博士論文

# Molecular Dynamics Study of Hamiltonian Mean Field Model Utilizing Processor with Many Cores

## （メニーコアプロセッサを用いた Hamiltonian Mean Field Model の分子動力学的研究）

青木尚登

# Contents

# Chapter 1

# Introduction

In recent years, accelerator devices such as GPU and Intel Xeon Phi are receiving attentions. Actually some supercomputers have those accelerator on them. These accelerators have relatively large number of cores on them. This thesis aims to make good use of such many core type accelerator devices to computational physics study. Up to now, the mainline of computational science is utilizing CPU, whose number of cores is up to several tens, and whose core clock is as high as several GHz. About accelerators, the core frequency is relatively low [1] and literally the number of cores is so large[2]. That makes difficult using legacy code. However, from the viewpoint of initial financial cost, power consumption and physical space, those accelerators have advantages, and if certain object for study to which accelerators are advantageous, it is better to utilize accelerators. Therefore, in this study, the performance for molecular dynamics (MD) simulation is researched. Then for mean field type systems, many core type accelerators are found to have more performance than CPU, and the author studied Hamiltonian mean field (HMF) model utilizing GPGPU.

HMF model is a toy model for studying systems with long range interaction such as self-gravitating systems and plasmas. HMF model as well as those long range interacting systems have non gaussian stationary states or quasi stationary states (QSSs) [1], and usually these are the main interesting target. Until now, numbers of stability analyses [2–4] and predictions of QSSs [5–7] are reported. However certain discrepancies exist between the predictions and numerical simulations. The main goal of this study is not precise prediction of QSS, but like thermodynamics, in which states are specified by limited number of quantities such as temperature, finding set of quantities which specifies QSS uniquely.

[1]The core frequency of NVIDIA Tesla K20 is 0.71 GHz.

[2]Tesla K20 has 2496 cores.

# Chapter 2

# Utilizing Processor with Many Cores for Computational Physics

## 2.1 Introduction

For computational physics, high-performance computer is a necessary tool. computer processor is improving obeying Moore's law [8]. Meanwhile, the understandings of physics improves as shown on table 2.1. The improvement of computers is closely related to our future.

In the history of computers, device miniaturization has been improving performance. When the improvement of performance per one core reached limit, CPU with multiple cores became mainstream.

Since the physical die size is limited, the number of cores was limited. Until several years ago, most of the high-performance computer used CPU with at most several tens of cores.

Meanwhile, GPU has large number of cores. The hierarchical architecture optimized inter core connection, and physical size is reduced. on 2007, NVIDIA released Compute Unified Device Architecture (CUDA), which enabled users to write programs for general-purpose computing on graphics processing units (GPGPU). Following such situation, Intel released its first many integrated core (MIC) device, Xeon Phi 5110P on 2012. The number of cores on the device is 60, which is not so large as GPU, but larger than normal CPU.

As seen from the above, two tendency exist: multiple high-performance cores and many numbers of non high performance cores. As shown in figure 2.1, many core processor is advantageous. In addition, the prices per 1

TFLOPS of many core accelerator is cheaper. For example, that of NVIDIA Tesla K80 is ¥200,000 [1], while that of Intel Xeon E5-2699 CPU is ¥630,000 [2] . These are part of the reasons that in recent years some super computers adopt many core accelerators. This tendency may accelerate and the computers equipped with many core accelerators will be common in the near future. Utilizing such accelerators for computational science is desirable.

However, there are problems that prevents users to use accelerators. Rewriting code for accelerators need costs such as learning and development time. In addition, even when a code for accelerator is prepared, it is not necessarily efficient.

In this study, the way to utilize many core accelerators for computational physics is studied. Through examining performances for short range system and mean field system, what research way utilizing many core accelerators is desirable.

| |
|---|
| $10^3$ ( $\sim$10nm) local thermal equilibrium<br>MFLOPS efforts, computer in 1970 physics in 1980s |
| $100^3$ ( $\sim$100nm) linear nonequilibrium MATERIAL<br>GFLOPS efforts, computer in 1985 physics in 1995$\sim$2005 |
| $1,000^3$ ( $\sim 1\mu$m) nonlinear nonequilibrium DEVICE<br>TFLOPS efforts, computer in 1995 physics in 2005 $\sim$ |
| $10,000^3$ ($\sim 10\mu$m) or more macroscopic complex system SYSTEM<br>PFLOPS efforts, computer in 2008 physics in 2010s |
| Furthermore, 10PFLOPS(Avogadro-scale computer) in 2011,<br>and EFLOPS($S^2$ computer) will come in 2020.<br>COMMUNITIES of Systems: Society in Silico($S^2$) |

Table 2.1: Scales from molecules to our world [9]. Size of simple molecules is taken as a unit of length $\sim$nm.

[1] taken from `http://www.amazon.com/dp/B00Q7O7PQA` as of 2015/2/3. The rate 117 ¥/$ is assumed and price in yen is rounded to the nearest ten thousand.

[2] taken from `http://www.amazon.com/dp/B00PDD1ZX0` as of 2015/2/3. The rate 117 ¥/$ is assumed and price in yen is rounded to the nearest ten thousand.

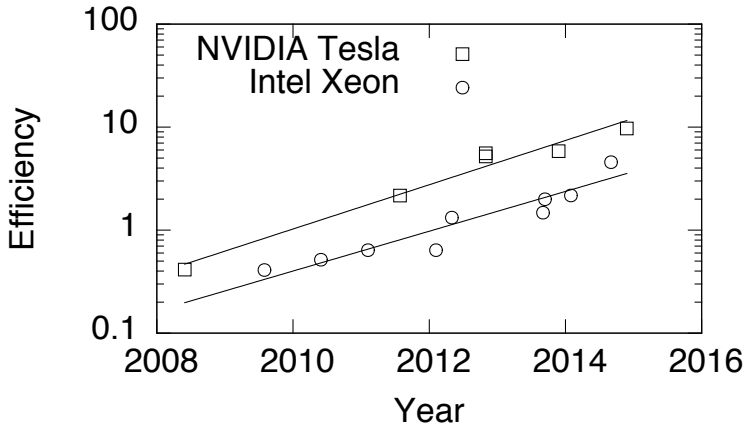


Figure 2.1: Scaling plot of the computation efficiency of typical GPU and CPU series. The computation efficiency is the computation performance divided by the thermal design power (TDP) [GFLOPS/W]. The lines are the fit by the Koomey's law $A \cdot 2^{\frac{t}{\tau}}$ [10], where $\tau$ denotes the time interval after which the computation efficiency doubles. The obtained fitting parameters $\tau$ for NVIDIA Tesla and for Intel Xeon are 1.4013 [year] and 1.5579 [year] respectively. For FLOPS and TDP, see table 2.2.

| Product | GFLOPS | TDP [Watt] | Reference |
|---|---|---|---|
| Tesla C1060 | 77.76 | 187.8 | [11] |
| Tesla C2050 | 515 | 238 | [12] |
| Tesla K20 | 1170 | 225 | [13] |
| Tesla K20X | 1310 | 235 | [14] |
| Tesla K40 | 1430 | 245 | [15] |
| Tesla K80 | 2910 | 300 | [16] |
| Xeon W3580 | 53.28 | 130 | [17] |
| Xeon X3480 | 48.96 | 95 | [18] |
| Xeon X5690 | 83.04 | 130 | [19] |
| Xeon E5-2687W | 198.4 | 150 | [20] |
| Xeon E5-1680 v2 | 192.0 | 130 | [21] |
| Xeon E5-2697 v2 | 259.2 | 130 | [22] |
| Xeon E7-2890 v2 | 336 | 155 | [23] |
| Xeon E5-2699 v3 | 662.4 | 145 | [24] |

Table 2.2: The specifications of NVIDIA Tesla series GPUs and Intel Xeon series CPUs.

### 2.1.1 The History of High Performance Computers

The computers have been contributing to the developments of human's knowledge. For example, the Earth Simulator contributed to the 2007 Nobel Prize of Intergovernmental Panel on Climate Change. The performance of computers is often measured by Floating-point Operations per Second (FLOPS). For example, TOP500 uses FLOPS for Linpack. The peak FLOPS is derived by the number of cores $\times$ core clock frequency $\times$ instructions per cycle (IPC). From this view point, we follow the development of high performance computers up to now.

Until around 2000, the number of cores per one CPU processor was 1 for most cases. For example, ASCI White, which became the first of TOP500 on 2000/11 had processors whose number of cores is 1. Until that time, mainly the core frequency had been increased and accordingly FLOPS had been improved. However, the improvement of core frequency is limited. When the quantity of heat is too large, the processor does not work correctly. There are two ways to change the situation.

One way is to improve IPC. The Earth Simulator, which was the first of TOP500 on 2002/6 consists of vector processors. A vector processor executes one instruction on multiple data. This way is called single instruction, multiple data (SIMD).

Another way is to increase the number of cores. Multiple cores are implemented on one processor. The reduced distance between cores makes the time for communication short. Blue Gene/L, which was the first of TOP500 on 2004/11 is an example.

Adopting the two ways, FLOPS has been developing. The number of cores has increased. IPC has been improved. The miniaturization has improved. However, these improvements increases the density of heat. Decreasing core frequency is one of the way to relax this problem. In order to obtain as high performance as other computers at the same period, the number of cores is made relatively high. This type of processors is called many core processors. The typical examples are GPU and Intel Xeon Phi. The TOP500 champions which are equipped with such devices are Tianhe-1A on 2010/11, Titan on 2012/11, and Tianhe-2 on 2013/6. The computers with many core processors which reached the top 10 of TOP500 are Tianhe-2, Titan, Piz Daint, Stampede, and CS-Storm. This amounts to the half of the top 10 computers. The many core processors have possibility to become the mainline of high performance computers.

### 2.1.2 Many Core Accelerators

#### NVIDIA GPU

Unfortunately, existing program code for CPU is not available for GPUs. We have to explicitly write code for utilizing device such as data transfer, execution of kernel function. A kernel function is a procedure of instructions processed parallel on GPU.

The architecture of GPU is hierarchical: NVIDIA GPU has multiple streaming multiprocessors (SM) and VRAM memory. Inside one SM, multiple CUDA cores and shared memory exist.

The architecture of GPGPU is mapped to the CUDA (Compute Unified Device Architecture) programming model. Block is group of threads. One block mapped to one SM. For some GPUs, multiple blocks are mapped to the same SM. The threads inside block are executed parallel. Especially the threads within one warp are executed concurrently. This manner is called "Single instruction, multiple threads" (SIMT). The set of threads processed by one core is called warp. The number of threads inside one warp is 32. When branching occurs inside one warp, it reduces efficiency. It is called warp divergence.

Different two blocks are not necessarily executed concurrently. For example when all the streaming multiprocessors are in use, other blocks must wait for certain block to be completed.

Grid is group of all the blocks, which is mapped to one GPU device. Global memory is globally accessible memory, which is mapped to Video RAM. Shared memory exists inside each blocks, and is locally accessible from threads inside the same block. Shared memory is mapped to memory on streaming multiprocessor, and is faster accessible than global memory, but shared memory outside the block is not accessible.

In order to obtain good performance, the large number of block threads is desirable. This is related to the memory latency. On each block, all the threads in one block are not processed simultaneously, but the threads in several warps. When a memory access instruction is encountered, SM switch warp in order to conceal memory access time.

#### Intel Xeon Phi

On Intel Xeon Phi, Linux operating system is running. Programs written in normal C/C++ code is available. Therefore existing code may run. However, there are problems. The frequency of core is relatively low compared with CPU. The number of cores is huge. Intel Xeon Phi also has hierarchical

architecture We should be aware that these matters may cause unexpected bottleneck.

For parallelization, frameworks such as Open MP, MPI are available. In this study, Open MP is used.

## 2.2 Method

We carry on MD simulations and evaluate the performance.

Simply put, MD is N-body simulation. It simulate the movement of particles, which correspond to molecules in certain material. On the computer, equation of motion is integrated numerically. There are many scheme for integrating differential equation: symplectic integrator, predictor corrector, and so on. In this study symplectic integrator is used. The basic idea of symplectic integrator is as following. Usually strict integration of the original Hamiltonian is impossible. However there exists approximate Hamiltonian for which the strict numerical integration is available. The strict integration of the approximate Hamiltonian is derived utilizing Suzuki-Trotter decomposition [25] to the strict integration of the original Hamiltonian.

