Journal of Computational Physics 230 (2011) 7191–7201

Contents lists available at ScienceDirect

Journal of Computational Physics

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

Pseudo-random number generation for Brownian Dynamics and Dissipative Particle Dynamics simulations on GPU devices

Carolyn L. Phillips [a], Joshua A. Anderson [b], Sharon C. Glotzer [a,b,c,*]

[a] Applied Physics, University of Michigan, Ann Arbor, Michigan 48109, USA
[b] Chemical Engineering, University of Michigan, Ann Arbor, Michigan 48109, USA
[c] Material Science and Engineering, University of Michigan, Ann Arbor, Michigan 48109, USA

ARTICLE INFO

Article history:
Received 1 December 2010
Received in revised form 7 April 2011
Accepted 19 May 2011
Available online 12 June 2011

Keywords:
GPU
Brownian Dynamics
Dissipative Particle Dynamics
Molecular dynamics
Random number generation

ABSTRACT

Brownian Dynamics (BD), also known as Langevin Dynamics, and Dissipative Particle Dynamics (DPD) are implicit solvent methods commonly used in models of soft matter and biomolecular systems. The interaction of the numerous solvent particles with larger particles is coarse-grained as a Langevin thermostat is applied to individual particles or to particle pairs. The Langevin thermostat requires a pseudo-random number generator (PRNG) to generate the stochastic force applied to each particle or pair of neighboring particles during each time step in the integration of Newton's equations of motion. In a Single-Instruction-Multiple-Thread (SIMT) GPU parallel computing environment, small batches of random numbers must be generated over thousands of threads and millions of kernel calls. In this communication we introduce a one-PRNG-per-kernel-call-per-thread scheme, in which a micro-stream of pseudorandom numbers is generated in each thread and kernel call. These high quality, statistically robust micro-streams require no global memory for state storage, are more computationally efficient than other PRNG schemes in memory-bound kernels, and uniquely enable the DPD simulation method without requiring communication between threads.

1. Introduction

In recent years GPUs have become affordable, easily programmable, general-purpose massively parallel processors. Originally designed for rendering graphics, the massively parallel architecture of GPUs makes them well suited for many scientific computing problems. Algorithms that exploit fine-grained parallelism have been accelerated by orders of magnitude including financial models [1], computational fluids dynamics [2], linear algebra performed by the GPU accelerated BLAS, LAPACK, and sparse matrix libraries [3,4], and Fast Fourier Transforms [3,4].

GPUs are Single-Instruction-Multiple-Thread (SIMT) parallel devices. In a GPU environment, algorithms with fine-grained parallelism distribute calculations over thousands of simultaneous *threads*, each sharing instructions but operating on different data. A *kernel* is a launch of a large group of these threads scheduled in blocks across a number of multiprocessors on the GPU device. Each thread has access to a fraction of the resources of its multiprocessor, can share a small amount of memory with other threads in the same block, but cannot communicate directly with other running threads. GPUs have a hierarchy of memory structures, from global memory, which is large, accessible by all threads and has a high bandwidth but a long latency, to shared memory, which is relatively small and low latency, but shared only by the threads

* Corresponding author at: Chemical Engineering, University of Michigan, Ann Arbor, Michigan 48109, USA. Tel.: +1 734 615 6296; fax: +1 734 764 7453.
E-mail address: sglotzer@umich.edu (S.C. Glotzer).

0021-9991/$ - see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcp.2011.05.021

in the same block. Recent GPU models have a L1/L2 cache hierarchy which improves the performance of spatially and temporally local accesses among threads. While the specifics of the GPU design change from one model to the next, the hierarchy of memory resources and the limited ability to communicate are likely to be common to most future SIMT architectures.

GPU-accelerated implementations of molecular dynamics (MD) have proven to be very fast compared to running a simulation on a single CPU core, achieving two orders of magnitude speed-ups [5] (http://codeblue.umich.edu/hoomd-blue/index.html). There is a great interest in expanding the algorithms and methods that can be accelerated by GPUs. However, reformulating methods developed for a serial environment, or even a parallel computing environment where work is distributed over many CPU cores, to a massively parallel SIMT environment is not always straightfoward. If a method cannot be effectively implemented in a multi-threaded environment, because of either communication requirements or a stubbornly serial step, the GPU speedup can be lost.

Two extensions of the molecular dynamics algorithm are Brownian Dynamics (BD) [6], also known as Langevin Dynamics, and Dissipative Particle Dynamics (DPD) [7,8]. BD and DPD are implicit solvent methods commonly used in models of soft matter and biomolecular systems [9]. BD and DPD enable longer simulation time-scales and larger systems to be studied by abstracting the interaction between the bath of solvent molecules and the larger particles of interest. Simulating the ballistic energy of the numerous solvent particles is replaced by a randomized coarse-grained force controlled by system temperature. Functionally, the interaction acts as a thermostat. BD and DPD are used to model polymers, proteins, nanoparticles, and colloidal systems [10–14].

BD and DPD both require random numbers to be generated at a rate of $\sim kp$ per time step, where p is the number of particles in the system and k is a constant. In a serial environment, these random numbers are typically drawn from a single stream of random numbers generated by a pseudo-random number generator (PRNG). Ignoring all issues of how the numbers generated would be delivered to or from the host CPU and GPU, using the test hardware of this paper, the GPU is capable of generating more than 140 times as many random numbers per second as the CPU. In general, it is the most efficient for random numbers to be generated as close in the hardware to their intended usage as possible. In a SIMT parallel computing environment where each particle calculation is assigned to a single short-lived thread, small batches of random numbers must be distributed over thousands of threads and millions of kernel calls.

In this communication, we introduce a novel scheme for generating such widely distributed, small batches of random numbers and then show how this scheme supports BD and DPD simulations. Our scheme, henceforth referred to as one-PRNG-per-kernel-call-per-thread (*pK-pT*) uses a disposable PRNG to produce a micro-stream of random numbers in each thread. The advantage of our scheme over other GPU PRNG schemes is that it does not use the GPU global memory and can accommodate a wide range of PRNG numerical algorithms. We implement our scheme with the *Saru* PRNG package [15] and the Tiny Encryption Algorithm, TEA [26,35]. Given statistically robust sub-algorithms, our scheme is statistically robust, is moderately faster than other schemes for the BD thermostat, and enables a significantly faster algorithm for the DPD thermostat. The *pK-pT* scheme is currently used to implement BD and DPD in the HOOMD-blue GPU-accelerated MD code package (http://codeblue.umich.edu/hoomd-blue/index.html).

