ISSN 2070-0482, Mathematical Models and Computer Simulations, 2012, Vol. 4, No. 3, pp. 272–287. © Pleiades Publishing, Ltd., 2012.
Original Russian Text © A.A. Zhmurov, V.A. Barsegov, S.V. Trifonov, Ya.A. Kholodov, A.S. Kholodov, 2011, published in Matematicheskoe Modelirovanie, 2011, Vol. 23, No. 10, pp. 133–156.

Langevin Dynamics Simulations of Micromechanics on Graphics Processors[1]

A. A. Zhmurov, V. A. Barsegov, S. V. Trifonov, Ya. A. Kholodov, and A. S. Kholodov

Moscow Institute of Physics and Technology

e-mail: kholodov@crec.mipt.ru

Received March 29, 2011

Abstract—Due to the very long timescales involved (ms-s), theoretical modeling of fundamental biological processes including folding, misfolding, and mechanical unraveling of biomolecules, under physiologically relevant conditions, is challenging even for distributed computing systems. Graphics Processing Units (GPUs) are emerging as an alternative programming platform to the more traditional CPUs as they provide high raw computational power that can be utilized in a wide range of scientific applications. Using a coarse-grained Self Organized Polymer (SOP) model, we have developed and tested the GPU-based implementation of Langevin simulations for proteins (SOP-GPU program). Simultaneous calculation of forces for all particles is implemented using either the particle based or the interacting pair based parallelization, which leads to a ~90-fold acceleration compared to an optimized CPU version of the program. We assess the computational performance of an end-to-end application of the SOP-GPU program, where all steps of the algorithm are running on the GPU, by profiling the associated simulation time and memory usage for a number of small proteins, long protein fibers, and large-size protein assemblies. The SOP-GPU package can now be used in the theoretical exploration of the mechanical properties of large-size protein systems to generate the force-extension and force-indentation profiles under the experimental conditions of force application, and to relate the results of singlemolecule experiments in vitro and in silico.

Keywords: graphics processing units, large-size protein systems simulations, self-organized polymer model, Langevin dynamics, SOP-GPU package.

DOI: 10.1134/S2070048212030106

1. INTRODUCTION

Mechanical functions of protein fibers such as fibronectin, fibrin fibers, microtubules, and actin filaments, are important in cytoskeletal support and cell motility [1–3], in cell adhesion and the formation of the extracellular matrix [4–7], and in blood clotting [8–10]. Physical properties of viral capsids of plant and animal viruses [11–13], retroviruses [14], and bacteriophages [15, 16], and the transitions between their stable and unstable states determine the life cycle of many viruses, including virus maturation, and infection of cells [17]. Understanding the microscopic origin of the unique viscoelastic properties of protein fibers and the crossover from an elastic to a plastic behavior in viral capsids, as well as the control of their mechanical response to an applied mechanical force constitute major areas of research in biochemistry and biophysics. Single-molecule techniques, such as AFM and laser tweezer-based force spectroscopy, have been used to study the mechanical properties of protein fibers [18–21] and viral capsids [15, 16, 22, 23]. Yet, due to the high complexity of these systems (~10^3–10^5 residues) and to their large size (~50–200 nm), these experiments yield results that are nearly impossible to interpret without first having some a priori information about their energy landscape [9].

Standard packages for all-atom Molecular Dynamics (MD) simulations, such as CHARMM [24] NAMD [25], and Gromacs [26] among others, are being used to access the submolecular behaviour of biomolecules. However, because all-atomic modeling is currently limited to a 10–50 nm length scale and 0.1–10 ms duration [27–29], these methods allow for the theoretical exploration of equilibrium properties of biomolecules, and reaching the biologically important ms–s timescale becomes virtually impossible even for a small system. More importantly, to fully explore the free energy landscape underlying a biological process of interest, one needs to generate a statistically representative set of trajectories. One possibility is to carry out MD simulations on manycore computer clusters, but it requires tremendous computational resources and long CPU times. For example, it takes 800000 CPU hours to obtain 20 short 1 ns MD

[1] The article was translated by the author.

trajectories for the southern bean mosaic virus, which contains as many as 4.5 million atoms, on an SGI Altix 4700 cluster [30]. These limitations exclude computations as an investigative tool in the study of a range of biological problems, such as the large deformations of protein fibers, the formation of biomolecular complexes and aggregates, and the mechanical failure of viral capsids, for which experimental data are already available, thereby rendering the direct comparison of the results of experiments in vitro and in silico impossible.

Although graphics processors have been originally designed for computationally intensive graphics rendering, they have evolved over the last few years into highly parallel, multithreaded computing devices. Recent technological advances in the throughput-oriented hardware architecture of GPUs with extremely high peak arithmetic performance, which employs IEEE floating point arithmetic, have unleashed tremendous computational power that can now be utilized in general purpose scientific applications. Unlike mainstream processor architecture, GPUs devote the majority of their logic units to performing actual calculations, rather than to cache memory and flow control. Massive multithreading, fast context switching, and high memory bandwidth have enabled GPUs to tolerate latencies associated with memory calls, and to run many computational cores simultaneously. Programming tools for modern GPUs include several platforms such as ATI Stream Computing [31], NVIDIA Compute Unified Device Architecture (CUDA) [32, 33], and Open Computing Language (OpenCL) [34]. CUDA, a parallel computing environment (a dialect of the C and C^{++} programming languages), provides a high level software platform that allows a programer to define kernels that are executed in parallel by independent computational threads.

Because GPUs differ from CPUs in several fundamental ways, CPU-based methods for molecular simulations of biomolecules cannot be easily translated nor simply adapted to a GPU. In these methods, particle-particle interactions are described by the same empirical potential energy function (force field), and the dynamics of the system in question is obtained by solving numerically the same equations of motion for all particles. Hence, there is a direct correspondence between the SIMD (Single Instruction Multiple Data) architecture of a GPU at the hardware level and the numerical routines (software) used to follow the molecular dynamics. It is then possible to execute "single instruction," i.e. calculation of the potential energy or evaluation of forces, or generation of random forces, or integration of the equations of motion, on "multiple data" sets (for all particles) at the same time over many iterations using many Arithmetic Logic Units (ALUs) running in parallel. This makes MD simulations a natural candidate to implementation on a GPU, but not all algorithms are amenable to this architecture. For an algorithm to execute efficiently on the GPU, it must be recast into a data-parallel form with independent threads running the same instruction stream on different data. There exist preliminary versions of standard packages for MD simulations of proteins implemented on a GPU, such as NAMD [28, 35, 36], Gromacs [47], and other applications [38–40]. Yet, there are no GPU-based implementations of Langevin dynamics simulations that we are aware of. In this paper, we develop and test such an implementation. Because the topology and the overall structure (geometry), rather than the atomic details, govern the force-driven molecular transitions in protein systems, we employ a coarse-grained description of proteins [41–43] using a Self Organized Polymer (SOP) model [44, 45].

Implementing the Langevin dynamics algorithm on a GPU requires detailed understanding of the device architecture. The methodology for the GPU-based implementation of Langevin simulations is presented in the next section, where we describe the particle based and the interacting pair based parallelization approaches to force computation as well as the numerical routines for generating pseudorandom numbers, constructing the Verlet lists, and integrating forward Langevin equations to the next time step. For purposes of presentation and to focus on the most essential computational aspects, we have simplified the presentation of the formalism as much as possible. A comparative analysis of the results of CPU- and GPU-based simulations of the mechanical unfolding for a test system represented by the all-strand *WW* domain is performed in Section 3, where we also assess the accuracy of the numerical integration. We discuss the results of the GPU-based computations in terms of the simulation time, memory usage, and computational speedup (CPU time versus GPU time) for a range of proteins including small proteins, such as the *WW* domain, the *Ig27* domain from human titin, the *C2A* domain from human synaptotagmin (Syt1), the γC chain and the double-D fragment of human fibrinogen (*Fb*), single-chain models of fibrin fibers (*Fb* monomer and dimer), and large-size protein assembly (viral capsid *HK*97). The main results are summarized in Section 4.

