Accelerating dissipative particle dynamics simulations on GPUs: Algorithms, numerics and applications

Yu-Hang Tang, George Em Karniadakis
Division of Applied Mathematics, Brown University, Providence, RI, USA

Abstract

We present a scalable dissipative particle dynamics simulation code, fully implemented on the Graphics Processing Units (GPUs) using a hybrid CUDA/MPI programming model, which achieves 10–30 times speedup on a single GPU over 16 CPU cores and almost linear weak scaling across a thousand nodes. A unified framework is developed within which the efficient generation of the neighbor list and maintaining particle data locality are addressed. Our algorithm generates strictly ordered neighbor lists in parallel, while the construction is deterministic and makes no use of atomic operations or sorting. Such neighbor list leads to optimal data loading efficiency when combined with a two-level particle reordering scheme. A faster in situ generation scheme for Gaussian random numbers is proposed using precomputed binary signatures. We designed custom transcendental functions that are fast and accurate for evaluating the pairwise interaction. The correctness and accuracy of the code is verified through a set of test cases simulating Poiseuille flow and spontaneous vesicle formation. Computer benchmarks demonstrate the speedup of our implementation over the CPU implementation as well as strong and weak scalability. A large-scale simulation of spontaneous vesicle formation consisting of 128 million particles was conducted to further illustrate the practicality of our code in real-world applications.

Program summary

Program title: GPU-accelerated DPD Package for LAMMPS
Catalogue identifier: AETN_v1_0
Program summary URL: http://cpc.cs.qub.ac.uk/summaries/AETN_v1_0.html
Program obtainable from: CPC Program Library, Queen’s University, Belfast, N. Ireland
Licensing provisions: GNU General Public License, version 3
No. of lines in distributed program, including test data, etc.: 1 602 716
No. of bytes in distributed program, including test data, etc.: 26 489 166
Distribution format: tar.gz
Programming language: C/C++, CUDA C/C++, MPI.
Computer: Any computers having nVidia GPGPUs with compute capability 3.0.
Operating system: Linux.
Has the code been vectorized or parallelized?: Yes. Number of processors used: 1024 16-core CPUs and 1024 GPUs
RAM: 500 Mbytes host memory, 2 Gbytes device memory
Supplementary material: The data for the examples discussed in the manuscript is available for download.
Classification: 6.5, 12, 16.1, 16.11.

(*) This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).
(*) Corresponding author. Tel.: +1 401 863 1217.
E-mail addresses: yuhang_tang@brown.edu (Y.-H. Tang), gk@dam.brown.edu (G.E. Karniadakis).
Nature of problem: Particle-based simulation of mesoscale systems involving nano/micro-fluids, polymers and spontaneous self-assembly process.

Solution method: The system is approximated by a number of coarse-grained particles interacting through pairwise potentials and bonded potentials. Classical mechanics is assumed following Newton’s laws. The evolution of the system is integrated using a time-stepping scheme such as Velocity-Verlet.

Restrictions: The code runs only on CUDA GPGPUs with compute capability 3.0.

Unusual features: Fully implemented on GPGPUs with significant speedup.

Running time: 78 h using 1024 GPGPUs for simulating a 128-million-particle system for 18.4 million time steps. © 2014 Elsevier B.V. All rights reserved.

1. Introduction

Particle-based simulation is a powerful tool in studying microscopic stochastic systems and has been popularized by the ever growing computing power provided by each new generation of hardware. Among the spectrum of parallel processors, the general purpose graphics processing unit (GPGPU) proves to be a particularly good fit for such simulation due to the massively parallel nature shared by inter-particle interaction evaluation and image rendering [1–4]. Specifically, the Compute Unified Device Architecture (CUDA) has provided a parallel programming model that could harness the processing power of the GPGPUs, which is tens of times more than the contemporary CPUs [5,6]. There is currently an array of molecular dynamics simulation applications that integrate the capability of using CUDA for part or all of the computation, such as AMBER, DL_POLY, HOOMD-BLUE, LAMMPS, NAMD and GROMACS [7,8,1,9–11].

The combination of fast hardware and highly optimized software package enables large-scale simulations of atomistic systems. For example, a protein folding simulation of 64 million particles using an all-atom resolution has been successfully carried out using NAMD for a total integration time of 100 ns [12]. Nevertheless, the classical all-atom molecular dynamics (AAMD) method is still prohibitively expensive for simulating atomistic systems at a length and time scale that is comparable to that in experiment. To bridge the gap between the microscopic time/length scale and the macroscopic ones, coarse-grained simulation techniques such as coarse-grained molecular dynamics (CGMD), dissipative particle dynamics (DPD) and smooth particle hydrodynamics (SPH) have been proposed in which a group of atoms in the AAMD model is represented by a single coarse-grained particle to reduce the computational complexity [13–17].

Among the various coarse-grained models, DPD has been our particular interest since it accurately reproduces the equilibrium and dynamic properties of systems in the area of biophysics, soft matter and fluid dynamics [18–20]. Unfortunately, little effort has been directed to the efficient implementation of DPD on the GPUs. Despite the intrinsic similarity in the governing principles, the DPD systems differ from their atomistic counterparts in that:

- A more complex functional form is used, i.e. a dissipative force, which involves the relative velocity in addition to the relative position of the interacting particles.

- A pairwise random force is essential to compensate for the reduced degrees of freedom in order to correctly reproduce the dynamics of the coarse-grained system.

- The system is sparse, which results in a much lower average neighbor count for the particles and makes it harder to achieve optimal performance on the GPU’s SIMD architecture.

So far, Goga et al. have presented a GPU parallelization of the SD and DPD types in the GROMACS tools [21], with which they observed a performance boost of 70% on the GPU over a single processor. Phillips et al. proposed a hash-function based pseudo-random number generator for the evaluation of the stochastic term [22]. Wu et al. reported a GPU implementation of DPD with parallel neighbor list updating [23]. Wang et al. reported a multi-GPU implementation, which achieves a speedup of 90x on three GPUs over a single-threaded CPU implementation from Material Studio [24]. However, no scalable implementation that runs across nodes has been reported so far.

In this paper, we demonstrate a scalable implementation of the DPD simulation on the latest Kepler GPGPU architecture using a CUDA–MPI hybrid programming model targeting optimal performance within a GPU and also across nodes. The paper is organized as follows: Section 2 presents a brief review of the functional forms of DPD. Section 3.1 presents an overview of the program flow, while detailed algorithms are given in Sections 3.2–3.6. Benchmarks and verifications of the code simulating simple flow and spontaneous vesicle formation are given in Section 4. In Section 5 we summarize our work.