**Symplectic Integrator**

In this study, symplectic integrator is used. In most cases, the interesting Hamiltonian is expressed like following:

$$H\left(\{\mathbf{q}_i\},\{\mathbf{p}_i\}\right) = K\left(\{\mathbf{p}_i\}\right) + U\left(\{\mathbf{q}_i\}\right). \tag{2.1}$$

From now on, this expression is assumed. Consider time evolution of phase space. Let $\rho_t\left(\{\mathbf{q}_i\},\{\mathbf{p}_i\}\right)$ the phase space distribution. The time evolution equation of $\rho_t$ i.e. Liouville's equation is expressed as

$$\frac{\mathrm{d}\rho_t}{\mathrm{d}t} = -\mathrm{i}\hat{\mathrm{L}}_H\rho_t, \tag{2.2}$$

where

$$\mathrm{i}\hat{\mathrm{L}}_X = \{\,\cdot\,, X\} = \sum_i \left(\frac{\partial X}{\partial p_i}\frac{\partial}{\partial q_i} - \frac{\partial X}{\partial q_i}\frac{\partial}{\partial p_i}\right) \tag{2.3}$$

is Liouville operator. Since the equation is linear form, strict solution is formally expressed as

$$\rho_{t+\Delta t} = \exp\left(-\Delta t\,\mathrm{i}\hat{\mathrm{L}}_H\right)\rho_t. \tag{2.4}$$

The exponential operator is transformed as

$$\exp\left(-\Delta t\,\mathrm{i}\hat{\mathrm{L}}_K - \Delta t\,\mathrm{i}\hat{\mathrm{L}}_U\right). \tag{2.5}$$

Applying Suzuki-Trotter decomposition [25], the operator is approximated as

$$\exp\left(-\Delta t\,\mathrm{i}\hat{\mathrm{L}}_K\right)\exp\left(-\Delta t\,\mathrm{i}\hat{\mathrm{L}}_U\right). \tag{2.6}$$

Each exponential operator has a simple form:

$$\exp\left(-\Delta t\,\mathrm{i}\hat{\mathrm{L}}_K\right)\rho\left(\{\mathbf{q}_i\},\{\mathbf{p}_i\}\right) = \rho\left(\left\{\mathbf{q}_i + \Delta t\frac{\partial K}{\partial \mathbf{p}_i}\right\},\{\mathbf{p}_i\}\right), \tag{2.7}$$

$$\exp\left(-\Delta t\mathrm{i}\hat{\mathrm{L}}_U\right)\rho\left(\{\mathbf{q}_i\},\{\mathbf{p}_i\}\right) = \rho\left(\{\mathbf{q}_i\},\left\{\mathbf{p}_i - \Delta t\frac{\partial U}{\partial \mathbf{q}_i}\right\}\right). \tag{2.8}$$

Therefore the approximate update is done as following:

- update momentum $\mathbf{p}_i \leftarrow \mathbf{p}_i - \Delta t\dfrac{\partial U}{\partial \mathbf{q}_i}$ ,
- update and then update position $\mathbf{q}_i \leftarrow \mathbf{q}_i + \Delta t\dfrac{\partial K}{\partial \mathbf{p}_i}$.

From the alternating update of $\mathbf{q}$ and $\mathbf{p}$ , this is also called Leapfrog integration.

Since this integration is the strict integration of approximate Hamiltonian, iterating the update does not conserve the energy calculated from the original Hamiltonian expression but the fluctuation level is of $\Delta t$ order. Higher order integration [26, 27] also exists.

### Data Structure on Memory

There are two types of data structure: Structure of Arrays (SoA) and Array of Structure (AoS).

For the example of HMF model, AoS means that structure of one particle variable is prepared, and array of the variable structure is allocated on memory, SoA mean that for each variables $(q, p, s_x, s_y, \cdots)$, separate arrays are allocated on memory. Schematic pictures are shown on figure 2.2.

For the implementation of MD simulations, SoA is chosen. The reason is explained in following. When a memory reading is needed, the necessary memory region is split into set of fixed-size region and processed for each region. This size of region is called bus width. Consider cases when we read set of data whose size is smaller than bus width. When the set of data is

continuous on memory, the peak memory reading speed is obtained. For cache enables processors including CPUs, the read data is stored in cache memory in advance. For GPUs, the set of memory access from parallel threads is coalesced and the bus width is fully used. Meanwhile, when the set of data is not continuous, the number of times for memory transfer is larger and the memory read time is bigger. For both short range system and mean field system, the continuous access of parameters on each particles is needed at many point. Therefore SoA is chosen.

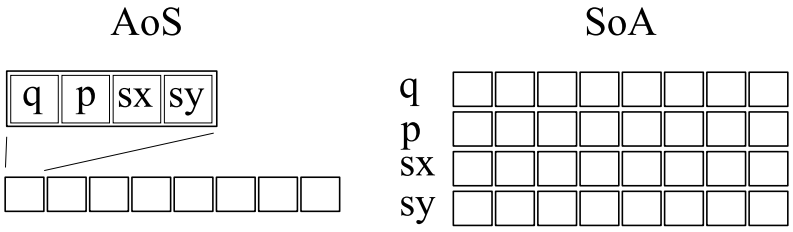


Figure 2.2: Schematic picture of AoS and SoA.

### System with Short Range Interaction

Usually actual molecules interact by short range interactions. For certain purpose, Lennard-Jones (LJ) potential is often used. First, implementation for single thread is explained, and then that for CPU parallelization and that for GPGPU are explained.

**LJ system** We introduce a system defined by the Hamiltonian

$$H = \sum_{i=1}^{N} \frac{\mathbf{p}_i^2}{2} + \sum_{i=1}^{N} \sum_{j=i+1}^{N} V\left(|\mathbf{r}_i - \mathbf{r}_j|\right) , \tag{2.9}$$

where $V$ is a truncated LJ interaction potential

$$V(r) = \begin{cases} 4\left[\dfrac{1}{r^{12}} - \dfrac{1}{r^6} + c_2 r^2 + c_0\right] & (r < r_c) \\ 0 & (\text{otherwise}) . \end{cases} \tag{2.10}$$

$r_c$ denotes interaction range. $c_0$ and $c_2$ are taken so that the following equations hold: $V(r_c) = 0$, $\frac{\partial V}{\partial r}(r_c) = 0$.

**Basic Implementation** This implementation is based on an efficient implementation [28]. In order to calculate force and potential between particles, the pairs of particles which are interacting should be listed. To speed up listing the interacting pair, the simulation box is split into pieces of cuboid, which is called cell. One has only to search neighbor cuboid cells for interacting particles.

Let the interaction radius $r_{\mathrm{c}}$. The list of pair is made reusable for a short time, by listing surplus pairs in addition: The two particles are not interacting each other, but the distance is equal to or shorter than $r_{\mathrm{r}}$. Here $r_{\mathrm{r}}(> r_{\mathrm{c}})$ denotes registering distance, and $r_{\mathrm{s}} = r_{\mathrm{r}} - r_{\mathrm{c}}$ denotes surplus distance.

Accordingly the side length of cuboid cell is set equal to or longer than $r_{\mathrm{r}}$. As explained above, one has only to search neighbor cells for the particle pairs. Here neighbor cells means the first and second nearest neighbor cells.

Once the pair list is built, one can update position and momentum by certain integration scheme. On the update of the particle positions, the largest displacement distance of each particle is measured. One sums up the largest distance each time. When the summation exceed $\frac{1}{2}r_{\mathrm{s}}$, some interacting two particles pair may appear which was not included in the list built last time, therefore the pair list is no longer reusable. Then, One has to update pair list in the manner mentioned above.

For further optimization, one sorts the data array of particles, just before building pair list. Since this makes physically neighbor particles neighbor on memory position, this decreases the probability of cache miss hit. Therefore this improves memory load speed. Since the key of sorting is integer cell number, bucket sort algorithm is used.

**CPU Parallelization** The implementation for parallel execution on CPU is based on that for single thread. There are two typical parallelization for short range systems: particle decomposition and domain composition. Particle decomposition means that different thread processes different particle. Domain decomposition means that different thread processes different domain. The efficient implementation [28] adopts domain decomposition.

**Implementation for GPU** Since the architecture of GPU is simple, the same implementation as CPU is not realistic. For example, dynamic memory allocation on kernel is not available, the memory size is small for FLOPS, and if-branching makes parallel efficiency worse. Therefore a simple implementation is introduced.

On this implementation, pair list is not built in advance. Instead, one searches neighbor cells for pair particles on each update.

For GPU Implementation, particle decomposition is used instead of domain decomposition. The reasons are explained in following.

First, coalesced memory access is expected: On particle decomposition, the chances for coalesced memory access is expected to be high. Assume that all the particles are sorted on memory according to the position of cuboid cell. The particles processed by threads in one warp are likely to be in the same cuboid cell. Therefore the data is likely to be continuous, and memory access is coalesced.

Second, warp divergence is expected to be suppressed. On domain decomposition, the numbers of particles on each domains are not uniform. Therefore, threads in one warp have different loop sizes and they must wait for the thread which has the biggest workload. As stated above, On particle decomposition, the particles in one warp is likely to be in the same cuboid cell. Therefore, the necessary particle list to check for interaction is same. Therefore warp divergence is relatively suppressed.

As described above, particle sorting is also done. For the parallelization of bucket sort, prefix sum algorithm [29] is used.

For the implementation code, see 5.1.

### System with Mean Field Interaction

As a typical mean field system HMF model is taken. For Hamiltonian and the equation of motion, see section 3.1.1.

**Implementation** The basic procedure of HMF simulation is as follows:

1. Update momentum $p_i \leftarrow p_i + (-M_x \sin q_i + M_y \cos q_i)\,\Delta t$
2. Update position $q_i \leftarrow q_i + p_i \Delta t$
3. Update mean field magnetization $\mathbf{M}$

There is certain freedom for implementation.

1. read $\mathbf{s}_i = (\cos q_i, \sin q_i)$ from memory or calculate it
2. parallel summation algorithm of $\mathbf{M}$

The first one is up to device. If memory reading speed is slower than the calculation of sine and cosine, then you may calculate $\mathbf{s}_i$ from $q_i$. Even if memory reading speed is faster, calculation of sine and cosine may conceal memory access time for other variables.

The second issue is simple for CPU. On single thread execution, one may use for-loop to calculate summation. One may use loop unrolling, and

software pipelining for optimization. For OpenMP and MPI, the reduction function is provided.

However on some machines, the reduction function for OpenMP is slower than the following reduction algorithm: calculate summation on each thread locally, write local summation value on globally shared array, calculate the summation of the array on a certain thread.

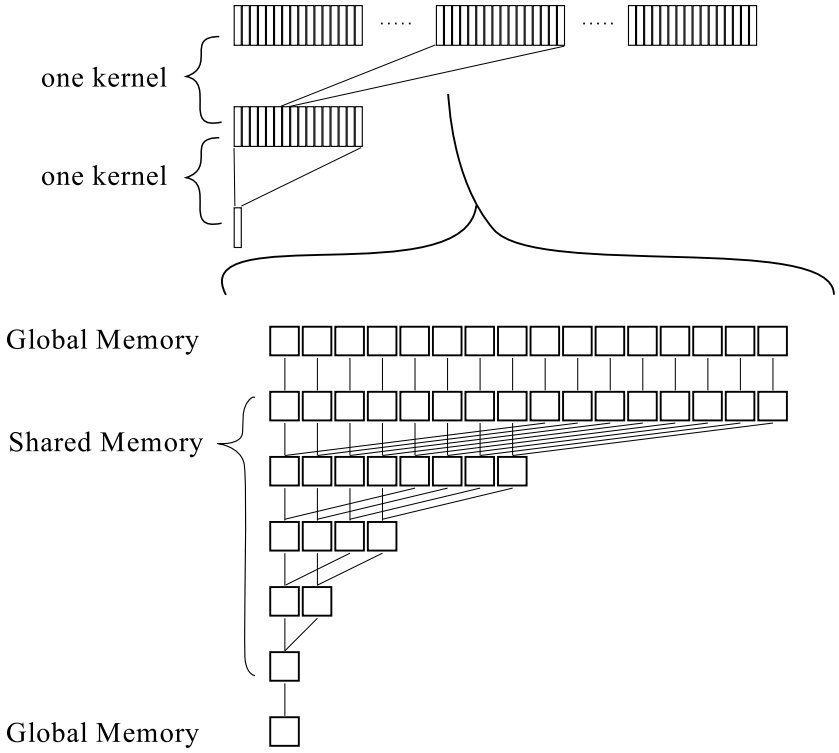


Figure 2.3: Schematic picture of parallel reduction algorithm [30]. The upper half shows the overall flow. The lower half shows the instructions of one block on one kernel function. Each threads in one block read each value on the global memory, and write it on the shared memory. Each threads of the former half of one block read a value on the former half of the shared memory and a value on the latter half of the shared memory. The sum of the two values is written on the shared memory. These instructions make the length of array half. Repeating this instructions, the sum in one block is obtained. When the number of blocks is not 1, the total sum is not obtained at once. Repeating the above kernel function, the total sum is obtained.

For GPU, a highly optimized reduction algorithm [30] is introduced In this study, the algorithm is applied.

**Floating-Point Operation Count and Memory Load** In order to evaluate the efficiency of calculation performance and memory load. For CUDA available GPUs, CUDA Profiling Tools Interface (CUPTI) [31] allows performance counting.

CUPTI provides a sample program callback_metric. The author modified the source code to perform calculation of simultaneous sine cosine function. In order to count double precision floating operation, metric name flop_count_dp is used. Then one gets flop_count_dp = 1900000. Since the total number of threads is 50000, flop of sincos is 38.

The overall count of floating operation is 86. The overall size of memory load is 4 floats i.e. 32 bytes for double precision floating-point operation.

### 2.2.1 Computation Configurations

In this study, various kinds of computer machines are used. The computation configurations are listed on table 2.3.

| Name | Table |
|---|---|
| Tesla K20 | 2.4 |
| Xeon E5-1620 v2 | 2.7 |
| Tesla C2050 | 2.5 |
| Xeon X5570 | 2.8 |
| Xeon Phi 5110 | 2.6 |
| Xeon E5-2680 v2 | 2.9 |
| SR16K OMP | 2.10 |
| SR16K MPI | 2.10 |

Table 2.3: Computation configuration names and configuration tables.

## 2.3 Results and Discussion

In this chapter, The performances for LJ system and HMF model are evaluated. MUPS denotes millions update per second.

