th Spring Conference on Computer Graphics, 2009, pp. 95–102.
[26] A.T. Áfra, L. Szirmay-Kalos, Stackless multi-BVH traversal for CPU, MIC and GPU ray tracing, Comput. Graphics Forum 33 (1) (2013) 129–140.
[27] M. Hapala, T. Davidovič, I. Wald, V. Havran, P. Slusallek, Efficient Stack-less BVH traversal for ray tracing, in: Proceedings of the 27th Spring Conference on Computer Graphics, 2011, pp. 7–12.
[28] R. Barringer, T. Akenine-Möller, Dynamic stackless binary tree traversal, J. Comput. Graph. Tech. 2 (1) (2013) 38–49.
[29] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G.D. Peterson, R. Roskies, J.R. Scott, N. Wilkens-Diehr, XSEDE: Accelerating scientific discovery, Comput. Sci. Eng. 16 (5) (2014) 62–74.
[30] http://nvlabs.github.io/cub.
[31] G.S. Grest, Q. Wang, P. in't Veld, D.J. Keffer, Effective potentials between nanoparticles in suspension, J. Chem. Phys. 134 (14) (2011) 144902.