2. Functional forms of DPD

In the DPD simulation, the movement of particles is governed by Newton’s equation of motion as [14]

$$\frac{d\mathbf{r}}{dt} = \mathbf{v}, \quad \frac{d\mathbf{v}}{dt} = \mathbf{a} = \frac{\mathbf{f}}{m}. \quad (1)$$

The force \(\mathbf{f}\) acting on each particle consists of three pairwise additive parts: a conservative component, a dissipative component and a random component, i.e.

$$\mathbf{f}_i = \sum_{i\neq j} \mathbf{F}_{ij} = \sum_{i\neq j} (\mathbf{F}_{ij}^C + \mathbf{F}_{ij}^D + \mathbf{F}_{ij}^R) \quad (2)$$

given by

$$\mathbf{F}_{ij}^C = a_{ij} w_C (r_{ij}) \mathbf{e}_i \quad (3)$$

$$\mathbf{F}_{ij}^D = -\gamma_{ij} w_D (r_{ij}) (\mathbf{v}_i \cdot \mathbf{e}_i) \mathbf{e}_i \quad (4)$$

$$\mathbf{F}_{ij}^R = \sigma_{ij} w_K (r_{ij}) \xi_{ij} \delta t^{-1} \mathbf{e}_i \quad (5)$$
Fig. 1. In the array of structure (AoS) layout, the coordinate vector for each particle is placed consecutively, whereas any specific component of the vector are separated by the other ones. In the structure of array (SoA) data layout, components are stored consecutively.

if $|\mathbf{r}_i| \leq r_c$, where $r_c$ is the cutoff distance, and

$$F^c_{ij} = F^0_{ij} = F^b_{ij} = 0, \quad |\mathbf{r}_i| > r_c.$$  \hspace{1cm} (6)

One of the two weight functions $w_D$ and $w_k$ can be chosen arbitrarily, while the other is then fixed through the fluctuation–dissipation constraint [25]

$$w_D(\mathbf{r}_i) = w_D^c(\mathbf{r}_i).$$  \hspace{1cm} (7)

In our implementation, $w_k$ is chosen to be a power of $w_c$ as

$$w_k(\mathbf{r}_i) = w_k^c(\mathbf{r}_i) = \left(1 - \frac{|\mathbf{r}_i|}{r_c}\right)^s.$$  \hspace{1cm} (8)

The exponent $s$ can be chosen arbitrarily, making it a convenient tuning parameter for reproducing the viscosity of a wide range of fluids. The coefficients $\sigma_{ij}$ and $\gamma_{ij}$ are related to each other by

$$\sigma_{ij}^2 = 2\gamma_{ij}k_BT$$  \hspace{1cm} as also dictated by the fluctuation–dissipation theorem.

3. Algorithms and implementation

3.1. Baseline architecture

We start with LAMMPS as the baseline code for the CPU portion of the program [9]. The GPU code is written in CUDA C due to the high level of availability and maturity. The basic control flow is shown in Algorithm 1 where the highlighted text indicates parts of the computation that involve the GPU.

Algorithm 1 Basic program control flow. The highlighted text indicates portions of the computation that involve the GPU.

1: \textbf{setup}
2: setup compute domain, determine particle ownership
3: CPU $\rightarrow$ GPU
4: obtain ghost particles information from neighboring processors
5: build neighbor list
6: compute forces
7: \textbf{main loop}
8: for $n$ steps do
9: time integration: phase 1
10: if time to rebuild neighbor list then
11: GPU $\rightarrow$ CPU
12: exchange stray particles
13: CPU $\rightarrow$ GPU
14: obtain ghost particles information from neighboring processors
15: rebuild neighbor list
16: else
17: exchange ghost particle information
18: end if
19: compute forces
20: time integration: phase 2
21: calculate statistics
22: if on demand then
23: display runtime info, dump trajectory, write restart file, etc.
24: end if
25: end for

force as illustrated in Fig. 1. Such layout results in strided access on the GPU, i.e. threads with consecutive thread IDs will access memory locations that are separated by a stride larger than one, and reduces the effective memory bandwidth. However, too much code modification is required if we were to change a data layout which is used essentially everywhere in the original program. As a compromise we used a pair of interleave/deinterleave kernels to convert data between the AoS and the SoA layouts at runtime. The kernels are carefully designed to achieve high memory bandwidth efficiency as summarized in Table 1.

3.2. Sorting, binning and reordering

3.2.1. Sorting

The parallel sorting primitive plays a fundamental role in our code for reordering particles and building the cell list. Several existing algorithm packages such as Thrust and CUDA Data Parallel Primitives Library (CUDA) provide high-level interfaces for this type of workload (CUDA actually uses Thrust::sort as its backend) [27,28]. However, neither package supports the CUDA streams. In other words, they can only launch kernels on the default stream 0. In our code, streaming is essential for achieving high GPU computing efficiency as it helps to hide communication and kernel launching latency. Moreover, our benchmark reveals that these packages are better optimized for large arrays with millions or billions of key/value pairs, while in practical DPD simulations the particle number per GPU does not exceed a few millions. Therefore, we implemented our own radix sort that is optimized specifically for smaller arrays. As shown in Algorithm 2, each successive invocation of three kernels GPU_radix_histogram, GPU_radix_prefix_sum and GPU_radix_permute completes one pass of the sorting algorithm, while 4 bit per pass was determined to be the optimal radix width through experimentation. The scan primitive by Sengupta et al, combined with the newly-introduced warp shuffle instructions is used to carry out the prefix summation [29]. The benchmark presented in Fig. 2 compares the
Table 1
Achieved bandwidth of the data marshaling and bandwidth-sensitive kernels. The benchmarks were done on a Kepler K20 GPU with a peak memory bandwidth of 208 GB/s.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Read (GB/s)</th>
<th>Write (GB/s)</th>
<th>Aggregate (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interleave</td>
<td>79.59</td>
<td>79.62</td>
<td>159.21</td>
</tr>
<tr>
<td>Deinterleave</td>
<td>72.21</td>
<td>70.80</td>
<td>143.01</td>
</tr>
<tr>
<td>NeighborListJoin</td>
<td>62.66</td>
<td>57.83</td>
<td>120.49</td>
</tr>
<tr>
<td>NeighborListTranspose</td>
<td>78.03</td>
<td>79.06</td>
<td>157.09</td>
</tr>
<tr>
<td>MergeXVT</td>
<td>95.78</td>
<td>54.66</td>
<td>150.44</td>
</tr>
</tbody>
</table>