2. LANGEVIN DYNAMICS SIMULATIONS ON A GPU

In this Section, we describe the particle based and the interacting pair based methods for the parallel computation of potentials and forces due to binary particle-particle interactions. We also outline the

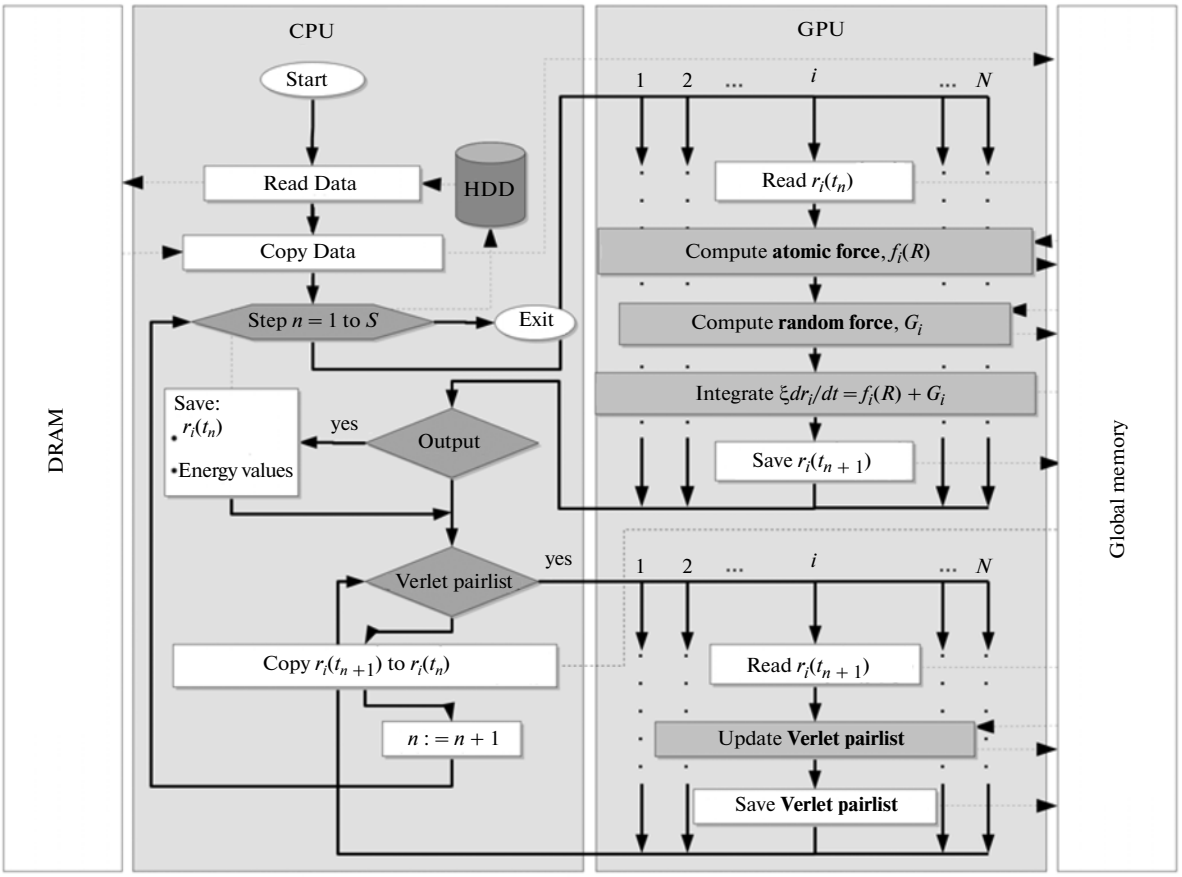

Fig. 1. The numerical algorithm and computational procedures for the Langevin simulations of biomolecules on a GPU. The computational workflow is shown using black arrows, and data transfer – read and write operations from and to the DRAM and the HDD (hard drive) on the CPU device and the GPU global memory – are represented by the dashed arrows. Designing CUDA kernels in volves the decomposition of work into small fragments that can be mapped into thread blocks, and further decomposition into warps and into independent threads of execution. The computational workflow for only one (i-th) thread running on the GPU and the workload division between CPU and GPU are shown in detail. The execution of the program is initiated on the CPU, which is used to prepare and store the initial and output data. The CPU device starts the launch of each computational kernel on the GPU for the calculation of forces, generation of the random forces and Verlet lists, and for the numerical integration of the Langevin equations of motion, using many independent threads running in parallel.

numerical procedures involved in the generation of the Verlet lists and the random forces, and in the numerical integration of the Langevin equations of motion. We designed the algorithm to process as many computational fragments simultaneously on a GPU as possible in order to move the GPU into full production efficiency and to minimize the GPU/CPU communication. The algorithm decomposition, along with the workload division between the GPU and the CPU, is diagrammed in Fig. 1, where we also summarize the computational workflow on a CPU and on a GPU, including the operation on the data files for the molecular topology, the particle energies and coordinates, and the data flow between the CPU DRAM and the GPU global memory (host-GPU data transfers).

In Langevin simulations of biomolecules, molecular forces are usually described by two-body (pair) potentials, such as the harmonic potential, the FENE potential [46], the Lennard-Jones potential, etc., which differ in their mathematical form. Hence, the same generic algorithm can be employed to compute these potentials. In the pseudocode listings below, the i-th residue for a protein of N residues has a unique index $i \in [0, N-1]$; for an array of particle coordinates r, $r[i]$ denotes coordinates of the i-th particle. We also use the following notations: global memory reads are represented by $\Leftarrow$, saving data to the GPU global memory is shown by $\Rightarrow$, <= denotes cached texture memory reads, and $\leftarrow$ represents local, shared, or constant memory invocation, or assignments of variables. Consider the following computational procedure for evaluating forces on a CPU:

***Algorithm 1*:** Calculation of pairwise forces (CPU implementation).

1. $P \leftarrow$ total number of pairs in a system for a given potential
2. **for** $p = 0$ to $P - 1$ **do**
3. $i \leftarrow pairs[p].i$ {index of the first particle in a particle pair}
4. $j \leftarrow pairs[p].j$ {index of the second particle in a particle pair}
5. $par \leftarrow pairs[p]$.parameters {parameters for the i–j pair}
6. $r \leftarrow r[i] - r[j]$
7. $df \leftarrow$ force (r, par)
8. $f[i] \leftarrow f[i] + df$
9. $f[j] \leftarrow f[j] - df$
10. **end for**

Information about the residue pairs is stored in array *pairs*, which has indices i and j, and variable *par* for constant parameters, which specify the potential energy. In the main cycle on a CPU (line 2), index p runs through all the pairs of interacting residues $p = 0, 1, ..., P - 1$. For each pair, the information about the coordinates ($r[i]$ and $r[j]$) and the constant parameters (*par*) is gathered in lines 3–5. The force increment *df* is computed in *force*(...) for a given pair p (line 7). This value is added (or subtracted) to (from) the force for the i-th particle ($f[i]$, line 8) and subtracted (or added) from (to) the force for the j-th ($f[j]$, line 9). On a CPU, the force values are computed only once (lines 7–9). Because force calculations are sequential, they do not overlap in time. By contrast, because on a GPU a two-body potential is computed for different pairs of residues in different threads, a naïve addition (lines 8 and 9) may cause memory conflicts when some or all the threads attempt to access the same address in the GPU global memory.

There are two main optimization strategies that allow one to avoid this situation. In the first approach, all the forces for one particle are computed in one thread, which requires running N threads to obtain the force values for all particles. We refer to this procedure as the particle based parallelization approach. The use of this approach results in the same force, acting on the i-th and j-th particles, but computed twice in the i-th and j-th threads [40]. Following a different strategy, which we refer to as the interacting pair based parallelization approach, force calculations are performed for all pairs in parallel using P independent theads, and $2P$ force values are saved to different locations in the GPU global memory. We pursued in detail both optimization strategies, which exploit the data-parallel aspects of GPU based computing.

2.1. The Particle Based Parallelization Approach

In this approach, N independent threads run on a GPU concurrently, each computing all the pair potentials for each particle and summing all force values (except for the random force) to obtain the total force. Although the force acting on the i-th and j-th particles is computed twice, the number of global memory calls is reduced by a factor of $2N$, and the time spent on recalculating the same potential is compensated by the time saved by not waiting to write the force data to and to read the data from the GPU global memory.

***Algorithm 2*:** Calculation of pairwise forces using particle based parallelization.

1. $f_i \leftarrow 0$ {resulting force}
2. $i \leftarrow$ GPU thread index {same as particle index}
3. $r_i <= r[i]$ {coordinates of the i-th particle}
4. $P_i \Leftarrow P_p[i]$ {number of pairs formed by the i-th particle}
5. **for** $p = 0$ to $P_i - 1$ **do**
6. $j \Leftarrow PairsMap[i][p].j$ {second particle in a pair}
7. $r_j \leq r[j]$ {coordinates of the j-th particle}
8. $par \Leftarrow PairsMap[i][j].parameters$ {parameters for the i–j pair}
9. $r \leftarrow r_i - r_j$
10. $df \leftarrow force\ (r, par)$
11. $f_i \leftarrow f_i + df$
12. **end for**
13. Output: f_i

The array P_p keeping track of the number of pairs for all residues and the matrix PairsMap of all particle pairs are pre-generated on a CPU and fetched to the GPU global memory. P_p is an N-dimensional vector of integers 0, 1, 2, ... Each element of this vector corresponds to a single particle, and the i-th integer value is the number of particles interacting with the i-th particle. PairsMap is the $N \times M$ matrix, where M is a

maximum number from array P_p. The i-th row of the matrix PairsMap corresponds to the i-th particle and contains the indices of all particles interacting with the i-th particle and the constant parameters for the potential energy function. Data in the PairsMap can be easily rearranged for coalescent memory reads. Coordinates of the second particle for each pair are accessed at random, which allows one to take advantage of the texture reference in global memory reads.

