ME759 Final Project Report
University of Wisconsin-Madison

GPU/OpenMP Accelerated Monte Carlo Simulation of Nucleation

Xinyi;Li

12/17

Abstract

In this project, I try to use OpenMP and GPU to accelerate the simulation of nucleation event in a molecular level. The simulation is hybrid Monte Carlo(MC)/Molecular Dynamics(MD) simulation. The MD part is accelerated by GPU through MD package OpenMM. And MC is speeded up by OpenMP.

Contents

1. Introduction

Nucleation is a key step to synthesize crystals and other solid materials. In order to successfully make targeted materials. We need to understand nucleation in a molecular level and use the information to effectively control the crystal growth.

Because nucleation happens in an atomic scale, it is extremely hard to track the nucleation in experiment. So instead of directly observe the nucleation event, I will use molecular simulation to simulate the event. Fig.1 describes a typical nucleation event.

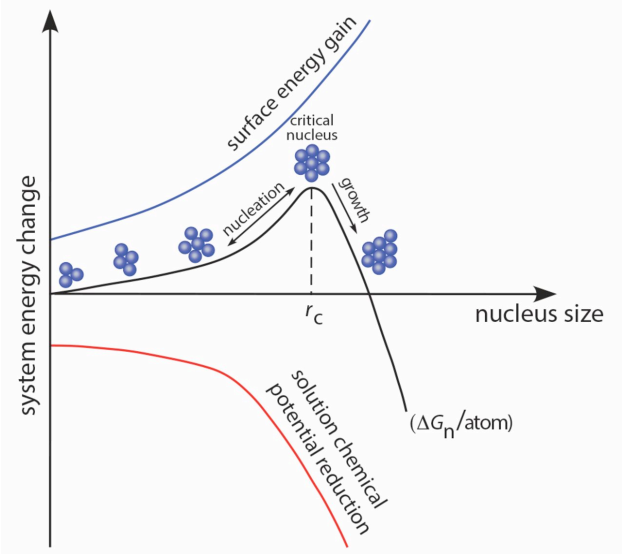

Fig.1 Typical Nucleation Process

Souce: http://elements.geoscienceworld.org/content/9/3/189/F3.large.jpg

2. Method

Because nucleation is a rare event, it would take years to run a brute force MD simulation until we can see the nucleation happens. So here I will use hybrid MC/MD to enhance the sampling. The basic working flow shows in Fig.2

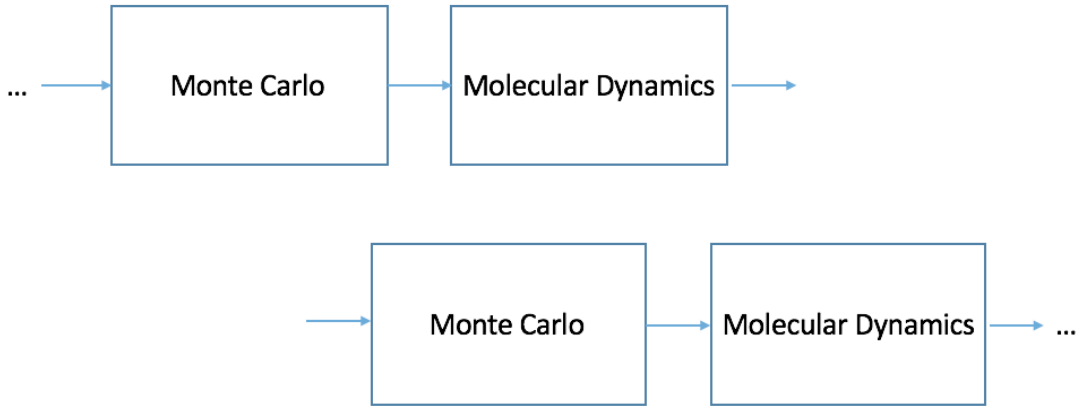

Fig.2 Working flow of hybrid MC/MD

2.1 Molecular Dynamic

MD Step is used to sample the space configurations. In order to sample the space configurations of all atoms, it's better to choose a method which can move all atoms collectively instead of a method can only move one atom. So here I use MD instead of MC. And to run fast MD, I use GPU-accelerated package OpenMM.

