Generation of random numbers on graphics processors: Forced indentation *in silico* of the bacteriophage $HK97$

A. Zhmurov[1,3], K. Rybnikov[2], Y. Kholodov[3] and V. Barsegov[1,3*]

[1]*Department of Chemistry, University of Massachusetts, Lowell, MA, 01854*

[2]*Department of Mathematics, University of Massachusetts, Lowell, MA, 01854*

[3]*Moscow Institute of Physics and Technology, Moscow region, Russia, 141700*

(Dated: September 22, 2010)

The use of graphics processing units (GPUs) in simulation applications offers a significant speed gain as compared to computations on central processing units (CPUs). Many simulations require generation of a large number of independent random variables at each step. We present two approaches for implementation of random number generators (RNGs) on the graphics processing unit (GPU). In the one-RNG-per-thread approach, one RNG produces a stream of random numbers in each thread of execution, whereas the one-RNG-for-all-threads method builds on the ability of different threads to communicate, thus, sharing random seeds across the entire GPU device. We used these approaches to implement Ran2, Hybrid Taus, and Lagged Fibonacci algorithms on the GPU. We profiled the performance of these generators in terms of the computational time, memory usage, and the speedup factor (CPU time/GPU time). These generators have been incorporated into the program for Langevin simulations of biomolecules fully implemented on the GPU. The ∼250-fold computational speedup on the GPU allowed us to carry out the single-molecule dynamic force measurements *in silico* to explore the mechanical properties of the bacteriophage $HK97$ in the experimental subsecond timescale using realistic force-loads. We found that the mechanical response of $HK97$ critically depends on the conditions of force application, including the rate of change and geometry of the mechanical perturbation. Hence, the GPU-based implementation of RNGs, presented here, in conjunction with Langevin simulations on the GPU makes it possible to compare directly the results of dynamic force measurements *in vitro* and *in silico*, and to interpret the experimental force spectra.

*Corresponding author; phone: 978-934-3661; fax: 978-934-3013; Valeri_Barsegov@uml.edu

I. INTRODUCTION

Graphics Processing Units (GPUs) are emerging as an alternative programming platform that provides high raw computational power for scientific applications [1–7]. The computational efficiency of contemporary GPUs reaching $\sim$1 TFlops for a single chip [8] enable one to utilize GPUs as performance accelerators in compute-intensive molecular simulations [1, 2, 6, 7]. The GPU-based calculations can be performed concurrently on many computational cores (Arithmetic Logic Units, ALUs) grouped into multiprocessors, each with its own flow control and cache units. For example, the number of multiprocessors per GPU can reach 15 on the most up-to-date graphics cards (e.g., GeForce GTX 480 from NVIDIA) bringing the total number of ALUs to 480 per chip. Although a GPU device has its own global memory with $\sim$10 times larger bandwidth compared to DRAM on a CPU, the number of memory invocations (per ALU) should be minimized to optimize the GPU performance. Hence, the computational task should be compute-intensive so that, most of the time, the GPU performs computations rather than reads/writes data [8]. This makes an N body problem into a prime candidate for the numerical implementation on the GPU.

Langevin Dynamics (LD) simulations, Monte Carlo (MC) simulations, and Molecular Dynamics (MD) simulations in implicit solvent, widely used to access the microscopic transitions in biomolecules, are among the many applications that can be implemented on a GPU. Since in MD simulations in implicit solvent (water) and in LD simulations the effect of solvent molecules is described implicitly, these methods require a reliable source of $3N$ normally distributed random numbers, $g_{i,\alpha}$ (i=1, 2, ..., N, and α=x, y, and z), generated on a GPU at each step of a simulation run, in order to compute the Gaussian random force $G_{i,\alpha}$. For example, in MD simulations in implicit water [9, 10], the dynamics of the i-th particle are governed by the equations of motion for the particle position, $d\mathbf{R}_i/dt$=$\mathbf{V}_i$, and velocity, $m_i d\mathbf{V}_i/dt$=$\xi\mathbf{V}_i$+$\mathbf{f}(\mathbf{R}_i)$+$\mathbf{G}_i(t)$, where m_i is the particle mass, ξ is the friction coefficient, and $\mathbf{f}(\mathbf{R}_i)$=$-\partial U/\partial\mathbf{R}_i$ is the molecular force exerted on the i-th particle due to the potential energy U. In LD simulations, the dynamics of the i-th C_α-particle are obtained by following the Langevin equation in the overdamped limit, $\xi d\mathbf{R}_i/dt$=$\mathbf{f}(\mathbf{R}_i)$+$\mathbf{G}_i(t)$ [11]. The equations of motion are propagated over many iterations of the same simulation algorithm. In MC simulations, the results of multiple trials, each driven by some random process, are combined to extract the average answer.

An algorithmic (pseudo-)random number generator (RNG), must have a long period and must meet the conflicting goals of being fast while providing a large amount of random numbers of proven statistical quality [12]. There is an extensive body of literature devoted to random

number generation on a CPU [13]. Yet, due to the fundamental differences in processor and memory architecture of the CPU and GPU devices, the CPU-based methods cannot be easily translated from the CPU to the GPU. One option is to have random numbers pre-generated on the CPU, and use these numbers in the simulations on the GPU. However, this requires a large amount of memory for an RNG. For a system of 10^4 particles in three dimensions, 3×10^4 random numbers are needed at each simulation step. To generate these numbers, say, every 10^2-10^3 steps requires $3\times10^6-3\times10^7$ random numbers, which takes $12-120MB$ of the GPU memory. This might be significant even for the most up-to-date GPUs, which have limited memory, e.g., GeForce GTX 200 series (NVIDIA) with $\sim1GB$ of memory.

We explored this option in Langevin simulations of N Brownian oscillators [11] using the Hybrid Taus and additive Lagged Fibonacci algorithms described below. We compared the computational time as a function of the system size N for three different implementations of Langevin simulations: (1) random numbers and dynamics are generated on the CPU, (2) random numbers, obtained on the CPU, are transfered to the GPU and used to generate dynamics on the GPU, and (3) random numbers and dynamics are generated on the GPU. The results for the $2.83GHz$ Intel Core i7 930 CPU and for the $1.4GHz$ GeForce GTX 480 GPU show that starting from $\approx10^2$ particles, it becomes computationally expensive to generate random numbers on the CPU and transfer them to the GPU in order to generate stochastic trajectories on the GPU (Fig. 1a). We found a substantial speedup for LD simulations fully implemented on the GPU, compared to the CPU-based implementation of the same LD algorithm, which also depends on the RNG choice and system size N (Fig. 1b). We observed a $\sim10-250$-fold speedup for Langevin simulations of $N=10^3-10^6$ Brownian particles on the GPU (Fig. 1b). Hence, for efficient molecular simulations in a stochastic thermostat, random numbers must be generated on the GPU device.

While there exist stand-alone implementations of RNGs on the GPU, to fully utilize computational resources of the GPU an RNG should be incorporated into the main simulation program. This allows to minimize read/write calls associated with invocation of the GPU global memory, and to generate streams of random numbers "on the fly", i.e. at each step of a simulation run, using fast GPU shared memory. Here, we describe the methodology for generating pseudo-random numbers on a GPU, which can be used in the GPU-based implementations of MD simulations in implicit solvent, LD simulations, and MC simulations of biomolecules. In the next Section, we focus on the Linear Congruential Generator (LCG), and the Ran2, Hybrid Taus, and Lagged Fibonacci algorithms. These algorithms are used in Section III to describe the methodology for the generation of (pseudo-)random numbers on the GPU. These RNGs have been incorporated

into the Langevin simulation program written in CUDA (dialect of C and C++ programming languages). Pseudocodes are given in the Supporting Information (SI). We test the GPU-based implementations of the LCG, Ran2, Hybrid Taus, and Lagged Fibonacci algorithms in Section IV, where we present the application-based assessment of their statistical properties using the Ornstein-Uhlenbeck process. We also profile these generators in terms of the computational time and memory usage. These algorithms and the C_α-based coarse-grained Self Organized Polymer (SOP) model [14–16] are used in Section V to perform single-molecule dynamic force measurements *in silico* to characterize the physical properties of the viral capsid $HK97$, a λ-like $dsDNA$ bacteriophage [17], under the experimental force-loading conditions. This is a model system for numerous studies of the kinetics of virus maturation, pressure-induced expansion, and the mechanism(s) of infection of cells [18, 19]. All the Langevin simulations were carried on the GPU GeForce GTX 480 (NVIDIA). The main results are discussed in Section VI. We conclude in Section VII.

II. PSEUDO-RANDOM NUMBER GENERATORS: THE GPU PERSPECTIVE

In this paper, we focus on algorithmic RNGs - the most common type of deterministic random number generators [20]. An RNG produces a sequence of random numbers, u_i, which is supposed to imitate independent uniform random variates from the unit interval $(0, 1)$. In implicit water models and in LD simulations of biomolecules, normally distributed random forces are used to emulate stochastic kicks from the solvent molecules. To generate the distribution of random forces, a common approach is to convert the uniformly distributed random variates (u_i) into the Gaussian distributed random variates (g_i) using a variety of methods [21–23]. Here, we adopt the most commonly used Box-Mueller transformation [23].

There are three main requirements for a numerical implementation of an RNG: (1) good statistical properties, (2) high computational speed, and (3) low memory usage. Because a deterministic sequence of random numbers comes eventually to a starting point, u_{n+p}=u_n (Poincaré recurrence), an RNG should have a long period p. An RNG must also be tested empirically for randomness, i.e., for the uniformity of distribution and for the independence [12]. The statistical tests of randomness are accumulated, e.g., in the DIEHARD test suite and in the TestU01 library [12, 24–26]. In addition, an RNG must pass application-based tests of randomness that offer exact solutions to the test applications. Using random numbers of poor statistical quality might result in insufficient sampling, unphysical correlations [27, 28], and unrealistic results, which might lead to errors in practical applications [29]. A good quality RNG should also

be computationally efficient so that random number generation does not become a major bottleneck. For example, in Langevin simulations of proteins on a GPU, one can follow a long trajectory over 10^{10} iterations. This requires $\sim 10^{15}$ random numbers for a system of $N{=}10^5$ particles. The condition of low memory usage is also important since contemporary graphics processors have low on-chip memory, $\sim 64KB$ per multiprocessor (graphics cards with Fermi architecture), compared to $\sim 2MB$ memory on the CPU. Hence, an efficient RNG algorithm must use a limited working area without invoking the relatively slow GPU global memory.