Force values are computed in parallel by N threads as follows. First, the i-th thread reads the coordinates r_i using a cached texture reference (line 3) and the number of particle pairs $P_p[i]$ (line 4). By cycling through all the pairs, formed by the i residue (lines 5–12), the thread reads the index j and the coordinates r_j of the j-th particle and the constant parameters (par, lines 6–8). These are used to compute the force increment df (line 10), which is added to the resulting force f (line 11) for the i-th particle. The parameters par depend on the potential used. For example, for a covalent bond, described by a harmonic potential $V_H = K_{ij}^{sp}(r_{ij} - r_{ij}^0)^2/2$, par contains the equilibrium distance r_{ij}^0 and the spring constant K_{ij}^{sp}.

***Verlet lists*:** In molecular simulations, the information about the covalent bonds and the native interactions (array P_p and matrix PairsMap), obtained from the PDB structure of a protein, does not change. However, the information about nonbonded long-range interactions, describing the gradual attraction and hardcore repulsion between pairs of atoms, needs to be updated from time to time. This is the most computationally demanding component of the algorithm, since the complexity of the calculation is $O(N^2)$. A common approach is to take advantage of the fact that long-range interactions vanish over some distance. This allows one to use pair lists that include pairs of particles that are closer than the cutoff distance (Verlet lists) [47]. In the particle based parallelization approach, the array *Pp* and the matrix PairsMap have to be regenerated on a GPU in order to accelerate the computation of the 11 potential energy using Verlet lists. This can be done by rearranging the pseudocode for particle based parallelization (Algorithm 2):

***Algorithm 3*:** Calculation of forces using particle based parallelization and Verlet lists.

1. $i \leftarrow$ GPU thread index {same as particle index}
2. $p_i \leftarrow 0$ {counter of residue pairs in Verlet list}
3. $r_i <= r[i]$ {coordinates of the i-th particle}
4. $P_{p,\,i} \Leftarrow P_{pp}[p_1]$ {number of all pairs for the i-th particle}
5. **for** $p_p = 0$ to $P_{p,\,i} - 1$ **do**
6. $j \Leftarrow PossiblePairsMap[i][p_p]$ {second j-th particle in a pair}
7. $r_j <= r[j]$ {coordinates of the j-th particle}
8. $r \leftarrow |r[i] - r[j]|$
9. **if** $r < cutoff$ **then**
10. $PossiblePairsMap[i][p_p] \Rightarrow$ PairsMap$[i][p_i]$
11. $p_i \leftarrow p_i + 1$
12. **end if**
13. **end for**
14. $p_i \Rightarrow P_p\,[i]$

Cycling over p_p includes all possible residue pairs to identify pairs that are within the cutoff distance. The interparticle distances are computed in line 8. The number of particles p_i that are within the cutoff distance is counted (line 11), and a newly found pair is added to the matrix PairsMap at the (i, p_i)-position, i.e. copied from the matrix PossiblePairsMap (the map of all pairs) to the matrix PairsMap (the map of pairs). Once the cycle is completed, pi is saved to the array $P_p[i]$, which stores the numbers of the residue pairs in the Verlet list.

2.2. The Interacting Pair Based Parallelization Approach

To avoid computing the two-body potentials on a GPU twice, one can design a different computational algorithm, where each thread calculates a single pair potential for two coupled residues. Then, forces acting on the interacting particles in opposite directions are computed only once, the force values obtained are saved to different locations in the GPU global memory, and all the forces exerted on each particle are summed up to obtain the total force. This approach requires additional memory calls and a gathering subroutine for the force summation, but it enables one to accelerate simulations when the number of residues N is of the same order of magnitude as the number of ALUs, and/or when the computation of pair potentials is expensive. In the following pseudocode for the force calculation, P (number of threads) is equal to

the number of interacting pairs for just one potential energy term, and each thread computes forces for one pair of residues:

***Algorithm 4*:** Calculation of pairwise forces using interacting pair based parallelization.

1. $p \leftarrow$ GPU thread index {same as pair index}
2. *pair* $\Leftarrow$ *pairs*[p]
3. *pair* $\Leftarrow$ *PairsParameters*[p] {parameters for one pair of residues p}
4. $i \leftarrow$ *pair.i* {the i-th particle in the pair}
5. $j \leftarrow$ *pair.j* {the j-th particle in the pair}
6. $shift_i \leftarrow pair.shift_i$ {position in array of forces for the i-th particle}
7. $shift_j \leftarrow pair.shift_j$ {position in array of forces for the j-th particle}
8. $r_i <= r[i]$ {coordinates of the i-th particle}
9. $r_j <= r[j]$ {coordinates of the j-th particle}
10. $r \leftarrow r_i - r_j$
11. $df \leftarrow$ force (r, *par*)
12. $df \Rightarrow F[i][shift_i]$ {saving force for the i-th particle}
13. $-df \Rightarrow F[j][shift_j]$ {saving force for the j-th particle}

Each thread identifies one pair potential using the thread index p, and reads the information about the potential from vectors *pairs* and PairsParameters about the constant parameters (par), the particle identity (i and j) and particle coordinates (r_i and r_j), and the global memory addresses for saving the force values ($shift_i$ and $shift_j$). In the array F, the force values are saved to the position defined by the particle index i or j (i-th or j-th row) and by the parameters $shift_i$ or $shift_j$ (columns). In the array pairs, each output position i and $shift_i$ for the i-th particle has to be unique so that each force value computed is saved to a different address in the GPU global memory. This allows one to avoid memory conflicts, but requires an additional gathering kernel for summing all the forces for a given particle obtained in the array F (in Algorithm 4):

***Algorithm 5*:** Gathering kernel for the force computation.

1. $i \leftarrow$ GPU thread index {same as particle index}
2. $f_i \leftarrow 0$ {resulting force due to one potential energy term}
3. $P_i \Leftarrow P_p[i]$ {number of particle pairs for the i-th particle}
4. **for** $p = 0$ to $P_i - 1$ **do**
5. $df \Leftarrow F[i][p]$
6. $f_i \Leftarrow f_i + df$
7. **end for**
8. Output: f_i

P_i counts the residue pairs for the i-th particle (for one potential energy term). The total force for the i-th residue (f_i) is calculated by summing over all the forces computed previously ($F[i][p]$, line 12–13). This part of the program can be incorporated into the integration kernel to minimize the number of computational kernels. On GPUs with the new Fermi architecture (from NVIDIA), the use of thread safe atomic addition of the computed force values to a specific location in the GPU global memory will help to remove the performance barriers associated with multiple memory calls.

***Verlet lists*:** In the interacting pair based parallelization approach, generating a Verlet list surmounts to forming the vector pairs of all residue pairs for one potential energy term. On a GPU, constructing this vector is a formidable task, since the exact position in the list, to which information about the next residue pair should be saved, is not known. One possibility is to use the **atomicAdd**(...) routine from the CUDA Software Development Kit [32], which allows one to add integers in the GPU global memory without running into memory conflicts even when many threads attempt to access the same memory address at the same time. However, when many threads run in parallel, identifying new pairs and saving them one after another may result in a Verlet list that is not ordered according to the particle index. This may result, in turn, in an inefficient utilization of the cache memory. To obtain an ordered Verlet list, it has to be sorted or updated on a CPU. It is more efficient to compute interparticle distances on a GPU, copy them to the CPU DRAM, and then generate a new list.

2.3. The Random Force

Langevin simulations require a reliable source of $3N$ normally distributed pseudorandom numbers, $g_{i,\alpha}$ ($\alpha = x$, y, and z) produced at each integration step, in order to compute the three components of a Gaus-

sian random force $G_{i,\alpha} = g_{i,\alpha}\sqrt{2k_BT\xi h}$, where T is system temperature, ξ is coefficient of friction, k_B is the friction coefficient, and h is the integration time step. A pseudorandom number generator (PRNG) produces a sequence of random numbers $u_{i,\alpha}$ uniformly distributed in the unit interval [0, 1]. This sequence, which imitates a sequence of independent and identically distributed (i.i.d.) random variables, is then translated into the sequence of normally distributed pseudorandom numbers with zero mean and unit variance ($g_{i,\alpha}$) using the Box-Mueller transformation [48]. While there exist stand alone implementations of good quality PRNGs on a GPU, in Langevin simulations a PRNG should be incorporated into the integration kernel to minimize read/write calls of the GPU global memory.