Algorithm 2 Pseudo code for the radix sort template function.

```plaintext
function RadixSort<RADIX>(KeyArray, ValArray, BitLength, TargetStream)
  BufferIn ← {KeyArray, ValArray}
  BufferOut ← InternalBuffer
  for bit = 0 to BitLength by Log2(RADIX) do
    gpuRadixHistogram(BufferIn, BufferOut, PrefixBuffer, bit) on TargetStream
    gpuRadixPrefixSum(PrefixBuffer) on TargetStream
    gpuRadixPermute(BufferIn, BufferOut, PrefixBuffer, bit) on TargetStream
    SwapPointer(BufferIn, BufferOut)
  end for
  if BufferOut ≠ (KeyArray, ValArray) then
    (KeyArray, ValArray) ← cudaMemcpy(BufferOut)
  end if
end function
```

Fig. 2. Sorting time: our streamed radix sort vs. Thrust.

Fig. 3. A 2-level Morton curve is obtained by ensuring that the number of bins for reordering is a multiple of the number of bins in the cell list by some integer power of 2.

Fig. 4. Performance gain due to particle reordering for the whole program and individual kernels. This metric also indirectly reflects the memory bandwidth consumption for each kernel.

3.2.2. Particle reordering

Unlike grid/lattice-based methods, particle-based simulation deals with systems whose structures are largely irregular. As a result, the memory access pattern of typical particle simulation codes exhibits poor locality. The situation turns out to be worse on GPU due to the SIMD nature of the GPU architecture and a lower per-core bandwidth. In fact, most particle-based simulation programs are memory-bound, while DPD suffers even more because both coordinate and velocity data are required for evaluating the pairwise interaction. Particle reordering along a space-filling curve has been proposed as an effective technique to mitigate the problem. For example, a space-filling curve pack (SFCPACK) algorithm is used in the HOOMD-blue simulation package to sort particles on a per-block basis [1]. Our code, on the other hand, uses a two-level Morton encoding (Z-curve) scheme to reorder particles as shown in Fig. 3. In this scheme, cells are first sorted according to their Morton order, while at the same time particles within a cell are also sorted by their relative position along a local Morton curve in each individual cell. In addition, the boundary of the cells used for particle reordering was chosen to coincide with that used for building the cell list. The two-level Morton encoding achieves good locality while at the same time ensures particles belonging to the same cell have consecutive indices. This feature is later exploited by our neighbor list building algorithm. Note that at this point only local particles have been sorted and in order to proceed we need to obtain information of ghost particles from neighboring processors; this will be discussed in Section 3.6. Fig. 5 summarizes the performance gain brought about by this particle reordering scheme for DPD fluid simulations at four different sizes compared to those without particle reordering using the same set of parameters. The overall program performance is boosted by almost 100%, while individual GPU kernels benefit even more depending on their level of memory consumption. The effect of particle reordering is further characterized in Fig. 5, which visualizes the interaction matrices for a system of 2048 particles with and without particle reordering. While diffusion tends to randomize the distribution of interactions
over the entire domain, reordering on the other hand strongly diagonalizes the matrix and aggregates off-diagonal components into concentrated blocks.

3.2.3. Cell list

Our program adopts the conventional approach of using the neighbor list to accelerate the evaluation of non-bond pairwise interactions. In addition, a cell list is used to further facilitate the construction of the neighbor list. The cell list construction starts with binning, i.e. partitioning the simulation box into a number of equally sized cells and assigning particles into their corresponding cells. This can be done efficiently using parallel sorting with the particles’ cell indices being the input key for sorting, while the determination of the cell index depends solely on the particle’s own position and hence is embarrassingly parallel. This algorithm is employed by the GPU module of the LAMMPS package [30], and contrasts the approach in Wu’s implementation where the atomic increment operation, a potential bottleneck at high parallelism, is used to build the cell list [23]. Alternatively, since particle indices are consecutive within each cell after the reordering process, the construction of the cell list can be performed by performing particle reordering followed by detecting the cell boundary in the one-dimensional particle array without the need for further sorting of cell indices. The boundary detection is done by launching one thread per particle, checking if its cell ID differs from the particle that precedes it. In our code we adopted the latter approach.

3.3. Neighbor list construction

3.3.1. Stencil

As the first step, a coarse-grained stencil list is generated, which contains one stencil for each cell. Each stencil stores the indices of the neighboring cells surrounding its master cell. Within each stencil the cell indices are sorted according to the Morton order as illustrated in Fig. 6. Such ordering of the stencil combined with the aforementioned particle reordering technique ensures that the particle indices in each stencil are monotonically increasing. Since the building of a coarse-grained stencil need only to be done once during the entire simulation, the sorting is simply implemented using bubble sort. The coarse-grained stencil list is then expanded into a fine-grained one, storing the actual indices of particles in each cell’s neighboring cells. This is particularly useful for our DPD application because the particle density in a DPD simulation is much lower than that of an AAMD simulation. For example, a cell of volume \((r_c + \Delta r)^3\) in a DPD system contains on average of 10 particles, where \(\Delta r\) is the skin distance for building the neighbor list. The fine-grained stencil allows the neighboring particles from multiple cells to be loaded into our neighbor list builder in a coalesced manner without incurring branch divergence. Both the coarse-grained and the fine-grained stencils are stored in a row-major fashion.

3.3.2. Neighbor list

For the actual job of building the neighbor list, we invented an atomic-free algorithm that is able to generate a fully ordered neighbor list in parallel. The algorithm is deterministic, meaning that the generated neighbor list will always be the same given the same initial configuration and runtime parameter. Such deterministic approach may benefit debugging and simulation cases that require reproducibility.

The neighbor list builder works on the fine-grained stencil. Instead of the common practice of assigning one block for each cell, a warp-centric programming model, in which each cell is taken care of by a single warp, is employed to optimally balance
A branched last iteration is not significant. Stencils contain 200 particles each so the performance hit due to if the stencil length is not a multiple of the batch size. Typical divergence occurs only in the last iteration of the middle loop arrangement minimizes the all threads looping through the shared inner loop performs the actual all-over-all distance checking with thread loading one particle to each warp. In the inner loop of our neighbor list builder, each thread ballots its distance checking predication to form a bit vector nHit. Shifting nHit by warpSize - laneId toward left, where laneId ∈ [0..warpSize - 1] is the ordinal number of a thread within a warp, and then counting the number of remaining set bits in the bit vector tells a thread the number of hits made by threads with a lower laneId than itself. The insertion point is then simply determined by adding this number to the existing number of neighbors for the i particle from previous iterations. The number of hits from an iteration is then accumulated to the total neighbor count for the i particle by a single thread, in our case lane 0, after the cooperative neighbor list insertion. In contrast to the atomic-based approach, this warp vote-based approach for committing particle indices into the neighbor list is completely deterministic and only involves fast intra-warp communication. Recall that the indices in the stencil are monotonically increasing. This implies that the neighbor list generated in this way is also strictly increasing for each particle, yet no sorting was performed explicitly on the neighbor list. A careful benchmark shows that our atomic-free builder executes twice as fast on average compared to an equivalent builder using the atomic increment as shown in Table 2.