A fast RNG employs simple logic and a few state variables to store its current state, but this may harm its statistical properties. On the other hand, using a more sophisticated algorithm with many arithmetic operations or combining several generators into a hybrid generator allows to improve statistics, but such generators are slower and use more memory. Here, we focus on some of the most widely used algorithms, Linear Congruential Generator (LCG) [13], and Ran2 [13], Hybrid Taus [13, 20, 30, 31], and Lagged Fibonacci algorithms [13, 32], briefly reviewed in Appendix A. LCG can be used in performance benchmarks since it employs a very fast algorithm. Ran2 is a standard choice for many applications due to its long period $p{>}10^{18}$, good statistical quality, and high computational performance on the CPU. However, Ran2 requires a large amount of on-chip GPU local and global memory to store its current state. Hybrid Taus is a prime example of how several simple algorithms can be combined to improve the statistical characteristics of the random numbers produced. It scores better in terms of the computational speed on the GPU than KISS, the best known combined generator [33], and its long period $p{>}10^{36}$ makes it a good choice for the GPU-based computations. Lagged Fibonacci employs simple logic while producing random numbers of high statistical quality [12]. It is used in distributed MC simulations, and it can also be utilized in GPU-based computations. We employed the additive Lagged Fibonacci RNG, which generates floating point variates directly, without the usual floating of random integers.

III. LCG, RAN2, HYBRID TAUS, AND LAGGED FIBONACCI ON A GPU

A. Basic ideas

To solve an N body problem on a GPU, an RNG should produce random numbers simultaneously for all particles. One possibility is to build an RNG into the main simulation kernel to maximize the amount of computations on a GPU while minimizing the number of calls of the GPU global memory (read/write operations). To fully utilize the GPU resources, the total

number of threads should be ∼10-times larger than the number of computational cores, so that none of the cores awaits for the others to complete their tasks. Here, we employ the cycle division paradigm [32]. The idea is to partition a single sequence of random numbers among many computational threads running concurrently across an entire GPU device, each producing a stream of random numbers. Since most RNG algorithms are based on sequential transformations of the current state (LCG, Hybrid Taus and Ran2), the most common way of partitioning the sequence is to provide each thread with different seeds while also separating the threads along the sequence to avoid the interstream correlations. This is the basis of the one-RNG-per-thread approach (Fig. 1a in SI). On the other hand, Mersenne Twister and Lagged Fibonacci algorithms, which employ recursive transformations, allow one to leap ahead in a sequence and to produce the $(n+1)$-st random number without knowing the n-th number [32–34]. The leap size, which, in general, depends on parameters of an RNG, can be adjusted to the number of threads (number of particles N), or multiples of N ($M\times N$). Then, all N random numbers can be obtained simultaneously, i.e. the j-th thread produces numbers j, $j+N$, $j+2N\ldots$, etc. At the end of each simulation step, threads must be syncronized to update the current RNG state. Hence, the same RNG state can be used by all threads, each updating just one elements of the state. We refer to this as the one-RNG-for-all-threads approach (Fig. 1b in SI).

B. One-RNG-per-thread approach

The idea is to run the same RNG algorithm in each thread to generate different subsequences of the same sequence of random numbers, but starting from different initial seeds. The CPU initiates N sets of random seeds (one for each RNG) and passes them to the GPU global memory (Fig. 2 in SI). To exclude correlations, these sets should come from an independent sequence of random numbers. Each thread on the GPU reads its random seeds from the GPU global memory and copies them to the GPU local (per thread) memory or shared (per thread block) memory. Then, each RNG generates random numbers without using the slow GPU global memory. At the end of a simulation step, each RNG saves its current state to the global memory and frees shared memory. Since each thread has its own RNG, there is no need for threads synchronization. However, when particles interact, threads must be synchronized. In the simulations, arrays of initial seeds and the current state should be arranged for coalescent memory read to speedup the global memory access.

In the one-RNG-per-thread setting, an RNG should be very light in terms of the memory usage. Small size of on-chip memory can be insufficient to store the current state of an RNG

with complex logic. The amount of memory required to store the current state is proportional to the number of threads (number of particles N). Hence a significant amount of memory has to be allocated for all RNGs to describe a large system. For example, LCG uses one integer seed to store its current state, which takes 4 bytes per thread (per generator) or $\sim 4MB$ of memory for 10^6 threads/particles, whereas Hybrid Taus uses 4 integers, i.e. $16MB$ of memory. These are acceptable numbers, given hundreds of megabytes of the GPU memory. By contrast, Ran2 uses 35 long integers and a total of 280 bytes per thread, or $\sim 280MB$ of memory (for 10^6 threads/particles). As a result, not all seeds can be stored in on-chip (local or shared) memory ($\sim 64KB$ on GPUs with Fermi architecture), and the GPU global memory has to be accessed to read/update the current state. In addition, less memory becomes accessible to other computational routines. This might prevent Ran2 from being used in the simulations of large systems on some graphics cards, including GeForce GTX 280 and GTX 295 (NVIDIA), with $768MB$ of global memory (per GPU). Yet, this is not an issue when using high-end graphics cards, such as Tesla C2070 with $6GB$ of global memory. In this paper, we utilized the one-RNG-per-thread approach to develop the GPU-based implementations of the LCG, Hybrid Taus and Ran2 algorithms (Fig. 2 in SI). Pseudocodes are presented in Section I in SI. Numerical values of the constant parameters for LCG, Ran2, and Hybrid Taus, can be found, respectively, in Appendix A [12], in Ref. [13], and in Section I in SI.

C. One-RNG-for-all-threads approach

In the one-RNG-for-all-threads approach, one can utilize a single RNG by allowing all computational threads to share the state of a generator. This can be used in algorithms that are based on the recursive transformations, i.e. $x_n{=}f(y_{n-r}, y_{n-r+1}, \ldots y_{n-k})$, where r is a recurrence degree and $k{>}r$ is a constant parameter, to obtain a random number at the n-th step from the state variables generated at the previous steps $n{-}r$, $n{-}r{+}1$, ..., $n{-}k$. If a sequence of random numbers is obtained simultaneously in N threads, each generating just one random number, then N random numbers are produced at each step. Given $k{>}N$, all the elements of the transformation have been obtained at the previous steps, in which case they can be accessed without threads synchronization. One of the algorithms that can be implemented on the GPU using the one-RNG-for-all-threads approach is additive Lagged Fibonacci (Fig. 3 in SI) [34]. A pseudocode is presented in Section II in SI. When one random number is computed in each thread and when $sl{>}N$ and $ll{-}sl{>}N$, where ll and sl are the long and short lags, N random numbers can be obtained simultaneously on the GPU device; sl and ll could be taken to be

sufficiently large to guarantee good statistical properties of the random numbers produced.

To initialize the Lagged Fibonacci RNG on the GPU, ll integers are allocated on the CPU. On the GPU, each thread reads two integers of the sequence (one for ll and the other for sl), generates the resulting integer, and saves it to the location in the GPU global memory, which corresponds to ll. Setting $sl{>}N$ and $ll{-}sl{>}N$ guarantees that the same position in the array of integers (current state variables) will not be accessed by different threads at the same time. The moving window of N random numbers, updated in N threads is moving along the array of state variables, leaping forward by N positions at each step. Importantly, a period of the Lagged Fibonacci generator, $p{\sim}2^{ll+31}$, can be adjusted to the system size N by assigning large values to sl and ll, so that $p{\gg}N{\times}S$, where S is the number of simulation steps. Varying ll and sl does not affect the execution time, but changes the size of the array of state variables, which scales linearly with ll, the amount of integers stored in the GPU global memory. Large ll values is not an issue even when $ll{\sim}10^6$, which corresponds to $\sim 4MB$ of the GPU global memory. Numerical values of the constant parameters for Lagged Fibonacci are given in Table I in SI.

IV. BENCHMARK TESTING

A. Test of randomness: Ornstein-Uhlenbeck process

To assess the statistical performance of the GPU-based realizations of LCG, Ran2, Hybrid Taus, and Lagged Fibonacci, we carried out Langevin simulations of N independent Brownian oscillators [11] on the GPU. Each particle evolves on the harmonic potential, $U(R_i){=}k_{sp}R_i^2/2$, where R_i is the i-th particle position and k_{sp} is the spring constant, and is subject to random force. We employed this analytically tractable model to compare directly the simulation output with the exact results that would be obtained with truly random numbers. The Langevin dynamics $\xi dR_i/dt{=}{-}\partial U(R_1,R_2,\ldots,R_N)/\partial R_i + G_i(t)$, widely used in the simulations of biomolecules [14–16, 35, 36], were obtained numerically using the first-order integration scheme [37],

$$R_i(t+\Delta t) = R_i(t) + f(R_i(t))\Delta t/\xi + g_i(t)\sqrt{2k_BT\xi/\Delta t}, \tag{1}$$

where $f(R_i){=}{-}\partial U(R_1,R_2,\ldots,R_N)/\partial R_i$ is the deterministic force, and g_i are the normally distributed random variates (with zero mean and unit variance) used to obtain the Gaussian random forces $G_i(t){=}g_i(t)\sqrt{2k_BT\xi/\Delta t}$. Numerical algorithms for the GPU-based implementation of Langevin simulations, i.e. evaluation of forces and numerical integration of the equations of motion (Eq. (1)), are presented in Ref. [38].

Numerical calculations for N=10^4 particles were carried out with the time step Δt=$1ps$ at room temperature T=$300K$, starting from the initial position R_0=$10nm$, and using the diffusion constant D=$0.25nm^2/ns$. A soft harmonic spring (k_{sp}=$0.01pN/nm$) allowed us to generate long $1ms$ trajectories over 10^9 steps. The average position $\langle R(t)\rangle$ and the two-point correlation function $C(t)$=$\langle R(t)R(0)\rangle$, obtained from the simulations, are compared in Fig. 2 with their exact counterparts [11, 39], $\langle R(t)\rangle$=$R_i(0)\exp[-t/\tau]$ and $C(t)$=$(k_BT/k_{sp})\exp[-t/\tau]$, where τ=ξ/k_{sp} is the characteristic time. We see that all RNGs describe well the exact Brownian dynamics except for the LCG. Indeed, $\langle R(t)\rangle$ and $C(t)$, obtained using Ran2, Hybrid Taus, and Lagged Fibonacci RNGs, practically collapse on the theoretical curve of these quantities. By contrast, using LCG results in a repeated pattern for $\langle R(t)\rangle$ and in the unphysically short-lived correlations in $C(t)$. At longer times, $\langle R(t)\rangle$ and $C(t)$, obtained from simulations, deviate from the theoretical curves due to a soft harmonic spring and insufficient sampling (Fig. 2).