The simplest approach for constructing a PRNG on a GPU is to initiate an independent generator in each thread (one-PRNG-per-thread approach) so that pseudorandom numbers can be produced during the numerical integration of the Langevin equations. First, a CPU generates N independent sets of random seeds for N PRNGs, and then transfers them to the GPU global memory. When $4N$ i.i.d. pseudorandom numbers $u_{i,\alpha}$ are needed for generating $3N$ normally distributed random numbers $g_{i,\alpha}$, each thread reads a corresponding set of random seeds to produce 4 normally distributed random numbers for each residue. Then, a PRNG updates its current state in the GPU global memory, which is used as an initial seed within the same thread at the next time step. In a different approach, random seeds for just one PRNG state can be shared among the computational threads on the entire GPU (one-PRNG-for-all-threads approach). Using both approaches, we have developed and tested GPU-based realizations of PRNG which are based on the Hybrid Taus, Ran2, Lagged Fibonacci and Mersenne Twister algorithms (manuscript in preparation). It has been shown that these algorithms pass a number of stringent statistical tests and produce pseudorandom numbers of very high statistical quality [49].

2.4. The Numerical Integration Kernel

On a GPU, the Langevin equations of motion can be solved simultaneously for all N particles in N threads working in parallel. When the particle based parallelization is utilized, the subroutines for the force computation can be incorporated into the integration kernel. This allows a programmer to use coordinate variables, stored locally in the GPU global memory, that are read only once at the beginning of the computational procedure and are passed to the next subroutine. Since all the interactions are more or less local, texture cache can be used as well to access the coordinates in the GPU global memory. When the interacting pair based parallelization is employed, the force computations can be performed in a separate kernel and the summation of all the forces (gathering kernel, Algorithm 5) can be done inside the integration kernel. Using one kernel for the force computation, the force summation and the numerical integration minimizes the number of kernel invocations on the CPU, thus, saving time for context switching on the GPU.

***Algorithm 6*:** Numerical integration of the Langevin equations of motion.

1. $i \leftarrow$ GPU thread index {same as particle index}

2. *rec* $r_i <= r[i](t_n)$ {reading coordinate of the i-th particle at the beginning of each step}

3. $f_i \leftarrow \sum_{\nu} f_{i,\nu}$ {total force exerted on the i-th particle due to several pair potentials}

4. $g_i \leftarrow (g_x, g_y, g_z)$ {3D vector with 3 normally distributed random numbers}

5. $r_i \leftarrow r_i + f_i h/\xi + g_i\sqrt{2k_BTh/\xi}$ {1-st order integration scheme}

6. $r_i \Rightarrow r[i](t_{n+1})$ {saving coordinates to global memory at the end of each step}

Each thread computes the displacement vector for just one particle. Once particle coordinates are retrieved from the GPU global memory via texture reference (line 2), they are used in the computational procedures that follow (the total force is computed in line 3). The corresponding forces $f_{i,\nu}$, where the index ν is running over different potential energy terms, are computed using ether the particle based or the interacting pair based parallelization approach. When the former approach is used, the entire computational procedure from the force computation to the random force generation, and to the numerical integration can be organized into a single kernel. In the latter case, an additional gathering kernel is needed to compute the total force (line 3). Continuing, particles are shifted to their new positions (line 5), which are saved to the GPU global memory (line 6). Since these coordinates are used at the next time step t_{n+1}, they have to be moved from the time layer $r[i](t_{n+1})$ to the time layer $r[i](t_n)$ at the end of each iteration.

3. THE SOP-GPU PROGRAM FOR LANGEVIN SIMULATIONS OF PROTEINS

3.1. The SOP Model

We employed the methodology for the GPU-based realization of Langevin dynamics to develop a CUDA program for biomolecular simulations fully implemented on a GPU. To describe the molecular force field, we adapted the Self Organized Polymer (SOP) model (SOP-GPU program) [44]. Previous studies have shown that the SOP model describes well the mechanical properties of proteins, including the Green Fluorescent Protein [54] and the tubulin dimmer [55]. In the SOP model, each residue is described using a single interaction center (C_α -atom). The potential energy function of a protein conformation V, specified in terms of the coordinates $\{r\} = r_1, r_2, ..., r_N$, is given by.

$$V = V_{FENE} + V_{NB}^{ATT} + V_{NB}^{REP} = -\sum_{i=1}^{N-1} \frac{k}{2} R_0^2 \log\left(1 - \frac{(r_{i,i+1} - r_{i,i+1}^0)^2}{R_0^2}\right) + \sum_{i=1}^{N-3} \sum_{j=i+3}^{N} \varepsilon_n \left[\left(\frac{r_{ij}^0}{r_{ij}}\right)^{12} - 2\left(\frac{r_{ij}^0}{r_{ij}}\right)^{6}\right] \Delta_{ij} + \sum_{i=1}^{N-2} \varepsilon_r \left(\frac{\sigma_{i,j+2}}{r_{i,j+2}}\right)^6 + \sum_{i=1}^{N-3} \sum_{j=i+3}^{N} \varepsilon_r \left(\frac{\sigma}{r_{ij}}\right)^6 (1 - \Delta_{ij}). \quad (1)$$

In Eq. (1), the finite extensible nonlinear elastic (FENE) potential V_{FENE} describes the backbone chain connectivity. The distance between two next-neighbor residues i and $i + 1$, is $r_{i,i+1}$, while $r_{i,i+1}^0$ is its value in the native (PDB) structure, and $R_0 = 2$ Å is the tolerance in the change of a covalent bond (first term in Eq. (1)). We used the Lennard–Jones potential (V_{NB}^{ATT}) to account for the non-covalent interactions that stabilize the native state (second term in Eq. (1)). We assumed that, if the noncovalently linked residues i and j ($|i-j| > 2$) are within the cutoff distance $R_C = 8$ Å, then $\Delta_{ij} = 1$, and zero otherwise. We used a uniform value for $\varepsilon_n = 1.5$ kcal/mol, which quantifies the strength of the non-bonded interactions. All the non-native interactions in the V_{NB}^{REP} potential are described as repulsive (third term in Eq. (1)). Additional constraint are imposed on the bond angle formed by residues $\sigma_{i,i+1}$, and $i + 2$ by including the repulsive potential with parameters $\varepsilon_r = 1$ kcal/mol and $i, i + 2 = 3.8$ Å, which determine the strength and the range of the repulsion. To ensure the self-avoidance of the protein chain, we set $\sigma_{i,i+2} = 3.8$ Å (last term in Eq. (1)).

3.2. Benchmark Simulations

We carried out test simulations of the mechanical unfolding for the all-β-strand domain *WW* from the human *Pin*1 protein (PDB code 1PIN, Table 1) using the SOP-GPU program. The rationale behind choosing this protein as a test system is two-fold. First, the *WW* domain is of particular interest to the field of protein folding and dynamics, and several research groups have expended considerable efforts to characterize the biophysical and biochemical properties of this protein [29, 56, 57]. Secondly, this is the smallest known independently folding all-β-domain and the all-β-protein architecture is the primordial structural state that can be studied experimentally using single-molecule force spectroscopic techniques such as AFM, and laser and optical tweezers [58, 59]. For these reasons, the WW domain has been extensively used in the theoretical exploration of protein folding and unfolding.

We consider the following principal sources of error: (1) precision issues arising from the differences in single precision (GPU) and double precision (CPU) IEEE floating point arithmetic, (2) possible read/write errors in the GPU global memory (hardware), and (3) accuracy of the SOP-GPU program, i.e. possible errors in the numerical routines (software). We report on our implementation of the SOP-GPU package on the NVIDIA GeForce GTX 295 (Section 2) and compare it against a dual Quad Core Xeon 2.66 GHz, considered to be representative of similar levels of technology. All CPU/GPU benchmarks have been obtained on a single GPU and a single CPU. To obtain the dynamics of the force-induced molecular elongation, the Langevin equations of motion for each residue r_i have been integrated numerically using the first-order integration scheme (in powers of the integration time step h) [60].

$$r_i(t + h) = r_i(t) + f(r_i(t))\xi h + G_i(t), \quad (2)$$

where $G_i(t)$ is the random force, and $f(r_i(t)) = -\dfrac{\partial V(r_i)}{\partial r_i}$ is the total force due to the covalent and the noncovalent interactions (Eq. (1)) exerted on the i-th particle. Benchmark simulations of the mechanical unfolding of the WW domain have been carried out at room temperature ($k_B T = 4.14$ pN/nm) over 4×10^8 iterations with

Number of residues, covalent bonds, native contacts stabilizing the folded state, and residue pairs for a range of proteins (*WW* -domain, *Ig*27, *C2A*-domain, γ*C* and β*C* chains), protein fibers (*Fb* monomer and dimer) and protein assembly (viral capsid *HK*97) used in the benchmark simulations.