### 2.3.1 Performance for LJ System

As shown in table 2.11, the performances for GPUs are less efficient than that for Xeon. From this results, utilizing GPUs for LJ simulation is not realistic at present.

| Target Product | NVIDIA Tesla K20 |
|---|---|
| FLOPS (single) | 3520 GFLOPS |
| FLOPS (double) | 1170 GFLOPS |
| Memory Bandwidth | 208 GB/s |
| Amount of Memory | 4800 MB |
| GPU Clock rate | 506 MHz |
| Number of CUDA Cores | 2496 Cores |
| TDP | 225 W |
| OS | CentOS 6.6 |
| CUDA Driver Version | 6.5 |
| CUDA Runtime Version | 6.5 |
| Compiler | nvcc 6.5.12 |
| Host Configuration | Xeon E5-1620 v2 |

Table 2.4: The configuration of Tesla K20.

| Target Product | NVIDIA Tesla C2050 |
|---|---|
| FLOPS (single) | 1030 GFLOPS |
| FLOPS (double) | 515 GFLOPS |
| Memory Bandwidth | 144 GB/s |
| Amount of Memory | 2687 MB |
| GPU Clock rate | 1147 MHz |
| Number of CUDA Cores | 448 Cores |
| TDP | 238 W |
| OS | CentOS 6.6 |
| CUDA Driver Version | 6.5 |
| CUDA Runtime Version | 6.5 |
| Compiler | nvcc 6.5.12 |
| Host Configuration | Xeon X5570 |

Table 2.5: The configuration of Tesla C2050.

| Target Product | Intel Xeon Phi 5110 |
|---|---|
| FLOPS (single) | 2006 GFLOPS |
| FLOPS (double) | 1003 GFLOPS |
| Memory Bandwidth | 240 GB/s |
| Amount of Memory | 6144 MB |
| Core Clock rate | 1100 MHz |
| Number of Cores | 57 Cores |
| TDP | 300W |
| Compiler | icc 14.0.1 |
| Host Configuration | Xeon E5-2680 |

Table 2.6: The configuration of Xeon Phi 5110

| Target Product | Intel Xeon E5-1620 v2 |
|---|---|
| FLOPS (double) | 118.4 GFLOPS |
| Core Clock rate | 3700 MHz |
| Number of Cores | 4 Cores |
| TDP | 130 W |
| OS | CentOS 6.6 |
| Compiler | gcc 4.4.7 |

Table 2.7: The configuration of Xeon E5-1620 v2

| Target Product | Intel Xeon X5570 |
|---|---|
| FLOPS (double) | 46.88 GFLOPS |
| Core Clock rate | 2930 MHz |
| Number of Cores | 4 Cores |
| TDP | 95 W |
| OS | CentOS 6.6 |
| Compiler | gcc 4.4.7 |

Table 2.8: The configuration of Xeon X5570

| Target Product | Intel Xeon E5-2680 v2 |
|---|---|
| FLOPS (double) | 118.4 GFLOPS |
| Core Clock rate | 2800 MHz |
| Number of Cores | 10 Cores |
| TDP | 130 W |
| Number of Cores for computation | 8 Cores |
| OS | CentOS 6.4 |
| Compiler | gcc 4.4.7 |

Table 2.9: The configuration of Xeon E5-2680 v2

| Target Product | Hitachi SR16000 L1 |
|---|---|
| FLOPS (double) | 448 GFLOPS |
| Memory Bandwidth | 224 GB/s |
| Amount of Memory | 64 GB |
| Core Clock rate | 3500 MHz |
| Number of Cores | 32 Cores |
| TDP | 4000 W |

Table 2.10: The configuration of SR16K. For SR16K OMP and for SR16K MPI, Open MP and MPI are used respectively.

| Device | MUPS | /TFLOPS | /KW |
|---|---|---|---|
| Tesla K20 | 22.9 | 19.6 | 102 |
| Tesla C2050 | 18.3 | 35.5 | 76.9 |
| Xeon E5-1620 v2 | 15.0 | 126.7 | 140 |
| Xeon E5-2680 v2 | 22.9 | 241.8 | 220 |

Table 2.11: The performances for LJ system. MUPS, MUPS divided by peak TFLOPS [MUPS/TFLOPS] and MUPS divided by power consumption [MUPS/KW] are shown. The number of particles is 65,536. The cut off radius $r_c$ is 2.5. The simulation box size is $L_x = L_y = L_z = 96$. The time slice $\Delta t$ is 0.0001. The number density is 0.7. For Xeon CPUs, an efficient implementation [32] is used.

### 2.3.2 Performance for HMF Model

As shown on figure 2.4, 2.5 and 2.6, performances for HMF on various devices are examined. The number of processing cores used is fixed to be maximum. Meanwhile the number of particles changes. The fitting function $\frac{N}{a+bN}$ is derived according to an idea similar to Amdahl's argument [33]: Let $A$ the processing time of particle-wise procedures. Let $M$ the number of processing cores. Let $a = \frac{A}{M}$. Let $b$ the processing time of system-wise procedures. Typical examples are thread synchronization. Therefore the processing time overall for $N$ particles is $\frac{AN}{M} + b = aN + b$, and the following estimation is derived:

$$\text{MUPS} = \frac{N}{aN + b}\,. \tag{2.11}$$

For convenience, call this idea Amdahl-like explanation.

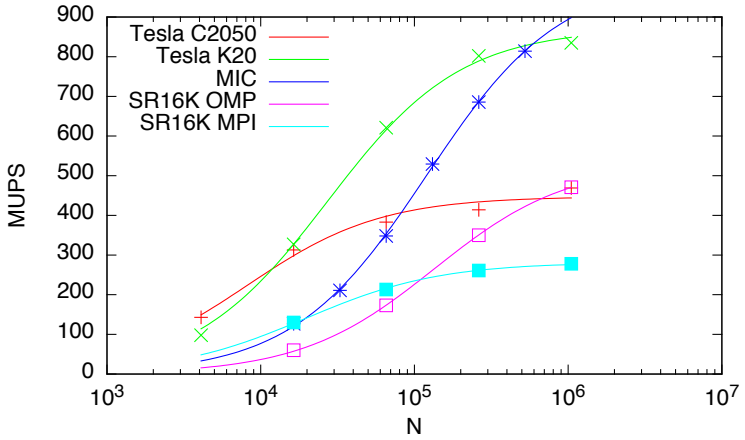


Figure 2.4: MUPS for HMF model as a function of number of particles. 1st order symplectic integrator of HMF model is applied. The time slice $\Delta t$ is 0.1. The points on the figure are obtained from simulation. The curves are fitting function $\frac{N}{aN+b}$ for each devices.

| Device | $a$ [ns] | $b$ [$\mu$s] |
|---|---|---|
| Tesla C2050 | 2.23483 | 18.289 |
| Tesla K20 | 1.14834 | 31.226 |
| Xeon Phi 5110 | 0.99899 | 119.792 |
| SR16000 OpenMP | 1.88267 | 254.077 |
| SR16000 MPI | 3.55610 | 70.047 |

Table 2.12: Fitting parameters $a$ and $b$ for each devices.

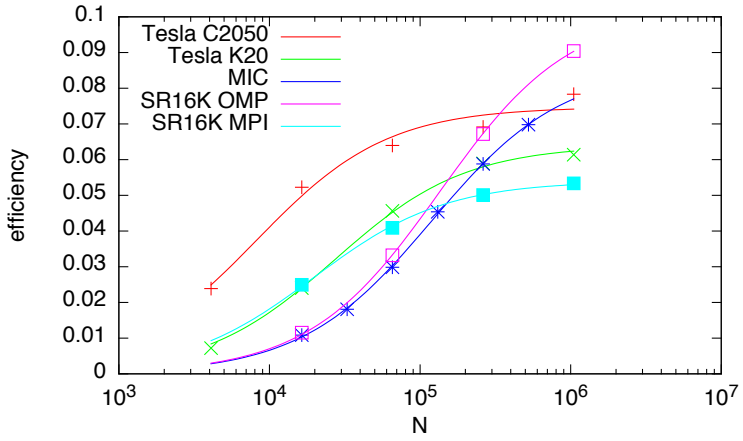


Figure 2.5: The efficiency of FLOPS as a function of the number of particles. The efficiency of FLOPS is the actual FLOPS divided by peak FLOPS. The actual FLOPS is measured using CUPTI as explained on the section 2.2.

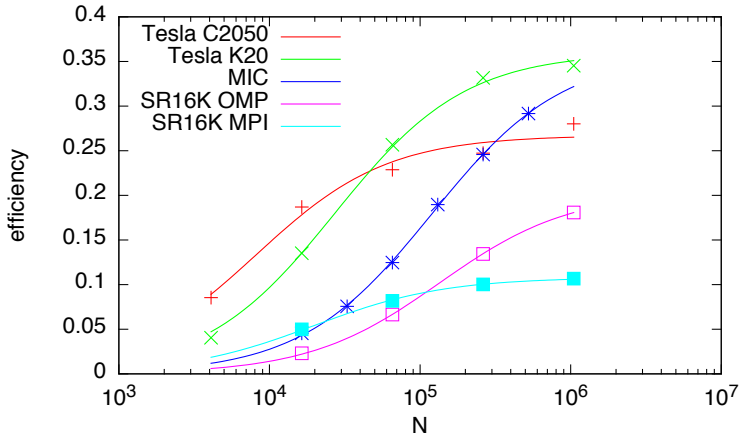


Figure 2.6: The efficiency of memory bandwidth as a function of the number of particles. The efficient of memory bandwidth is the actual memory load speed divided by peak bandwidth.

Amdahl-like estimation seems to work well for all the devices. However, small discrepancy exist for GPU devices. According to Amdahl-like explanation, it is assumed that certain part exists whose time $b$ is independent of the number of particles $N$. For GPU, summation algorithm is implemented by binary tree [30], and $b$ changes as the number of reduction steps.

About FLOPS efficiency, on the limit of infinite number of particles $N \to \infty$, SR 16000 MPI has the best performance. However, the performance for limited number of particles is also important.

For $N = 65536$, Tesla C2050 has the best performance.

About memory bandwidth efficiency both for $N \to \infty$ and $N = 65536$, many core accelerators have the best performance.

As stated above, the performance for the limited number of particles is also important from the viewpoint of practicality. It is the $N$ independent term $b$ that influences such performance. Hardwares and softwares that make $b$ small is preferable. As can be seen in table 2.12, especially GPUs have small $b$. Possible factors are as follows. First, many core accelerator has hierarchical architecture. For GPU, the instructions are processed on each streaming multiprocessor independently, and synchronization is designed available only within a streaming multiprocessor. This enables fine synchronization, and reduces unnecessary overall synchronization time. Intel MIC also has hierarchical architecture. However, the implementation for normal CPU is used this time, and the advantage does not make effect.

Secondly, for many core accelerator, core clock is relatively low. Generally speaking, memory bandwidth is slower than calculation performance. For example on SR16000, memory reading speed of double precision float is 28 G/s = (224 GB/s / 8 byte) , while the calculation performance is 448 G/s. Actually, the position $\mathbf{s}_i = (\cos\theta_i, \sin\theta_i)$ is not read from memory, but is calculated from $\theta_i$. Making the core frequency low, data hazard is relaxed and instructions on pipeline are processed efficiently.

These are considered to be the $b$ reducing factors.

## 2.4 Summary

In recent years, accelerator devices such as GPU and Intel Xeon Phi are receiving attentions. From the viewpoint of initial financial cost, power consumption, and physical space, such accelerators are advantageous, but actually it is not necessarily true. This time the performance for molecular dynamics simulation is examined. For LJ system, accelerators are not advantageous. However, for HMF model they are useful. The performance for HMF model is estimated by an idea similar to Amdahl's law [33]: particle-

wise time $a$ and system-wise time $b$. For practically use, it is important to reduce $b$ as well as $a$. $b$ is related to synchronization time, and it seems the hierarchical architecture of many core accelerator makes $b$ small. In addition, for many core accelerators, the low core frequency reduced data hazard and makes instructions on pipeline processed efficiently. From now on, searching for topics for which many core accelerators are advantageous, and utilizing them efficiently is an assignment.

# Chapter 3

# Molecular Dynamics Study of Hamiltonian Mean Field Model

## 3.1 Introduction

In this section, a brief introduction to HMF model is given and then a verbose explanation is given.

Most of the interactions in nature are finite range type. On systems with such short range interaction, since the stationary state is thermal equilibrium, one can analyze the system using Maxwell-Boltzmann distribution.

Meanwhile long range interaction also exists. Typical examples are self-gravitating systems [34], plasmas [35], and fluids [36]. Since the partition function diverges, the thermodynamic limit of such systems is not well defined [37]. Therefore long range systems are different from short range systems. Typical phenomena are stationary states [35, 38, 39], and negative specific heat [40]. In order to understand these phenomena, toy models such as HMF model [41], and sheet model [42] are studied.

HMF model is a system of N-rotators with ferromagnetic XY interaction. HMF model was originally introduced as a symplectic coupled map systems [43], which is studied under the context of collective chaos. Then it was transformed to a time-continuous Hamiltonian system. Its similarity to the physical long-range systems such as self-gravitating systems [44], and the collective behaviour in it [41] have attracted much interest.

On this model, long lasting states exist. They are called quasi stationary states (QSSs). The duration of QSSs increases as the number of particle $N$ gets larger. When thermodynamic limit $N \to \infty$ is taken, the duration diverges [1].

Until now, numbers of researches explaining QSSs are reported. Station-

ary states distribution is fit by Tsallis distribution [1], which is a generalization of Maxwell-Boltzmann distribution. Lynden-Bell equilibrium [45] , which is derived by assuming violent relaxation is applied to HMF model [5], and QSS prediction is reported [6]. Fermi distribution is derived by the idea of Lynden-Bell equilibrium. However, certain QSS distributions are not well explained by it. In such situation, prediction using Core Halo distribution is also reported [7].

Meanwhile, a number of stability analyses for HMF are reported [2–4]. In addition to utilizing spectral stability, formal stability is used those studies. The spectral stability analysis of Vlasov equation on Fourier space. The formal stability is optimization of Casimir invariants.

In this study, various QSSs arising from various initial conditions are researched. Then, QSSs those were not reported in former studies are found. The author tried to characterize QSSs by some number of quantities like the former studies, and it is found that set of energy, magnetization and temperature-like quantity is useful.