B. Computational performance

We benchmarked the computational efficiency of the GPU-based realizations of the Ran2, Hybrid Taus, and Lagged Fibonacci algorithms using Langevin simulations of N Brownian oscillators in three dimensions. For each system size N, we ran one trajectory for 10^6 simulation steps. All N threads were synchronized at the end of each step to emulate an LD simulation run of a biomolecule on a GPU. The execution time and memory usage are profiled in Fig. 3. We find that Ran2 is the most demanding generator. The use of Ran2 in Langevin simulations of a system of N=10^4 particles requires additional $\sim$264 hours of wall-clock time to obtain a single trajectory over 10^9 steps. The memory demand for Ran2 is quite high, i.e. $>250MB$ for N=10^6 (Fig. 3b). Because in biomolecular simulations a large memory area is needed to store parameters of the force field, Verlet lists, interparticle distances, etc., the high memory demand might prevent one from using Ran2 in the simulations of a large system. Also, implementing Ran2 in Langevin simulations on the GPU does not lead to a substantial speedup (Fig. 3a). By contrast, Hybrid Taus and Lagged Fibonacci RNGs are both light and fast in terms of the memory usage and the execution time (Fig. 3). These generators require a small amount of memory, i.e. $<15-20MB$, even for a large system of N=10^6 particles (Fig. 3b).

In general, the number of memory calls scales linearly with N. Because on a GPU the computational speed even of a fast RNG is determined mostly by the number of memory calls, multiple reads/writes from/to the GPU global memory can prolong significantly the computational time. We profiled the LCG, Ran2, Hybrid Taus, and Lagged Fibonacci RNGs, which use, respectively,

1, 40, 4, and ∼3 state variables per thread, in terms of the number of memory calls per simulation step. The state size for Lagged Fibonacci depends on the choice of ll and sl (Appendix A). The LCG, Hybrid Taus and Lagged Fibonacci use 4−16 bytes/thread, which is quite reasonable even for a large system of N=10^6 particles. However, Ran2 requires 280 bytes/thread which is significant for a large system (Table I). Since Ran2 has large size of the state, saving/updating its current state using the GPU local or shared memory is not efficient computationally. Also, Ran2 employs long 64-bit variables, which doubles the amount of data (memory), and requires 4/4 read/write memory calls (7/7 read/write calls are needed to generate 4 random numbers). Hybrid Taus uses the GPU global memory only when it is initialized, and when it updates its current state. Since it uses 4 state variables, 4/4 read/write calls per thread are required regardless of the amount of random numbers produced (Table I). The Lagged Fibonacci RNG uses 2 random seeds, which results in 2/1 read/write calls per random number (8/4 read/write calls for 4 random numbers). The execution time for Hybrid the Taus and Lagged Fibonacci RNGs scales sublinearly with N (i.e. remains constant) for N<10^4 particles due to insufficient parallelization of the GPU device, but grows linearly with N for larger systems when all ALUs on the GPU become fully subscribed (Fig. 4). It takes about the same time to generate random numbers using these generators and to propagate Langevin dynamics to the next step (Fig. 4). This is a high performance level given the fact that the potential function does not involve long-range interactions.

V. FORCED INDENTATION *IN SILICO*: BACTERIOPHAGE $HK97$

We employed the Hybrid Taus and additive Lagged Fibonacci generators to develop the GPU-based implementation of Langevin simulations using a C_α-based Self Organized Polymer (SOP) coarse-grained model of the protein chain. All the steps of the algorithm, fully implemented on the GPU, have been converted into a standard CUDA code (SOP-GPU package) [38]. The SOP model (Appendix B [14]) describes well the mechanical properties of proteins, including the Green Fluorescent Protein [40], the tubulin dimer [41], and kinesin [42]. This model has also been used to explore the kinetics and to resolve the free energy landscape of tetrahymena ribozyme [14], riboswitch aptamers [43], GroEL [44], protein kinase A [45], and myosin V [46]. We used the SOP model and Langevin simulations on the GPU to carry out single-molecule dynamic force measurements *in slilco* of the mechanical indentation of the bacteriophage $HK97$. All-atom MD simulations cannot be used to resolve the micromechanics of supramolecular biological assemblies under the experimentally relevant force-loads (v_f≈0.1−10$\mu m/s$) in the experimental subsecond

timescale [38, 47]. Besides, topology and the arrangement of the secondary structure elements into the overall structure, rather than the atomic details, govern the large-scale conformational transitions [38, 48].

Bacteriophage $HK97$ (115, 140 residues) [18] is made of 420 copies of the $gp5$ protein [17] and is formed by 60 icosahedral units, each composed of 7 domains A through G. Domains $A-F$ form 60 hexamers, and domain G binds to 5 G chains to form 12 pentamers. Each subunit is joined to two of its neighbors by ligation of $Lys169$ to $Asp356$, which results in formation of the topologically linked protein rings (catenanes). The capsid outer radius is $X\approx 32nm$ and the average wall thickness is $\Delta X\approx 2.1nm$ (in the head II state). The $HK97$ varus maturation involves pressure-induced capsid expansion due to the $dsDNA$ packaging [18]. Yet, the mechanical properties of this infectious agent have not been investigated neither experimentally nor computationally. We probed the mechanical reaction of the bacteriophage $HK97$ in the head II state (Protein Data Bank code: 2FT1) by indenting it with the time-dependent force $f_{ext}(t)=r_f t$, where $r_f=\kappa v_f$ is the force-loading rate, and κ and v_f are, respectively, the cantilever spring constant and the tip velocity. We analyzed the dependence of the physical properties of $HK97$ on the rate of change r_f and geometry of mechanical perturbation. The effect of geometry was studied using the spherical tip of different radius R.

We generated the force-indentation curves (FZ curves), which quantify the mechanical response of $HK97$ as a function of the distance traveled by the cantilever Z [49, 50]. We varied the speed and radius of the cantilever tip by setting $v_f=2.5\mu m/s$ (in the experimental range [51]), $25\mu m/s$, and $250\mu m/s$, and using $R=5nm$, $10nm$, and $25nm$. For each set of values of v_f and R, we generated 3 force-indentation curves at room temperature ($T=300K$) with the time step $\Delta t=20ps$ using the bulk water viscosity, which corresponds to the friction coefficient $\xi=7.0\times 10^5 pNps/nm$. It took $\sim$34 days (10^9 steps) and $\sim$3.4 days (10^8 steps) of wall-clock time to generate a singe indentation trajectory of length $20ms$ and $2ms$ for $v_f=2.5$ and $25\mu m/s$, respectively, on the GPU (GeForce GTX 480). For comparison, it would take $\sim$120 and $\sim$12 months, respectively, to complete the same jobs on the CPU Intel Core i7 930. The typical FZ curves, the number of native contacts Q, and the capsid spring constant K are displayed in Fig. 5. We estimated the values of K, which quantifies the elastic component of the mechanical response of $HK97$, using the formula $1/K_{FZ}=1/\kappa+1/K$ [49, 52] for the spring constant for the combined system (capsid plus tip), K_{FZ}, extracted from the FZ curves.

The mechanical response of $HK97$ shows stochastic variation at a slow force-load $v_f=2.5\mu m/s$ (Fig. 5a), but it becomes more "deterministic" when v_f is increased (Fig. 5b and c). The slope of the FZ curves (proportional to K), while increasing with v_f and R, fluctuates for all values of

v_f and R used, which implies that the capsid elasticity is a dynamic, rather than static, property. Interestingly, K increases from $0.01-0.025pN/nm$ at v_f=$2.5\mu m/s$ to $0.05-0.075pN/nm$ at v_f=$25\mu m/s$, and to $0.2-0.35pN/nm$ at v_f=$250\mu m/s$ (Table II). This result demonstrates that the capsid wall becomes stiffer when indented faster, and implies that its elasticity also depends on the geometry of a force-bearing load. The buckling transitions were observed only at the slowest force-load of v_f=$2.5\mu m/s$ (Fig. 5a) when a large $25nm$ tip was used. When the capsid buckles, K first increases and then dicreases with Z while the number of native contacts Q, stabilizing the virus shell structure, decreases monotonically with Z. The buckling transitions set in at Z≈$25nm$, at which point K drops to zero signifying a loss of mechanical resistance (Fig. 5a). At v_f=$2.5\mu m/s$ and for R=$10nm$, the indentation is monotonic (no buckling); however, at v_f=$2.5\mu m/s$ and for R=$5nm$, the indentation continues up to Z≈$55nm$, at which point the mechanical fracture occurs (Fig. 5a). The local fracture around the area where the tip penetrates the viral shell is associated with partial unfolding and disruption of some of the native contacts, which is reflected in the sudden drop in Q (Fig. 5a). The native contacts form again as the tip passes through the capsid wall, resulting in the increase in Q, which implies that the fracture is reversible.

The buckling transitions were not detected at the faster loads v_f=$25\mu m/s$ and $250\mu m/s$ (Figs. 5b and c). At v_f=$25\mu m/s$, the dependence of the mechanical reaction on Z is monotonic only for a large $10nm$ and $25nm$ tip. For a small $5nm$ tip, a gradual indentation is interrupted by the capsid fracture at Z≈$65nm$. This results in a loss of capsid elasticity (Fig. 5b), which is reflected in a sudden drop in Q due to the disruption of the native contacts, and in the decrease of K to zero (Fig. 5b). At v_f=$250\mu m/s$, we obtained the monotonic FZ curves only when the capsid was indented with a large tip (R=$25nm$). Indenting with smaller $5nm$ and $10nm$ tips resulted in the capsid fracture at Z≈$65nm$ and Z≈$120nm$, respectively (Fig. 5c). The structural damage was localized to the residue positions affected by the tip moving downward, and the recovery of the native contacts was partial for R=$10nm$ and full for R=$5nm$ (Fig. 5c). The structural analysis of bacteriophage $HK97$ has revealed that the ratio of the wall thickness to the outer radius is $\Delta X/X$≈0.065≪1. This allowed us to use the thin-shell approximation to connect the spring constant K with the Young's modulus Y, using the formula K=$\alpha Y\Delta X^2/X$, where α is the proportionality factor [53]. Assuming that α≈1, we estimated the modulus Y, which characterizes the "in-plane" elasticity of the viral shell. We also evaluated the energy costs for the structural damage (spherical cavity) ΔE_f and for the buckling ΔE_b, and calculated the critical pressure p_c=f_c/A, where A is the contact area on the capsid outer surface impacted by the cantilever tip. The numerical values of Y, p_c, ΔE_b, and ΔE_f are accumulated in Table II.