In Section 2, we discuss prior work implementing PRNGs in parallel environments and introduce our *pK-pT* scheme. In Section 3, we briefly introduce how HOOMD-blue accelerates MD, the BD and DPD methods, and how each method uses PRNG schemes. In Section 4, we introduce a particular implementation of the *pK-pT* scheme based on the *Saru* PRNG and TEA, address how we validated our *pK-pT* scheme, and discuss two micro-benchmarks that we used to measure the performance of the *pK-pT* scheme against a one-PRNG-per-thread scheme. In Section 5, we provide concluding remarks.

2. Pseudo-random number generation

Generating robust pseudo-random number streams on a CPU is a well-studied topic [16–18]. In general, most algorithm design choices involve making a trade-off between the statistical robustness of the random number stream and the computational cost of generating the stream. Most trade-off discussions focus on the generation of the PRNG stream without much consideration of how the stream is initialized. With the advent of massively parallel architectures, however, new questions arise regarding how to handle random number generation and both of these considerations are important.

2.1. Parallel processor PRNG schemes

In prior work in multi-processor parallel computing environments, a commonly used scheme is the one-PRNG-per-processor scheme. In this scheme, each processor maintains a uniquely seeded and therefore independent random number stream. For example, for MD methods that require random numbers, the Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) code package (http://lammps.sandia.gov) uses a Marsaglia PRNG [19] with a unique seed generated for each processor. Alternatively, Matsumoto and Nishimura [20] proposed that rather than create independent seeds for each instance of their Mersenne Twister PRNG, instead the processor id could be encoded into the characteristic polynomial used to generate the random number stream.

Another common scheme has been one-PRNG-for-all-processors [21]. Here, a leap-frogging or blocking method is used to partition a single random number stream over many processors. This PRNG scheme is limited to using sufficiently robust PRNGs that also have an efficient method for advancing the state.

In a GPU SIMT environment, these methods, respectively, become one-PRNG-per-*thread* and one-PRNG-for-all-*threads*. Multiple approaches have been considered for the generation of random numbers in different types of applications [22–26]. Ref. [22] focuses on either generating a bank of random numbers written to global memory or for use with long calculations executed over a single long kernel call. Langdon [23] proposes using a Parker–Miller PRNG, seeding each thread with a master seed plus the thread number, and discarding the first three random numbers generated. For a BD application, Zhmurov et al. [24] proposes using a separate PRNG on the CPU to generate a sequence of random seeds used to initialize a PRNG, such as Hybrid Tau or Ran2, in each thread. The drawback of these methods for BD is that the random stream of numbers is used over many very short kernel calls. The state of the PRNG associated with each thread is loaded from memory at the beginning of the kernel call and stored to memory at the end of the kernel call. Per-thread resources, possibly even global memory, may be exhausted when storing large PRNG state vectors, thus making their use in real applications impractical. Ran2, for example, has a state size of 35 long integers (64-bits) or 280 bytes per thread, mandating either small thread blocks or expensive repeated reads and writes to global memory [24]. The Marsaglia PRNG used by LAMMPS has an even larger state size of 100 doubles and two integers per stream and, thus is also impractical for a one-PRNG-per-thread scheme. Nearly all of the robust serial PRNGs also have huge state sizes, making them also unsuitable for a one-PRNG-per-thread scheme [18].

For one-PRNG-for-all-threads schemes, either the PRNG must be computationally efficient to advance or partition, or a separate kernel must be called to periodically generate random numbers and "bank" them in global memory for later use. As an example of a stream that can be shared amongst the threads, Zhmurov et al. [24] consider the use of a lagged Fibonacci algorithm. For this PRNG, each thread must still load and store state information, albeit storing far fewer total variables than if this PRNG was used in a one-PRNG-per-thread scheme. Banking random numbers requires a data management scheme in the simulation kernel for expending and refreshing the bank. Also, without a parallel implementation of the PRNG, banking random numbers in global memory [22] will become a bottleneck in a massively parallel algorithm. And even if pre-generating such a bank had no computational cost, merely loading a random number into a thread takes more time than generating it in the thread from state information.

2.2. One-PRNG-per-kernel-call-per-thread scheme, pK-pT

In the *pK-pT* scheme, instead of loading, modifying, and storing the PRNG state in global memory, we propose to *create* a per-thread PRNG state by applying a *hash-based PRNG* to data unique to each thread and kernel call and then applying a *streaming PRNG* to generate a micro-stream of random numbers. At the end of the thread lifetime, the PRNG state can be discarded. In other words, the state of the PRNG is not stored in global memory.

A hash-based PRNG is a function that applies a sequence of integer operations to an input vector so as to generate a single deterministic output that is effectively decorrelated from the input. In order to achieve a sufficient degree of decorrelation, a large number of operations are applied to the input. In comparison, most streaming PRNGs use few integer operations to efficiently generate statistically random streams.

The usual purpose of a hash-based PRNG is for message encryption. A block of data is passed through a cryptographic hash such as the Secure Hash Function, SHA, with variants SHA-0,-1, or -2, the Message Digest Function 5 (MD5), or the Tiny Encryption Algorithm (TEA) and its extension XTEA, to generate encrypted text that cannot be decrypted without knowledge of the key used. The demands on cryptographic hash functions are very similar to the demands of streaming PRNGs, as any measurable patterns in the output makes the encryption vulnerable to attack.

Using a cryptographic hash alone to generate a small set of random numbers in a massively parallel GPU environment was explored in references [25,26]. Tzeng and Wei [25] used MD5 as a random number generator on a GPU for graphics applications and demonstrated that it produces a high quality set of random numbers. Subsequently, Zafar et al. [26] compared the use of TEA and XTEA against MD5 on a GPU as a random number generator and found that TEA and XTEA also produce random number sets of comparable statistical quality with fewer computations. The limitation of using a hash-based PRNG alone is that the number of random numbers that can be generated in the thread is restricted to the size of the algorithm output, unless the computationally-heavy hash-based PRNG is applied multiple times. By using the hash-based PRNG to create the state for a streaming PRNG, an arbitrary length micro-stream of quality random numbers can be produced for reasonable computational effort.