Another issue to be considered here is the layout of the neighbor table. A row-major layout assigns one consecutive line of memory for each particle, while a column-major layout, on the other hand, stores the nth neighbors of all the particles consecutively in the nth row. On the GPU we typically assign each thread a particle when evaluating the pairwise interaction, which implies that a column-major layout is more efficient because fetching such a list results in fully coalesced memory loads. However, a column-major table is much less efficient when being written to because an entire warp works on the same i particle in the inner loop in our neighbor list builder. In this case, each insertion of the j indices from a thread generates a write request of 32 bytes, among which only 4 bytes are actually useful data, resulting in a memory efficiency of merely 12.5%. To combine the advantages from both layouts, we adopt a write–transpose–read approach in which the neighbor table is built in a row–major fashion, and then transposed for maximum loading efficiency in the subsequent computation. In practice, we only locally in-place transpose each 32 by 32 tile of the neighbor table matrix. We made this compromise because a full transposition of such a large matrix is complicated with suboptimal efficiency [31]. A schematic representation of the local transposition is presented in Fig. 8.

Further, a double-insertion trick can be utilized by the neighbor list builder to help suppress branch divergence in the force kernel. This heuristic depends on the assumption that particles which were within the actual cutoff distance when building the neighbor list are more likely to be found within the cutoff distance again in subsequent time steps than those which initially lie between the cutoff and the skin distance. To separate the two types of neighbors in the neighbor list, the warps in our neighbor list builder can be modified to perform two commits per inner loop iteration, one for the core particles, and the other one for the skin particles. The core particle indices are stored in the original order, while the skin particle indices were stored at the back of each row in reversed order. The neighbor count integer used in the shared slots in the neighbor list builder is also split into two short integers correspondingly. In this way we still need only one table for storing the neighbor list. A separate kernel is used to join the two

---

**Table 2**

Performance comparison of the neighbor list builders using warp ballot or atomics increment.

<table>
<thead>
<tr>
<th>System config</th>
<th>Time (ms) Ballot</th>
<th>Time (ms) Atomic</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>ρ = 3, n = 64k</td>
<td>0.76</td>
<td>1.55</td>
<td>2.02</td>
</tr>
<tr>
<td>ρ = 3, n = 512k</td>
<td>5.28</td>
<td>10.77</td>
<td>2.04</td>
</tr>
<tr>
<td>ρ = 6, n = 64k</td>
<td>1.31</td>
<td>2.92</td>
<td>2.24</td>
</tr>
<tr>
<td>ρ = 6, n = 512k</td>
<td>9.89</td>
<td>21.72</td>
<td>2.20</td>
</tr>
<tr>
<td>ρ = 50, n = 64k</td>
<td>14.58</td>
<td>30.15</td>
<td>2.07</td>
</tr>
<tr>
<td>ρ = 50, n = 512k</td>
<td>90.28</td>
<td>196.26</td>
<td>2.17</td>
</tr>
</tbody>
</table>

---

As illustrated in Fig. 7, each warp in our neighbor list builder works in a triple loop. The outer loop walks through the i particles in its working cell with a batch size of 32, with each thread loading one particle’s index and coordinate into a slot in the shared memory. The middle loop scans over the cell’s stencil with each thread loading one j particle into its private registers at a time. The inner loop performs the actual all-over-all distance checking with all threads looping through the shared j particles together. Such loop arrangement minimizes the long tail effect because branch divergence occurs only in the last iteration of the middle loop if the stencil length is not a multiple of the batch size. Typical stencils contain 200 particles each so the performance hit due to a branched last iteration is not significant.

Because multiple j particles may be found staying within the cutoff distance of one i particle at the same time, a commit of the j index into the correct position of the neighbor list intuitively requires the use of atomic increment operation to resolve potential conflict. This can also be thought of as each thread needs the global hit/miss information from other threads to determine the insertion point for its own commit. However, realizing that in our algorithm at any given time there is only one warp evaluating the neighbors for a given particle, we can reduce the hard problem of global information gathering into a simpler one of collecting a boolean value from a warp. The warp vote function, __ballot(), fits naturally for this job by enabling fast communication between the threads and eliminating the use of atomic instructions. The ballot intrinsics takes a predicate value from each active threads of a warp and returns to each thread an unsigned integer whose Nth bit is set if the predicate evaluates to true for the Nth thread. In the inner loop of our neighbor list builder, each thread ballots its distance checking predication to form a bit vector nHit. Shifting nHit by warpSize - laneId toward left, where laneId ∈ [0..warpSize - 1] is the ordinal number of a thread within a warp, and then counting the number of remaining set bits in the bit vector tells a thread the number of hits made by threads with a lower laneId than itself. The insertion point is then simply determined by adding this number to the existing number of neighbors for the i particle from previous iterations. The number of hits from an iteration is then accumulated to the total neighbor count for the i particle by a single thread, in our case lane 0, after the cooperative neighbor list insertion. In contrast to the atomic-based approach, this warp vote-based approach for committing particle indices into the neighbor list is completely deterministic and only involves fast intra-warp communication. Recall that the indices in the stencil are monotonically increasing. This implies that the neighbor list generated in this way is also strictly increasing for each particle, yet no sorting was performed explicitly on the neighbor list. A careful benchmark shows that our atomic-free builder executes twice as fast on average compared to an equivalent builder using the atomic increment as shown in Table 2.