VI. DISCUSSION

A. Choosing RNG for GPU-based computations

Random number generators are used in many computer applications such as simulations of stochastic systems, probabilistic algorithms, and numerical analysis among many others. The highly parallel architecture of the GPU provides an alternative computational platform that allows one to utilize multiple ALUs on a single processor. This comes at a price of having smaller cache memory and reduced flow control. Hence, to harvest raw computational power offered by the GPU, one needs to re-design computational algorithms that have been used on the CPU for many decades. Here, we described two general methods for generating pseudo-random numbers on the GPU.

In the one-RNG-per-thread approach, the same RNG algorithm is executed in each computational thread (for each particle), a procedure used in the CPU-based methods. In the one-RNG-for-all-threads setting, one can utilize the ability of different threads to communicate across the entire GPU device. We employed these methods to develop the GPU-based realizations of the Ran2, and Hybrid Taus generators (Fig. 2 in SI), and the additive Lagged Fibonacci RNG (Fig. 3 in SI). The Hybrid Taus and Lagged Fibonacci generators provide random numbers at a computational speed almost equal to that of the LCG, and the associated memory demand is rather low (Fig. 3). The long period of these RNGs is sufficient to describe the dynamics of a very large system ($N{>}10^6$ particles) on a long timescale ($>10^9$ steps). Ran2 is a well tested generator of proven statistical quality [13], but it works only $\sim$10$-$15-times faster on the GPU and requires a large memory area (Fig. 3). By contrast, employing the Hybrid Taus and Lagged Fibonacci algorithms results in an impressive 200$-$250-fold speedup for a large system of as many as 10^6 particles (Fig. 1).

As an application-based test of randomness, we carried out Langevin simulations of N Brownian oscillators (Ornstein-Uhlenbeck process). We found an excellent agreement between the stochastic trajectories, obtained analytically and computationally by using the Hybrid Taus, Ran2 and Lagged Fibonacci algorithms (Fig. 2). We also applied some stringent statistical tests to access the statistical properties of random numbers produced using the developed GPU-based implementation of the Hybrid Taus and Lagged Fibonacci RNGs. We found that the Hybrid Taus RNG does not fail a single tests in the DIEHARD test suite [24] and passes both the BigCrush battery and the SmallCrush battery of tests in the TestUO1 package [12]. The Lagged Fibonacci RNG, even with a small short lag sl=1252, does not fail any test in the

DIEHARD test suite, and passes the BigCrush in the TestU01 package. We recommend these generators for Langevin simulations, for Monte Carlo simulations, and for MD simulations in implicit solvent of large biomolecular systems. Given their high statistical quality, these RNGs is a reasonable choice for the GPU-based implementations of molecular simulations. These RNGs can also be used in parallel tempering algorithms, including variants of the Replica Exchange method. Using the Hybrid Taus algorithm results in a faster acceleration, compared to the Lagged Fibonacci generator, but the latter can be ported to the GPU with MIMD architecture (Multiple Instruction Multiple Data).

B. Dynamic signatures of the force-indentation spectra

Streered Molecular Dynamics (SMD) simulations are currently limited to a $10-50nm$ length scale and $0.1-10\mu s$ duration [1, 54, 55]. Hence, it is virtually impossible to span the experimental millisecond timescale using SMD simulations. For example, it takes $800,000$ CPU hours to obtain $1ns$ MD trajectories for the southern bean mosaic virus (4.5×10^6 atoms) on an SGI Altix cluster [47]. Computational approaches based on the elastic network normal mode analysis allow mostly for the theoretical exploration of equilibrium properties of biomolecules [56, 57]. We utilized the structure-based coarse-grained description of proteins, where each residue position is specified by a single interaction center (C_α-atom), to carry out single-molecule forced indentation experiments *in silico* of the bacteriophage $HK97$. The hundred-fold computational acceleration achieved on the GPU (GeForce GTX 480), compared to the heavily tuned CPU version of the same program (Fig. 1), allowed us to explore the physical properties of this larga-size supramolecular biological assembly (10^5 particles) in the subsecond timescale [38]. We used the experimental force-loading conditions, employed in the AFM based dynamic force measurements (force-ramp), including the cantilever spring constant κ, and the spherical tip velocity v_f and size R. We found that the microscopic mechanical response of the virion $HK97$ depends rather sensitively on the rate (v_f) and geometry (R) of the force application.

We observed a whole spectrum of biomechanical reactions for $HK97$ in the far-from-equilibrium regime from the gradual indentation at low and moderately high forces to buckling at the intermediate forces, and to the mechanical fracture at high forces (Fig. 5). These dynamic signatures in the theoretical force spectra might reflect the general physical properties, shared by many virus shells. We found that virus shell elasticity is a fluctuating dynamic property, rather than an average static characteristic, which also varies with the rate of change of the mechanical perturbation. The spring constant of $\approx 0.01-0.02N/m$ for the bacteriophage $HK97$,

obtained at the experimental pulling speed v_f=2.5$\mu m/s$ used in the AFM based dynamic force spectroscopy, agrees with the experimental estimates of this parameter for empty viral shells [52]. Our finding that K might also change with size of the load-bearing tip implies that the spring constant of a virus shell K, reported in the experimental AFM studies, is a local characteristic. Indeed, the larger the tip the more structural units cooperate to withstand the external mechanical perturbation. Hence, larger tips comparable in size with the dimensions of the viral shell in question should be used to average over local variations in the mechanical response. In addition, K might vary depending on where on the shell surface the tip presses against the virus shell [47], but we leave this important aspect for future studies.

We found that a temporary loss of the elastic response of a virus shell, when K rapidly decreases to zero, might occur as a result of buckling transition or mechanical fracture. In the event of buckling, the capsid shell rapidly regains its elasticity, which results in the subsequent increase of K. A sudden drop in K indicates, rather, the onset of the mechanical fracture due to the structural damage associated with partial disruption of the network of native contacts, which results in the local unfolding transitions. This process is reversible, as the native contacts tend to reform soon after the cantilever tip has passed through the capsid wall (Fig. 5). These results agree well with the experimental observations on other virions [49, 52]. We did observe the expected crossover from purely elastic behavior at low forces to plastic behavior at higher forces [47], which also follows from an observation that K tends to decrease at longer Z values, but this effect is not well-pronounced, which might be due, in part, to presence of the topological links [58]. This rare feature of the molecular architecture of bacteriophage $HK97$ adds to the structural integrity and enhances the elastic component of this supramolecular assembly. In fact, the observed sudden drops in Q were mostly due to the disruption of the intracapsomer native contacts, which stabilize the native folded state of the capsid structural units, rather than the intercapsomer contacts, which mediate the capsomer-capsomer protein interactions. Detailed analysis of the transient structures, obtained in the course of the force-driven indentation of bacteriophage $HK97$, and structural underpinnings underlying the mechanical failure will be presented in a separate paper.

The onset of mechanical failure (buckling) is controlled by a universal physical characteristic - the Foppl-von Kármán (FvK) number γ [59]. For a thin spherical shell, it is defined as γ=YX^2/k, where k is the "out-of-plane" bending modulus. In general, for a buckling transition to occur in a shell of fixed radius X, the ratio of the extent of "in-plane" stretching (YX^2) and the degree of "out-of-plane" bending (k) must be large so that γ exceeds some critical value $\sim 10^3$ [59]. The results obtained for v_f=2.5$\mu m/s$ show that the buckling regime sets in when

the capsid is indented with the tip comparable with the capsid size, i.e. $R{\sim}X$ (Fig. 5a). In this case, the tip pushing downward excites mostly the in-plane stretching degrees of freedom, and $YX^2{\gg}k$. On the other hand, the fracture occurs when smaller tips are used, i.e. when $R{<}X$. Here, the tip motion excites the out-of-plane bending modes, and $YX^2{\ll}k$. Hence, both dynamic regimes can be accessed by controling the geometry of the force application. In addition, the results obtained for the faster force-loads (v_f=25 and 250$\mu m/s$) indicate strongly that whether the mechanical failure (buckling or fracture) occurs also depends on the rate of change of the applied force $f(t)$ (Fig. 5b and c). Hence, theoretical models should be extended to account for dynamic coupling of the in-plane modes and the out-of-plane modes of motion and for the far-from-equilibrium conditions of propagation and distribution of the mechanical stress on the spherical surface.

The Young's modulus Y was also found to depend on the rate of change and geometry of the pushing force (Table II). At the experimental value of v_f=2.5$\mu m/s$, the modulus Y=60$-$160MPa for $HK97$ is comparable with Y=140MPa for the empty CCMV virus capsid [52], but is less than Y=1.8GPa for the bacteriophage ϕ29 [49] (Table II). The empty shell $HK97$ is capable of withstanding the mechanical pressures of the order of 60$-$140atm, which is comparable with effective pressure inside the pacteriophage ϕ29 due to DNA packaging. These results show that dynamic force assays *in silico*, carried out under the experimentally relevant force-loads, can be used to explore the limits of the elasticity of virus shells and to estimate the maximum internal pressure due to the encapsulated genetic material. We found that with the tip-sphere moving at v_f=2.5$\mu m/s$, the energy it would cost to create a structural damage, i.e. a spherical cavity (of radius 5nm) on the outer surface of the capsid $HK97$, or the energy for fracture, ΔE_f=2.7$\times 10^{-17} Nm$, is about twice the energy required to buckle the capsid, ΔE_b=1.4$\times 10^{-17} Nm$. Interestingly, for the same cavity size, the energy for fracture grows with the rate of change of the applied force (Table II).

VII. CONCLUSION

The development of new Fermi architecture (NVIDIA) and Larrabee architecture (Intel), is an important step for general purpose GPU computing. The high speed interconnection network will provide a fast interface for threads communication. These advances will enable the programmer to distribute a computational workload among the many cores on the GPU more efficiently, and to reach an even higher performance level. In this regard, the developed GPU-based implementation of additive Lagged Fibonacci RNG can be ported to new graph-

ics processors with minor modifications. In a context of biomolecular simulations, this will make it possible to compute random forces using the much needed synchronization of threads over the entire GPU device. This makes the one-RNG-for-all-threads method of generation of pseudo-random numbers on the GPU, where the thread synchronization is utilized, all the more important. This method can also be used to develop the GPU-based implementations of the Mersenne Twister algorithm [60], and several other algorithms, including multiple recursive generator (MRG) and linear/generalized shift feedback register (LSFR/GSFR) generators, such as 4-lag Lagged Fibonacci algorithm [12, 33].