As long as the micro-stream of random numbers for each calculation can be associated with a unique set of integers, a hash-based PRNG can be applied once to the set to create a unique PRNG state for a micro-stream. Refs. [25,26], for example, use lattice point coordinates as inputs. For a MD application, an example of a unique set of input integers is (1) a global user chosen seed, (2) an integer that is incremented in each subsequent kernel call (e.g. the time step), and (3) an integer unique to each micro-stream across a single kernel call (e.g. the thread index). Seeds (1), (2), and (3) define each micro-stream uniquely over a set of simulations, sequence of time steps, and sequence of threads. No memory bandwidth is spent accessing these three integers since seeds (1) and (2) are broadcast to all threads, and since seed (3) uses data already available in the thread.

This scheme takes advantage of the massive parallelism of a GPU, whereby the long memory latencies can be hidden by arithmetic instructions performed in other concurrently running threads. An ideal ratio of arithmetic to global memory accesses enables full utilization of both the memory bandwidth and instruction throughput [3]. While the ideal ratio is device dependent, the value for one specific GPU, the NVIDIA Tesla C2050, is 27.8 instructions per 32 bits loaded. In a simulation large enough that the streaming multiprocessors are sufficiently occupied, a hash-based PRNG that requires 28 instructions to initialize the state vector for a streaming PRNG can be just as computationally efficient as a one-PRNG-per-thread scheme that loads a small 32-bit state.

Also, as the computational throughput in GPU devices is currently increasing faster than the memory bandwidth, trading loads and store for computations is likely to remain a winning strategy. For a kernel that is memory bound without considering the generation of random numbers, the *pK-pT* scheme will generally outperform any scheme requiring memory accesses.

3. SIMT Molecular Dynamics, Brownian Dynamics, and Dissipative Particle Dynamics

In molecular dynamics (MD) algorithms, the trajectory of a system of particles is generated by solving Newton's equations of motion repeatedly for suitably small time steps. HOOMD-blue, which implements all computations on the GPU, realizes a classical MD algorithm by employing a velocity Verlet integration scheme. The calculation and summation of short-range forces between pairs of particles is reduced to an $\Theta(N)$ calculation by using cell lists and neighbor lists. A given particle's neighbor list contains the indices of all particles within interaction range of that particle. The net pair-wise force acting on a particle can be determined by consulting only those particles in that particle's neighbor list.

In HOOMD-blue, the basic kernels deployed over a single time step are (1) a first update of the position and velocity of the particles (2) the calculation and summation of the forces on each particle, and (3) a second update of the velocity of the particles. When needed (typically every 9 time steps), (4) the cell list for the system is recalculated and (5) the neighbor list for each particle is built. All the kernels are handled by assigning a thread to each particle [5] (http://codeblue.umich.edu/hoomd-blue/index.html).

One significant difference between SIMT MD algorithms and a streaming serial MD algorithm is how the pair-wise force summation is handled. On the CPU, the calculation and summation of pair-wise forces is generally made more efficient by calculating the force only once per particle pair [9]. If particle *A* and particle *B* interact, the interaction force can be calculated while looping over the neighbors of particle *A*, and then the equal and opposite force acting on particle *B* can be written to a globally stored force array. In contrast, most massively parallel molecular dynamics algorithms that employ one thread per particle have each thread calculate all the forces acting on a given particle [5,27–30], with the exception of Refs. [31,32]. This doubles the number of non-bonded force computations, but converts a gather-scatter memory access pattern to a simple gather memory access pattern [33] leading to an overall faster algorithm.

3.1. Brownian Dynamics

The BD thermostat models the solvent-colloidal particle interaction by applying a Langevin force [6,9]. In BD, a temperature dependent random force and a drag force proportional to the particle velocity is applied to each particle at each time step of the simulation. The force F applied to each spatial component of particle i is

$$F_i = -\gamma v_i + R_i\sqrt{3}\sqrt{\frac{2k_B T\gamma}{\Delta t}} \tag{1}$$

where γ is a coupling constant, T is the temperature of the solvent, Δt is the time step size, v_i is the velocity of the particle, k_B is the Boltzmann constant, and R_i is a random number uniformly distributed in the range $[-1,1]$. To compute this force for a given particle, three random numbers (one for each x, y, and z component) and the velocity of the particle are required. We choose to incorporate the force calculation into the second update of the velocity of the particles, kernel (3).

The one-PRNG-per-thread, -for-N-threads, and -for-all-thread schemes are all suitable for generating the single microstream of three random numbers required per particle per time step. Algorithm 1 provides a description of a BD kernel using a one-PRNG-per-thread scheme.

To use the *pK-pT* scheme for BD, the hash-based PRNG is applied to (1) a global seed, (2) the time step of the simulation, and (3) the particle index (i.e. the thread index) to generate an initialized state vector for the streaming PRNG. The PRNG state data is created and discarded inside the thread, and therefore does not need to access global memory. Algorithm 2 provides a description of a BD kernel using a *pK-pT* scheme.

3.2. Dissipative Particle Dynamics

The net sum of all the Langevin forces applied to a system for the BD thermostat is not zero because the random forces are not computed pairwise, so momentum is not conserved and therefore hydrodynamic behavior is not preserved (without the introduction of interaction tensors [34]). In contrast, the DPD thermostat method models the solvent-colloidal particle or bead interaction by applying the Langevin force to all pairs of interacting particles. Equal and opposite random and drag

forces are applied to particle pairs, so the momentum is conserved both locally and system-wide. The force F applied to a pair of particles i and j in DPD, where $r_{ij} < r_{interaction}$ is

$$F_{ij} = F_c(r_{ij}) - \gamma[\omega(r_{ij})]^2(\vec{v}_{ij}\cdot\hat{r}_{ij}) - R_{ij}\omega(r_{ij})\sqrt{3}\sqrt{\frac{2k_bT\gamma}{\Delta t}}$$

where F_c is a conservative force, γ is a coupling constant, T is the temperature of the solvent, dt is the time step size, $\vec{v}_{ij}$ is a velocity difference vector $\vec{v}_{ij} = \vec{v}_i - \vec{v}_j$, $\hat{r}_{ij}$ is a unit vector $\hat{r}_{ij} = (\vec{r}_i - \vec{r}_j)/|\vec{r}_i - \vec{r}_j|$, $\omega(r_{ij})$ is a function of the distance between particle i and j, and R_{ij} is a random number uniformly distributed in the range $[-1,1]$. A single random number is required for each pair-wise applied Langevin force between interacting particles. Depending on the density of the system and the interaction radius, this may require generating ~50–100 random numbers per particle.