2.1 Monte Carlo

Besides sampling the space configuration, I also need to sample cluster size. MD is not capable of simulating system with flexible atom numbers, here I choose MC, more specifically, grand canonical Monte Carlo to run this type of move. And such MC moves are designed specifically for cluster growth. Each MC move consists of four parts, which are showed in Fig.3.

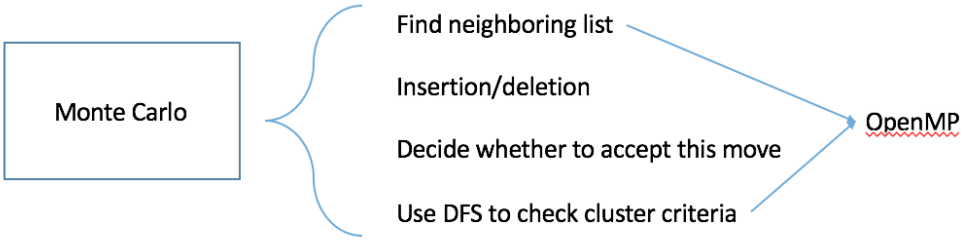

Fig.3 Monte Carlo Steps

There are four steps in Monte Carlo. In first step, I need to calculate atom-atom distances around one specific atom, and in the last step, I need to loop all the atom pairs to generate the adjacency matrix to be used in depth-first search(DFS). For both steps, we can use OpenMP to accelerate the simulation. Fig.4 shows the procedure with acceleration.

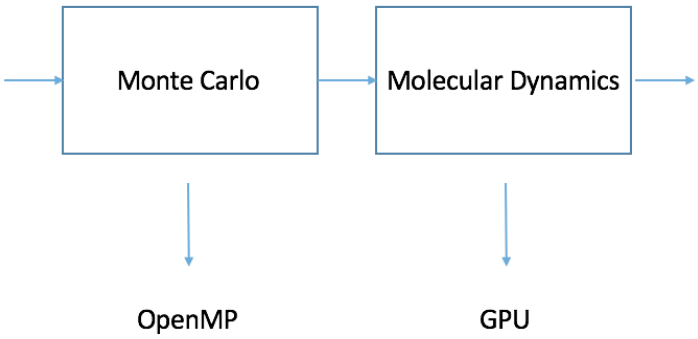

Fig.4 Procedure with acceleration

3. Implementation

All implementations are based on GPU-accelerated MD package OpenMM. In package OpenMM, MD steps are written into CUDA device kernel. And the interface is mainly written in C++ and python. Because the package provides a good python interface for programming MC, before the project, I wrote a hybrid MC/MD python code to run the simulation. In the code, I call OpenMM to run MD and running MC in python.

Later, I found that when cluster gets larger, calculating neighboring list and DFS becomes much more cumbersome, so I decided to rewrite the code in C++ with parallelization to improve the efficiency.

4. Results

I implemented C++ and OpenMP code for my project. But it turns out there is a function in OpenMM C++ library doesn't behave properly. More specifically, the function to set atom positions does not always update atom positions with parameters I input. So instead of testing the whole program, I just picked the two parts of the program to test the performance of the code. One part is calculating DFS and checking cluster criteria, and the other part is calculating the neighboring list. Fig.5 and 6 are results from testing.