### 3.1.1 Hamiltonian and Equation of Motion

The Hamiltonian of HMF model is expressed as

$$H\left(\{q_i\},\{p_i\}\right)=\sum_{i=1}^{N}\frac{{p_i}^2}{2}+\frac{1}{2N}\sum_{i=1}^{N}\sum_{j=1}^{N}\left[1-\cos\left(q_i-q_j\right)\right]. \tag{3.1}$$

This is transformed as

$$H\left(\{q_i\},\{p_i\}\right)=\sum_{i=1}^{N}\frac{{p_i}^2}{2}+\frac{1-\mathbf{M}\left(\{q_i\}\right)^2}{2}N, \tag{3.2}$$

where

$$\mathbf{M}\left(\{q_i\}\right)=\frac{1}{N}\sum_{i=1}^{N}\left(\cos q_i,\sin q_i\right) \tag{3.3}$$

denotes magnetization. $M=\sqrt{{M_x}^2+{M_y}^2}$ is used as order parameter.

The motion of equation is derived as follows:

$$\begin{aligned}\frac{\mathrm{d}p_i}{\mathrm{d}t} &= -\frac{\partial H}{\partial q_i} \\ &= \frac{N}{2}\frac{\partial}{\partial q_i}\left(M_x^2 + M_y^2\right) \\ &= N\left(M_x\frac{\partial M_x}{\partial q_i} + M_y\frac{\partial M_y}{\partial q_i}\right) \\ &= -M_x \sin q_i + M_y \cos q_i \quad , \end{aligned} \tag{3.4}$$

$$\begin{aligned}\frac{\mathrm{d}q_i}{\mathrm{d}t} &= \frac{\partial H}{\partial p_i} \\ &= p_i \quad . \end{aligned} \tag{3.5}$$

### 3.1.2 Quasi Stationary States

It is reported that HMF model has quasi stationary states [1]. The time evolution of $M$ has two plateaus. The latter one is thermal equilibrium. However the former one is not thermal equilibrium. The duration of the former state diverges with $N$. Therefore, it is expected that when the limit $N \to \infty$ is taken, the duration diverges, and the system does not get out of such state. This is called quasi stationary state (QSS).

### 3.1.3 Vlasov Equation and Stationary Condition

Vlasov equation is the time evolution equation of one body distribution for collisionless system. As is well-known, Boltzmann equation is the time evolution equation of one-body distribution. Since HMF model is collisionless system, collisionless Boltzman equation holds. That is Vlasov equation, which is expressed as following:

$$\frac{\partial f}{\partial t} + p\frac{\partial f}{\partial q} + \left(-M_x\left[f\right]\sin q + M_y\left[f\right]\cos q\right)\frac{\partial f}{\partial p} = 0\,, \tag{3.6}$$

where $f\left(q,p,t\right)$ is the one-body distribution function, $\mathbf{M}\left[f\right] = \left(M_x\left[f\right], M_y\left[f\right]\right) = \int \mathbf{s}\left(q\right) f\left(q,p,t\right)\mathrm{d}q\mathrm{d}p$ is magnetization as a functional of distribution, and $\mathbf{s}\left(q\right) = \left(\cos q, \sin q\right)$ is single spin.

HMF model has quasi stationary states. On the limit $N \to \infty$, the duration of QSSs diverges. It is naive to consider that QSSs are considered as the stationary solution of Vlasov equation [1].

[1] Note that exceptions such as traveling clusters [46, 47] exist.

Consider (non periodic) stationary solutions, i.e. $f(q,p,t) = f(q,p)$. Substituting it to (3.6), one gets

$$p\frac{\partial f}{\partial q} + \left(-M_x[f]\sin q + M_y[f]\cos q\right)\frac{\partial f}{\partial p} = 0\,. \tag{3.7}$$

Here is a sufficient condition solution

$$f(q,p) = f\left(h\left(q,p;\mathbf{M}[f]\right)\right)\,, \tag{3.8}$$

where

$$h(q,p;\mathbf{M}) = \frac{p^2}{2} - \mathbf{M}\cdot(\cos q, \sin q) \tag{3.9}$$

denotes single-spin Hamiltonian. This result is reasonable: On stationary state, $\mathbf{M}$ is constant and accordingly single-spin Hamiltonian $h(q,p;\mathbf{M})$ is time-independent. Therefore each spin moves on the single-spin micro canonical orbit. In order to keep the same single-spin distribution function, the density at the same single-spin energy should be constant.

Actually, since micro canonical orbit $(q(t), p(t))$ satisfacts $\frac{\mathrm{d}q}{\mathrm{d}t} = p$ , $\frac{\mathrm{d}p}{\mathrm{d}t} = -M_x\sin q + M_y\cos q$, the left hand side of (3.7) is expressed as Lagrange derivative: $\frac{\mathrm{d}}{\mathrm{d}t}f(q(t), p(t)) = \frac{\partial f}{\partial q}\frac{\mathrm{d}q}{\mathrm{d}t} + \frac{\partial f}{\partial p}\frac{\mathrm{d}p}{\mathrm{d}t}$. Therefore the necessary and sufficient condition is that $f$ is constant alongside micro canonical orbit.

### 3.1.4 Lynden-Bell Equilibrium and Initial Condition

It is reported that Lynden-Bell equilibrium [45] explains QSS [5]. The basic idea comes from violent relaxation: Since Vlasov equation is solely an incompressive advection type equation, from the Lagrange view i.e. moving alongside with each particles, the $\mu$ space phase density does not change. Therefore, relaxation is only made by mixing of the phase space.

Two level initial distribution is useful to examine that idea Water bag initial condition (WBIC) is often used. Uniform density is put on a rectangle:

$$f(q,p) = \begin{cases} \dfrac{1}{4\Delta q\Delta p} & (|q| < \Delta q, |p| < \Delta p) \\ 0 & (\text{ otherwise }) \end{cases}\,. \tag{3.10}$$

where $\Delta q$ and $\Delta p$ denote half width of q and that of p respectively. In this study, WBIC is used. For the generation of $\Delta q$ and $\Delta p$ from $M_0$ and $e$, please see subsection 5.3.1.

Consider applying Lynden-Bell equilibrium to WBIC. The mixing entropy is introduced:

$$s\left[f\right]=-\int \mathrm{d}p\mathrm{d}q\left[\frac{f}{f_0}\ln\frac{f}{f_0}+\left(1-\frac{f}{f_0}\right)\ln\left(1-\frac{f}{f_0}\right)\right], \tag{3.11}$$

where $f_0=\dfrac{1}{4\Delta q\Delta p}$. Maximizing this entropy

$$S\left(e\right)=\max_f\left(s\left(f\right)\middle|e\left[f\right]=e,P\left[f\right]=0,\int \mathrm{d}p\mathrm{d}qf=1\right) \tag{3.12}$$

derives stationary states, where $f_0=\dfrac{1}{4\Delta q\Delta p}$, $e\left[f\right]=\displaystyle\int\frac{p^2}{2}f\mathrm{d}p\mathrm{d}q-\frac{\mathbf{M}^2-1}{2}$ and $P\left[f\right]=\int pf\mathrm{d}p\mathrm{d}q$. The derived stationary distribution is

$$f\left(q,p\right)=f_0\frac{\exp\left(-\beta\left(h\left(q,p;\mathbf{M}\left[f\right]\right)-\mu\right)\right)}{1+\exp\left(-\beta\left(h\left(q,p;\mathbf{M}\left[f\right]\right)-\mu\right)\right)}. \tag{3.13}$$

Core-halo distribution [7] is also reported:

$$\begin{aligned} f\left(q,p\right)=&\eta_0\chi\left(h_F-h\left(q,p;\mathbf{M}\left[f\right]\right)\right) &(3.14)\\ &+\eta_1\chi\left(h_h-h\left(q,p;\mathbf{M}\left[f\right]\right)\right)\chi\left(h\left(q,p;\mathbf{M}\left[f\right]\right)-h_F\right). &(3.15)\end{aligned}$$

Although this distribution explains the simulation data well, discrepancies exist.

## 3.2 Method

In this study MD simulation of HMF is performed instead of integrating Vlasov equation.

For the implementation, see section 2.2. For the implementation code, see section 5.2. Time slice $\Delta t$ is 0.1. The 4th order symplectic integrator is used. The water bag initial condition (see section 3.1.4) is taken. The devices used in this chapter are Tesla C2050 and Tesla K20 (see section 2.2.1).

### 3.2.1 Energy Moment

In order to characterize QSSs, energy moment is introduced in following.

### Area Density

As explained on 3.1.3, The distribution of stationary state is convertible to the density profile of one body microcanonical orbit and vice-versa.

There is a slight difference between specifying orbit and specifying one-body energy $h(q,p;\mathbf{M})$. For example $(q,p)$ and $(q,-p)$ have the same one body energy $h(q,p;\mathbf{M}) = h(q,-p;\mathbf{M})$ , but they have different orbits. Nevertheless assuming symmetric profile of $p$, the difference is taken away. Then, area density function is introduced:

$$f(h;\mathbf{M}) = \frac{\rho(h;\mathbf{M})}{D(h;\mathbf{M})} \quad , \tag{3.16}$$

where $\rho(h;\mathbf{M}) = \int f(q,p)\,\delta(h(q,p;\mathbf{M}) - h)\,\mathrm{d}q\mathrm{d}p$ is one body energy distribution and $D(h;\mathbf{M}) = \int \delta(h(q,p) - h)\,\mathrm{d}q\mathrm{d}p$ is the density of states (see section 5.3.2).

### Moment

In order to specify QSS by a set of finite variables, the author considered moment expansion of $f(h;\mathbf{M})$. Both energy area density function $f(h;\mathbf{M})$ and energy distribution function $\rho(h;\mathbf{M})$ specifies stationary states uniquely by its definition. $f$ is advantageous for analysis: $f$ is not influenced by $D$. Therefore moment of $f$ is considered. Three definitions of moment are conceivable: moment about $-M$[2], moment about 0 and moment about "average"[3]. The definition of moment about $c$ is

$$\mu_n^c \equiv \frac{\int (h-c)^n f(h;\mathbf{M})\,\mathrm{d}h}{\int f(h;\mathbf{M})\,\mathrm{d}h} \tag{3.17}$$

$$= \frac{\int (h-c)^n \frac{\rho(h;\mathbf{M})}{D(h;\mathbf{M})}\mathrm{d}h}{\int \frac{\rho(h;\mathbf{M})}{D(h;\mathbf{M})}\mathrm{d}h} \tag{3.18}$$

$$= \frac{\left\langle \frac{(h-c)^n}{D(h;\mathbf{M})} \right\rangle}{\left\langle \frac{1}{D(h;\mathbf{M})} \right\rangle}. \tag{3.19}$$

[2] $-M$ is the minimum value of $h$.

[3] The word average is quoted because it is different from normal average. It is explained later.

For the "average" $c$ is $c_{\text{avg}} = \int hf(h;\mathbf{M})\,\mathrm{d}h / \int f(h;\mathbf{M})\,\mathrm{d}h$. For your information, the normal average is $\int h\rho(h;\mathbf{M})\,\mathrm{d}h / \int \rho(h;\mathbf{M})\,\mathrm{d}h$.

On the following paragraphs, the three definitions of moment are compared for Maxwell-Boltzmann case and step function case.

**Maxwell-Boltzmann Distribution** For thermal equilibrium states, $f(h;\mathbf{M})$ is Maxwell-Boltzmann type $f(h;\mathbf{M}) \propto \exp(-\beta h) \quad (-M \leq h)$. The moment about $c$ is

$$\mu_n^c = \frac{\dfrac{\partial^n}{\partial\beta^n}\dfrac{\exp(-\beta(-M-c))}{\beta}}{\dfrac{\exp(-\beta(-M-c))}{\beta}}. \tag{3.20}$$

For the case $c = -M$, it is quite simple:

$$\mu_n^{-h} = \frac{(-1)^n n!}{\beta^{n+1}}. \tag{3.21}$$

From this fact, moment about $-h$ is preferable for analysis.

**Step Function** Consider a case when the area density function is step function: $f(h;\mathbf{M}) = 1/\Delta x \quad (-M \leq h \leq \Delta x - M)$. The moment is

$$\mu_n^c = \frac{(\Delta x - M - c)^{n+1} - (-M-c)^{n+1}}{(n+1)\Delta x}. \tag{3.22}$$

The moment about $-h$ is quite simple:

$$\mu_n^{-M} = \frac{\Delta x^n}{n+1}. \tag{3.23}$$

Therefore, for step function too, the moment about $-h$ is preferable.

From now on, we call $\mu_n^{-M}$ simply $\mu_n$.

## 3.3 Results and Discussion

### 3.3.1 Necessary Times for Calculation

Let $\tau_{\text{X}}^s$ the necessary simulation time for state X. Therefore necessary calculation time is $\tau^c = 3\dfrac{N\tau_{\text{X}}^s}{\Delta t} \times \text{MUPS} \times 10^{-6}$ [s]. The number "3" on the expression comes from the 3 loops on 1 update of 4th order symplectic integrator.

Let the number of particles $N = 65536$. For early stage QSS stated later, $\tau_{\text{E}}^s = 6553.6$. For late stage QSS stated later, $\tau_{\text{L}}^s = 10^7$. The necessary calculation time is shown on 3.1.

| Device | MUPS | $\tau_{\mathrm{E}}^{c}$ [s] | $\tau_{\mathrm{L}}^{c}$ [h] |
|---|---|---|---|
| Tesla C2050 | 383 | 34 | 14.3 |
| Tela K20 | 621 | 21 | 8.8 |
| Xeon Phi 5110 | 348 | 37 | 15.7 |
| SR16000 MPI | 213 | 61 | 25.6 |
| SR16000 OpenMP | 173 | 75 | 31.6 |

Table 3.1: The necessary calculation time. $N = 65536$.

### 3.3.2 Various QSS Distributions

Changing WBIC parameters, various QSS distributions are realized as shown on Figure 3.1. The QSS magnetization $M_{\mathrm{QSS}}$ is shown on Figure 3.2.