The developed GPU-based realizations of the Hybrid Taus and Lagged Fibonacci generators enable one to carry out Langevin simulations of large-size supramolecular assemblies in the experimental subsecond timescale. The presented formalism can be applied to study the biomechanical reactions in a range of biological systems, including molecular motors, nucleosomes, and proteasomes among many others. Understanding the micromechanical properties of nanometer-scale protein shells of plant and animal viruses is important for virus biology, materials engineering, and for nanotechnological applications. Here, we utilized the Self Organized Polymer (SOP) model and Langevin simulations, fully implemented on the GPU (SOP-GPU package), to perform the dynamic force spectroscopic measurements *in silico* of the mechanical properties of the bacteriophage $HK97$ using realistic conditions of force application. These computer experiments mimic the AFM based dynamic force measurements on biomolecules *in vitro* [49, 51, 52, 61, 62], thus, allowing for direct comparison of the simulation output with the experimental data. The observed dynamic signatures for continuous transitions (indentation), phase transitions (buckling), and structural failure (fracture) can be used to provide meaningful interpretation of the force peaks and kinks in the experimental force-indentation curves.

Acknowledgements: Acknowledgement is made to the donors of the American Chemical Society Petroleum Research Fund (grant PRF #47624−$G6$) for partial support of this research (VB). This work was also supported in part by the grant (#09−0712132) from the Russian Foundation for Basic Research (VB, YK and AZ).

Appendix A: LCG, Ran2, Hybrid Taus and Lagged Fibonacci algorithms

LCG: The Linear Congruential Generators (LCGs) use a transitional formula,

$$x_n = (ax_{n-1} + c) \bmod m, \tag{A1}$$

where m is the maximum period, and a=1664525 and c=1013904223 are constant parameters [13]. To produce a uniformly distributed random number, x_n is divided by 2^{32}. Assuming a 32-bit integer, the maximum period can be at most p=2^{32}, which is far too low. LCGs also have known statistical flows [12]. If m=2^{32}, one can neglect mod m operation as the returned value is low-order 32 bits of the true 64-bit product. Then, the transitional formula reads x_n=ax_{n-1}+c, which is the so-called Quick and Dirty or ranqd2 generator (simplified LCG). Quick and Dirty LCG is very fast as it takes only a single multiplication and a single addition to produce a random number, and it uses one integer to describe its current state.

Ran2: Ran2, one of the most popular RNGs, combines two LCGs and employs randomization using some shuffling procedure [13]. Ran2 has a long period and provides random numbers of very good statistical properties [12]. It is one of a very few generators that does not fail a single statistical test. Ran2 is reasonably fast, but it involves long integer arithmetic (64-bit logic) - a computational bottleneck for contemporary GPUs, and it requires a large amount of memory to store its current state.

Hybrid Taus: Hybrid Taus [20] is a combined generator that uses LCG and Tausworthe algorithms. Tausworthe taus88 is a fast equidistributed modulo 2 generator [30, 31], which produces random numbers by generating a sequence of bits from a linear recurrence modulo 2, and forming the resulting number by taking a block of successive bits. In the space of binary vectors, the n-th element of a vector is constructed using the linear transformation,

$$y_n = a_1 y_{n-1} + a_2 y_{n-2} + \ldots a_k y_{n-k}, \tag{A2}$$

where a_n are constant coefficients. Given initial values, $y_0, y_1, \ldots y_{n-1}$, the n-th random integer is obtained as x_n=$\sum_{j=1}^{L} y_{ns+j-1} 2^{-j}$, where s is a positive integers and L=32 is the integer size (machine word size). Computing x_n involves performing s steps of the recurrence, which might be costly computationally. Fast implementation can be obtained for a certain choice of parameters: when a_k=a_q=a_0=1, where 0<2q< k and a_n=0 for 0<s≤k−q<k≤L, the algorithm can be simplified to a series of binary operations [31]. Statistical characteristics of random numbers produced using taus88 alone are poor, but combining taus88 with LCG removes all the statistical defects [20]. In general, statistical properties of a combined generator are better than those of its components. When periods of all components are co-prime numbers, a period of a combined generator is the product of periods of all components. A similar approach is used in the KISS generator [33]. However, multiple 32-bit multiplications, used in KISS, might harm its performance on the GPU. The period of the Hybrid Taus is the lowest common multiplier of the periods of three Tausworthe steps and one LCG. We used parameters that result in the

periods $p_1{\approx}2^{31}$, $p_2{\approx}2^{30}$, and $p_3{\approx}2^{28}$ for the Tausworthe generators and the period $p_4{=}2^{32}$ for the LCG, which makes the period of the combined generator equal $\sim 2^{121}{>}10^{36}$. Hybrid Taus uses small memory area since only four integers are needed to store its current state.

Lagged Fibonacci: The Lagged Fibonacci algorithm is defined by the recursive relation,

$$x_n = f(x_{n-sl}, x_{n-ll}) \bmod m, \tag{A3}$$

where sl and ll are the short lag and the long lag, respectively ($ll{>}sl$), m defines the maximum period and f is a function that takes two integers x_{n-sl} and x_{n-ll} to produce integer x_n. The most commonly used functions are multiplication, $f(x_{n-sl}, x_{n-ll}){=}x_{n-sl}{*}x_{n-ll}$ (multiplicative Lagged Fibonacci), and addition, $f(x_{n-sl}, x_{n-ll}){=}x_{n-sl}{+}x_{n-ll}$ (additive Lagged Fibonacci). Random numbers are generated from the initial set of ll integer seeds. To achieve the maximum period $\sim 2^{ll-1}{\times}m$, the long lag ll should be set equal the base of a Mersenne exponent, and the short lag sl should be taken so that the characteristic polynomial $x^{ll}{+}x^{sl}{+}1$ is primitive. Also, sl should not be too small nor too close to ll. It is recommended that $sl{\approx}\rho{\times}ll$, where $\rho{\approx}0.618$ [12]. When single precision arithmetic is used, the mod m operation can be omitted by setting $m{=}2^{32}$.

Appendix B: Self Organized Polymer (SOP) model

We adapted the Self Organized Polymer (SOP) model [14–16], where each residue is described using a single interaction center (C_α-atom). The potential energy function of a protein conformation U_{SOP}, specified in terms of the coordinates $\{r\}{=}r_1, r_2, \ldots, r_N$, is given by

$$\begin{aligned} U_{SOP} &= U_{FENE} + U_{NB}^{ATT} + U_{NB}^{REP} = \\ &- \sum_{i=1}^{N-1} \frac{k}{2} R_0^2 \log\left(1 - \frac{\left(r_{i,i+1} - r_{i,i+1}^0\right)^2}{R_0^2}\right) + \sum_{i=1}^{N-3}\sum_{j=i+3}^{N} \varepsilon_n \left[\left(\frac{r_{ij}^0}{r_{ij}}\right)^{12} - 2\left(\frac{r_{ij}^0}{r_{ij}}\right)^{6}\right]\Delta_{ij} \\ &+ \sum_{i=1}^{N-2}\sum_{j=i+2}^{N} \varepsilon_r \left(\frac{\sigma_{ij}}{r_{ij}}\right)^6 + \sum_{i=1}^{N-3}\sum_{j=i+3}^{N} \varepsilon_r \left(\frac{\sigma}{r_{ij}}\right)^6 (1 - \Delta_{ij}). \end{aligned} \tag{B1}$$

In Eq. (B1), the finite extensible nonlinear elastic (FENE) potential U_{FENE} with the spring constant $k{=}14N/m$ describes the backbone chain connectivity. The distance between residues i and $i{+}1$ is $r_{i,i+1}$, $r_{i,i+1}^0$ is its value in the native (PDB) structure, and $R_0{=}2$Å is the tolerance in the change of a covalent bond distance. We used the Lennard-Jones potential U_{NB}^{ATT} to account for the non-covalent interactions that stabilize the native state. We assumed that, if the noncovalently linked residues i and j ($|i-j|{>}2$) are within the cutoff $R_C{=}8$Å, then

Δ_{ij}=1, and zero otherwise. The value of ϵ_n (=1.5$kcal/mol$) quantifies the strength of the non-bonded interactions. All the non-native interactions in the potential U_{NB}^{REP} are treated as repulsive. An additional constraint is imposed on the bond angle formed by residues i, i+1, and i+2 by including the repulsive potential with parameters ϵ_r=1$kcal/mol$ and $\sigma_{i,i+2}$=3.8Å, which determine the strength and the range of the repulsion. To ensure the self-avoidance of the protein chain, we set σ=3.8Å .

[1] Stone, J. E.; Phillips, J. C.; Freddolino, P. L.; Hardy, D. J.; Trabuco, L. G.; Schulten, K. *J. Comput. Chem.* **2007**, *28*, 2618–2640.

[2] Friedrichs, M. S.; Eastman, P.; Vaidyanathan, V.; Houston, M.; Legrand, S.; Beberg, A. L.; Ensign, D. L.; Bruins, C. M.; Pande, V. S. *J. Comput. Chem.* **2009**, *30*, 864–872.

[3] Anderson, J. A.; Lorentz, C. D.; Travesset, A. *J. Comput. Phys.* **2008**, *227*, 5342–5359.

[4] van Meel, J. A.; Arnold, A.; Frenkel, D.; Zwart, S. F. P.; Belleman, R. *Mol. Simulat.* **2008**, *34*, 259–266.

[5] Harvey, M. J.; Fabritilis, G. D. *J. Chem. Theory Comput.* **2009**, *5*, 2371–2377.

[6] Anderson, A. G.; III, W. A. G.; Schröder, P. *Comput. Phys. Commun.* **2007**, *177*, 298–306.

[7] Yang, J.; Wang, Y.; Chen, Y. *J. Comput. Phys.* **2007**, *221*, 799–804.

[8] Kirk, D. B.; Hwu, W.-M. W. *Programming Massively Parallel Processors. A Hands-on Approach.*; Morgan Kaufmann, 2010.

[9] Brooks, B. R.; Bruccoleri, R. E.; Olafson, B. D.; States, D. J.; Swaminathan, S.; Karplus, M. *J. Comput. Chem.* **1983**, *4*, 187–217.

[10] Haberthür, U.; Caflisch, A. *J. Comput. Chem.* **2008**, *29*, 701–715.

[11] Doi, M.; Edwards, S. F. *The Theory of Polymer Dynamics*; International Series of Monographs on Physics; Oxford Science Publications, 1988.

[12] L'Ecuyer, P.; Simard, R. *ACM T. Math. Software.* **2007**, *33*, 22.

[13] Press, W. H.; Teukolsky, S. A.; Vetterling, W. T.; Flannery, B. P. *Numerical Recipes in C*, 2nd ed.; The Art of Scientific Computing; Cambridge University Press, 1992.

[14] Hyeon, C.; Dima, R. I.; Thirumalai, D. *Structure* **2006**, *14*, 1633–1645.

[15] M. Mickler, R. I. D.; Dietz, H.; Hyeon, C.; Thirumalai, D.; Rief, M. *Proc. Natl. Acad. Sci. USA* **2007**, *104*, 20268–20273.