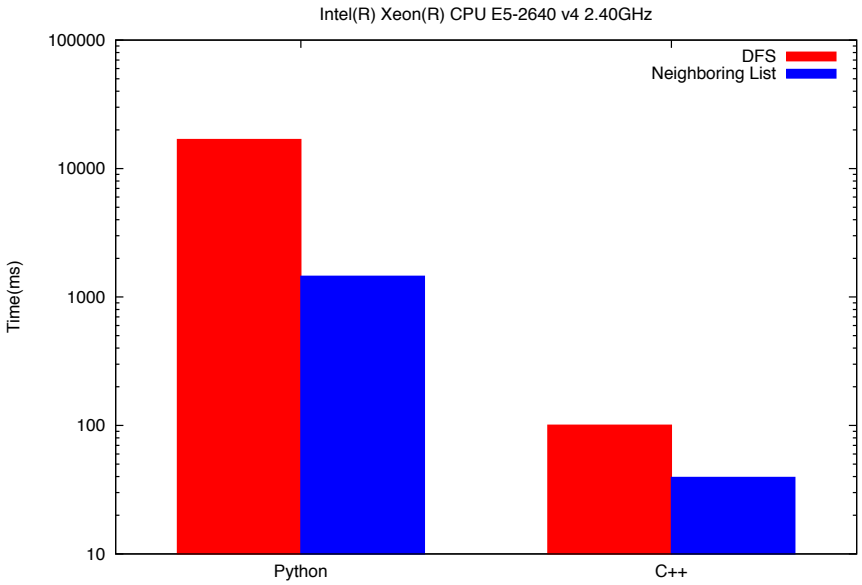

Fig.5 Performance comparison C++/Python

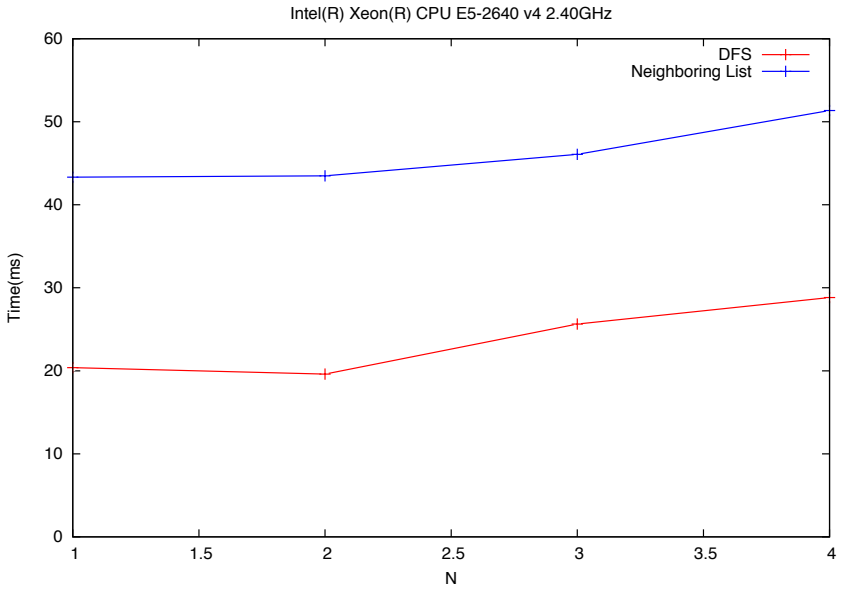

Fig.6 OpenMP performance w.r.t number of threads

From the plot, we can see that C++ is more than 100 times faster than python. That's not surprising. In python, looping list is very expensive because the memory of list is not continuous. But in C++, vector has continuous addresses, and looping all atom positions is much cheaper.

However, OpenMP does not help to increase the efficiency. One reason is that the testing system is too small (30 atoms). If we can expand the system to hundreds of atoms, maybe we can see the efficiency gain from using OpenMP. And also, in DFS searching, I need to save the adjacency matrix during looping all atom pairs. Saving the adjacency matrix requires to use some critical/atomic clauses and causes a lot of overheading. That's another reason why we did not see the improvement of the performance from parallelization.

5. Conclusion and Future Work

I rewrote the hybrid MC/MD code from python to C++ and parallelize the code by OpenMP. Because the problem of the function from OpenMM library, the code could not run properly. So I pick the parts of DFS and calculating neighboring list to test the code efficiency. C++ is more than 100 times faster than python. But parallelizaton does not help to improve the performance because of the small testing size and overheading.

In the future, I will email OpenMM developer to solve the library bug in OpenMP and test the simulation in a larger system size.

About running the code in Euler:

Because the OpenMM library bug, I couldn't run the simulation at this time. But for testing purpose, I just picked two functions and such two functions can be executed without any issues.

I compiled the testing code and executed it in my own machine without any issue. But running such code requires installing OpenMM. I did install the OpenMM in Euler. But because of some unknown reason, the compiler could not find "-lOpenMM". It must be something related to path setting. I will try to figure it out later.