Digital Modelling Of Protein Systems Using Monte-Carlo Simulation

Anna Yaremko
Faculty of Applied Sciences
Ukrainian Catholic University
L'viv, Ukraine
yaremko.pn@ucu.edu.ua

Bohdan Pavliuk
Faculty of Applied Sciences
Ukrainian Catholic University
L'viv, Ukraine
pavliuk.pn@ucu.edu.ua

Iryna Kokhan
Faculty of Applied Sciences
Ukrainian Catholic University
L'viv, Ukraine
kokhan.pn@ucu.edu.ua

Petro Mozil
Faculty of Applied Sciences
Ukrainian Catholic University
L'viv, Ukraine
mozil.pn@ucu.edu.ua

Taras Patsagan
Institute of Condensed Matter Physics
National Academy of Sciences of Ukraine
L'viv, Ukraine
tarpa@icmp.lviv.ua

***Abstract*—This document describes the general workflow of particle simulation using patchy particles and the structure of the particular realization of it for two ensembles: NVT and μVT. The implementation relies on CUDA to improve performance, and a big part of the work focuses on describing the optimizations for parallel implementation of the simulation.**

***Index Terms*—Digital simulation, Monte Carlo simulation parallelization**

I. Introduction

In this work, we simulate patchy particle systems with the Monte-Carlo Metropolis algorithm. The general algorithm is described in [4].

The patchy particle model is useful for particle simulations at different scales. Particles have anisotropic properties, the most popular being chemical potential, patches upon the particles, or both. The particles are hard spheres that repel each other, and the patches attract.

The interaction can be described by two potentials: interaction between spheres and interaction between patches. A square-well potential is often used due to its simplicity of calculation.

Monte-Carlo particle simulation is one of the two popular methods for particle simulation, the other being Molecular Dynamics (MD). For generating moves, the system uses a Markov chain and then accepts the move at random. The Markov chain does not have to follow an actual physically allowed process and thus could be optimized more than MD.

We first implement the algorithm for patchy particle simulation and then optimize the algorithm. We found three ways for optimization:

- **Parallelization of random number generation.**
- **Parallelization of energy calculation.**
- **Parallelization of particle moves.**

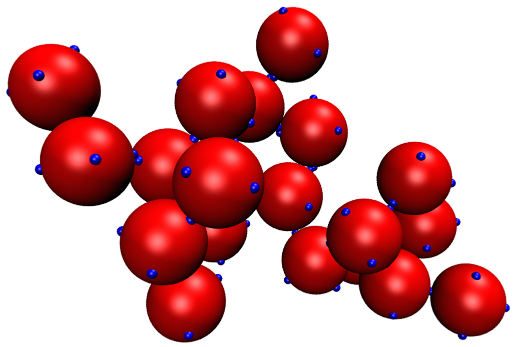
Fig. 1. An illustration of patchy particles. Image by [5].

II. Theoretical overview

A. Definitions

With the aim of preventing misunderstandings, here we provide definitions for some of the ambiguous terms:

1) **Device memory** – a term used by CUDA for memory on the GPU.
2) **Host memory** – a term used by CUDA for memory on the CPU.
3) **Single Bond Per Patch Condition** – a condition required by simulation algorithms to simplify calculations. It states that a patch can only bond with one patch at a time. It is enforced by setting the radius of the patch to ≤ 0.119 of the radius of the particle.
4) **Cell** – a leaf of the octree data structure. A part of the space (usually of cubic shape) that contains a certain number of particles.

B. Ensembles overview

The NVT ensemble, known as the canonical ensemble, fixes the volume, temperature, and number of particles in the system. The fixed number of particles is especially useful for simulation as it simplifies calculations of the system's energy.

In the NVT ensemble, there are two types of actions:

- moving a particle,
- rotating a particle.

Both of these actions could be performed as a single one, known as a rototranslation, but for the purpose of clarity, we implement these as two separate actions. For each move, only the repulsive potential between hard spheres and attraction between particles is calculated.

The μVT ensemble, known as the grand canonical ensemble, fixes the volume and the temperature as NVT does, but instead of the number of particles, μVT fixes the chemical potential of the system.

In the μVT ensemble, there are three types of actions:

- moving a particle,
- rotating a particle,
- adding or deleting a particle.

For the μVT ensemble, the moves related to the NVT ensemble are still valid, so the calculation of probabilities of moving and rotating a particle is unchanged. However, there is also a new function describing the probability of adding or deleting a particle.