### 3.3.3 Two Stages of QSS

For certain cases of initial conditions, two QSSs are realized in addition to the final thermal equilibrium state. The magnetizations of two stages are almost 0. As can be seen in figure 3.3, the fluctuation level of magnetization is different. The time duration of each stage is examined and shown on figure 3.4 , 3.5 and 3.6. The scaling of duration $\tau \propto N$ also appears when $M > 0$, and the scaling of duration $\tau \propto N^{1.7}$ appears when $M = 0$ is known and analyzed [3, 48].

As can be seen in time series, on early stage, the fluctuation of relatively large. Since both stages have almost 0 magnetization, all the particles move almost parallel to $q$ axis. As shown on figure 3.7, the distribution of momentum $p$ is examined, and the two distributions are not so different each other. However, as can be seen in figure 3.8 and figure 3.9 the phase space distributions are quite different.

Particularly for early stage, traveling clusters [46, 47] are observed. The phase diagram of collective oscillation is researched [49]. The time evolution of magnetization shown on figure 3.10 is quite similar to acoustic beat wave. Assuming the traveling clusters are the cause of beat wave, the traveling time period of clusters are closely related. From the time series, the corresponding absolute values of momentum are 0.48 and 0.52, which are within the area of two clusters.

As for late stage, clusters faded away, and beat wave form is not observed. Still, as can be seen in figure 3.11, the collective oscillation exists, though the amplitude is smaller than that of early stage. Absolute momentum value corresponding to the oscillation frequency is 0.55, which is within the vicinity of the peak of momentum distribution.

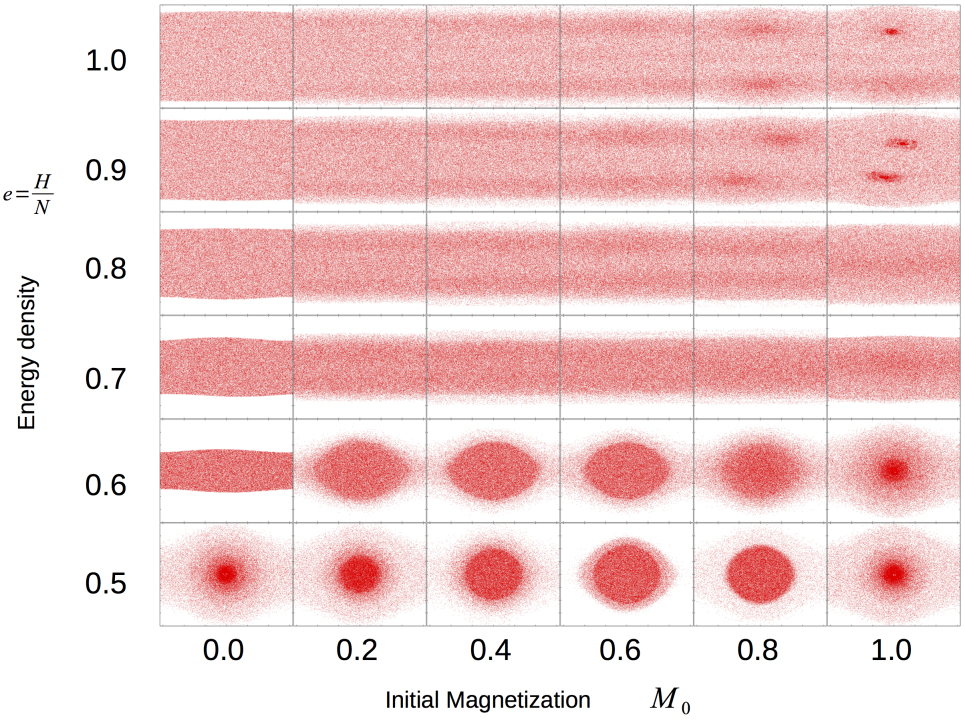


Figure 3.1: A montage picture of various QSS Snapshots arising from WBIC. The horizontal axis and the vertical axis of each snapshot picture show the angle $q$ $(-\pi \leq q \leq \pi)$ and the momentum $p$ $(-2 \leq p \leq 2)$ respectively. The parameters are as follows: $N = 65536$. $t = 6553.6$.

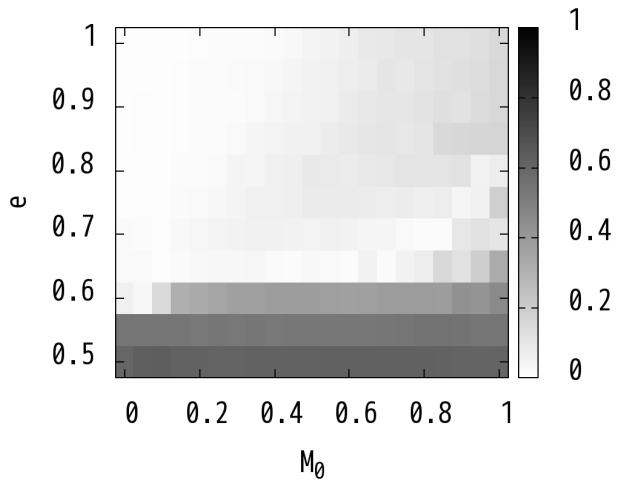


Figure 3.2: QSS magnetization $M$ as a function of $e, M_0$. The binwidth of $M_0$ and that of $e$ are both 0.05. The parameters are as follows: $N = 65536$, $t = 6553.6$.

As described above, the early stage QSS has traveling clusters As time passes, the traveling clusters fades and the oscillation becomes small.

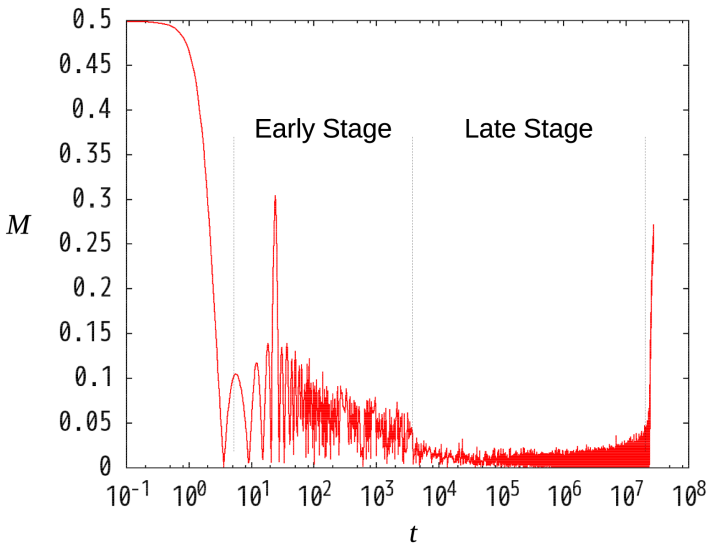


Figure 3.3: Time series of magnetization $M$. Two QSS stages are observed. The parameters are as follows: $e = 0.7$, $M_0 = 0.5$, $N = 65536$.

### 3.3.4 Non Unique QSS

When the initial condition is specified, thermal equilibrium is uniquely specified. For HMF model, the final canonical temperature is specified by the energy density. However for some cases, QSS is not uniquely specified even by the macroscopic initial condition variable set $(e, M_0)$. As figure 3.12 shows, the same initial parameter $(e, M_0)$ does not necessarily result int the same QSS. This tendency arises on certain region as figure 3.13 shows.

Comparing time series shown on figure 3.14, magnetization is rather close until $t \sim 10$, and the variation of magnetization arises after that.

### 3.3.5 Set of Quantities Specifying QSS Uniquely

Although numbers of stability studies for QSS of HMF are reported, the number of QSS characterizing variables is not finite. Therefore extending the idea previously stated for non thermal equilibrium cases is difficult. As a small step, QSSs is restricted to be WBIC arising ones.

$(e, M_0)$ is WBIC parameter set. Therefore if one has certain $M_0$ specifying variable(s), QSS is specified uniquely. However it is not a necessary condition:

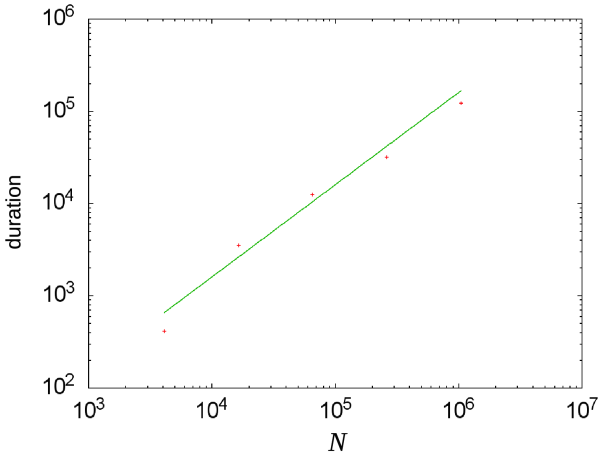


Figure 3.4: Time duration of early stage QSS as a function of $N$. The red points are obtained from simulation. The green line shows fitting function $kN$. The parameters are as follows: $e = 0.7$, $M_0 = 0.5$, $N = 65536$.

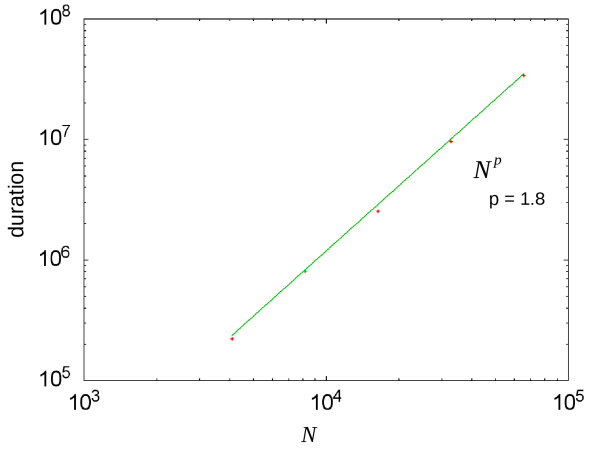


Figure 3.5: Time duration of late stage QSS as a function of $N$. The red points are obtained from simulation. The green line shows fitting function $kN^p$ , where $p = 1.8$. The parameters are as follows: $e = 0.7$, $M_0 = 0$, $N = 65536$.

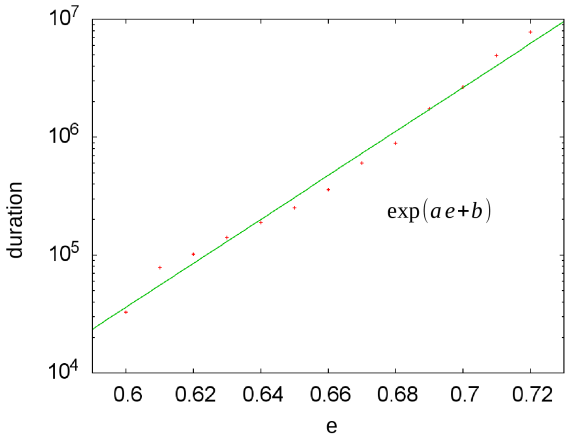


Figure 3.6: Time duration of late stage QSS as a function of energy density $e$. The red points are obtained from simulation. The green line shows fitting function $\exp(ae+b)$ , where $a = 42.932, b = -15.268$. The parameters are as follows: $M_0 = 0$, $N = 16384$.

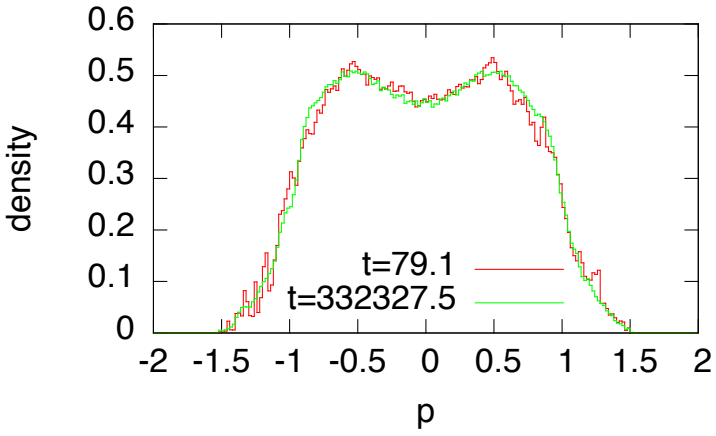


Figure 3.7: Density profile of momentum $\rho(p)$. $\rho(p)$ is normalized so that the equation $\int \rho(p)\,\mathrm{d}p = 1$ holds. The binwidth is 0.02. $t = 79.1$ is within early stage QSS. $t = 332327.5$ is within late stage QSS. Discrepancy between the two is rather small, despite of large time difference. The parameters are as follows: $e = 0.7$, $M_0 = 0.5$, $N = 1048576$.

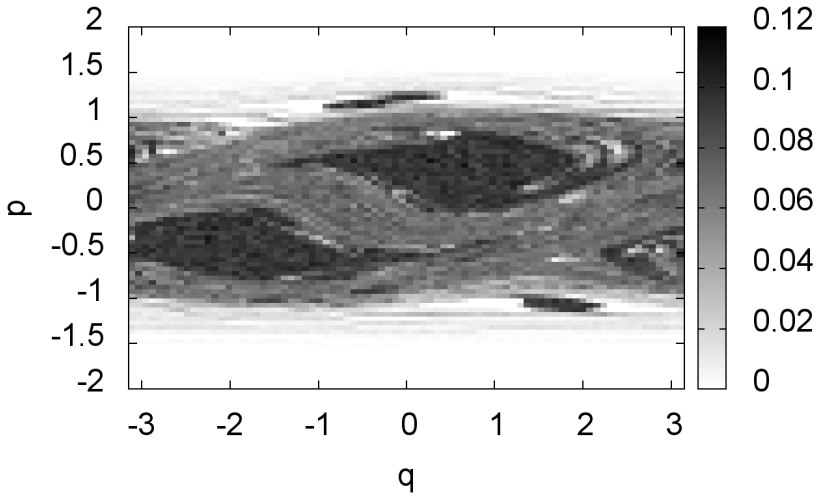


Figure 3.8: One body phase space distribution of early stage QSS $f(q,p)$. $f(q,p)$ is drawn as brightness. $f(q,p)$ is normalized so that $\int f(q,p)\,\mathrm{d}q\mathrm{d}p = 1$. The binwidth of $q$ and that of $p$ are $\pi/40$ and 0.05 respectively. Traveling clusters [46, 47] are observed. In order to obtain a clear picture of clusters, the large number of particles is set here. The parameters are as follows: $e = 0.7$. $M_0 = 0.5$. $N = 1048576$. $t = 79.1$.