Another issue to be considered here is the layout of the neighbor table. A row-major layout assigns one consecutive line of memory for each particle, while a column-major layout, on the other hand, stores the nth neighbors of all the particles consecutively in the nth row. On the GPU we typically assign each thread a particle when evaluating the pairwise interaction, which implies that a column-major layout is more efficient because fetching such a list results in fully coalesced memory loads. However, a column-major table is much less efficient when being written to because an entire warp works on the same i particle in the inner loop in our neighbor list builder. In this case, each insertion of the j indices from a thread generates a write request of 32 bytes, among which only 4 bytes are actually useful data, resulting in a memory efficiency of merely 12.5%. To combine the advantages from both layouts, we adopt a write–transpose–read approach in which the neighbor table is built in a row–major fashion, and then transposed for maximum loading efficiency in the subsequent computation. In practice, we only locally in-place transpose each 32 by 32 tile of the neighbor table matrix. We made this compromise because a full transposition of such a large matrix is complicated with suboptimal efficiency [31]. A schematic representation of the local transposition is presented in Fig. 8.

Further, a double-insertion trick can be utilized by the neighbor list builder to help suppress branch divergence in the force kernel. This heuristic depends on the assumption that particles which were within the actual cutoff distance when building the neighbor list are more likely to be found within the cutoff distance again in subsequent time steps than those which initially lie between the cutoff and the skin distance. To separate the two types of neighbors in the neighbor list, the warps in our neighbor list builder can be modified to perform two commits per inner loop iteration, one for the core particles, and the other one for the skin particles. The core particle indices are stored in the original order, while the skin particle indices were stored at the back of each row in reversed order. The neighbor count integer used in the shared slots in the neighbor list builder is also split into two short integers correspondingly. In this way we still need only one table for storing the neighbor list. A separate kernel is used to join the two

---

**Fig. 7.** A graphical visualization of the triple loop in the atomics-free neighbor list builder.
Algorithm 3 Atomics-free neighbor list builder

1: function \texttt{GPUBuildNeighborList} \\
2: \hspace{1em} \textbf{for} each cell in parallel \textbf{do} \\
3: \hspace{2em} \texttt{shared} slot[warpSize] \\
4: \hspace{2em} \texttt{l \leftarrow threadIdx.x \mod warpSize} \\
5: \hspace{2em} \texttt{n \leftarrow number of particles in the cell} \\
6: \hspace{2em} \texttt{p \leftarrow 0} \\
7: \hspace{2em} // load working cell \\
8: \hspace{2em} \texttt{while} \ p < n \ \texttt{do} \\
9: \hspace{3em} \texttt{part \leftarrow Min(n - p, warpSize)} \\
10: \hspace{3em} \texttt{if} \ l < part \ \texttt{then} \\
11: \hspace{4em} \texttt{slot[i].l \leftarrow cell[p + 1]} \\
12: \hspace{4em} \texttt{slot[i].r \leftarrow \text{TEXTURE(coordinate, slot[i].i)}} \\
13: \hspace{4em} \texttt{slot[i].nn \leftarrow 0} \\
14: \hspace{3em} \texttt{end if} \\
15: \hspace{3em} // load stencil \\
16: \hspace{3em} \texttt{for} \ k = l \ \texttt{to nStencil} \ \texttt{by warpSize} \ \texttt{in parallel} \ \texttt{do} \\
17: \hspace{4em} \texttt{j \leftarrow stencil[k]} \\
18: \hspace{4em} \texttt{r \leftarrow \text{TEXTURE(coordinate, j)}} \\
19: \hspace{4em} // distance check \\
20: \hspace{4em} \texttt{for} \ i = 0 \ \texttt{to part by 1} \ \texttt{do} \\
21: \hspace{5em} \texttt{dr^2 \leftarrow (r - slot[i].r)^2} \\
22: \hspace{5em} \texttt{hit \leftarrow dr^2 < r^2 \ ? \ \text{true} : \text{false}} \\
23: \hspace{5em} \texttt{nHit \leftarrow \text{BALLOT(hit)}} \\
24: \hspace{5em} \texttt{nAhead \leftarrow \text{SHIIFTLEFT(nHit, warpSize - 1)}} \\
25: \hspace{5em} \texttt{plns \leftarrow slot[i].nn + \text{POPC}(nAhead)}} \\
26: \hspace{5em} \texttt{if} \ hit \ \texttt{then} \\
27: \hspace{6em} \texttt{neighborList[ slot[i].i \ || \ plns ] \leftarrow j} \\
28: \hspace{5em} \texttt{end if} \\
29: \hspace{5em} \texttt{accumulate \ POPC(nHit) to \ slot[i].nn by \ lane 0} \\
30: \hspace{4em} \texttt{end for} \\
31: \hspace{4em} \texttt{if} \ l < part \ \texttt{then} \\
32: \hspace{5em} \texttt{store \ slot[i].nn to \ global \ storage} \\
33: \hspace{4em} \texttt{end if} \\
34: \hspace{2em} \texttt{p \leftarrow p + warpSize} \\
35: \hspace{2em} \texttt{end while} \\
36: \hspace{2em} \texttt{end for} \\
37: \hspace{1em} \texttt{end function} \\

parts together, making the splitting trick completely transparent to the rest of the code. Note that the naive way of simply forcing every thread to enter the force evaluation branch regardless of the actual distance results in worse performance, since it issues more stress on the memory bandwidth and the texture units. It is even erroneous in DPD since the functional form does not accept $|r_{ij}|$ greater than $r_c$.

3.4. Precision model

We use a hybrid precision model for the neighbor list builder and force evaluation kernel to optimize GPU memory bandwidth usage. Force evaluation in DPD is expensive in terms of both memory bandwidth consumption and arithmetic complexity. The memory bandwidth consumption is high because both the coordinate and the velocity of the interacting particles are involved in the functional envelope. For each pair of interacting particles an input of 2 integer and 6 fp64 values, or a total of 56 bytes, is required to supply the required information. We used the standard approach of texture data mapping to alleviate the problem. Note that the coordinate and velocity are stored in double precision in our code everywhere else, but they are converted to single precision before being mapped as textures for force evaluation. Specifically, the single-precision coordinates are cast from the double-precision difference between the actual particle coordinates and the geometric center of the owning processor’s compute domain. The double-precision velocity is cast to single precision directly. Our approach differs from that of Götz et al. [32] in that the computation is still done in full double precision. The neighbor list builder works entirely in single precision because the distance comparison is not sensitive to precision. The possibility of using the newly-introduced non-coherent cache pathway to load such data was also explored. However, a microbenchmark revealed that on the current Kepler GPU the latency for a cache hit in the non-coherent cache (140 cycles) is much longer than that of a texture fetch (108 cycles), though is still shorter than that of a regular global memory access (220 cycles) [33].