While the DPD algorithm effectively applies a pair-wise force to the system of particles, that force cannot be simply incorporated into the existing pair-wise force summation kernel (2) using a one-PRNG-per-thread, per-N-threads, or for-all-threads scheme. In kernel (2), the same random force must be applied in both threads that compute the ij interaction. This implies that either some model of communication between threads, some method of two threads (and only those two threads) loading the same random number from a bank, or for this algorithm and only this step, a separate fine-grained decomposition based on one-thread-per-force calculation, with all the accompanying data structures, must be implemented. The first option, modifying kernel (2) to include a scatter communication using atomic operations to write to a global force array, is the simplest. This communication significantly slows the kernel execution time, as shown by our later benchmarks. Algorithm 3 provides a description of such a DPD kernel using a one-PRNG-per-thread scheme.

The DPD pair-wise force can be simply incorporated into the existing pair-wise force summation kernel (2) if using the *pK-pT* scheme. The *pK-pT* scheme enables two threads to generate the same random number, and thus the same stochastic force, without requiring coordination between the threads. To use the *pK-pT* scheme for DPD, the hash-based PRNG is applied to (1) a global seed, (2) the time step of the simulation, and (3) the index of the first (4) and second particle in the interaction. This generates an initialized state vector for the streaming PRNG that is the same in both threads. Algorithm 4 provides a description of a DPD kernel using a *pK-pT* scheme.

Algorithm 1: One-PRNG-per-thread (Brownian Dynamics)

```
if tid ⩽ n_particles then
  load RNGstate ← d_RNGstate[tid]
  load v ← d_velocity[tid]
  load f ← d_force[tid]
  R_x ← callRNG()
  R_y ← callRNG()
  R_z ← callRNG()
  f = f + F(v,R_x,R_y,R_z)
  v = update_velocity(v,f)
  store d_RNGstate[tid] ← RNGstate
  store d_velocity[tid] ← v
  store d_force[tid] ← f
end if
```

Algorithm 2: One-PRNG-per-kernel-call-per-thread (Brownian Dynamics)

```
Require: timestep, seed are broadcast
  if tid ⩽ n_particles then
    load v ← d_velocity[tid]
    load f ← d_force[tid]
    RNGstate ← hashRNG(timestep,seed,tid)
    R_x ← callRNG()
    R_y ← callRNG()
    R_z ← callRNG()
    f = f + F(v,R_x,R_y,R_z)
    v = update_velocity(v,f)
    store d_velocity[tid] ← v
    store d_force[tid] ← f
end if
```

Algorithm 3: One-PRNG-per-thread (Dissipative Particle Dynamics)

if $tid \leqslant n_{\text{particles}}$ **then**
  load $v \leftarrow d\_velocity[tid]$
  load $x \leftarrow d\_position[tid]$
  load $nlist\_length \leftarrow d\_nlist\_length[tid]$
  $f_{net} \leftarrow 0$
  load $RNGstate \leftarrow d\_RNGstate[tid]$
  for $i < nlist\_length$ **do**
    load $nid \leftarrow d\_nlist[tid,i]$
    load $v_{neigh} \leftarrow d\_velocity[nid]$
    load $x_{neigh} \leftarrow d\_position[nid]$
    $R \leftarrow \text{callRNG}()$
    $f_{tid,nid} = F(v, x, v_{neigh}, x_{neigh}, R)$
    $f_{net} \leftarrow f_{net} + f_{tid,nid}$
    atomic $force[nid] \leftarrow force[nid] - f_{tid,nid}$
  end for
  atomic $force[tid] \leftarrow force[tid] + f_{net}$
  store $d\_RNGstate[tid] \leftarrow RNGstate$
end if

Algorithm 4: One-PRNG-per-kernel-call-per-thread (Dissipative Particle Dynamics)

Require: timestep, seed are broadcast
  if $tid \leqslant n_{\text{particles}}$ **then**
    load $v \leftarrow d\_velocity[tid]$
    load $x \leftarrow d\_position[tid]$
    load $f \leftarrow d\_force[tid]$
    load $nlist\_length \leftarrow d\_nlist\_length[tid]$
    for $i < nlist\_length$ **do**
      load $nid \leftarrow d\_nlist[tid,i]$
      load $v_{neigh} \leftarrow d\_velocity[nid]$
      load $x_{neigh} \leftarrow d\_position[nid]$
      $RNGstate \leftarrow \text{hashRNG}(timestep, seed, tid, nid)$
      $R \leftarrow \text{callRNG}()$
      $f_{tid,nid} = F(v, x, v_{neigh}, x_{neigh}, R)$
      $f \leftarrow f + f_{tid,nid}$
    end for
    store $force[tid] \leftarrow f$
  end if

4. Validation

4.1. SaruSaru and SaruTEA PRNG

For performance testing the *pK-pT* scheme we considered two combinations of a hash-based and streaming PRNG. The PRNG package *Saru* [15]contains both a hash-based and streaming PRNG. The streaming PRNG of *Saru* was used in conjunction with its native hash-based PRNGs and with an implementation of TEA with eight rounds (TEA8) as recommended by reference [26]. These two combinations will be referred to as *SaruSaru* and *Saru*TEA, respectively.

Streaming PRNGs are tested for statistical randomness by applying a large set of theoretical and practical tests to its output stream. Hash-based PRNG can be tested by applying the same tests to concatenations of their outputs while systematically varying the inputs. TestU01 is a software library offering a collection of utilities for the empirical statistical testing of uniform random number generator [18] (other test batteries include DIEHARD, NIST, Rabbit, and Gorilla). Each independent PRNG component we considered has been tested and found statistically robust.

Saru provides a choice of three hash-based PRNGs, able to transform one, two or three seeds (s_1, s_2, s_3) into a two integer state for its streaming random number generator, using bitwise xor, multiplication, addition, bit shifting, and type conversion, via 12, 30, and 45 issued instructions, respectively. The hash-based PRNG for *Saru* was tested successfully by its author against TestU01's Crush [18,15]. The streaming PRNG in *Saru* has a state of two 32-bit words, and it uses a linear congruential generator (LCG) and an Offset Weyl Sequence (OWS) to advance the two words. *Saru* mixes the words and further transforms

the output by xors and a multiply. This streaming PRNG has a period of $3666320093 \cdot 2^{32} \approx 2^{63.77}$ and requires 13 issued instructions to advance the state and generate an output. The streaming PRNG for *Saru* was tested successfully by its author against DIEHARD, Rabbit, Gorilla, and TestU01's SmallCrush, Crush, and BigCrush [18,15].