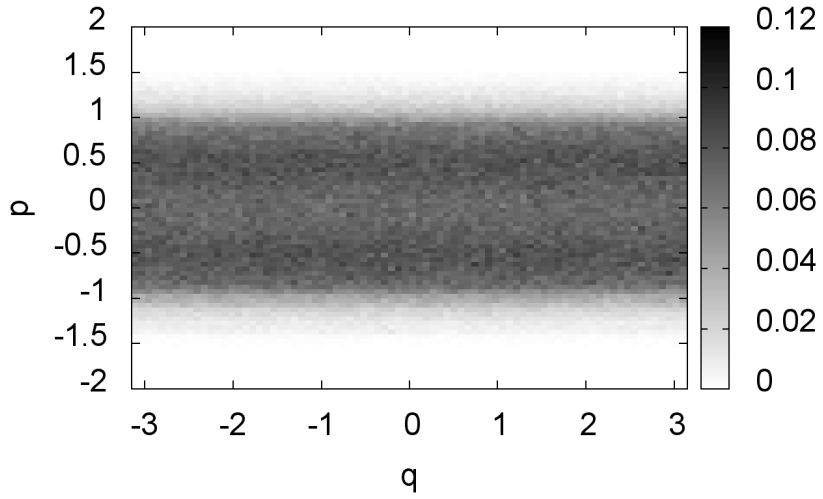


Figure 3.9: One body phase space distribution of late stage QSS $f(q,p)$. $f(q,p)$ is drawn as brightness. $f(q,p)$ is normalized so that $\int f(q,p)\,\mathrm{d}q\mathrm{d}p = 1$. The binwidth of $q$ and that of $p$ are $\pi/40$ and 0.05 respectively. Clusters faded away, which is present at early stage . The parameters are as follows: $e = 0.7$. $M_0 = 0.5$. $N = 1048576$. $t = 332327.5$.

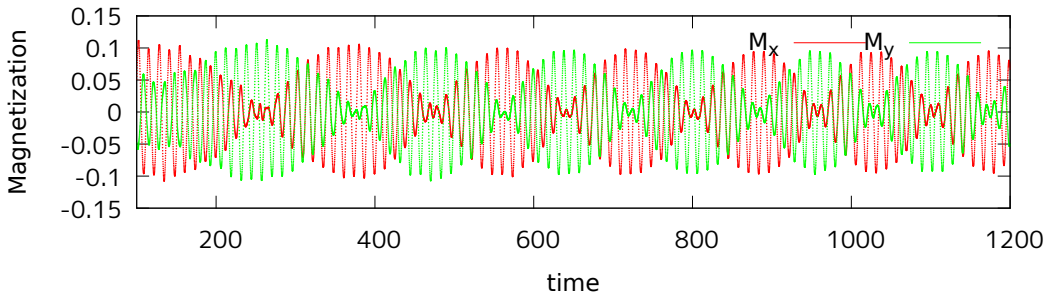


Figure 3.10: Time evolution of magnetization on early stage QSS. It seems similar to acoustic beat wave. The number of the wave peaks in $100 < t < 1100$ is 80, and the number of the envelop peaks in $260 < t < 1105$ is 5. Therefore assuming this is the composition of two close frequencies, the frequencies are 0.077 and 0.083. Assuming $M = 0$, the corresponding absolute values of momentum $p$ are 0.48 and 0.52. The parameters are as follows: $e = 0.7$, $M_0 = 0.5$, $N = 1048576$.

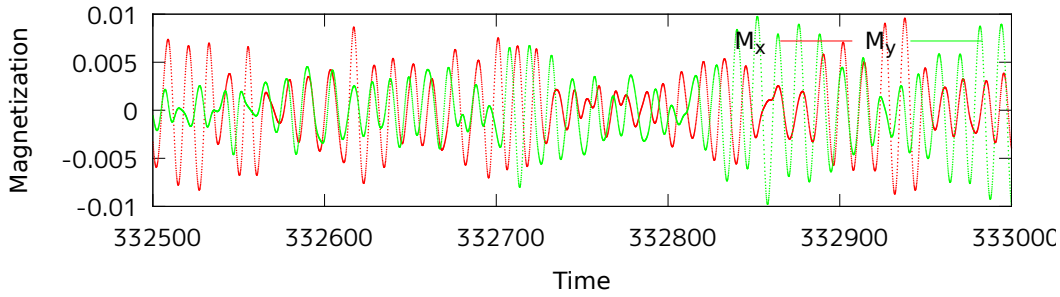


Figure 3.11: Time evolution of magnetization on early stage QSS. It does not seem to be a beat wave form. The number of the wave peaks in $332500 < t < 333000$ is 44. Therefore the frequency is 0.088. Assuming $M = 0$, the corresponding absolute value of momentum $p$ is 0.55. The parameters are as follows: $e = 0.7$, $M_0 = 0.5$, $N = 1048576$.

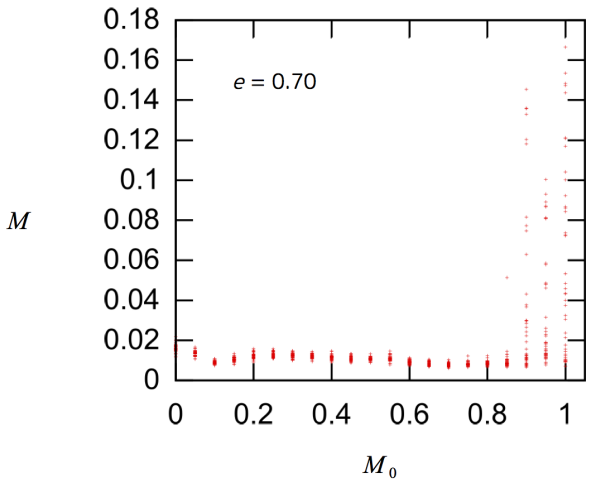


Figure 3.12: QSS magnetization $M$ as a function of initial magnetization $M_0$. $e = 0.70, N = 65536$. $N_s$ the number of samples per one parameter set is 50. Even when the same WBIC parameter set $(e, M_0)$ is chosen, the realized QSS can differ for certain cases.

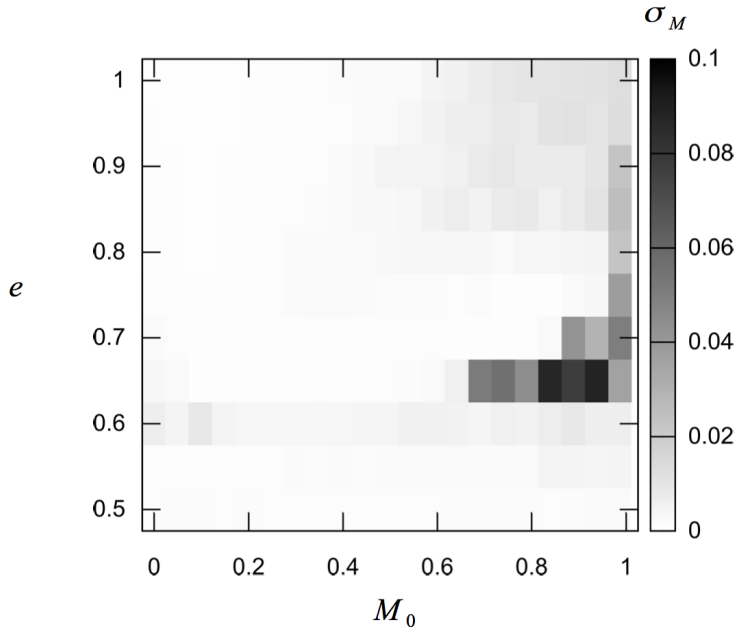


Figure 3.13: Uncorrected sample standard deviation of magnetization $\sigma_M = \sqrt{\frac{1}{N_\mathrm{s}} \sum_{i=1}^{N_\mathrm{s}} \left(M_i - \bar{M}\right)^2}$. $\sigma_M$ is drawn as brightness. The binwidth of $M_0$ and that of $e$ are both 0.05. The number of samples $N_\mathrm{s}$ is 50.

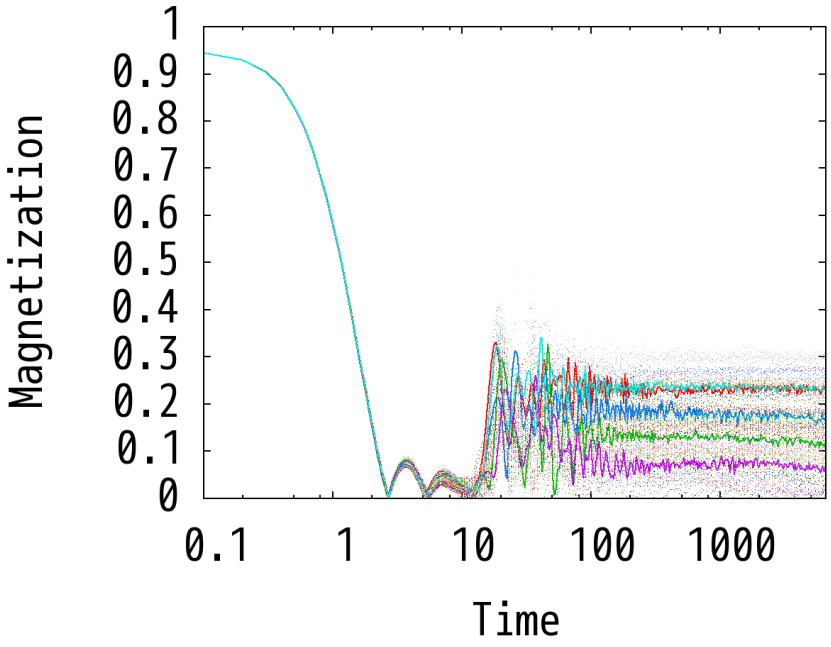


Figure 3.14: Time series of magnetization $M$ for various random seeds. The number of samples is 100. The dotted points are obtained from 100 simulations. The 5 lines shows interpolation lines of 5 samples for eye guide. The parameters are as follows: $e = 0.65$, $M_0 = 0.95$, $N = 65536$.

part of information of initial condition can possibly be lost during relaxation. Note that $(e, M)$ is not sufficient. For example, when $e$ is sufficiently large, $M = 0$ for any $M_0$ , and $f(h; \mathbf{M})$ changes as $M_0$. The goal is to specify QSSs by $e$, $M$ and certain limited number of $\mu_n$. Since the system is rotational symmetric, $M_y = 0$ is assumed and $M = M_x$ holds.

Dependence of $M_0$ on energy moment $\mu_2$ is inspected. As can be seen in figure 3.15, energy moment $\mu_2$ remember initial magnetization $M_0$ partly.

Specifying $\mu_n$ for any positive integer $n$ determines energy distribution $f(h; \mathbf{M})$. Therefore if $\mu_n$ is specified by certain set of quantities, the set of quantities specifies energy distribution.

As seen in figure 3.16 and 3.17, $\mu_4$ is uniquely specified by $(e, M, \mu_2)$. Moreover that is true for $\mu_n$ $(n \leq 10)$. This fact is not proved for any positive integer $n$ this time. Nevertheless, assuming that, energy distribution $f(h; \mathbf{M})$ is uniquely specified by $(e, M, \mu_2)$.

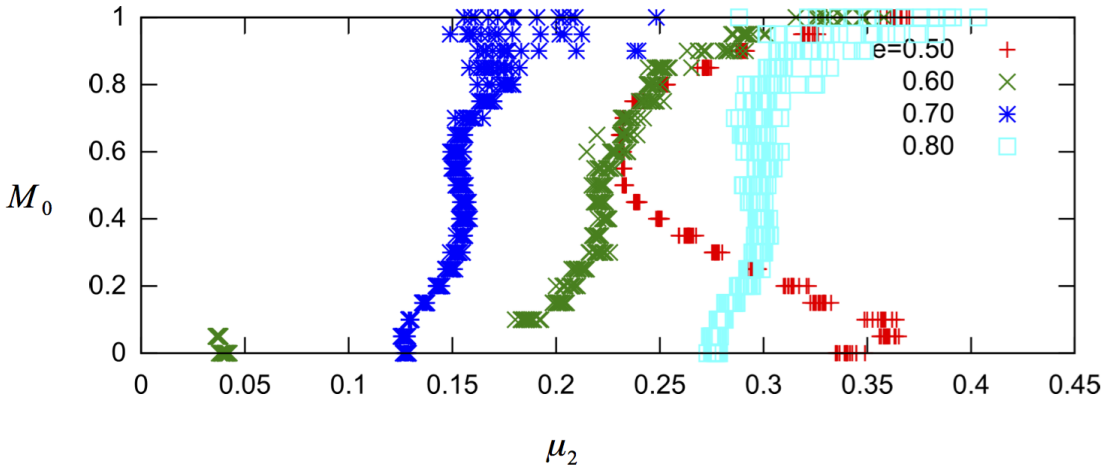


Figure 3.15: Initial magnetization $M_0$ as a function of $\mu_2$. Note that for the same $e$, $M_0$ does not make $M$ change largely, which can be seen in figure 3.2. The number of particles $N$ is 65536.