Protein	*WW*[1]	*Ig*27[2]	*C2A*[3]	γ*C*[4]	*D*-dimer[5]	*Fb*[6]	$(Fb)_2^7$	*HK*97[8]
PDB-code	1PIN	1TIT	2R83[9]	1M1J[10]	1FZB	3GHG	3GHG	1FT1[11]
Residues	34	89	126	517	1062	1913	3849	115140
Covalent bonds	33	88	125	521	1072	1932	3839	114720
Native contacts	65	255	328	1770	3498	5709	12560	467904
Pairs	463	3573	7422	131101	558833	1821212	7389077	16178028[12]

Note: [1] All-β-strand *WW* -domain.
[2] *Ig*27 domain of human titin.
[3] *C2A*-domain from human synaptotagmin *Syt*1.
[4] β*C* and γ*C* domains from human fibrinogen *Fb*.
[5] Double-*D* fragment *(D–D* interface) of human fibrinogen *Fb*.
[6] Human fibrinogen monomer *Fb*.
[7] Human fibrinogen dimer $(Fb)_2$ created from two *Fb* monomers (3GHG) and the *D–D* interface (1FZB).
[8] *HK*97 is Head II viral capsid.
[9] *C2A* domain of human *Syt*1 protein.
[10] β*C* and γ*C* chains in human *Fb* starting fom the CY S ring in the *D*-domain.
[11] PDB code for a structural unit; the full *HK*97 capsid structure can be found in the Viper [64] database.
[12] Based on a cut-off distance of 200 Å.

the time step $h = 20$ ps, using the standard bulk water viscosity ($\xi = 7.0 \times 10^5$ pNps/nm). Each trajectory has been generated by fixing the *N*-terminal end and pulling the C-terminal end of the *WW* domain with the time-dependent mechanical force $f_{ext}(t) = r_f t$ in the direction corresponding to the end-to-end vector, and using the force-loading rate $r_f = \kappa v_0$, where $\kappa = 35$ pN/nm is the cantilever spring constant and $v_0 = 2.5$ μm/s is the pulling speed.

The results of the CPU- and the GPU-based computations are presented in Fig. 2, where we compare the force-extension curves $f(R)$ and the average temperature $\langle T \rangle$ for two representative trajectories of unfolding, and the distributions of unfolding forces $p(f^*)$ sampled from 260 trajectories on a CPU and 300 trajectories on a GPU. Temperature conservation ($d\langle T \rangle/dt$), mechanical work performed on the system ($w = \int_{R_0}^{R_{fin}} f(R)dR$) and the distribution of peak forces (f^*'s) are rigorous physical metrics for measuring the precision of molecular simulations. Aside from small deviations due to the different initial conditions, the profiles of $f(R)$ and $\langle T \rangle$, obtained on the CPU, are close to the profiles of the same quantities, generated on the GPU. A small drop in $\langle T \rangle$ is due to the onset of the unfolding transition in the *WW* domain, which occurs at $t \approx 0.15$ ms. Both the CPU- and the GPU-based calculations of the histogram of unfolding forces $p(f^*)$ result in similar values of the average force, i.e. $\langle f^* \rangle \approx 120.56$ pN on the CPU and $\langle f^* \rangle \approx 120.94$ pN on the GPU, but slightly different standard deviations of $\sigma_{f^*} \approx 5.83$ pN on the CPU versus $\sigma_{f^*} \approx 6.58$ pN on the GPU due to the small sample size (Fig. 2). The magnitude of the critical force for unfolding is well within the 60 pN – 200 pN force range observed for mostly β-strand single domain proteins [58, 61].

3.3. Accuracy of the Numerical Integrators

Langevin simulations fully implemented on a GPU enable one to obtain long trajectories of protein dynamics generated over as many as 10^9–10^{10} iterations. Consequently, there emerges a question about the numerical accuracy of the integration scheme used. In Langevin simulations of proteins, the equations of motion are solved numerically using the first-order integrator (Eq. (2)) [60]. However, the magnitude of the associated numerical error, which may, potentially, add up over many billions of iterations, is not known. We assessed the numerical accuracy of integration protocols by considering the mechanical unfolding of a protein. To test the results of simulations of the protein extension $\Delta X(t) = X(t) - X_0$ against the theoretical predictions, we used an exactly solvable model of a Brownian particle $X(t)$ evolving in a one-dimensional harmonic potential, $V(X) = K_{sp}(X - X_0)^2/2$, where X_0 is the equilibrium position and K_{sp} is the molecular spring constant [62].

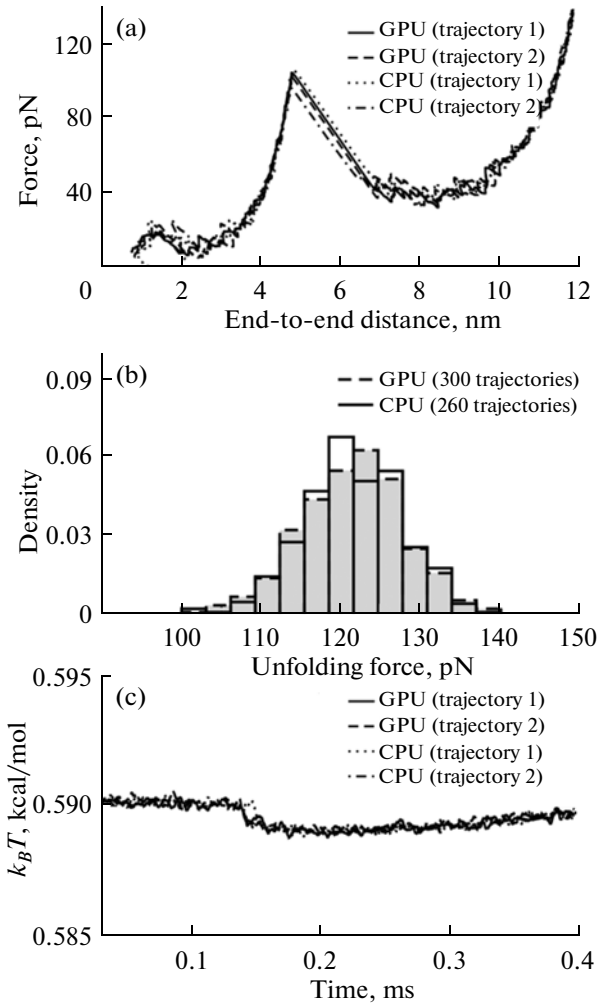

Fig. 2. Comparison of the results of pulling simulations for the all *WW*-domain (Table) obtained on a CPU and on a GPU (the color code is explained in the graphs). Panel (a): Representative examples of the dependence of mechanical tension experienced by the protein chain *f* as a function of the molecular extension R (force-extension curves) obtained using running averages over 500 data points. Panel (b): The histogram based estimates of the distribution of unfolding forces $p(f^*)$, i.e., peak forces f^* extracted from the force-extension curves. The histograms have been constructed using the bandwidth (bin size) of $h_{f^*} \approx 3.6$ pN. Panel (c): Representative examples of the time dependence of the average temperature of the protein chain $\langle T(t)\rangle$ (in units of k_BT), which correspond to the force-extension curves (panel (a)), obtained using running averages over 500 data points.

Under the non-equilibrium conditions of the time-dependent force application, $f_{ext}(t) = r_f t$, the average particle position (the end-to-end distance), computed theoretically, is given by

$$\langle X(t)\rangle_{th} = X_0 e^{-t/\tau} + \frac{r_f \tau}{\xi}(t - \tau(1 - e^{-t/\tau})), \tag{3}$$

where $\tau = \xi/K_{sp}$ is the characteristic timescale. In pulling simulations, the average particle position at the step $n + 1$, $\langle X(t_{n+1})\rangle$, where $t_N = nh$, can be obtained recursively from the average position obtained at the previous n-th step, $\langle X(t_n)\rangle$, using the first-order integration scheme

$$\langle X(t_{n+1})\rangle_{sim} = \langle X(t_n)\rangle + \left(\frac{r_f t_n}{\xi} - \frac{\langle X(t_n)\rangle}{\tau}\right)h \tag{4}$$

or the second-order integration scheme,

$$\langle X(t_{n+1})\rangle_{sim} = \langle X(t_n)\rangle + \left(\frac{r_f t_n}{\xi} - \frac{\langle X(t_n)\rangle}{\tau}\right)h + \left(\frac{\langle X(t_n)\rangle}{2\tau^2} - \frac{r_f t_n}{2\tau\xi}\right)h^2. \quad (5)$$

Hence, Eq. (3), (4), and (5) can be used to assess the accuracy of the numerical integrators.

We carried out calculations of $\langle \Delta X(t)\rangle$ at room temperature for $n = 10^9$ iterations using $X_0 = 0$ as initial condition, which corresponds to the initial molecular extension of $\Delta X(0) = 0$, $K_{sp} = 40$ pN/nm, which is within the 20–50 pN/nm range of values observed in the experimental force-extension curves of proteins. We used the time steps of $h = 1$, 25 and 50 ps. We set $\kappa = 10$ pN/nm and $v_0 = 1$ μm/s, which translates to $r_f = 10^{-5}$ pN/ns. These are typical values of a cantilever spring constant and a pulling speed used in AFM experiments. We set the diffusion constant to $D = k_B T/\xi = 1.5 \times 10^{-11}$ cm^2/s, which corresponds to the force-driven ~150 nm extension of the fibrinogen molecule observed in AFM experiments over time $t = 0.1$ s [21]. The slow force-driven "diffusion of the molecular extension," $\Delta X(t)$, described by the Brownian particle model with $D \approx 10^{-11}$ cm^2/s, should not be confused with the free Brownian diffusion of protein molecules in aqueous solution, for which D is in the 10^{-6}–10^{-8} cm^2/s range.