The Tiny Encryption Algorithm [35], TEA, has a 64-bit input and uses a 128 bit key to generate a 64 bit output. Each round, or unit set of integer and bitwise operations, of TEA uses bitwise xor, multiplication, addition, and bit shifting to mix the input with the key. Each round of TEA involves 17 issued instructions, for a total of 136 issued instructions for TEA8. In Ref. [26], eight rounds were sufficient to produce high quality random numbers when tested against NIST and DIEHARD. As TEA was designed to minimize memory footprint while maximizing speed, it is particularly ideal for GPU applications.

The hash-based PRNGs are seeded as follow. For the Saru hash-based PRNG, we premix certain inputs so as to reduce three (or four) unique integers to two (or three) integers and then used the two (or three) input hash-based PRNG. For both BD and DPD, the global seed, *gs*, provided by user is hashed once by $gs = gs \times 0x12345677 + 0x12345$; $gs = gs \wedge (gs \gg 16)$; $gs = gs \times 0x45679$. This hash is performed as a precaution as users tend to use a very restricted set of seeds. For BD, seed s_1 is set to the particle index. Then we add the time step and the global seed to generate s_2. For DPD, s_1 is set to the smaller particle index, and s_2 is set to the larger particle index, then we again add the hashed global seed and the time step to generate s_3.

For the TEA8 hash-based PRNG, we use the key provided by reference [26], or $\{k_1 = 0xA341316C, k_2 = 0xC8013EA4, k_3 = 0xAD90777D, k_4 = 0x7E95761E\}$. For BD, the two inputs are set to the particle id and the time step. The first part of the key, k_1 is then set to the hashed global seed. For DPD, the two inputs are set to the smaller and larger particle index, in that order, and the first and second part of the key are set to the time step and hashed global seed. Using parts of the key for the extra inputs allows a single application of TEA8 to be used rather than using a nested hash function which requires further iterations of TEA8.

Both the *Saru* hash-based PRNG and TEA8 output two 32-bit words. The state of the *Saru* streaming PRNG is simply set to the output.

Even if the hash-based PRNG and the streaming PRNG are individually statistically robust, we must also validate statistical randomness between multiple streams generated by this scheme. This is effectively a validation of the hash-based seeding of the PRNG, insuring that different, highly correlated seed values do not generate correlations. This is also a validation that the streaming PRNG does not inadvertently undo the mixing of the hash-based PRNG. It is also important to test PRNG streams in a manner that reflects their use. The PRNG output used by a massively parallel simulation might best be visualized as a matrix. Each column represents the PRNG stream of a single thread over sequential kernel calls. Each row represents the PRNG values generated over a single time step. We want to analyze the randomness of this matrix both across the rows and across the columns. In practice, this matrix of values could be considered not just 2D, but 3D or 4D since the seeding values themselves are multidimensional. PRNGs in parallel environments should be sliced and concatenated into an appropriate single stream for testing. The same tool set can be applied to streams formed by reading through the matrix by different paths.

The two implementations used for the *pK-pT* scheme were tested in three ways. First, we assumed 16,000 particles and generated micro-streams of three random numbers per particle. The micro-streams of all the particles were concatenated and then concatenated again over subsequent time steps in order to test for whole system correlations. Second, we concatenated the micro-streams for a single particle over many time steps to test for correlations in the stochastic force that may be applied to a single particle. Third, to model the DPD stream, we assumed that a single particle was interacting with the same 50 particles for all time, and concatenated those random numbers over time. We applied the TestU01 SmallCrush, Crush, and BigCrush to each stream, generating a net total of 319 test statistics and p-values per stream. All three of these stream types passed the TestU01 SmallCrush, Crush, and BigCrush test batteries, with only spurious non-systemic failures of a few subtests for some seed values, a failure rate in the range of the expected number of failures for TestU01.

4.2. Benchmarks

To test the performance of the *pK-pT* scheme we used micro-benchmarks that emulate the kernels of the MD code [5] (http://codeblue.umich.edu/hoomd-blue/index.html). We compared the performance of the *pK-pT* scheme against a one-PRNG-per-thread (*pT*) scheme, where the small state (two 32-bit unsigned integers) of the *Saru* streaming PRNG is loaded and stored during each kernel call. The *pT* scheme used represents a best case memory access pattern relative to the per-thread, per-N-threads, or for-all-threads schemes. We also compared the *pK-pT* scheme using *SaruSaru* and *Saru*TEA to a *pT* scheme using the *XORWOW* PRNG provided in the recently released CURAND library, available in the CUDA™ 3.2 toolkit. As the current seeding method provided in the CURAND library for *XORWOW* is very slow (requiring $\approx$ 50 times longer than what is needed to load and store the state), a *pK-pT* scheme with the *XORWOW* PRNG was not considered.

All benchmarks were performed on a custom built workstation with an AMD Athlon™ II X4 630 CPU operating at 2.8 GHz on a mainboard with the nForce 980a chipset, 4 GB of DDR3 RAM operating at 1333 MHz, and an NVIDIA® GeForce® GTX 480 operating at stock settings with a processor clock of 1401 MHz and a memory clock of 1848 MHz. The system runs an x86_64 installation of CentOS 5.5, the CUDA toolkit version 3.2.9 and corresponding GPU device driver 260.24. The GTX 480 is the latest "Fermi" architecture which is the first generation of NVIDIA GPUs to include atomic operations for floats, enabling Algorithm 3.

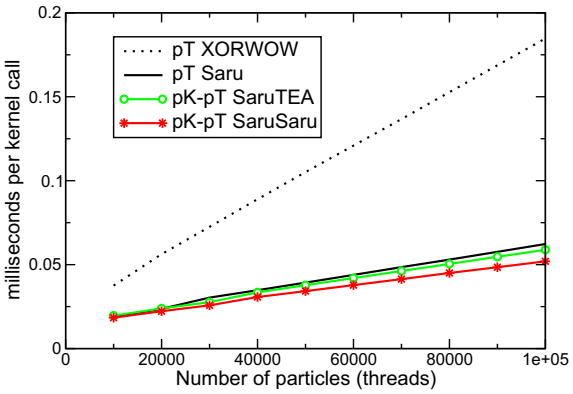

Fig. 1. For a Brownian Dynamics micro-benchmark, a one-PRNG-per-thread, *pT*, scheme using the Saru and XORWOW streaming PRNG, is compared to a one-PRNG-per-kernel-per-thread, *pK-pT*, scheme using *SaruSaru* and *Saru*TEA combined PRNGs.