[16] Dima, R. I.; Joshi, H. *Proc. Natl. Acad. Sci. USA* **2008**, *105*, 15743–15748.

[17] Gertsman, I.; Gan, L.; Guttman, M.; Lee, K.; Speir, J. A.; Duda, R. L.; Hendrix, R. W.;

Komives, E. A.; Johnson, J. E. *Nature* **2009**, *458*, 646–650.

[18] Steven, A. C.; B., H. J.; Cheng, N.; Trus, B. L.; Conway, J. F. *Curr. Opin. Struct. Biol.* **2005**, *15*, 227–236.

[19] Vlad, R. A. J. D. H.; Bahar, I. *Structure* **2005**, *13*, 413–421.

[20] *GPU Gems 3*; Nguyen, H., Ed.; Addison-Wesley, 2008.

[21] Tsang, W. W.; Marsaglia, G. *J. Stat. Softw.* **2000**, *5*, .

[22] Marsaglia, G.; Bray, T. A. *SIAM Rev.* **1964**, *6*, 260–264.

[23] Box, G. E. P.; Miller, M. E. *Ann. Math. Stat.* **1958**, *29*, 610–611.

[24] Marsaglia, G. *DIEHARD: a battery of tests of Randomness.*, 1996, See: http://stat.fsu.edu/ geo/diehard.html.

[25] Mascagni, M.; Srinivasan, A. *ACM T. Math. Software.* **2000**, *26*, 436–461.

[26] Soto, J. *Statistical testing of random number generators*, 1999, See: http://csrc.nist.gov/rng/.

[27] Selke, W.; Talapov, A. L.; Shchur, L. N. *JETP Lett.* **1993**, *58*, 665–668.

[28] Grassberger, P. *Phys. Lett. A* **1993**, *181*, 43–46.

[29] Ferrenberg, A. M.; Landau, D. P.; Wong, Y. J. *Phys. Rev. Lett.* **1992**, *69*, 3382–3384.

[30] Tausworthe, R. C. *Math. Comput.* **1965**, *19*, 201–209.

[31] L'Ecuyer, P. *Math. Comput.* **1996**, *65*, 203–213.

[32] Mascagni, M.; Srinivasan, A. *Parallel Comput.* **2004**, *30*, 899–916.

[33] Marsaglia, G. *Random numbers for C: The END?*, 1999, Published on sci.crypt.

[34] L'Ecuyer, P.; Blouin, F.; Couture, R. *ACM T. Model. Comput. S.* **1993**, *3*, 87–98.

[35] Veitshans, T.; Klimov, D.; Thirumalai, D. *Fold. Des.* **1997**, *2*, 1–22.

[36] Barsegov, V.; Klimov, D.; Thirumalai, D. *Biophys. J.* **2006**, *90*, 3827–3841.

[37] Ermak, D. L.; McCammon, J. A. *J. Chem. Phys.* **1978**, *69*, 1352–1360.

[38] Zhmurov, A.; Dima, R. I.; Kholodov, Y.; Barsegov, V. *Proteins* **2010**, *78*, 2984–2999.

[39] Risken, H. *The Fokker-Planck Equation*, 2nd ed.; Springer-Verlag, 1989.

[40] Mickler, M.; Dima, R. I.; Dietz, H.; Hyeon, C.; Thirumalai, D.; Rief, M. *Proc. Natl. Acad. Sci. USA* **2007**, *104*, 20268–20273.

[41] Dima, R. I.; Joshi, H. *Proc. Natl. Acad. Sci. USA* **2008**, *105*, 15743–15748.

[42] Hyeon, C.; Onuchic, J. N. *Proc. Natl. Acad. Sci. USA* **2007**, *104*, 2175–2180.

[43] Lin, J. C.; Thirumalai, D. *J. Am. Chem. Soc.* **2008**, *130*, 14080–14084.

[44] Hyeon, C.; Lorimer, G. H.; Thirumalai, D. *Proc. Natl. Acad. Sci. USA* **2006**, *103*, 18939–18944.

[45] Hyeon, C.; Jennings, P. A.; Adams, J. A.; Onuchic, J. N. *Proc. Natl. Acad. Sci. USA* **2009**, *106*,

3023–3028.

[46] Tehver, R.; Thirumalai, D. *Structure* **2010**, *18*, 471–481.

[47] Zink, M.; Grubmueller, H. *Biophys. J.* **2009**, *96*, 1767–1777.

[48] Klimov, D. K.; Thirumalai, D. *Proc. Natl. Acad. Sci. USA* **2000**, *97*, 7254–7259.

[49] Ivanovska, I. L.; de Pablo, P. J.; Ibarra, B.; Sgalari, G.; MacKintosh, F. C.; et al., *Proc. Natl. Acad. Sci. USA* **2004**, *101*, 7600–7605.

[50] Ivanovska, I.; Wuite, G.; Joensson, B.; Evilevitch, A. *Proc. Natl. Acad. Sci. USA* **2007**, *104*, 9603–9608.

[51] Weisel, J. W. *Science* **2008**, *320*, 456–457.

[52] Michel, J. P.; Ivanovska, I. L.; Gibbons, M. M.; Klug, W. S.; Knobler, C. M.; et al., *Proc. Natl. Acad. Sci. USA* **2006**, *103*, 6184–6189.

[53] Landau, L. D.; Lifshitz, E. M. *Theory of Elasticity*, 3rd ed.; Pergamon, New York, 1986.

[54] Isralewitz, B.; Gao, M.; Schulten, K. *Curr. Opin. Struct. Biol.* **2001**, *11*, 224–230.

[55] Freddolino, P. L.; Liu, F.; Gruebele, M.; Schulten, K. *Biophys. J.* **2008**, *94*, L75–L77.

[56] Phelps, D. K.; Speelman, B.; Post, C. B. *Curr. Opin. Struct. Biol.* **2000**, *10*, 170–173.

[57] Bahar, I.; Rader, A. J. *Curr. Opin. Struct. Biol.* **2005**, *15*, 1–7.

[58] Wikoff, W. R.; Liljas, L.; Duda, R. L.; Tsiruta, H.; Hendrix, R. W.; Johnson, J. E. *Science* **2000**, *289*, 2129–2133.

[59] Lidmar, J.; Mirny, L.; Nelson, D. R. *Phys. Rev. E* **2003**, *68*, 051910–051919.

[60] Matsumoto, M.; Nishimura, T. *ACM T. Model. Comput. S.* **1998**, *8*, 3–30.

[61] Kuznetsov, Y. G.; Gurnon, J. R. abd Etten, J. L. V.; McPherson, A. *J. Struct. Biol.* **2005**, *149*, 256–263.

[62] Schwaiger, I.; Sattler, C.; Hostetter, D. R.; Rief, M. *Nat. Mater.* **2002**, *1*, 232–235.

FIGURE CAPTIONS

Fig. 1. Panel *a*: The computational time (in ms/step) for Langevin Dynamics (LD) of N Brownian oscillators with the Hybrid Taus and additive Lagged Fibonacci generators of (pseudo)-random numbers. We considered the three implementations, where (1) random numbers and LD are generated on the CPU (Hybrid Taus (CPU)+Dynamics (CPU)), (2) random numbers are obtained on the CPU, transfered to the GPU and used to propagate LD on the GPU (Hybrid Taus (CPU)+Dynamics (GPU)), and (3) random numbers and LD are generated on the GPU (Hybrid Taus (GPU)+Dynamics (GPU) and Lagged Fibonacci (GPU)+Dynamics (GPU)). Panel *b*: The computational speedup (CPU time versus GPU time) for LD simulations fully implemented on the GPU and on the single CPU core. We compared the two options when an RNG (Hybrid Taus or Lagged Fibonacci) is organized in a separate kernel or is inside the main (integration) kernel. We ran long trajectories (10^6 steps) to converge the speedup factor.

Fig. 2. The average particle position $\langle X(t)\rangle$ (panels *a* and *b*) and two-point correlation function $C(t)$ (panel *c*) for a system of N=10^4 Brownian oscillators. Theoretical curves of $\langle X(t)\rangle$ and $C(t)$ are compared with the simulation results obtained using the LCG, Hybrid Taus, Ran2, and Lagged Fibonacci algorithms. Equilibrium fluctuations in $\langle X(t)\rangle$ in a longer timescale, obtained using LCG, are magnified in panel *b*. A repeating pattern due to correlations among N streams of random numbers is clearly observed.

Fig. 3. The computational performance of the developed realizations of the LCG, Ran2, Hybrid Taus, and Lagged Fibonacci algorithms in Langevin simulations of N three-dimensional Brownian oscillators on the GPU. Panel *a*: The execution time (in ms/step); threads have been synchronized on the CPU at the end of each step to imitate an LD simulation run of a biomolecule. As a reference, we display the CPU time for Langevin simulations with Ran2 and Hybrid Taus generators. Panel *b*: The memory demand, i.e. the amount of memory needed for an RNG to store its current state. Step-wise increases in the memory usage for Lagged Fibonacci are due to the change of constant parameters (Table I in SI).

Fig. 4. Computational time (in ms/step) for the GPU-based implementation of Langevin simulations of N three-dimensional Brownian oscillators using Hybrid Taus RNG (panel *a*) and Lagged Fibonacci RNG (panel *b*). The simulation time for Langevin Dynamics is compared with the time for generating random numbers using Hybrid Taus RNG or Lagged Fibonacci RNG, and with the time required to obtain deterministic (Newtonian) dynamics without random numbers (computational speedup is displayed in Fig. 1b).

Fig. 5. The force-indentation profiles showing the dependence of force F on the cantilever displacement Z (FZ curves) for the bacteriophage $HK97$, obtained using Langevin simulations and Lagged Fibonacci RNG fully implemented on the GPU. To mimic the dynamic force measurements *in vitro*, we indented the capsid using a spherical tip (gray balls) of radius R=$5nm$, $10nm$, and $25nm$. The cantilever with the spring constant κ=$50N/m$ is moving downwards in the direction shown by the gray arrows, approaching the viral shell with the constant velocity v_f=$2.5\mu m/s$ (panel a), $25\mu m/s$ (panel b), and $2.5\mu m/s$ (panel c). Also shown are the transient structures formed in the course of a single indentation trajectory. These show the geometric changes to the $HK97$ conformation due to the continuous indentation (panel b), buckling (panel a) and fracture (panel c). The insets show the number of native contacts Q and the spring constant of the virus capsid K as a function of Z.