We found that the average extensions $\langle \Delta X(t_{n+1})\rangle_{sim}$, calculated using the first order (Eq. (4)) and the second order (Eq. (5)) integrators, agree very well with each other and with the theoretical curve of this quantity $\langle \Delta X(t)\rangle_{th}$ (Eq. (3)) for all values of h. The simulated data points of $\langle \Delta X(t_{n+1})\rangle_{sim}$ practically collapsed on the theoretical curve $\langle \Delta X(t)\rangle_{th}$ in the entire range of time (data not shown). To quantify any reduction in accuracy, we estimated the relative error of the molecular extension, $|\langle \Delta X(t_{n+1})\rangle_{sim} - \langle \Delta X(t)\rangle_{th}|/\langle \Delta X(t)\rangle_{th}$, accumulated at the end of each trajectory and averaged over 10^5 runs. The relative error was found to be less than 2×10^{-5} (1×10^{-5}) for the first-order (second-order) scheme, significantly below the 10^{-3} level considered the acceptable maximum for relative error in biomolecular simulations. These results show that in a stochastic thermostat (random force) the numerical integration errors, associated with the calculation of the global mechanical reaction coordinate $X(t)$, are minimal and/or cancel out, and that single precision arithmetic is adequate for production runs.

Hence, in the context of long simulations of biomolecules on a GPU, the first-order integrator (Ermak-McCammon algorithm) can be used to describe accurately their mechanical properties under physiologically relevant conditions of force application.

3.4. Performance Measurements

We have compared the overall performance of an end-to-end application of the SOP-GPU program with the heavily tuned CPU-based implementation of the SOP model (SOP-CPU program) in describe ing the Langevin dynamics of the domain WW at equilibrium (Fig. 3, Table 1). To fully occupy the GPU resources, we profiled the computational performance of the SOP-GPU program as a function of the number of independent trajectories running concurrently on a single GPU. We refer to this as the "many-runs-per-GPU" approach. Alternatively, we could have assessed the performance of the program by running one trajectory on a single GPU, but for a range of systems of different size N, which we refer to as the "one-run-per-GPU" approach. The results obtained indicate that for a small system of 34 residues (WW domain, Table 1), the use of a single GPU device allows one to accelerate simulations starting from 3 independent runs (for small systems, there is insufficient parallelism to fully load the GPU), which is also equivalent to running one trajectory on a single GPU for a system of ~10^2 residues, such as the domains *Ig27* and *C2A* (Table 1). While the simulation time on the CPU scales linearly with the number of runs, the scaling in this regime on the GPU is sublinear (nearly constant) until the number of runs is ~500. At this point, depending on the number of threads per thread block, the GPU shows significant performance gains relative to the CPU reaching its maximum 80–90-fold value (the speedup is shown in the inset of Fig. 3b). We ran the simulations long enough to converge the speedup ratio ($n = 10^6$ steps of size $h = 40$ ps). Beyond this point, the GPU device is fully subscribed and the execution time scales linearly with the number of runs as on the CPU.

In general, the total number of threads of execution $M_{th} = m_B B$, defined by the number of thread blocks m_B of size B, is roughly equal the product of the system size N and the number of trajectories s running concurrently on the GPU, i.e. $M_{th} = Ns$. Because it is impossible to predict which block size B will result in the best performance, we carried out benchmark computations for $B = 64$, 128, 256 and 512 for purposes of performance comparison. Our results indicate that all ALUs must to be fully loaded and that M_{th} should exceed the number of ALUs on a single GPU by a factor of 10^{-15}. For example, on a graphics card with 240 ALUs (GeForce GTX 280 or GTX 295, or Tesla C1060), $M_{th} \approx 5000$–10000, and in the case of

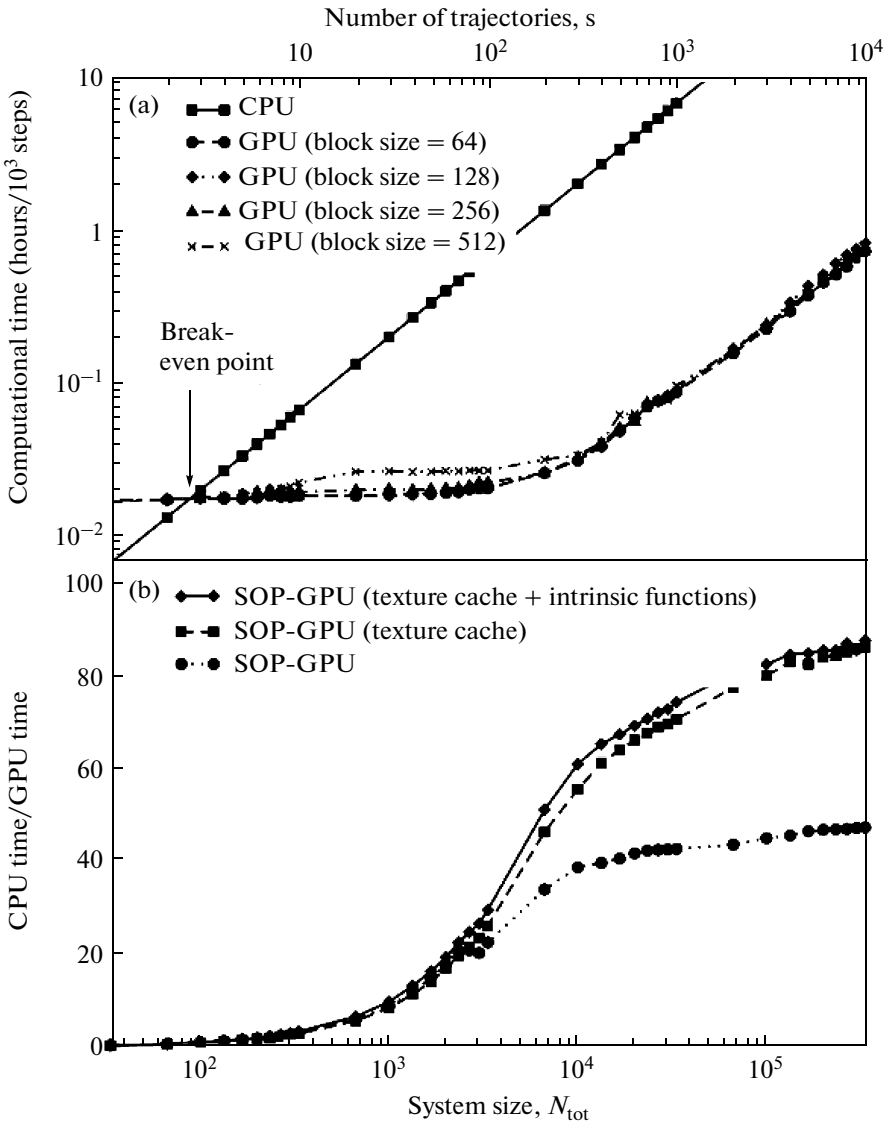

Fig. 3. The many-runs-per-GPU approach: shown are the simulation time on the CPU and on the GPU on a log-log scale and the relative CPU/GPU performance (computational speedup) of the end-to-end application of the SOP-GPU program (the inset) as a function of the number of equilibrium simulation runs for the all-β-strand *WW* domain. While a single CPU core generates only one trajectory at a time, a GPU device is capable of running many independent trajectories at the same time. The relative CPU/GPU performance is tested for the thread block size of $B = 16$, 64, and 256 computational threads of execution.

WW domain ($N = 34$) this translates to $s \approx 100$ (Fig. 3). This implies that the use of small thread blocks is more advantageous when $M_{th} \leq 3000$, e.g., when simulating one trajectory for a system of the size of fibrinogen monomer *Fb* ($N \approx 2000$) or 30 runs for a system with $N \approx 100$ (*Ig27*), or ~50 runs for a smaller system such as the *WW* domain ($N = 34$). However, larger thread blocks should be used when $M_{th} > 3000$ to simulate a few trajectories for larger systems such as the fibrinogen dimer $(Fb)_2$ ($N = 3849$) or to obtain one trajectory for a very large system, e.g., the viral capsid *HK*97 ($N = 115$, 140, see Table 1).