C. Implementation details

We only implemented the NVT and μVT ensembles in this work. The project uses an acceleration structure akin to an oct-tree to optimize particle intersection calculation and implements the square-well and Lennard-Jones potentials – a mathematical model that describes the interaction between a pair of neutral atoms or molecules, which is widely used in molecular physics to approximate the behavior of non-bonded interactions in a system.

We use the Metropolis heuristic, which adds a chance of rejecting the particle movement even after a proper move, and the probability of that is calculated by taking the difference between the system's energy before and after the move.

The particles are stored in a container with the number of particles and the size of the simulation box. The box also contains a pointer to an array in device memory. On each successful particle move, the particle is copied to that array as if updating the position of the particle.

The particles are represented by hard spheres, and the patches on them are just points on a sphere, represented by a quaternion. The particles have three properties:

- radius,
- position,
- patches.

Note that the particle record does not contain an orientation position.

The patch has one property:

- orientation.

Each cell has a list of particle indices.

The cell view contains all the info about the system:

- the particle box,
- the split cells,
- functions for particle energy calculation and movement.

We use the Kern-Frenkel model to simulate patch-patch interactions. In this model, each particle can have an arbitrary number of patches at arbitrary positions on the particles. Each particle's radius is 0.119 of the radius of its parent particle. When the particle is rotated, the patches' positions are simply rotated. Since the position of the patch is a unit quaternion, we could simply rotate them on the particle.

For calculating the patch-patch potential, we first calculate the vector between the particles. If its magnitude is more than 1.119, the patches cannot interact, so we return 0. Otherwise, we calculate the degree between the vector and orientation of the first patch. If the cosine of this angle is more than the given value (the $\cos\max^{\theta}$), we go on to calculate the same value of cosine for the patch on the second particle. If the cosine is more than $\cos\max^{\theta}$, we return the energy of the interaction, else return 0.

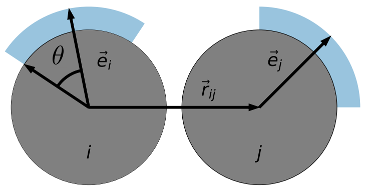

Fig. 2. Illustration of the Kern-Frenkel calculation of patch-patch interaction.

For the particle-particle interaction, we use a simple square well potential. It is implemented as:

$$V(x) = \begin{cases} -V_0 & \text{for } 0 \leq x \leq \theta^{\max}; \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

where $\theta^{\max}$ is the given parameter for the width of the square well, and V_0 is the parameter that describes the energy of the interaction.

Since, for the two ensembles (**NVT** and μ**VT**), the Single Bond Per Patch Condition must be met, in our simulation, the radius of the particles is exactly 0.119 of the particle radius. We can, thus, simply return the said value for the patch-patch energy after finding a single valid patch-patch bond.

III. Potential optimizations

The simulation can be parallelized in two ways: parallelizing energy calculation and simulating multiple particles at once. Parallelizing energy calculation is very straightforward, but parallelizing moves for each particle are not. However, because we use square-well potential, it can be done – if particles are further than a given range, they do not affect each other. Another way is optimizing **RNG** since the host-side generation of random numbers is sequential, and the demand for random numbers is high in Monte-Carlo simulations.

A. *Parallel **RNG***

Generating random numbers on the GPU is not complicated since CUDA provides CURand, a library for random number generation both on the host and on the device. It also provides quasi-random number generators.

The algorithm of parallel random number generation is simple – to generate N random numbers N **RNG** states must be created in the device memory. Then, call the device code that generates the random numbers and copy them to an appropriate location.

The biggest problem with this is that when more than approximately 1000 random numbers are needed, more than 1000 threads are to be run in parallel. Since this may not be possible on older devices, our implementation launches 256 threads at once in multiple blocks. Blocks in CUDA are not necessarily executed in parallel. Thus, the generation may slow down for a great amount of random numbers. Still, it speeds up the simulation approximately thrice as fast as host **RNG**.

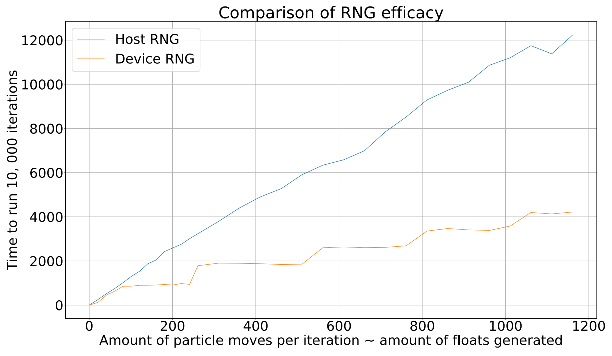

Fig. 3. Comparison of host vs device RNG runtimes speed.