TABLE I: The memory usage (in bytes/thread) and the number of GPU global memory calls, i.e. the numbers of read/write operations per one random number (M_1) and for 4 random numbers (M_2), for the LCG, Hybrid Taus, Ran2, and Lagged Fibonacci algorithms (4 random numbers are needed per particle to generate 3 components of the Gaussian random force).

Parameter	LCG	Hybrid Taus	Ran2	Lagged Fibonacci
bytes/thread	4	16	280	12
M_1	1/1	4/4	4/4	3/1
M_2	1/1	4/4	7/7	12/4

TABLE II: The average parameters characterizing the microscopic elastic behavior of the bacteriophage $HK97$, namely, the spring constant K and the Young's modulus Y. Also presented are the average values of the critical pressure p_c, and the energy change due to the buckling transitions ΔE_b and the capsid fracture ΔE_f (values of R are given in parentheses).

v_f, $\mu m/s$	K, N/m	Y, MPa	p_c, MPa	$\Delta E_b/10^{-17}$, Nm	$\Delta E_f/10^{-17}$, Nm
2.5	0.01−0.025	64−160	6 ($5nm$)	1.4 ($25nm$)	2.7 ($5nm$)
25	0.05−0.075	320−480	9 ($5nm$)	-	3.0 ($5nm$)
250	0.2−0.35	1280−2230	13 ($5nm$), 18 ($10nm$)	-	3.4 ($5nm$), 5.7 ($10nm$)

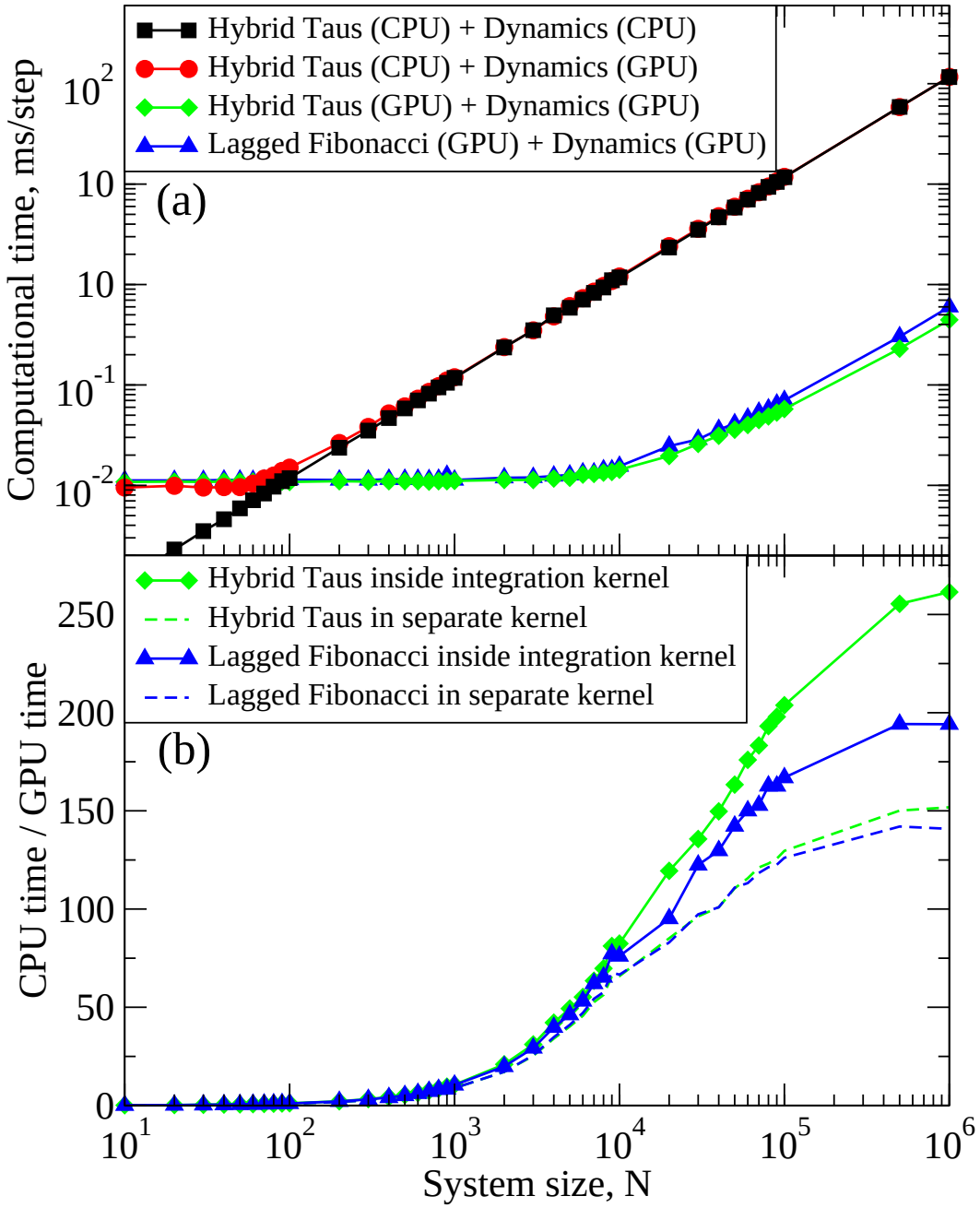
Hybrid Taus (CPU) + Dynamics (CPU)
Hybrid Taus (CPU) + Dynamics (GPU)
Hybrid Taus (GPU) + Dynamics (GPU)
Lagged Fibonacci (GPU) + Dynamics (GPU)
(a)
Computational time, ms/step
Hybrid Taus inside integration kernel
Hybrid Taus in separate kernel
Lagged Fibonacci inside integration kernel
Lagged Fibonacci in separate kernel
(b)
CPU time / GPU time
System size, N

Figure 1

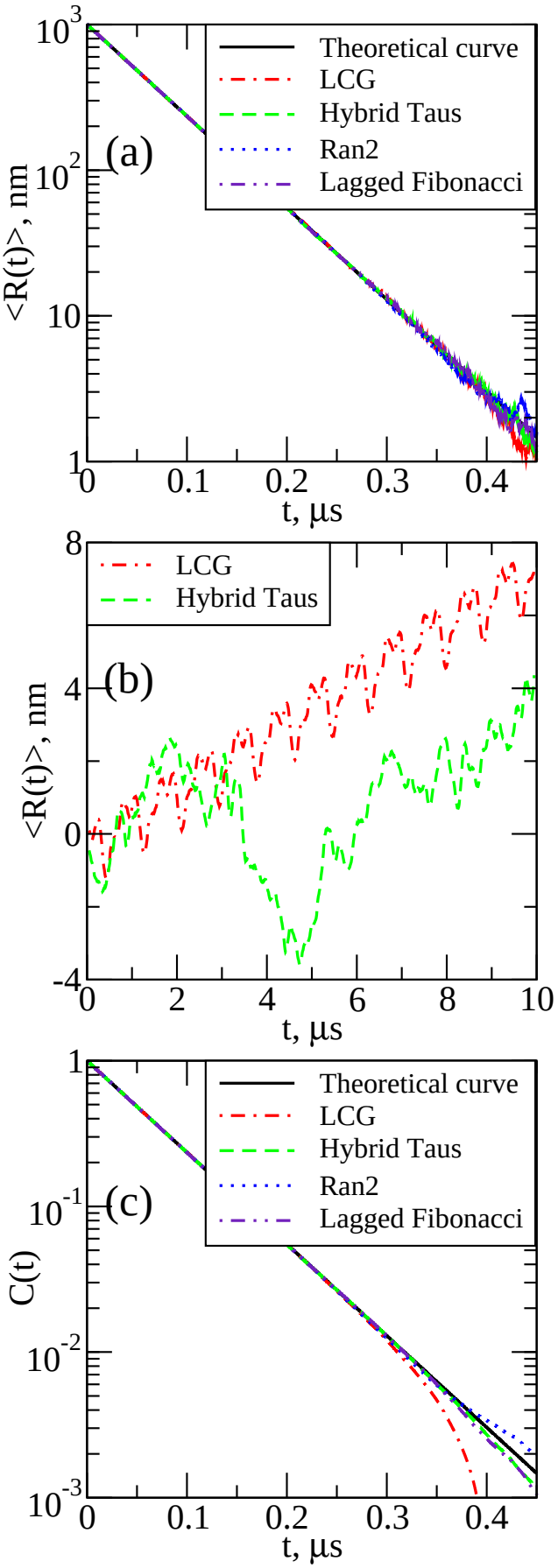
(a)
Theoretical curve
LCG
Hybrid Taus
Ran2
Lagged Fibonacci
<R(t)>, nm
t, μs
(b)
LCG
Hybrid Taus
<R(t)>, nm
t, μs
(c)
Theoretical curve
LCG
Hybrid Taus
Ran2
Lagged Fibonacci
C(t)
t, μs