4.2.1. Brownian Dynamics

For the BD micro-benchmark kernel, (Algorithms 1 and 2), the current force and velocity vectors for a particle are retrieved from memory, the Langevin force is calculated and added to the force vector, the velocity of the particle is updated, and both the force and velocity vector are written back to memory. This minimal kernel requires 48 bytes of memory transfer in each thread (presuming the force and velocity vector are both three 32-bit floats). Retrieving and storing the state of the *Saru* streaming PRNG and *XORWOW* PRNG requires an additional 16 bytes and 80 bytes, respectively, of memory transfer. The threads are synchronized at the end of each kernel call and the kernel is called a thousand times to average out statistical variation.

The *SaruSaru* and *Saru*TEA *pK-pT* scheme reduces the memory access by 25% relative to the *Saru pT* scheme. In Fig. 1, the average time spent per kernel invocation is shown for systems of 10,000 to 100,000 particles. The *SaruSaru* and *Saru*TEA*pK-pT* scheme out-performs the *pT* scheme over the entire range. For systems of 100,000 particles, the *SaruSaru pK-pT* scheme is 16% faster than the *Saru pT* scheme. The *Saru*TEA scheme, with its three times larger hash-based PRNG, is only 7% faster than the *Saru pT* scheme. In contrast, the *XORWOW* PRNG in a *pT* scheme is three times slower than the *Saru* PRNG *pT* scheme due to loading and storing the larger PRNG state.

4.2.2. Dissipative Particle Dynamics

For the DPD micro-benchmark, the DPD thermostat is applied to each particle as in a MD force summation kernel [5], as described in Algorithms 3 and 4. Each thread loads its particle's neighbor list one entry at time. The position and velocity of each neighbor is then loaded via texture reads, and the pair-wise Langevin forces are computed. For this benchmark, the conservative force is not computed. The threads are synchronized at the end of each kernel call and the kernel is called a thousand times to average out statistical variation.

For the *pT* scheme, the current velocity and position is loaded at the beginning of the kernel, and a half-neighbor-list is used. Half-neighbor-lists are neighbor-lists where, if particles i and j are neighbors, particle i is in particle j's neighbor-list or vice-versa, but not both. Floating point atomic adds are used for all writes to the force array in global memory and are bundled load-add-store operations. The PRNG state is loaded at the beginning of the kernel, a single stream is used during the kernel, and the PRNG state is stored at the end of the kernel. The total average memory transfer per thread is 72 bytes + $(N/2) * 60$ bytes, where N is the number of neighbors per particle.[1]

For the *pK-pT* scheme, the current velocity, position, and force of the particle is loaded at the beginning of the kernel call and a full neighbor-list is used for each particle. Full neighbor-lists are neighbor lists where, if particles i and j are neighbors, particle i is in particle j's neighbor list, and vice versa. The PRNG is continually re-seeded for each pair as described above. At the end of each kernel call, the net force on the particle is written to global memory. The total memory transfer of each thread is 56 bytes + $N * 36$ bytes, where N is the number of neighbors per particle.

In simulations of diffusive particles, after a certain amount of time there is no correlation between the indices of nearby particles. HOOMD-blue solves this by periodically reorganizing particle data so that nearby particles are likely to have nearby indices [5]. Therefore, two topologies of neighbor list were constructed to bound the possible neighbor lists. In one case, particles are assumed to only interact with neighbors with nearby indices ($\pm N/2$). In the other case, the neighbor list was randomly constructed. In both cases the half-neighbor list was constructed so that each particle is assigned roughly (or for the case of sequential neighbors, exactly) the same number of neighbor calculations to balance the load of work among threads. The two types of neighbor lists lead to significantly different patterns of cache use and data collisions in global memory accessed by the computation kernel. The number of neighbors, N, was set to 50.

[1] velocities and positions accessed by texture reads and therefore are float4's.

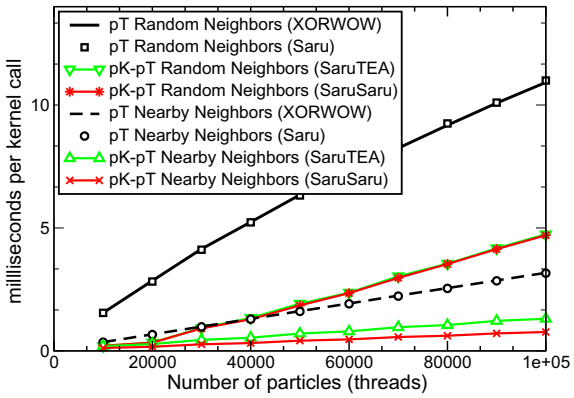

Fig. 2. For a Dissipative Particle Dynamics micro-benchmark, a one-PRNG-per-thread, *pT*, scheme using the *Saru* and *XORWOW* streaming PRNG, is compared to a one-PRNG-per-kernel-per-thread, *pK-pT*, scheme using *SaruSaru* and *Saru*TEA combined PRNGs. Two types of neighbor lists, with different topologies and therefore different patterns of memory usage, are shown to bound actual simulation performance.

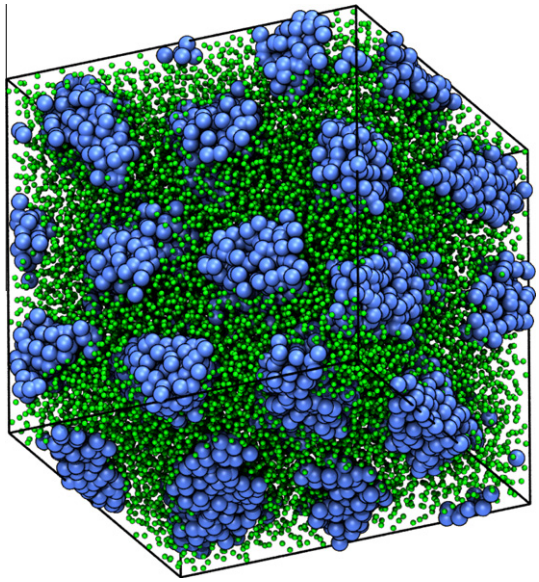

Fig. 3. Pictured is the full benchmark system for the DPD simulation method, an A_3B_7 block copolymer system of 2400 polymers (24,000 particles) self-assembled into the hexagonally packed cylinder phase. Details on this system can be found in Ref. [36,37].