B. *Parallel energy calculation*

Since an octree to optimize energy calculation is readily available, parallelizing the calculation is simple: simply iterate over the particles of each of the neighboring cells in parallel. In essence, we split the work of calculating the energy between particles in between threads in such a way that each thread only iterates over the particles in one cell.

To figure out which cell the particle belongs to, we use the following formula:

$$\mathbf{p\_idx} = \frac{p.x}{cell\_size.x} * cells\_per\_axis^2 + \frac{p.y}{cell\_size.y} * cells\_per\_axis + \frac{p.z}{cell\_size.z}$$

To check which cells neighbor the particle, we simply add/subtract the size of a cell's side to the particle's position. The parallelization process is straightforward since it is just packing a triple for loop into a CUDA kernel call.

When computing sparse systems, however, it may happen that the CPU simulation is faster – the overhead of synchronizing particle positions on the GPU may not be worth the faster calculations. For this reason, we only calculate the energy on the device when a cell has more than 80 particles in it.

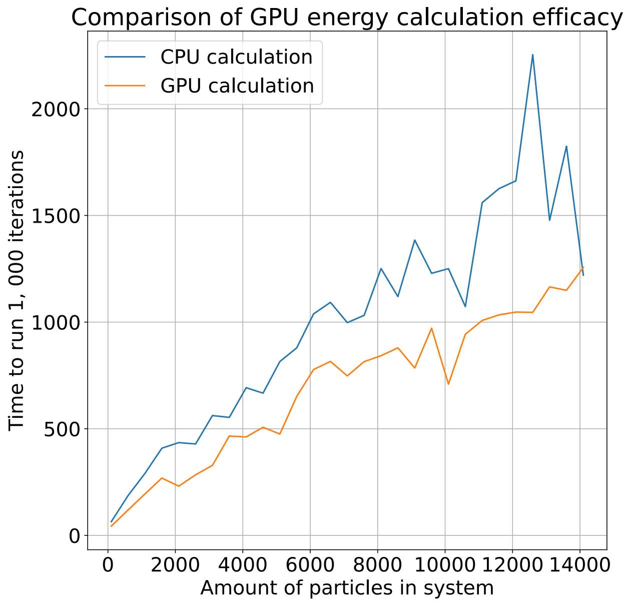

Fig. 4. Comparison of CPU vs device GPU calculations speed.

C. *Parallelizing particle movement*

This task is the most complicated conceptually since particles influence each other irrespective of distance in a real system. For the system with square-well potentials, however, for a particle of radius r_1, only particles in the range $2 * r_1 + \sigma$ matter. Hence, if we set the maximal move distance for a particle to be λ, we can move two particles of radii r_1 and r_2 at a distance greater than $2 * \max(r_1, r_2) + \sigma + \lambda$ in parallel, without side effects.

There are two possible approaches to this operation:

- Select N particles at random while they are at a great enough distance.
- Select random particles from cells selected in a ”chessboard-like” pattern.

The first method is useful, since no change of selection must be done. For the second method, the cells selected must alternate, and that introduces complexity. The first method, however, may be fairly inefficient, as it requires more random number generation.

Using parallel particle movement on the CPU doesn't lead us to any speedup, the reason for the result has to be suboptimal use of movement particles, because all threads lock our whole structure to lead it unchanged by other threads.

We also have domain decomposition calculation on GPU, but only positions, without implementation of parallel energy calculation, since this idea is architecturally incompatible with the previously implemented parallel energy calculation of particles.

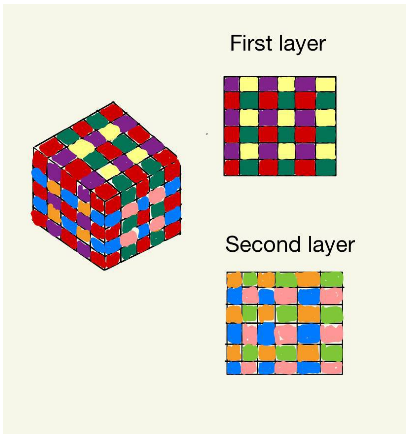

Fig. 5. Illustration of alternating cell selection.