Figure 2

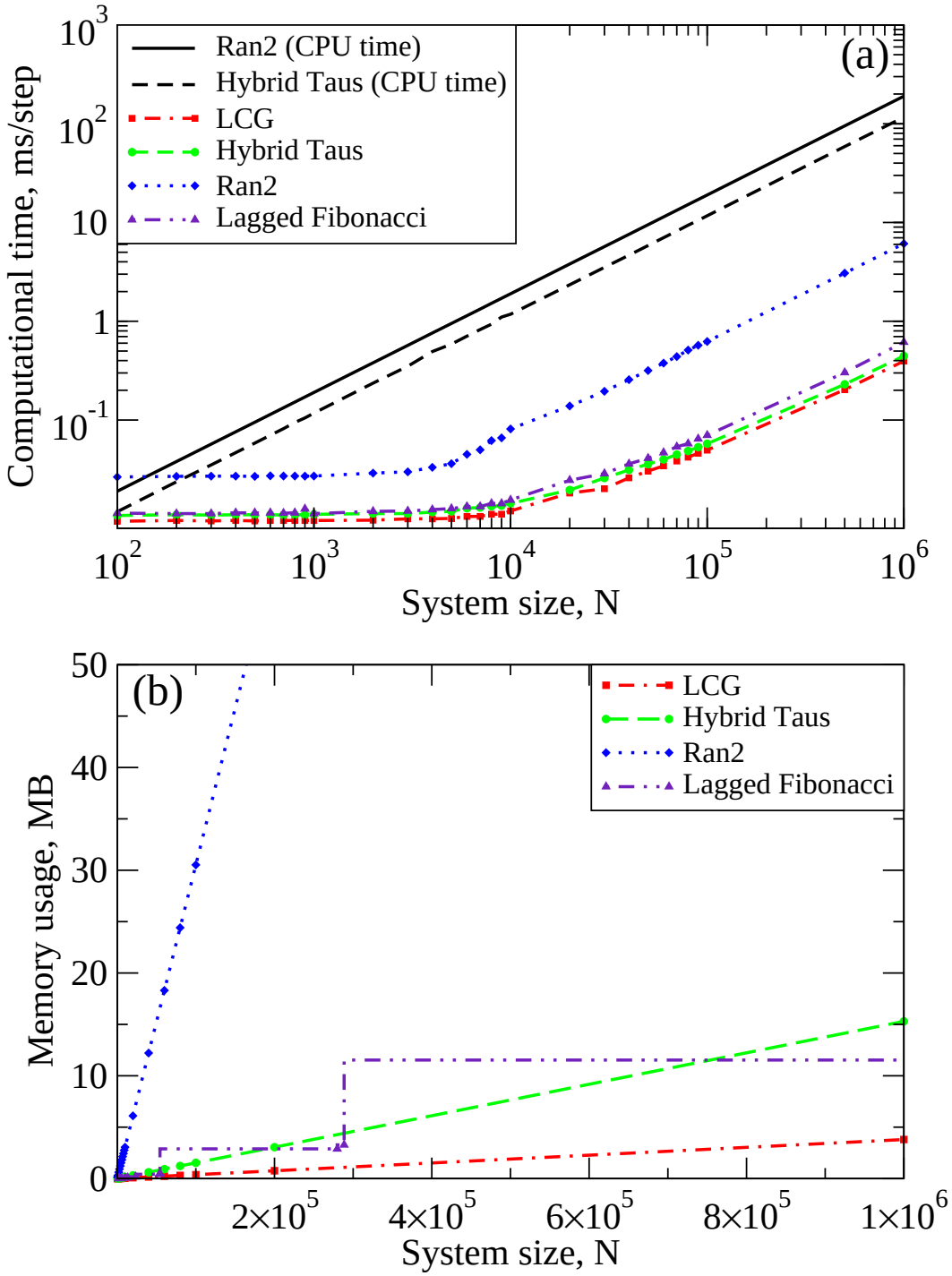
(a)
Ran2 (CPU time)
Hybrid Taus (CPU time)
LCG
Hybrid Taus
Ran2
Lagged Fibonacci
Computational time, ms/step
System size, N
(b)
LCG
Hybrid Taus
Ran2
Lagged Fibonacci
Memory usage, MB
System size, N

Figure 3

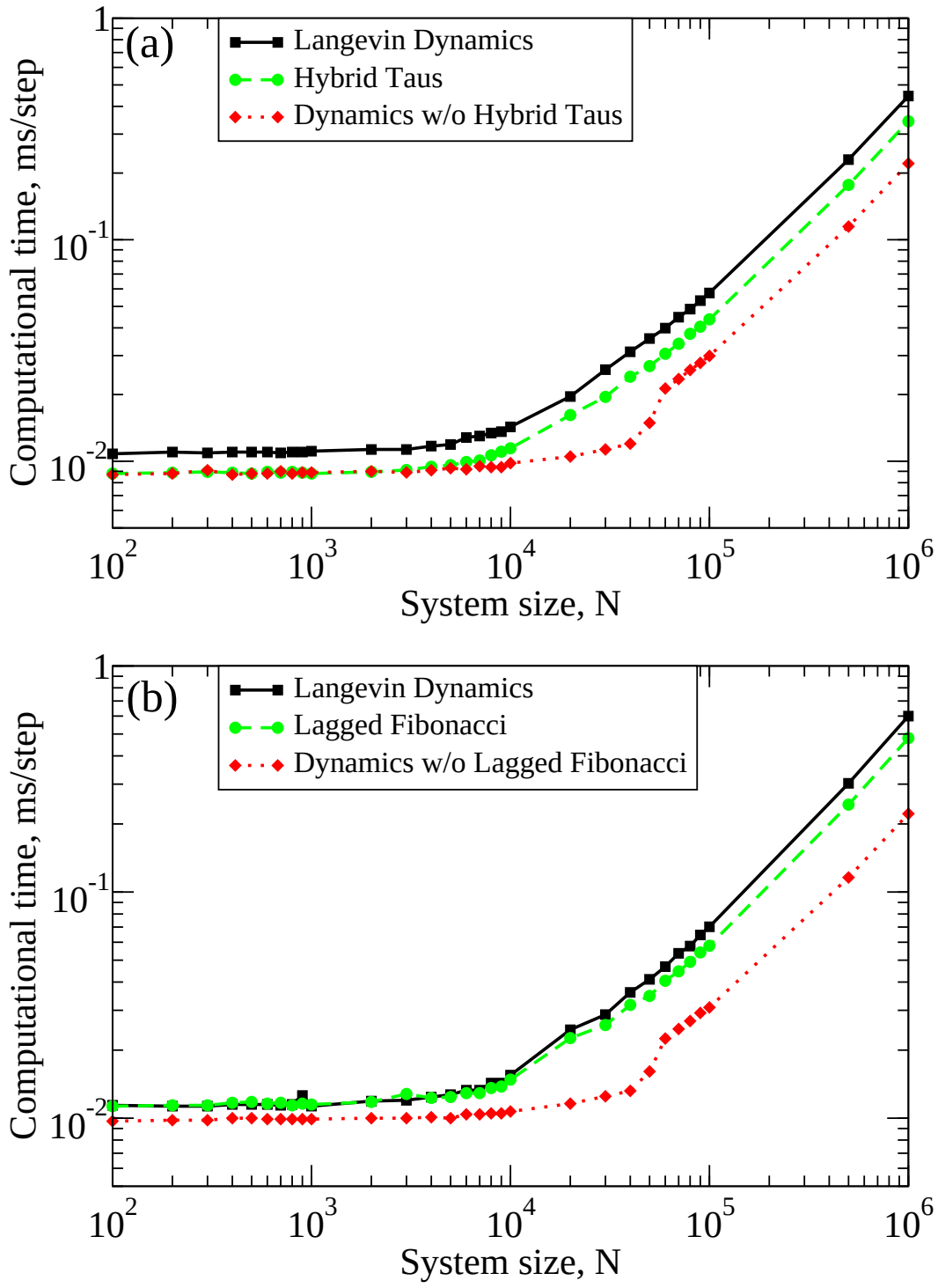
(a)
Langevin Dynamics
Hybrid Taus
Dynamics w/o Hybrid Taus
Computational time, ms/step
System size, N
(b)
Langevin Dynamics
Lagged Fibonacci
Dynamics w/o Lagged Fibonacci
Computational time, ms/step
System size, N

Figure 4

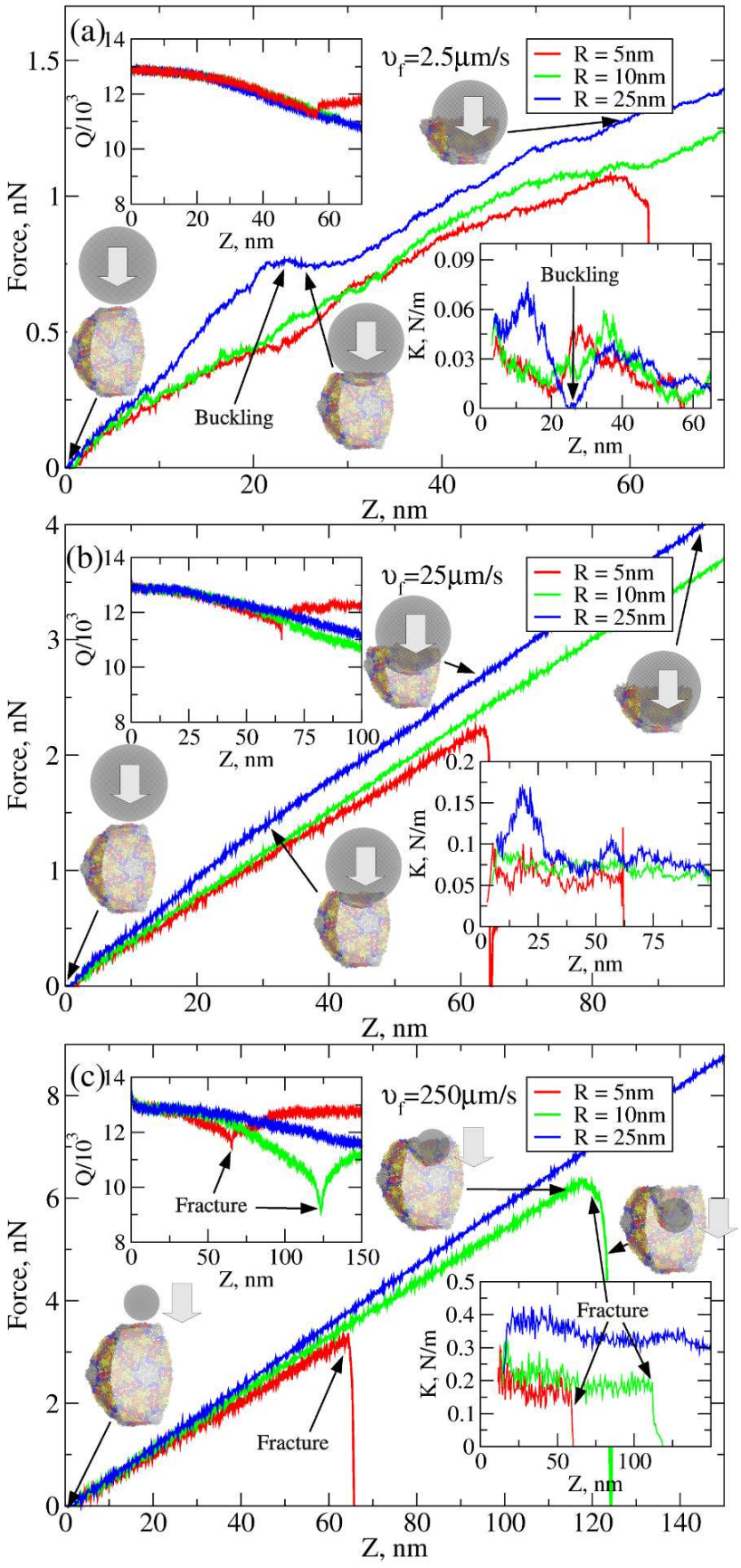
(a)
υf=2.5μm/s
R = 5nm
R = 10nm
R = 25nm
Force, nN
Z, nm
Q/10^3
K, N/m
Buckling
(b)
υf=25μm/s
R = 5nm
R = 10nm
R = 25nm
Force, nN
Z, nm
Q/10^3
K, N/m
(c)
υf=250μm/s
R = 5nm
R = 10nm
R = 25nm
Force, nN
Z, nm
Q/10^3
K, N/m
Fracture

Figure 5