3.5. Numerical optimization

The arithmetic complexity of the DPD pairwise interaction is high for two reasons. First, for each pair of the random force $F_R^{ij}$, one Gaussian random number has to be consumed. This translates into one square root, one natural logarithm, one trigonometric function and the generation of two uniform random numbers assuming the use of the Box–Muller algorithm [34]. Second, the power function is needed to evaluate the weight function $w_S(r_{ij}) = w_S^2(r_{ij}) = (1 - r_{ij}/r_c)^s$ if the exponent $s$ takes a value other than positive integers.

Using a separate RNG library to generate the random numbers beforehand and storing them in double precision requires a storage
space that is twice the size of the neighbor list. Besides, it is inevitable for pairs of interacting particles to get separated onto different processors during domain partitioning. In this case, we have to reproduce the same random number on different processors to save the communication. Generally, communication is expensive for the GPU so it is preferred if the random numbers can be saved during onesimulation of 65,536 DPD particles and compared to that generated by MATLAB.

The quality of the random numbers generated by TEA increases with the number of iterative rounds performed. Due to the limited sampling range of the input seeds in DPD (most simulations contain no more than a few million particles), at least 8 rounds of hashing are required to obtain sufficient randomness. We took a different approach of sampling from a wider source of entropy for the input seeds to guarantee randomness rather than simply increasing the number of iterative rounds. In particular, for each particle we take the bit-reversed particle identity tag as the first integer, and an interleaving of the first 11 bits of the mantissa of the three-dimensional velocity components as the second integer. Besides, we employ a pre-processing step to blend the two integers using 16 rounds of the TEA hashing. The two integer outputs from this preprocessing step are then combined using bitwise exclusive or into a single 32-bit binary signature for every particle. The overhead of such preprocessing is negligible since it is embarrassingly parallel with complexity $O(N)$. Only 4 rounds of TEA hashing are needed when the preprocessed binary signatures are used. In order to validate this approach, we employed the same technique as described previously. However, there are facts that we can exploit to accelerate the math functions. First of all, we have perfect control over the range of input for the functions, i.e. the uniform random numbers are always between 0 and 1. In addition, even though we are producing double precision floating points as the final output, it does not necessarily mean that we need to compute in the full precision during the intermediate steps. By looking at the Box–Muller equation $z_1 = \sqrt{-2 \ln u_1 \cos(2 \pi u_2)}$, it is clear that the radial component of the number depends only on $u_1$, while the phase component depends only on $u_2$, both of which contain only 32 bits of entropy as specified by the TEA algorithm. It is through the combination of the two parts that we obtain a result of full double precision.

Hence, we implement our own reduced-precision natural logarithm and cosine as follows:

(a) Instead of converting the 32-bit unsigned integer $v_1$ into a double precision floating point number $u_1 \in [0, 1)$ and then performing the natural logarithm, we apply a binary logarithm directly to $v_1$. The evaluation can be split into the integral part and the fractional part as $\log_2 v_1 = \log_2 2^{I+F} = I + \log_2 2^F$, where $I$ is an integer and $F$ is a fractional number between 0 and 1. The integer part $F$ can be obtained trivially using the relation $I = 31 - clz(v_1)$, where $clz$ stands for consecutive leading zeros, which is a hardware-implemented PTX instruction on the Kepler GPU. The fractional part is approximated with an 8th order Chebyshev polynomial as $\log_2 x = z \times P(z)$, where $z = \frac{x^{1/2} - 1}{x^{1/2} + 1}$ [37]. The natural logarithm is derived by subtracting the binary logarithmic result by 32 and then multiplying by ln 2. The implementation compiles into 47 branch-free PTX instructions, which contrasts the CUDA native implementation that compiles into 89 PTX instructions with 5 conditional branches and one floating point reciprocal. The maximum relative error for our implementation is $4.21 \times 10^{-12}$ over the entire range of unsigned integer, yielding a binary precision of 42.8 bits.

(b) In the custom cosine function, the highest bit of $v_2$ is taken as a sign bit, while the low 31 bits are converted into a double precision floating point number $u_2 = v_2[0:30] \times 2^{-31} \in [0, 1)$. This simplifies the range reduction for the evaluation of cosine, enabling us to approximate the function with a 11th order Chebyshev polynomial, which has only 6 terms thanks to the axial symmetry of the cosine function. The resulting function compiles into 23 branch-free PTX instructions using only cheap arithmetics such as bit shifting, floating point multiply and fused multiply–addition. The CUDA native implementation, on the other hand, compiles into 54 PTX
instructions with 3 conditional branches and potential conflict in the constant cache. Our implementation gives a maximum relative error of $1.10 \times 10^{-16}$ over the range $[0,1]$, or a binary accuracy of 44.1 bits.

The double-precision power function used in evaluating the weight function $w_i(r_i)$ is one of the most time-consuming functions in the CUDA math library. In order to conform to the IEEE floating point standard, the CUDA native implementation has to deal with the full range of inputs as well as possible exceptions. As a result, the native power function compiles into 258 PTX instructions with 18 conditional branches, 9 uniform branches, 3 reciprocal and 1 division with an average execution latency of 1982 clock cycles. Again in our implementation we seek to exploit the knowledge over the input range for optimization: the exception conditions that the base or the exponent would be NaN or Inf unless there are serious problems in the underlying physics of the model. Our custom double precision power function is based on the base-2 logarithmic intrinsic and exponential using the identity $a^b = 2^{b \log_2 a}$ as shown in Algorithm 5. The logarithm part is similar to that used in our Box–Muller implementation except that the order of the Chebyshev polynomial is increased to 14. The exponential part is approximated by a 11th order Chebyshev polynomial. Both routines give a maximum error of less than 1 unit in the last place (ULP). Special care has been taken to ensure the accuracy of the composite function when combining the results. The maximum error of our power function is 6 ULP given an input range of $a \in [10^{-102}, 10^{102}]$ and $b \in [0,3]$, or 11 ULP given an input range of $a \in [10^{-51}, 10^{51}]$ and $b \in [0,6]$. This should suffice for the purpose of DPRD simulation, where typically $a \in [10^{-10}, 2]$ and $b \in [0.25, 3.0]$. The advantages of the aforementioned custom transcendental functions over the CUDA native ones are characterized in detail by Table 3.