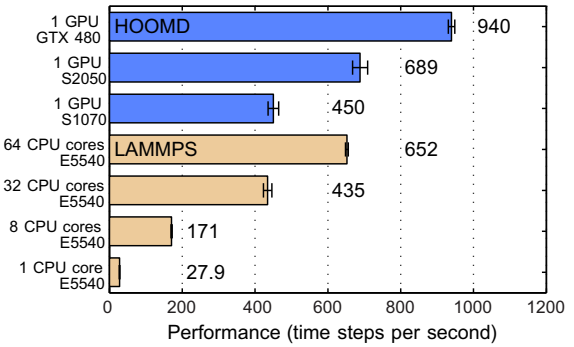

Fig. 4. Benchmarks comparing HOOMD-blue to LAMMPS, a parallelized CPU molecular dynamics code package, for the DPD benchmark system of Fig. 3.

The *pT* scheme, which uses both a gather and scatter memory access pattern, has 15% fewer memory transfers than the *pK-pT* scheme, which only uses a gather memory access pattern. This is because the *pT* scheme uses half-neighbor lists. However, a quarter of the memory transfers for the *pT* scheme are significantly slower atomic operations.

In Fig. 2, the average time spent per kernel invocation of this micro-benchmark is shown for systems of 10,000 to 100,000 particles. The *SaruSaru* and *Saru*TEA *pK-pT* scheme out-performs the *pT* scheme over the entire range. For nearby neighbors, the *SaruSaru pK-pT* scheme is 3 to 4 times faster than the *pT* scheme. For randomly dispersed neighbors, the *SaruSaru pK-pT* scheme is 2 to 7 times faster than the *pT* scheme. The *Saru*TEA *pK-pT* is 60–70% slower than the *SaruSaru pK-pT* scheme for nearby neighbors, but performs equivalently for randomly dispersed neighbors. In effect, the inefficient memory operations necessary to load data for randomly dispersed particles interleave with arithmetic operations on the GPU multiprocessor so as to completely hide the larger TEA8 hash-based PRNG.

Unlike the BD kernel, the time to load and store the PRNG state is negligible relative to the length of the kernel. Thus, the performance difference of the kernels using the *pT* scheme and the *Saru* or *XORWOW* PRNG is indistinguishable.

5. Conclusions

In this communication, we introduce a one-PRNG-per-kernel-per-thread scheme, which allows the generation of millions of micro-streams of random numbers in a SIMT massively parallel computing environment without having to load and store a PRNG state vector in each thread. This scheme is currently used in HOOMD-blue to support BD and DPD simulations. The one-PRNG-per-kernel-per-thread scheme uniquely supports DPD simulations without changing the one-thread-per-particle fine grained parallelism used for the MD algorithm or requiring communication and coordination between threads via atomic operations. By eliminating the need to load and store PRNG state information in the kernel, the one-PRNG-per-kernel-call-per-thread scheme is moderately faster than other schemes for calculating small batches of random numbers in the BD algorithm. By eliminating the need to communicate and coordinate between threads, the one-PRNG-per-kernel-per-thread scheme is 2-7 times faster than a one-PRNG-per-thread scheme for the DPD thermostat.

The one-PRNG-per-kernel-per-thread scheme utilizes both a hash-based and a streaming PRNG in each kernel call. While for our simulation software package we chose the *SaruSaru* PRNG, which has both PRNG functionalities, the components of the PRNG scheme can be changed to meet the needs of the application. Replacing the *Saru* hash-based PRNG with TEA8, for example, is a reasonable alternative that more naturally handles up to six unique inputs. Replacing the streaming PRNG with a one that has a longer period may also be more suitable for some applications. This PRNG scheme also allows for a mix of different types of PRNGs to be used in a massively parallel application if desired.

To our knowledge, HOOMD-blue is the only GPU MD code package that can perform Dissipative Particle Dynamics simulations. An example of a full DPD benchmark simulation comparing GPU and parallelized CPU performance is shown in Figs. 3 and 4. Fig. 3 shows a block copolymer system of 2400 A_3B_7 polymers (24,000 particles) that have self-assembled into the hexagonally packed cylinder phase [36,37] simulated using the DPD method. In Fig. 4, we compare the performance, measured in time steps per second, of HOOMD-blue and a optimized parallel multiprocessor (CPU) MD code package LAMMPS (http://lammps.sandia.gov) in simulating this system. The GPU code run on a single NVIDIA GTX 480 is significantly faster than the CPU code parallelized over 64 cores. By enabling BD and DPD to be performed in HOOMD-blue, a broad range of mesoscale coarse-grained simulations can now be accelerated in a massively parallel architecture, thus also accelerating the understanding of soft matter and biomolecular systems and discovery of new materials.

Acknowledgments

We acknowledge many useful discussions with Steve Worley regarding use of his *Saru* PRNG package. We also acknowledge useful discussions John Stone regarding GPU computing and with Marc Olano regarding implementing TEA8. We thank our reviewer for providing helpful comments. C.L.P was supported by the US Department of Energy Computational Science Graduate Fellowship and the US Department of Energy, Office of Basic Energy Sciences, Division of Materials Sciences and Engineering, under award DE- FG02-02ER46000. This material is also based upon work supported by the DOD/DDRE under Award No. N00244-09-1-0062. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the DOD/DDRE.

References

[1] A. Gaikwad, I.M. Toke, GPU based sparse grid technique for solving multidimensional options pricing PDEs, in: Proceedings of the 2nd Workshop on High Performance Computational Finance, WHPCF'09, New York, NY, USA, ACM, 2009, pp. 6:1–6:9.
[2] J.C. Thibault, I. Senocak, CUDA implementation of a Navier–Stokes solver on multi-GPU desktop platforms for incompressible flows, in: 47th AIAA Aerospace Sciences Meeting, Orlando, FL, January 2010.
[3] D.B. Kirk, W.W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kaufman Publishers Inc.,, San Francisco, CA, USA, 2010.
[4] <http://www.nvidia.com/cudazone>. <http://developer.nvidia.com/getcuda> (CUDA toolkit).
[5] 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.
[6] M.P. Allen, D.J. Tildesley, Computer Simulation of Liquids, Oxford Science Publications, Oxford University Press, USA, 1989.