IV. Results

Composing the random numbers generation, energy calculation and domain decomposition gave us a speedup of approximately 3.5 times. The amount of communication on each iteration was a bottleneck – to achieve further speedup, a rework of how we stored particles is required.

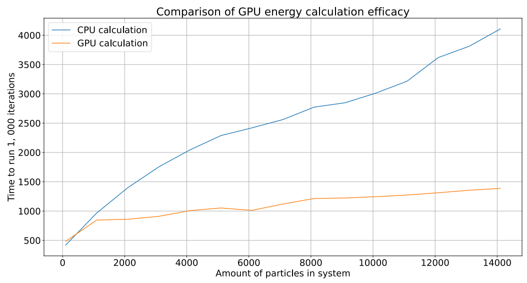

Fig. 6. Comparison of run times for initial and final versions.

To validate the correctness of our algorithm, we calculated the radial calculation function and normed it in time.

We used the following parameters for the system:

- particle radius is 0.5,
- T = 4,
- 2 patches, top and bottom,
- box: 30x30x30,
- 5 cells per axis, 125 total,
- 16000 particles,
- Yukawa potential: $A = -0.5,\ \alpha = 1.5,\ \sigma = 1.0$.

V. Conslusion

In this work, we implemented a program to simulate patchy particle systems. We implemented two ensembles: μVT and NVT. To speed up the simulation, we implemented energy calculation on GPU and parallel particle movement through domain decomposition.

The main concern for parallelization of this task is the amount of communication for each step. Each time a particle is moved, its data has to be updated in host memory, and its new position has to be moved to device memory. To somewhat optimize the constant communication, batched the particle movement to the device, but the batches could not be too big since then the old particle positions would not represent the current state of the system. Our implementation using the GPU has achieved a speedup of 3.5 times as compared to code running just on the host.

The code is available on GitHub [7].

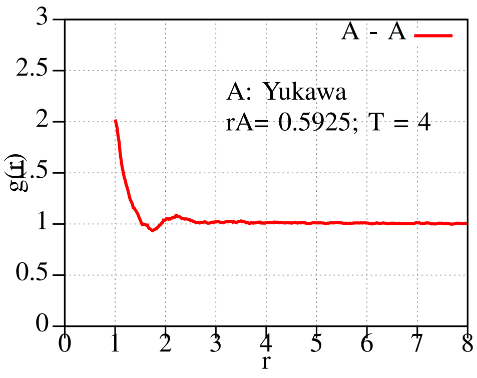

Fig. 7. RDF for a system with Yukawa potential.

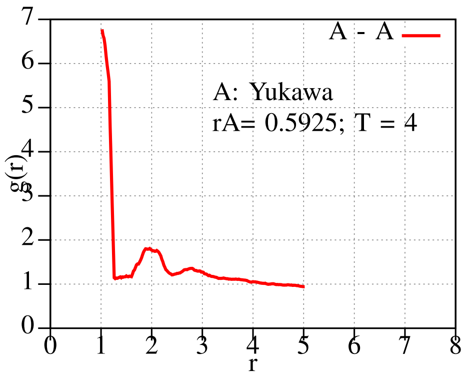

Fig. 8. RDF for a system with Yukawa potential and kern-Frenkel patch potential.

References

[1] Allen, M.P., Tildesley, D.J., Computer simulation of liquids. Oxford university press, 2017. https://doi.org/10.1093/oso/9780198803195.001.0001
[2] Bianchi, E., Blaak, R. and Likos, C.N., Patchy colloids: state of the art and perspectives. Physical Chemistry Chemical Physics, 13 (2011) 6397. https://doi.org/10.1039/c0cp02296a
[3] Gong, Z., Hueckel, T., Yi, G.R. and Sacanna, S., Patchy particles made by colloidal fusion. Nature, 550 (2017) 234. https://doi.org/10.1038/nature23901
[4] Lorenzo Rovigatti, John Russo, Flavio Romano, How to simulate patchy particles https://arxiv.org/abs/1802.04980
[5] Image of patchy particles https://www.nist.gov/sites/default/files/images/photogallery/patchy_particles.jpg
[6] Illustration of the Kern-Frenkel energy calculation https://hoomd-blue.readthedocs.io/en/v3.11.0/tutorial/07-Modelling-Patchy-Particles/01-Kern-Frenkel-Model.html
[7] Code for the particle simulation program https://github.com/tootonee/particle_sim