Information of initial configuration is to a certain degree remembered as a phase distribution. That remembered information is extracted by energy momentum $\mu_2$, and $(e, M, \mu_2)$ succeeded to characterize QSSs arising from WBIC uniquely.

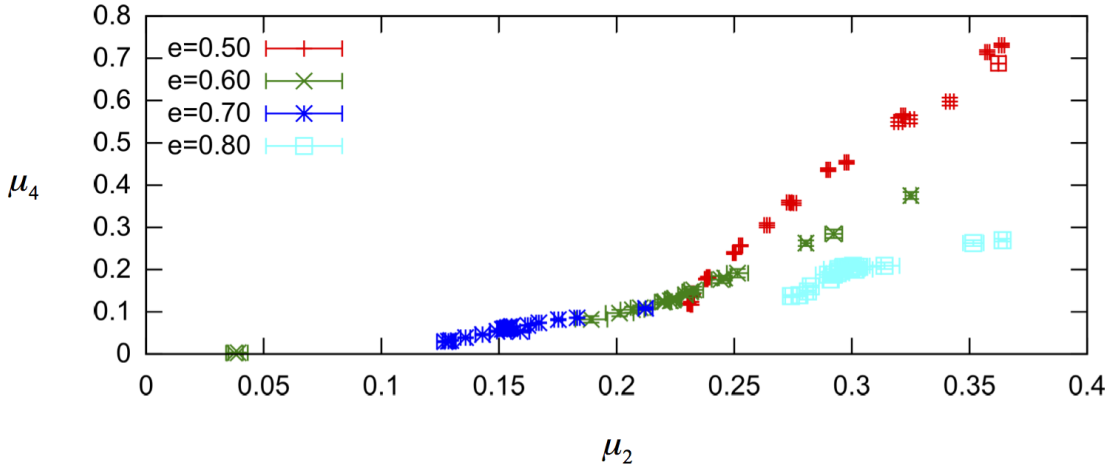


Figure 3.16: $\mu_4$ as a function of $\mu_2$ is shown. It seems that for each energy density $e$, $\mu_4$ is specified uniquely by $\mu_2$. However for $e = 0.70$, it is not true. The number of particles $N$ is 65536.

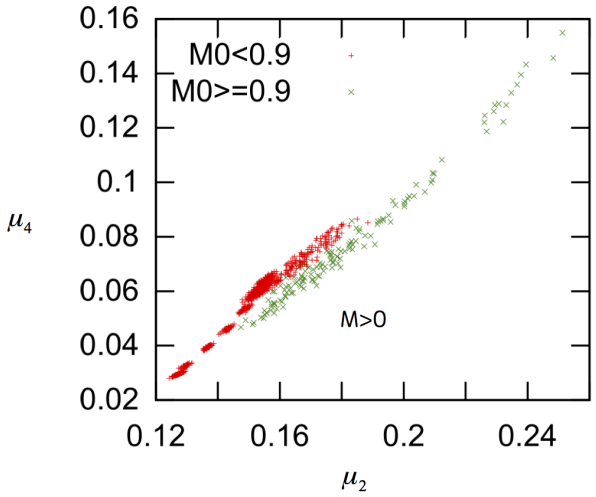


Figure 3.17: $\mu_4$ as a function of $\mu_2$ for $e = 0.70$ is shown. The red points are obtained from simulations where $M_0 < 0.9$. The green points are obtained from simulations where $M_0 \geq 0.9$. For each cases, $\mu_4$ is uniquely specified by $\mu_2$. However for all points it is not true. For $M_0 < 0.9$, $M = 0$ and for $M_0 \geq 0.9$, $M \geq 0$. Therefore using $M$ as an additional quantity, $\mu_4$ is uniquely specified. The number of particles $N$ is 65536.

## 3.4 Summary

HMF model is a toy model for long range interacting systems. The author aimed at finding set of variables which specifies QSS. Utilizing GPU, various QSS is examined. Then two stages of QSSs and non unique QSS are found. For two stages of QSSs, on the early stage QSS the phase distribution is nonuniform and that makes collective oscillation [49]. As time passes the nonuniformity is relaxed and collective oscillation disappears. About non unique QSS, time series of magnetization suggest that there is fork when $M = 0$. Introducing energy moment, the author succeeded int specifying QSSs uniquely. For long range systems, this energy moment has a possibilty to play a good role and examining it is a future work.

# Chapter 4

# Summary and Perspective

In recent years, accelerator devices such as GPU and Intel Xeon Phi are receiving attentions. From the viewpoint of initial financial cost, power consumption, and physical space, such accelerators are advantageous, but actually it is not necessarily true. This time the performance for MD simulation is examined. For LJ system, accelerators are not advantageous. However, for HMF model they are useful. The performance for HMF model is estimated by an idea similar to Amdahl's law [33]: particle-wise time $a$ and system-wise time $b$. For practically use, it is important to reduce $b$ as well as $a$. $b$ is related to synchronization time, and it seems the hierarchical architecture of many core accelerator makes $b$ small. In addition, for many core accelerators, the low core frequency reduced data hazard and makes instructions on pipeline processed efficiently. From now on, searching for topics for which many core accelerators are advantageous, and utilizing them efficiently is an assignment.

HMF model is a toy model for long range interacting systems. The author aimed at finding set of variables which specifies QSS. Utilizing GPU, various QSS is examined. Then two stages of QSSs and non unique QSS are found. For two stages of QSSs, on the early stage QSS the phase distribution is nonuniform and that makes collective oscillation [49]. As time passes the nonuniformity is relaxed and collective oscillation disappears. About non unique QSS, time series of magnetization suggest that there is fork when $M = 0$. Introducing energy moment, the author succeeded int specifying QSSs uniquely. For long range systems, this energy moment has a possibility to play a good role and examining it is a future work.