Algorithm 5 Branchless power function with error $\leq 11$ ULP. The __hiloint2double, __double2hiloint and __fma functions are double-precision floating point intrinsics from the CUDA math library.

```c
// fast computing for integer power of 2
double power2( int n ) { return __hiloint2double((1023+n)<<20, 0 ); }

// 11th order 2^x = P(x)
if( error < 1 ULP for x in [0,1.0])
  double exp2_frac( double x ) {...}

// fast converging 14-order log2(x) = z*P(z^-2)
if(error < 1ULP for x in [1,2])
  double log2_frac( double x ) {...}

double fastpow(double a, double b)
{
  int hi, lo;
  __double2hiloint( a, hi, lo );  // extract exponent
  double I = ( hi >> 20 ) - 1023;
  // reset exponent, do fractional log
  hi = ( hi & 0X000FFFFF ) | 0X3FF00000;
  double F = log2_frac( __hiloint2double(hi,lo) );
  // multiply by exponent, separate to 2 parts to prevent mantissa precision loss
  double II = floor( b * ( I + F ) );
  // prevent mantissa precision loss
  return power2(II) * exp2_frac( __fma( b, F, __fma( b, I, -II )) );
}
```

3.6. Communication

In our code, each processor maintains a sendlist for every neighboring processor recording the indices of local particles whose information is to be sent during the communication. The determination of the border particles can be implemented relatively straightforward on CPUs, whereas on GPUs it requires the cooperation of multiple kernels. Nevertheless, we still implemented it on GPU because the border determination must be done after particle reordering where particle indices are reassigned. A total of seven kernels are used as outlined in Algorithm 6 for border determination. Communication of ghost particle position and velocity between two border-determination steps involves only the last three steps of the algorithm. A CUDA-aware MPI implementation, i.e. the Cray MPI, was tested. However it was found that manually pipelined execution and data transfer between the host and device yields better performance. Peer-to-peer memory transfer is not utilized because each MPI rank handles only a single GPU.

Algorithm 6 Exchange of border information.

1: function BORDERDETERMINATION
2: flag particles to be sent using a 1-bit boolean
3: multi-block segmented prefix sum: down-sweep phase
4: multi-block segmented prefix sum: blocks
5: multi-block segmented prefix sum: up-sweep phase
6: send list generation: parallel compaction using scan result
7: pack information of particles in send list
8: MPI communication...
9: unpack list of incoming ghost particles
10: end function

4. Code verification & benchmark

4.1. Method

Benchmarks were carried out on the TITAN supercomputer located at the Oak Ridge National Laboratory [38]. The TITAN is a Cray XK-7 system with 18688 nodes. Each node is equipped with one AMD Opteron 6274 CPU and one NVIDIA Kepler K20X GPU. Each Opteron 6274 CPU has 16 integer cores and 8 floating point cores running at 2.2 GHz along with 16 MB of L3 cache. The Kepler K20X, with 2688 CUDA cores, has a theoretical double-precision performance of 1.31 TFLOPS and a peak memory bandwidth of 250 GB/s. Only the main loop is timed in the benchmark since
Table 3
Comparison between our custom double-precision math routines (prefixed with fast) and the CUDAnative ones. This microbenchmark is done through a chain of dependent statements according to Ref. [33].

<table>
<thead>
<tr>
<th>Function</th>
<th>Instruction#</th>
<th>Conditional branch</th>
<th>Latency (cycles)</th>
<th>Max rel. error</th>
</tr>
</thead>
<tbody>
<tr>
<td>fastcospi</td>
<td>23</td>
<td>0</td>
<td>90</td>
<td>$1.10 \times 10^{-10}$</td>
</tr>
<tr>
<td>cospi</td>
<td>54</td>
<td>3</td>
<td>420</td>
<td>1 ULP</td>
</tr>
<tr>
<td>fastlog</td>
<td>47</td>
<td>0</td>
<td>380</td>
<td>$4.21 \times 10^{-12}$</td>
</tr>
<tr>
<td>log</td>
<td>89</td>
<td>5</td>
<td>603</td>
<td>1 ULP</td>
</tr>
<tr>
<td>fastpow</td>
<td>79</td>
<td>0</td>
<td>494</td>
<td>11 ULP</td>
</tr>
<tr>
<td>pow</td>
<td>258</td>
<td>18</td>
<td>1982</td>
<td>2 ULP</td>
</tr>
</tbody>
</table>

Fig. 11. Speedup of our code on a single Kepler K20x GPU over 16 AMD Opteron 6274 CPU cores for various system size, cutoff distance and particle number density. Results obtained by using two neighbor list updating frequencies, i.e. 3 and 5, are represented by dotted and solid lines, respectively.

Fig. 12. Speedup contributed by each of the optimizing techniques mentioned in Section 3 for systems containing 64k, 128k, 512k, and 2m particles.

Fig. 13. Weak scaling performance of our code for various cutoff distance and particle number density. The system size is kept at 1 million particles per node.

Fig. 14. Strong scaling performance of our code for various cutoff distance and particle number density. The system size is fixed at 2 million particles regardless of the node number.

4.2. Flow simulation

To verify the correctness of our code, we aimed at reproducing the fluid viscosity measured from the double Poiseuille flow used by Backer et al. [39]. The parameters are chosen as $\sigma = 4.5$, $\rho = 6.0$, $k_B T = 0.5$ and $\delta t = 0.001$. The conservative force is left out in the same way as in the original paper. A total of 4608 particles were simulated in a simulation box of dimension $12r_c \times 8r_c \times 8r_c$ in the x, y and z directions, respectively. A body force $g_z = 0.055$ was applied to drive the flow in the periodical space. The viscosity of the fluid was evaluated from 10 parallel simulations with different initial configurations and random seeds. We obtained a viscosity of $2.089 \pm 0.009$, which is in excellent agreement with the published value of $2.09 \pm 0.02$. 
A variety of vesicles and membrane-like structures were grown from the 128 million particles simulation. The BBBAABBB triblock copolymer was initially randomly distributed in a cubic system of size 299.4 × 299.4 × 299.4. The concentration of the surfactant is 10%. Scale bars are in reduced DPD units.

Another test case of transient double Poiseuille flow was employed to examine the accuracy of our code over long-time integration. The parameters are chosen as $\rho = 5.0$, $a_0 = 15.0$, $\sigma = 3.0$, $k_B T = 1.0$, $r_c = 1.0$ and $\delta t = 0.01$. The system consists of 262,144 particles in a simulation box of 59.4123 × 7.42654 × 118.825 in the x, y and z directions. Velocity profiles along the z direction were collected for time $T = 100$, 200, 500, 1000, 2000 and 10,000. For each time point the velocity profile is sampled from a time window of $[T - 0.5, T + 0.5]$. The simulation was repeated 10 times with different initial configurations and random seeds. The simulation gives good matching with the analytic solution given in Eq. (10) by Sigalotti et al. [40].

$$u(z, t) = \frac{F d^2}{8 \nu} \left(1 - \frac{2z}{d}\right)^2 - \sum_{n=0}^{\infty} \frac{4 (-1)^n F d^2}{\nu \pi^2 (2n + 1)^3} \cdot \cos \left(\frac{(2n + 1) \pi z}{d}\right) \cdot \exp \left[- \frac{(2n + 1)^2 \pi^2 \nu t}{d^2}\right].$$  \hspace{1em} (10)

A comparison of analytical and DPD simulation results is shown in Fig. 10.