[7] P.J. Hoogerbrugge, J.M.V.A. Koelman, Simulating microscopic hydrodynamic phenomena with dissipative particle dynamics, Europhys. Lett. 19 (1992) 155.
[8] J.M.V.A. Koelman, P.J. Hoogerbrugge, Dynamic simulations of hard-sphere suspensions under steady shear, Europhys. Lett. 21 (1993) 363.
[9] D. Frenkel, B. Smit, Understanding Molecular Simulation, Academic Press Inc., Orlando, FL, USA, 2001.
[10] K. Huang, C. Lin, H. Tsao, Y. Sheng, The interactions between surfactants and vesicles: dissipative particle dynamics, J. Chem. Phys. 130 (24) (2009) 245101.
[11] M. Kenward, K.D. Dorfman, Brownian dynamics simulations of single-stranded DNA hairpins, J. Chem. Phys. 130 (9) (2009) 095101.
[12] C. Singh, A.M. Jackson, F. Stellacci, S.C. Glotzer, Exploiting substrate stress to modify nanoscale SAM patterns, J. Am. Chem. Soc. 131 (45) (2009) 16377–16379.
[13] C.R. Iacovella, C.L. Phillips, S.C. Glotzer, Stability of the double gyroid phase to nanoparticle polydispersity in polymer-tethered nanosphere systems, Soft Matter 6 (2010) 1693–1703.
[14] L. Gu, S. Xu, Z. Sun, J.T. Wang, Brownian dynamics simulation of the crystallization dynamics of charged colloidal particles, J. Colloid Interface Sci. 350 (2) (2010) 409–416.
[15] Steve Worley, Private communication, and Saru code package.
[16] D.E. Knuth, The art of computer programming, Seminumerical algorithms, third ed., vol. 2, Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA, 1997.
[17] W. Press, B. Flannery, S. Teukolsky, W. Vetterling, Numerical Recipes in C: The Art of Scientific Computing, Cambridge University Press, 1992.
[18] P. L'Ecuyer, R. Simard, Testu01: a C library for empirical testing of random number generators, ACM Trans. Math. Softw. 33 (4) (2007) 22.
[19] G. Marsaglia, A. Zaman, W.W. Tsang, Toward a universal random number generator, Stat. Probab. Lett. 9 (1) (1990) 35–39.
[20] M. Matsumoto, T. Nishimura, Dynamic creation of pseudorandom number generators, in: Monte Carlo and Quasi-Monte Carlo Methods, Springer, 1998, pp. 56–69.
[21] B.L. Holian, O.E. Percus, T.T. Warnock, P.A. Whitlock, Pseudorandom number generator for massively parallel molecular-dynamics simulations, Phys. Rev. E 50 (2) (1994) 1607–1615.
[22] H. Nguyen, GPU Gems 3, Addison-Wesley Professional, 2007.
[23] W.B. Langdon, A fast high quality pseudo random number generator for graphics processing units, in: IEEE Congress on Evolutionary Computation, CEC 2008, 2008, pp. 459–465.
[24] A. Zhmurov, K. Rybnikov, Y. Kholodov, V. Barsegov, Efficient pseudo-random number generators for biomolecular simulations on graphics processors, 2010. <arXiv:1003.1123v1[physics.chem-ph]>; A. Zhmurov, K. Rybnikov, Y. Kholodov, V. Barsegov, Generation of random numbers on graphics processors: forced indentation in silico of the bacteriophage HK97, J. Phys. Chem. B. 115 (18) (2011) 5278–5288.
[25] S. Tzeng, L. Wei, Parallel white noise generation on a GPU via cryptographic hash. I3D 2008, Association for Computing Machinery, Inc., October 2007.
[26] F. Zafar, A. Curtis, M. Olano, GPU random numbers via the tiny encryption algorithm, in: HPG 2010: Proceedings of the ACM SIGGRAPH/Eurographics Symposium on High Performance Graphics, June 2009.
[27] W. Liu, B. Schmidt, G. Voss, W. Müller-Wittig, Accelerating molecular dynamics simulations using graphics processing units with CUDA, Comput. Phys. Commun. 179 (9) (2008) 634–641.
[28] J. Davis, A. Ozsoy, S. Patel, M. Taufer, Towards large-scale molecular dynamics simulations on graphics processors, in: Sanguthevar Rajasekaran (Ed.), Bioinformatics and Computational Biology, Lecture Notes in Computer Science, vol. 5462, Springer, Berlin/ Heidelberg, 2009, pp. 176–186.
[29] D.C. Rapaport, Enhanced molecular dynamics performance with a programmable graphics processor, Comput. Phys. Commun. 182 (4) (2011) 926–934.
[30] A. Sunarso, T. Tsuji, S. Chono, GPU-accelerated molecular dynamics simulation for study of liquid crystalline flows, J. Comput. Phys. 229 (15) (2010) 5486–5497.
[31] M.S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A.L. Beberg, D.L. Ensign, C.M. Bruns, V.S. Pande, Accelerating molecular dynamic simulation on graphics processing units, J. Comput. Chem. 30 (6) (2009) 864–872.
[32] P. Eastman, V.S. Pande, Efficient nonbonded interactions for molecular dynamics on a graphics processing unit, J. Comput. Chem. 31 (6) (2010) 1268–1272.
[33] J.E. Stone, D.J. Hardy, I.S. Ufimtsev, K. Schulten, GPU-accelerated molecular modeling coming of age, J. Mol. Graph. Model. 29 (2) (2010) 116–125.
[34] D.L. Ermak, J.A. McCammon, Brownian dynamics with hydrodynamic interactions, J. Chem. Phys. 69 (4) (1978) 1352–1360.
[35] David Wheeler, Roger Needham, Tea, A Tiny Encryption Algorithm, Springer Verlag, 1995.
[36] R.D. Groot, T.J. Madden, D.J. Tildesley, On the role of hydrodynamic interactions in block copolymer microphase separation, J. Chem. Phys 110 (1999) 9739–9749 (This paper was shown incorrect by [37]).
[37] M.A. Horsch, Z. Zhang, C. R. Iacovella, S.C. Glotzer, Hydrodynamics and microphase ordering in block copolymers: are hydrodynamics required for ordered phases with periodicity in more than one dimension? J. Chem. Phys (2004).