To profile the associated computational time and memory demand for the SOP-GPU software we ran a series of benchmark simulations on several different protein systems (Table 1) using the one-run-per-GPU approach. The execution time of an end-to-end application of the program as a function of particle count was recorded using the standard CUDA runtime profiling tool. To more closely assess the performance characteristics of the SOP-GPU code as a function of the system size N, we analyzed one simulation run for each protein system, generated over $n = 10^6$ steps of size $h = 40$ ps and block size of $B = 64$ threads (Fig. 4). For all test systems of less than ~3.000 residues the associated simulation time remains roughly the same, i.e.

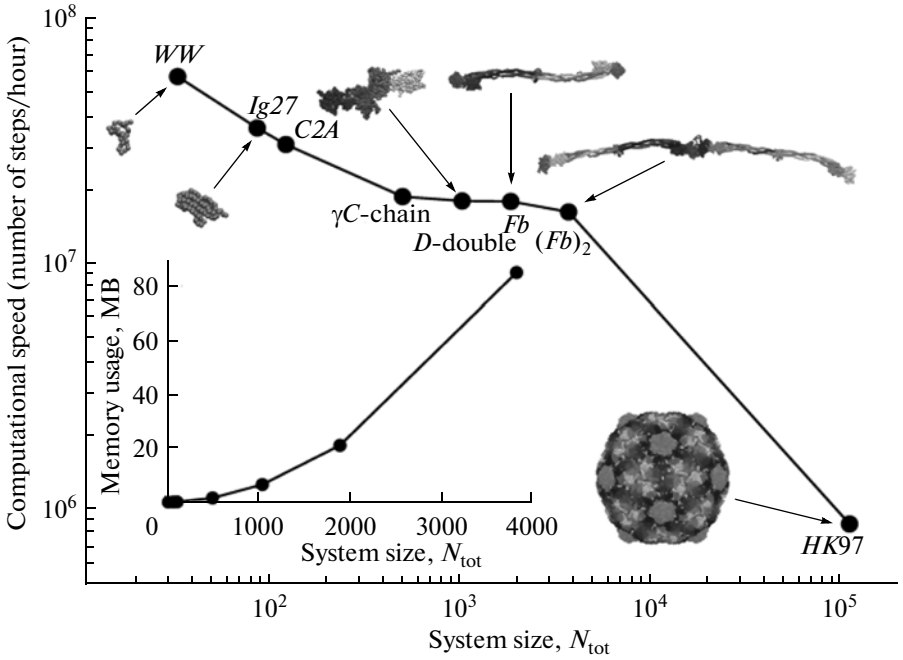

Fig. 4. The one-run-per-GPU approach: shown are the simulation time on the GPU on a log-log scale and the memory usage (the inset) on a GPU as a function of the system size (number of residues) N_{tot}. Results are presented for several test systems, including small proteins (the *WW*-domain, the *Ig27* domain from human titin, the domain *C2A* from human synaptotagmin Syt1), large proteins fragments (the γ*C* chain and the double-D fragment from human fibrinogen *Fb*), long protein fibers (the *Fb* monomer and dimer), and large-size protein assembly (the viral capsid *HK*97). The information about the native state topology for these biomolecules is summarized in Table.

(1–3) ~10^{-4} seconds per step. This is not surprising since $M_{th} > 3000$ is the amount of threads needed to fully utilize the GPU resources. For larger systems ($N > 3000$), the computational time scales roughly linearly with N (Fig. 4). Small variation in runtime versus N around the monotonic linear dependence is due to the different native topology of the test systems, i.e. the number of native and non-native contacts in the PDB structure (Table 1). The amount of on-board memory in contemporary graphics cards – ~1 GB (GeForce GTX 200 series) and 4GB (Tesla C1060) – is sufficient for Langevin simulations of large biomolecular systems comparable in size with the fibrinogen dimer $(Fb)_2$. The amount of on-board memory will most likely increase with the next generation of graphics cards with new Fermi architecture from NVIDIA (up to ~6 GB) [63].

4. CONCLUSION

We have developed and tested, to the best of our knowledge, the first GPU-based implementation of Langevin simulations of biomolecules, where the particle based and the interacting pair based approaches have been employed in the parallel computation of forces due to the covalent and non-covalent interactions governed by the standard pair potentials. We have presented the numerical routines for the generation of pseudorandom numbers using the Hybrid Taus algorithm to describe random collisions of a biomolecule with solvent molecules, for the construction of Verlet lists, and for the numerical integration of Langevin equations of motion based on the first order scheme. Although we focused on the C_α-based coarse-grained SOP model of a protein, which involves only two-body potentials in the potential energy function, the developed formalism can be used in conjunction with more sophisticated biomolecular force fields to explore, e.g., protein-protein and protein-DNA interactions, and it can also be extended to include the three-body (angle) potentials to describe side chains. In addition, the numerical integration kernel can be modified to follow the Langevin dynamics in the underdamped limit, in order to describe the thermodynamics of biomolecular transitions.

The developed formalism has been mapped into a standard CUDA code for Langevin simulations of biomolecules (SOP-GPU program). Benchmark simulations have shown that for a test system of the all-*WW* domain the results of simulations of the mechanical denaturation on the GPU agree well with the

results obtained on the CPU (Fig. 2). In a separate work, we also compared the forced unfolding data, obtained on the CPU and on the GPU, for the human synaptotagmin Syt1 [54] and for the human fibrinogen *Fb* molecules [55], and found that the results of the CPU- and GPU-based computations agree very well. Using an exactly solvable model of a Brownian particle evolving in a harmonic potential, we have assessed the accuracy of the numerical integration of Langevin equations of motion in describing the force-induced elongation of a protein chain. We found that using the first-order integrator is sufficient to accurately describe the force-driven elongation of a protein over many billions of iterations.

GPUs can be utilized to generate a few trajectories of Langevin dynamics for a large system of many thousands of residues (one-run-per-GPU approach). This is the major approach used by many researchers in this field. As it was also shown, GPU can be utilized to obtain many trajectories for a small system composed of a few hundreds of amino acids (many-runs-per-GPU approach). This approach can be used to get a statistically significant set of results for the direct comparison of the theoretical and experimental distribution functions. Important point made in this work is that an efficient utilization of texcture cache allows one to obtain on a GPU a computation speed-up of about 90 times compared to CPU.

The results obtained attest to the accuracy of the SOP-GPU program. The SOP-GPU software can now be utilized to describe the mechanical properties of proteins, the strength of the noncovalent bonds that stabilize protein-protein complexes and aggregates, and the physical properties of large-size protein assemblies. A combination of the SOP model and the GPU-based computations enables one to carry out molecular simulations in reasonable wall-clock time in order to explore the unfolding micromechanics of protein on the timescale of 0.01–0.1s. This allows one to interpret the experimental force-extension curves and force-indentation profiles of biomolecules, obtained in dynamic force spectroscopy assays, thus, bridging the gap between theory and experiments. For example, on a GPU GeForce GTX 280 or GTX 295 it takes only ~4 days to generate a single unfolding trajectory for the fibrinogen monomer *Fb* using experimental pulling speed of 2.5 μm/s. By contrast, it would take as long as ~8 months to obtain just one trajectory using the CPU version of the program on a 2.66 GHz Intel Core i7 with 6GB of memory. It takes ~20 days to generate one force-indentation curve for the viral capsid *HK*97 on a single GPU (GeForce GTX 200 series, Tesla C1060) using experimental pulling speed of 10 m/s. Identical simulation run on a CPU would take about 5 years.

Beyond that, we note that due to rapid evolution of GPU hardware, the simulation time on the GPU will decrease significantly with the introduction of the MIMD (Multiple Instruction Multiple Data) based Fermi architecture (NVIDIA) in the near future [63]. Presented SOP-GPU program can be easily adapted for the new hardware which will presumably allow one to carry out simulations of many more biological systems for which experimental results already obtained. The presented methodology can be expanded for the simulations of other biological systems such as molecular motors, nucleosomes, lyposomes, etc.

ACKNOWLEDGMENTS

This project was supported by the grant (no. 09–0712132) from the Russian Foundation for Basic Research.

REFERENCES

1. T. P. Stossel, J. Condeelis, L. Cooley, J. H. Hartwig, A. Noegel, M. Schleicher, and S. S. Shapiro, Nat. Rev. Mol. Cell Biol. **2**, 138–145 (2001).
2. C. P. Johnson, H. Y. Tang, C. Carag, D. W. Speicher, and D. E. Discher, Science **317**, 663–666 (2007).
3. R. Paul, P. Heil, J. P. Spatz, and U. S. Schwarz, Biophys. J. **94**, 1470–1482 (2008).
4. D. Leckband, Curr. Opin. Struct. Biol. **14**, 523–530 (2004).
5. R. P. McEver, Curr. Opin. Cell Biol. **14**, 581–586 (2002).
6. B. T. Marshall, M. Long, J. W. Piper, T. Yago, R. P. McEver, C. Zhu, Nature **423**, 190–193 (2003).
7. V. Barsegov and D. Thirumalai, Proc. Natl. Acad. Sci. USA **102**, 1835–1839 (2005).
8. J. W. Weisel, Biophys. Chem. **112**, 267–276 (2004).
9. J. W. Weisel, Science **320**, 456–457 (2008).
10. S. T. Lord, Curr. Opin. Hematol. **14**, 236–241 (2007); G. E. P. Box and M. E. Miller, "A Note on the Generation of Normal Random Deviates," Ann. Math. Stat **29**, 610–611 (1958).
11. M. R. Falvo, S. Washburn, R. Superfine, M. Finch, J. F. P. Brooks, et al., Biophys. J. **72**, 1396–1403 (1997).
12. C. Uetrecht, C. Versluis, N. R. Watts, W. H. Roos, G. J. L. Wuite, et al., Proc. Natl. Acad. Sci. USA **105**, 9216–9220 (2008).