4.3. Single node speedup

Three different particle densities, 3, 6, and 50, and two cutoff distances, 1.0 and 1.5, were used to fully characterize the performance of our code. Note that a particle density of 50 is only observed for atomistic systems, yet such comparison allows us to estimate the performance of our algorithm when applied to an atomistic system. A time step of 0.005, along with two neighbor list updating frequencies, 3 and 5, was used. Systems whose size ranged from 8192 to 2 million particles were simulated. To enable a fair competition between the GPU and CPU implementations, the speedup is measured for one Kepler GPU over all available CPU cores on a single node. Speedup is measured as the ratio of wall time elapsed for carrying out a specific simulation. As shown in Fig. 11, the GPU code performs better for larger systems, with the best speedup of over 30 measured with 2 million particles. This is not surprising since overhead, especially kernel launching latency, is independent of the system size. Another interesting fact is that the speedup increases rapidly once the system size grows over 200,000. We speculate that the GPU may be experiencing cache depletion above this point.

We also measured the speedup contributed by each of the optimizing techniques mentioned in Section 3 for systems containing 64k, 128k, 512k, and 2m particles. The measurement was carried out by first substituting all aforementioned optimized kernels with their conventional counterparts, and then re-enabling them in the order as presented in the previous section. A more transferable metric, million particles-steps per second, or MPS/second, is used because it facilitates performance characterization and comparison across systems of different particle numbers. The result as presented in Fig. 12 shows that performance enhancement ranging from 80% to 130% was achieved when combining all five techniques. It is also shown that larger systems tend to benefit more from the optimization.
4.4. Weak scaling and strong scaling

The same set of test cases from the single node benchmark were used to establish the weak scaling and strong scaling benchmarks. For weak scaling, particles per node was kept at 1 million for $\rho = 3$ and 5, and was kept at 128k for $\rho = 50$. A nearly linear speedup is observed for up to 1024 nodes as demonstrated by Fig. 13.

Systems containing 2 million particles at different particle densities were simulated for the strong scaling benchmark. The result is shown in Fig. 14. Parallel efficiency is satisfactory for up to 64 nodes, but then it deteriorates gradually. The efficiency declination is attributed primarily to the overhead of copying data between the main memory and the GPUs when preparing the MPI communication. An attempt to optimize such latency...
using the newly-introduced GPU Direct RDMA pathway combined with the CUDA-aware Cray MPI was made. However, no speedup was observed because RDMA is only efficient for smaller message sizes due to its limited bandwidth despite lower communication latency. The strong scaling limit for our code is around 512–1024 nodes, which corresponds to the point when communication overhead dominates the entire simulation. Increasing node count beyond this point results in worse performance.

4.5. Amphiphilic polymer self-assembly

A simulation of spontaneous vesicle formation using 134,217,728 particles is performed to further demonstrate the capability of our new DPD program to study complex fluids. An aqueous surfactant solution system consisting of water and an amphiphilic triblock copolymer were modeled. The particles representing water, hydrophilic block and hydrophobic block are denoted as S, A, B, respectively. The triblock copolymer has a chain configuration of BBBAABB. The system was initialized to be a random distribution of water particles and polymer molecules. Bounce-forward boundary condition was applied instead of the periodic boundary condition. Beads within a molecule are tied together using a harmonic potential $F = K(r - r_0)$, where the coefficients were chosen as $r_0 = 0.38$ and $K = 80$. The DPD parameters were chosen as $\rho = 5.0, \sigma = 3.0, \gamma = 4.5, k_B T = 1.0$, and $\delta t = 0.01$. The pairwise repulsive force magnitudes $a_{ij}$ were adapted from Ref. [41] and scaled inversely proportionally with regard to the particle density:

$$
\begin{pmatrix}
A & B & S \\
A & 15 & 120 & 15 \\
B & 120 & 15 & 120 \\
S & 15 & 120 & 15 \\
\end{pmatrix}
$$

The simulation took 54 h to complete on 1024 nodes of TITAN. A total of 12,400,000 time steps, or a time span of 124,000 in reduced DPD units, were sampled. The time evolution of the system as illustrated in Fig. 15 proves that a mixture of membranes, simple vesicles and multi-compartment vesicles with a remarkable morphological variation were grown from the initially random distribution. The structure of the vesicles as visualized by a transmission electron microscopy-style rendering technique exhibits many similarities to those of the onion-like multilamellar multi-compartment vesicles observed by Bangham et al. [42] as shown in Fig. 16. Fig. 17 demonstrates that the single-compartment vesicle in Fig. 16(a) is actually formed through the fusion and spontaneous wrapping of membrane-like structures.

5. Conclusion

We presented a complete design of the DPD simulation code on the CUDA GPUs. The program is derived from LAMMPS and it achieves substantial speedup over the original CPU version. Novel algorithms are used in all parts of the program including force evaluation, data layout and numerics. An atomics-free, deterministic algorithm for constructing the neighbor list is proposed. We have also been able to address locality and neighbor list building in a tightly-coupled framework. Numerical optimization targeting random number generation and functional form evaluation were proposed to fully utilize the computing power of the GPU. The code executes on large-scale supercomputers with near-optimal weak scaling efficiency, while strong-scaling efficiency is compromised by the GPU–CPU data transfer overhead. A large-scale simulation of spontaneous vesicle formation demonstrated the practicality of our code. Future work includes optimizing communication latency through overlapping computation with communication as well as load balancing for non-trivial domain geometry. The code will be contributed as a LAMMPS user module that is freely available to the general public.

Acknowledgments

This work was supported by the new Department of Energy Collaboratory on Mathematics for Mesoscopic Modeling of Materials (CM4). Y.H.T. acknowledges discussion and technical support from Michael L. Parks, Steve Plimpton and Christian Trott at Sandia National Laboratories. Simulations were carried out at the Oak Ridge Leadership Computing Facility through the Innovative and Novel
Computational Impact on Theory and Experiment program at Oak Ridge National Laboratory under project BIP017.

References