13. Y. G. Kuznetsov, S. Daijogo, J. Zhou, B. L. Semler, A. J. McPherson, Mol. Biol. **347**, 41–52 (2007).
14. N. Kol, Y. Shi, D. Barlam, R. Z. Shneck, M. S. Kay, et al., Biophys. J. **92**, 1777–1783 (2007).
15. I. L. Ivanovska, P. J. de Pablo, B. Ibarra, G. Sgalari, F. C. MacKintosh, et al., Proc. Natl. Acad. Sci. USA **101**, 7600–7605 (2004).
16. I. Ivanovska, G. Wuite, B. Joensson, and A. Evilevitch, Proc. Natl. Acad. Sci. USA **104**, 9603–9608 (2007).
17. A. C. Steven, B. H. J., N. Cheng, B. L. Trus, and J. F. Conway, Curr. Opin. Struct. Biol. **15**, 227–236 (2005).
18. M. Carrion-Vazquez, H. Li, H. Lu, P. E. Marszalek, A. F. Oberhauser, and J. M. Fernandez, Nat. Struct. Biol. **10**, 738–743 (2003).
19. I. Schwaiger, C. Sattler, D. R. Hostetter, and M. Rief, Nature Mat. **1**, 232–235 (2000).
20. J. Brujic, R. I. Hermans, K. A. Walther, and J. M. Fernandez, Nature Phys. **2**, 282–286 (2006).
21. A. E. X. Brown, R. I. Litvinov, D. E. Discher, and J. W. Weisel, Biophys. J. **92**, L39–L41 (2007).
22. D. E. Smith, S. J. Tans, S. B. Smith, S. Grimes, D. L. Anderson, and C. Bustamante, Nature **413**, 748–752 (2001).
23. W. H. Roos, I. L. Ivanovska, A. Evilevitch, and G. J. L. Wuite, Cell. Mol. Life Sci. **64**, 1484–1497 (2007).
24. B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus, J. Comp. Chem. **4**, 187–217 (1983).
25. J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten, J. Comp. Chem. **26**, 1781–1802 (2005).
26. H. J. C. Berendsen, D. van der Spoel, and R. van Drunen, Comp. Phys. Comm. **91**, 43–56 (1995).
27. B. Isralewitz, M. Gao, and K. Schulten, Curr. Opin. Struct. Biol. **11**, 224–230 (2001).
28. J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, J. Comp. Chem. **28**, 2618–240 (2007).
29. P. L. Freddolino, F. Liu, M. Gruebele, and K. Schulten, Biophys. J. **94**, L75–L77 (2008).
30. M. Zink and H. Grubmueller, Biophys. J. **96**, 1767–1777 (2009).
31. ATI Stream Computing Technical Overview, AMD, 2009.
32. NVIDIA CUDA Programming Guide, version 2.3.1, NVIDIA, 2009.
33. NVIDIA CUDA C Programming Best Practices Guide, version 2.3.1, NVIDIA, 2009.
34. A. Munshi, The OpenCL Specification, version, 1.0, Khronos OpenCL Working Group, 2009.
35. C. I. Rodrigues, D. J. Hardy, J. E. Stone, K. Schulten, and W.-M. W. Hwu, "GPU Acceleration of Cutoff Pair Potentials for Molecular Modeling Applications. CF', 08," in *Proceedings of the 5th conference on Computing frontiers* (New York, NY, USA, 2008), pp. 273–282.
36. J. C. Phillips, J. E. Stone, and K. Schulten, "Adapting a Message-Driven Parallel Application to GPU-Accelerated Clusters. SC', 08," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Piscataway, NJ, USA, 2008), pp. 1–9.
37. M. S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A. L. Beberg, D. L. Ensign, C. M. Bruins, and V. S. Pande, J. Comp. Chem. **30**, 864–872 (2009).
38. J. E. Davis, A. Ozsoy, S. Patel, and M. Taufer, "Towards Large-Scale Molecular Dynamics Simulations on Graphics Processors. BICoB', 09," in *Proceedings of the 1st International Conference on Bioinformatics and Computational Biology* (Berlin, Heidelberg, 2009), pp. 176–186.
39. J. A. van Meel, A. Arnold, D. Frenkel, S. F. P. Zwart, and R. Belleman, Mol. Simul. **34**, 259–266 (2008).
40. J. A. Anderson, C. D. Lorentz, and A. Travesset, J. Comp. Phys. **227**, 5342–5359 (2008).
41. V. Tozzini, Curr. Opin. Struct. Biol. **15**, 144–150 (2005).
42. C. Clementi, H. Nymeyer, and J. N. Onuchic, J. Mol. Biol. **298**, 937–953 (2000).
43. T. Veitshans, D. Klimov, and D. Thirumalai, Folding and Design **2**, 1–22 (1997).
44. C. Hyeon, R. I. Dima, and D. Thirumalai, Structure **14**, 1633–1645 (2006).
45. C. Hyeon and J. N. Onuchic, Proc. Natl. Acad. Sci. USA **104**, 2175–2180 (2007).
46. D. van der Spoel, E. Lindahl, B. Hess, C. Kutzner, A. R. van Buuren, E. Apol, P. J. Meulenhoff, D. P. Tieleman, A. L. T. M. Sijbers, K. A. Feenstra, R. van Drunen, and H. J. C. Berendsen, GROMACS User Manual, version 4.0, The GROMACS development team, 2009.
47. D. Levesque, L. Verlet, and Kürkijarvi, J. Phys. Rev. A **7**, 1690–1700 (1973).
48. G. E. P. Box and M. E. Miller, Ann. Math. Stat. **29**, 610–611 (1958).
49. *GPU Gems 3*, Ed. by H. Nguyen, (Addison-Wesley, 2008).
50. R. C. Tausworthe, Math. of Comp. **19**, 201–209 (1965).
51. P. L'Ecuyer, Math. of Comp. **65**, 203–213 (1996).
52. G. Marsaglia, Published on sci.crypt.
53. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, Numerical Recipes in C, *The Art of Scientific Computing*, 2nd ed., (Cambridge University Press, 1992).

54. R. I. D. Mickler, H. Dietz, C. Hyeon, D. Thirumalai, M. Rief, Proc. Natl. Acad. Sci. USA **104**, 20268–20273 (2007).
55. R. I. Dima and H. Joshi, Proc. Natl. Acad. Sci. USA **105**, 15743–15748 (2008).
56. N. Ferguson, C. M. Johnson, M. Macias, H. Oschkinat, and A. R. Fersht, Proc. Natl. Acad. Sci. USA **98**, 13002–13007 (2001).
57. J. Karanicolas, III C. L. B., Proc. Natl. Acad. Sci. USA **100**, 3954–3959 (2003).
58. M. Rief, M. Gautel, F. Oesterhelt, J. Fernandez, and H. Gaub, Science, **276**, 1109–1112 (1997).
59. H. Dietz and M. Rief, Proc. Natl. Acad. Sci. USA **101**, 16192–16197 (2004).
60. D. L. Ermak and J. A. McCammon, J. Chem. Phys., **69**, 1352–1360 (1978).
61. M. Carrion-Vazquez, A. F. Oberhauser, T. E. Fisher, P. E. Marszalek, H. Li, J. M. Fernandez, Prog. Biophys. Mol. Biol. **74**, 63–91 (2000).
62. M. Doi and S. Edwards, *The Theory of Polymer Dynamics, International Series of Monographs on Physics* (Oxford Science Publications, 1988).
63. NVIDIA'S Next generation CUDA Compute Architecture: Fermi, version 1.1, NVIDIA, 2009.
64. M. Carrillo-Tripp, C. M. Shepherd, I. A. Borelli, S. Venkataraman, G. Lander, P. Natarajan, J. E. Johnson, C. L. Brooks I., and V. S. Reddy, Nucl. Acid. Res. **37**, D436–D442 (2009).
65. L. Duan, A. Zhmurov, V. Barsegov, and R. I. Dima, J. Phys. Chem. B. **115**, 10133–10346 (2011).
66. A. Zhmurov, A. E. X. Brown, R. I. Litvinov, R. I. Dima, J. W. Weisel, and V. Barsegov, Structure **19**, 1615–1624 (2011).