# Appendix

## 5.1 Implementation Code for LJ Simulation

Template library thrust [50] is used in the code.

```
#include <unistd.h>
#include <cstdio>
#include <cstdlib>
#include <ctime>
#include <cmath>
#include <csignal>
#include <vector>
#include <algorithm>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/scan.h>
#include <thrust/fill.h>
#include <thrust/copy.h>
// #define MD_DEBUG
// #define THRUST_DEBUG
#define CHECK_CUDA_ERROR(err) check_cuda_error
    (err, __FILE__, __LINE__)
static void check_cuda_error(const cudaError_t
    err, const char* const file, const int
    line)
{
  if(err!=cudaSuccess) {
    const char* str_err = cudaGetErrorString(
        err);
    char what_arg[128];
    sprintf(what_arg, "[%s:%d] %s\n", file,
        line, str_err);
    throw thrust::system::system_error(err,
        thrust::cuda_category(), what_arg);
  }
}
#ifdef MD_DEBUG
#define CHECK_CUDA_KERNEL_ERROR()
    CHECK_CUDA_ERROR(cudaDeviceSynchronize())
#else
#define CHECK_CUDA_KERNEL_ERROR()
#endif

static int sigint_status = 0;
static void sigint_catch(const int sig)
{
  sigint_status++;
  fprintf(stderr, "[Info] SIGINT (%d)\n",
      sigint_status);
  if(sigint_status == 3) {
    cudaDeviceReset();
    exit(1);
  }
}
namespace md
{
  typedef double float_t;
  typedef int int_t;
  typedef unsigned int uint_t;
}

namespace mylib
{
  struct add
  {
    template<typename T>
    __device__ static T join(const T x, const
        T y) {
      return x + y;
    }
  };
  struct multiply
  {
    template<typename T>
    __device__ static T join(const T x, const
        T y) {
      return x * y;
    }
  };
  struct max
  {
    template<typename T>
    __device__ static T join(const T x, const
        T y) {
      return ::max(x, y);
    }
  };
  struct min
  {
    template<typename T>
    __device__ static T join(const T x, const
        T y) {
      return ::min(x, y);
    }
  };

  namespace func {

    template<typename T, typename F, bool
        if_sync>
    __device__ static int reduce_half
    (
     int active_threads,
     T* const sum1_shared
     )
    {
      T sum1_self = sum1_shared[threadIdx.x];
      const int half_active_threads =
          active_threads >> 1;
      active_threads = (active_threads+1) >>
          1;
      if(if_sync) { __syncthreads(); }
      if(threadIdx.x < half_active_threads){
        T sum1_read = sum1_shared[threadIdx.x+
            active_threads];
        sum1_shared[threadIdx.x] = F::join(
            sum1_self, sum1_read);
      }
      return active_threads;
    }
    template<typename T, typename F>
    __device__ static void reduce
    (
     T* const sum1_shared
     )
    {
      int active_threads = blockDim.x;
```

```
      while (active_threads > 32) {
        active_threads = func::reduce_half<T,
            F, true>
          (active_threads, sum1_shared);
      }
      active_threads = func::reduce_half<T, F,
          false>
        (active_threads, sum1_shared);
      active_threads = func::reduce_half<T, F,
          false>
        (active_threads, sum1_shared);
      active_threads = func::reduce_half<T, F,
          false>
        (active_threads, sum1_shared);
      active_threads = func::reduce_half<T, F,
          false>
        (active_threads, sum1_shared);
      active_threads = func::reduce_half<T, F,
          false>
        (active_threads, sum1_shared);
    }
  }
  namespace kernel
  {
    // required shared mem size : blocksize *
        sizeof(T)
    template<typename T, typename F>
    __global__ static void reduce
    (
     const T* const input1_global, T* const
         output1_global
    )
    {
      extern __shared__ T sum1_shared[];
      const int idx = blockDim.x * blockIdx.x
          + threadIdx.x;
      sum1_shared[threadIdx.x] = input1_global
          [idx];
      mylib::func::reduce<T, F>(sum1_shared);
      if(threadIdx.x==0) {
        output1_global[blockIdx.x] =
            sum1_shared[0];
      }
    }
  }
}

__global__ static void init_kernel
(
 const md::float_t particle_interval,
 const md::int_t num_particles_line,
 const md::float_t pos_x_0,
 const md::float_t pos_y_0,
 const md::float_t pos_z_0,
 md::float_t* const pos_x,
 md::float_t* const pos_y,
 md::float_t* const pos_z,
 md::float_t* const mom_x,
 md::float_t* const mom_y,
 md::float_t* const mom_z,
 md::float_t* const f_x,
 md::float_t* const f_y,
 md::float_t* const f_z
)
{
  const int idx = blockDim.x * blockIdx.x +
      threadIdx.x;
  md::int_t n_line = idx;
  pos_x[idx] = (n_line % num_particles_line )
      * ( particle_interval ) + pos_x_0;
  n_line /= num_particles_line;
  pos_y[idx] = (n_line % num_particles_line )
      * ( particle_interval ) + pos_y_0;
  n_line /= num_particles_line;
  pos_z[idx] = (n_line % num_particles_line )
      * ( particle_interval ) + pos_z_0;
  mom_x[idx] = 0;
  mom_y[idx] = 0;
  mom_z[idx] = 0;
  f_x[idx] = 0;
  f_y[idx] = 0;
  f_z[idx] = 0;
}

__global__ static void detect_cell_num_kernel
(
 const md::float_t cell_size,
 const md::float_t cell_size_inv,
 const md::int_t log_num_cells_x,
 const md::int_t log_num_cells_y,
 const md::int_t log_num_cells_z,
 md::float_t* const pos_x_global,
 md::float_t* const pos_y_global,
 md::float_t* const pos_z_global,
 md::int_t* const cell_num_global,
 md::int_t* const num_particles_cell_global,
 md::int_t* const
      num_particles_cell_virt_global,
 md::int_t* const
      num_particles_cell_imag_global,
 md::int_t* const addr_in_cell_global
)
{
  const md::int_t num_cells_x = 1<<
      log_num_cells_x;
  const md::int_t num_cells_y = 1<<
      log_num_cells_y;
  const md::int_t num_cells_z = 1<<
      log_num_cells_z;
  const uint idx = blockDim.x * blockIdx.x +
      threadIdx.x;
  md::float_t pos_x_idx = pos_x_global[idx];
  md::float_t pos_y_idx = pos_y_global[idx];
  md::float_t pos_z_idx = pos_z_global[idx];
  md::int_t n_x = pos_x_idx * cell_size_inv;
  md::int_t n_y = pos_y_idx * cell_size_inv;
  md::int_t n_z = pos_z_idx * cell_size_inv;
  /* mod periodic */
  if (n_x==0) {
    const md::float_t periodic_mod_x = (
        num_cells_x-2)*cell_size;
    pos_x_idx += periodic_mod_x;
    pos_x_global[idx] = pos_x_idx;
    n_x = num_cells_x - 2;
  } else if (n_x==num_cells_x-1){
    const md::float_t periodic_mod_x = (
        num_cells_x-2)*cell_size;
    pos_x_idx -= periodic_mod_x;
    pos_x_global[idx] = pos_x_idx;
    n_x = 1;
  }
  if (n_y==0) {
    const md::float_t periodic_mod_y = (
        num_cells_y-2)*cell_size;
    pos_y_idx += periodic_mod_y;
    pos_y_global[idx] = pos_y_idx;
    n_y = num_cells_y - 2;
  } else if (n_y==num_cells_y-1){
    const md::float_t periodic_mod_y = (
        num_cells_y-2)*cell_size;
    pos_y_idx -= periodic_mod_y;
    pos_y_global[idx] = pos_y_idx;
    n_y = 1;
  }
  if (n_z==0) {
    const md::float_t periodic_mod_z = (
        num_cells_z-2)*cell_size;
    pos_z_idx += periodic_mod_z;
    pos_z_global[idx] = pos_z_idx;
    n_z = num_cells_z - 2;
  } else if (n_z==num_cells_z-1){
    const md::float_t periodic_mod_z = (
        num_cells_z-2)*cell_size;
    pos_z_idx -= periodic_mod_z;
    pos_z_global[idx] = pos_z_idx;
    n_z = 1;
  }
  const md::int_t n = n_x | ( ( n_y | ( n_z <<
      log_num_cells_y ) ) << log_num_cells_x
      );
  cell_num_global[idx] = n;
  addr_in_cell_global[idx] = atomicAdd(&
      num_particles_cell_global[n], 1);
  atomicAdd(&num_particles_cell_virt_global[n
      ], 1);
  /* mirroring */
  for(int mz=0; mz<3; ++mz) {
    bool if_z;
    md::int_t _n_z;
    switch(mz) {
    case 0:
      if(n_z==1) {
        if_z = true;
        _n_z = num_cells_z - 1;
        break;
      } else {
        continue;
      }
    case 1:
      if_z = false;
      _n_z = n_z;
      break;
    case 2:
      if(n_z==num_cells_z-2){
```

```
        if_z = true;
        _n_z = 0;
        break;
      } else {
        continue;
      }
    }
    for(int my=0; my<3; ++my) {
      md::int_t if_y;
      md::int_t _n_y;
      switch(my) {
      case 0:
        if(n_y==1) {
          if_y = true;
          _n_y = num_cells_y - 1;
          break;
        } else {
          continue;
        }
      case 1:
        if_y = false;
        _n_y = n_y;
        break;
      case 2:
        if(n_y==num_cells_x-2) {
          if_y = true;
          _n_y = 0;
          break;
        } else {
          continue;
        }
      }
      for(int mx=0; mx<3; ++mx) {
        md::int_t if_x;
        md::int_t _n_x;
        switch(mx) {
        case 0:
          if(n_x==1){
            if_x = true;
            _n_x = num_cells_x - 1;
            break;
          } else {
            continue;
          }
        case 1:
          if_x = false;
          _n_x = n_x;
          break;
        case 2:
          if(n_x==num_cells_x-2){
            if_x = true;
            _n_x = 0;
            break;
          } else {
            continue;
          }
        }
        if(if_x||if_y||if_z) {
          const md::int_t _n = _n_x | ( ( _n_y
                | ( _n_z << log_num_cells_y )
              ) << log_num_cells_x );
          atomicAdd(&
                num_particles_cell_virt_global[
                _n], 1);
          atomicAdd(&
                num_particles_cell_imag_global[
                _n], 1);
        } /* end if_x if_y if_z */
      } /* end for mx */
    } /* end for my */
  } /* end for mz */

}

__global__ static void
     sort_and_update_addr_kernel
(
 const md::float_t cell_size,
 const md::int_t log_num_cells_x,
 const md::int_t log_num_cells_y,
 const md::int_t log_num_cells_z,
 md::float_t* const pos_x_global,
 md::float_t* const pos_y_global,
 md::float_t* const pos_z_global,
 md::float_t* const mom_x_global,
 md::float_t* const mom_y_global,
 md::float_t* const mom_z_global,
 md::int_t* const cell_num_global,
 md::int_t* const addr_in_cell_global,
 md::float_t* const pos_x_new_global,
 md::float_t* const pos_y_new_global,
 md::float_t* const pos_z_new_global,
 md::float_t* const mom_x_new_global,
 md::float_t* const mom_y_new_global,
 md::float_t* const mom_z_new_global,
 md::int_t* const cell_num_new_global,
 md::int_t* const addr_in_cell_new_global,
 const md::int_t* const cell_begin_global,
 const md::int_t* const cell_begin_virt_global
      ,
 const md::int_t* const cell_begin_imag_global
      ,
 // md::int_t* const logical_addr_global,
 md::int_t* const logical_addr_virt_global,
 md::int_t* const physical_addr_global
 )
{
  const uint idx =  blockDim.x * blockIdx.x +
       threadIdx.x;

  const md::int_t cell_num = cell_num_global[
       idx];

  const md::int_t addr_in_cell_idx =
       addr_in_cell_global[idx];

  const md::float_t pos_x_idx = pos_x_global[
       idx];
  const md::float_t pos_y_idx = pos_y_global[
       idx];
  const md::float_t pos_z_idx = pos_z_global[
       idx];

  const md::float_t mom_x_idx = mom_x_global[
       idx];
  const md::float_t mom_y_idx = mom_y_global[
       idx];
  const md::float_t mom_z_idx = mom_z_global[
       idx];

  const md::int_t cell_begin_log_idx =
       cell_begin_virt_global[cell_num];
  const md::int_t cell_begin_phys_idx =
       cell_begin_global[cell_num];

  const md::int_t logical_addr_idx =
       cell_begin_log_idx + addr_in_cell_idx;
  const md::int_t physical_addr_idx =
       cell_begin_phys_idx + addr_in_cell_idx;
  // logical_addr_global[physical_addr_idx] =
       logical_addr_idx;
  logical_addr_virt_global[physical_addr_idx]
       = logical_addr_idx;
  physical_addr_global[logical_addr_idx] =
       physical_addr_idx;

  /* copy */
  pos_x_new_global[physical_addr_idx] =
       pos_x_idx;
  pos_y_new_global[physical_addr_idx] =
       pos_y_idx;
  pos_z_new_global[physical_addr_idx] =
       pos_z_idx;

  mom_x_new_global[physical_addr_idx] =
       mom_x_idx;
  mom_y_new_global[physical_addr_idx] =
       mom_y_idx;
  mom_z_new_global[physical_addr_idx] =
       mom_z_idx;

  cell_num_new_global[physical_addr_idx] =
       cell_num;
  addr_in_cell_new_global[physical_addr_idx] =
       addr_in_cell_idx;

  /* mirroring */

  md::float_t _pos_x_idx;
  md::float_t _pos_y_idx;
  md::float_t _pos_z_idx;

  const md::int_t num_cells_x = 1<<
       log_num_cells_x;
  const md::int_t num_cells_y = 1<<
       log_num_cells_y;
  const md::int_t num_cells_z = 1<<
       log_num_cells_z;

  const md::int_t n_x = cell_num & (
       num_cells_x-1);
  const md::int_t n_y = (cell_num >>
       log_num_cells_x) & (num_cells_y-1);
  const md::int_t n_z
    = (cell_num >> (log_num_cells_x+
        log_num_cells_y) ) & (num_cells_z-1);

  const md::float_t periodic_mod_x = (
       num_cells_x-2)*cell_size;
  const md::float_t periodic_mod_y = (
       num_cells_y-2)*cell_size;
  const md::float_t periodic_mod_z = (
       num_cells_z-2)*cell_size;

  for(int mz=0; mz<3; ++mz) {

    bool if_z;
    md::int_t _n_z;

    switch(mz) {

    case 0:
      if(n_z==1) {
        if_z = true;
        _n_z = num_cells_z - 1;
```

```
      _pos_z_idx = pos_z_idx +
          periodic_mod_z;
      break;
    } else {
      continue;
    }

  case 1:
    if_z = false;
    _n_z = n_z;
    _pos_z_idx = pos_z_idx;
    break;
  case 2:
    if(n_z==num_cells_z-2){
      if_z = true;
      _n_z = 0;
      _pos_z_idx = pos_z_idx -
          periodic_mod_z;
      break;
    } else {
      continue;
    }
  }
  for(int my=0; my<3; ++my) {
    md::int_t if_y;
    md::int_t _n_y;
    switch(my) {
    case 0:
      if(n_y==1) {
        if_y = true;
        _n_y = num_cells_y - 1;
        _pos_y_idx = pos_y_idx +
            periodic_mod_y;
        break;
      } else {
        continue;
      }
    case 1:
      if_y = false;
      _n_y = n_y;
      _pos_y_idx = pos_y_idx;
      break;
    case 2:
      if(n_y==num_cells_x-2) {
        if_y = true;
        _n_y = 0;
        _pos_y_idx = pos_y_idx -
            periodic_mod_y;
        break;
      } else {
        continue;
      }
    }
    for(int mx=0; mx<3; ++mx) {
      md::int_t if_x;
      md::int_t _n_x;
      switch(mx) {
      case 0:
        if(n_x==1){
          if_x = true;
          _n_x = num_cells_x - 1;
          _pos_x_idx = pos_x_idx +
              periodic_mod_x;
          break;
        }else{
          continue;
        }
      case 1:
        if_x = false;
        _n_x = n_x;
        _pos_x_idx = pos_x_idx;
        break;
      case 2:
        if(n_x==num_cells_x-2){
          if_x = true;
          _n_x = 0;
          _pos_x_idx = pos_x_idx -
              periodic_mod_x;
          break;
        } else {
          continue;
        }
      }
      if(if_x||if_y||if_z) {
        const md::int_t cell_num_virt = _n_x
            | ( ( _n_y | ( _n_z <<
            log_num_cells_y ) ) <<
            log_num_cells_x );
        const md::int_t cell_begin_virt =
            cell_begin_virt_global[
            cell_num_virt];
        const md::int_t cell_begin_imag =
            cell_begin_imag_global[
            cell_num_virt];
        const md::int_t logical_addr_virt =
            cell_begin_virt +
            addr_in_cell_idx;
        const md::int_t physical_addr_virt =
            cell_begin_imag +
            addr_in_cell_idx;
        pos_x_new_global[physical_addr_virt]
            = _pos_x_idx;
        pos_y_new_global[physical_addr_virt]
            = _pos_y_idx;
        pos_z_new_global[physical_addr_virt]
            = _pos_z_idx;
        cell_num_new_global[
            physical_addr_virt] =
            cell_num_virt;
        logical_addr_virt_global[
            physical_addr_virt] =
            logical_addr_virt;
        physical_addr_global[
            logical_addr_virt] =
            physical_addr_virt;
      } /* end if_x if_y if_z */
    } /* end for mx */
  } /* end for my */
} /* end for mz */

}


__global__ static void update_addr_kernel
(
 const md::float_t cell_size,
 const md::int_t log_num_cells_x,
 const md::int_t log_num_cells_y,
 const md::int_t log_num_cells_z,
 md::float_t* const pos_x_global,
 md::float_t* const pos_y_global,
 md::float_t* const pos_z_global,
 md::int_t* const cell_num_global,
 const md::int_t* const addr_in_cell_global,
 const md::int_t* const cell_begin_virt_global,
 const md::int_t* const cell_begin_imag_global,
 md::int_t* const logical_addr_virt_global,
 md::int_t* const physical_addr_global
 )
{
  const uint idx = blockDim.x * blockIdx.x +
      threadIdx.x;
  const md::int_t cell_num = cell_num_global[
      idx];
  const md::int_t addr_in_cell_idx =
      addr_in_cell_global[idx];
  const md::int_t cell_begin_idx =
      cell_begin_virt_global[cell_num];
  md::float_t pos_x_idx = pos_x_global[idx];
  md::float_t pos_y_idx = pos_y_global[idx];
  md::float_t pos_z_idx = pos_z_global[idx];
  const md::int_t logical_addr_idx =
      cell_begin_idx + addr_in_cell_idx;
  logical_addr_virt_global[idx] =
      logical_addr_idx;
  physical_addr_global[logical_addr_idx] = idx
      ;
  /* mirroring */
  md::float_t _pos_x_idx;
  md::float_t _pos_y_idx;
  md::float_t _pos_z_idx;
  const md::int_t num_cells_x = 1<<
      log_num_cells_x;
  const md::int_t num_cells_y = 1<<
      log_num_cells_y;
  const md::int_t num_cells_z = 1<<
      log_num_cells_z;
  const md::int_t n_x = cell_num & (
      num_cells_x-1);
  const md::int_t n_y = (cell_num >>
      log_num_cells_x) & (num_cells_y-1);
  const md::int_t n_z
    = (cell_num >> (log_num_cells_x+
        log_num_cells_y) ) & (num_cells_z-1);

  const md::float_t periodic_mod_x = (
      num_cells_x-2)*cell_size;
  const md::float_t periodic_mod_y = (
      num_cells_y-2)*cell_size;
  const md::float_t periodic_mod_z = (
      num_cells_z-2)*cell_size;
  for(int mz=0; mz<3; ++mz) {
    bool if_z;
    md::int_t _n_z;
    switch(mz) {
    case 0:
      if(n_z==1) {
        if_z = true;
```

```
      _n_z = num_cells_z - 1;
      _pos_z_idx = pos_z_idx +
          periodic_mod_z;
      break;
    } else {
      continue;
    }

  case 1:
    if_z = false;
    _n_z = n_z;
    _pos_z_idx = pos_z_idx;
    break;
  case 2:
    if(n_z==num_cells_z-2){
      if_z = true;
      _n_z = 0;
      _pos_z_idx = pos_z_idx -
          periodic_mod_z;
      break;
    } else {
      continue;
    }
  }
  for(int my=0; my<3; ++my) {
    md::int_t if_y;
    md::int_t _n_y;
    switch(my) {
    case 0:
      if(n_y==1) {
        if_y = true;
        _n_y = num_cells_y - 1;
        _pos_y_idx = pos_y_idx +
            periodic_mod_y;
        break;
      } else {
        continue;
      }
    case 1:
      if_y = false;
      _n_y = n_y;
      _pos_y_idx = pos_y_idx;
      break;
    case 2:
      if(n_y==num_cells_x-2) {
        if_y = true;
        _n_y = 0;
        _pos_y_idx = pos_y_idx -
            periodic_mod_y;
        break;
      } else {
        continue;
      }
    }
    for(int mx=0; mx<3; ++mx) {
      md::int_t if_x;
      md::int_t _n_x;
      switch(mx) {
      case 0:
        if(n_x==1){
          if_x = true;
          _n_x = num_cells_x - 1;
          _pos_x_idx = pos_x_idx +
              periodic_mod_x;
          break;
        }else{
          continue;
        }
      case 1:
        if_x = false;
        _n_x = n_x;
        _pos_x_idx = pos_x_idx;
        break;
      case 2:
        if(n_x==num_cells_x-2){
          if_x = true;
          _n_x = 0;
          _pos_x_idx = pos_x_idx -
              periodic_mod_x;
          break;
        } else {
          continue;
        }
      }
      if(if_x||if_y||if_z) {
        const md::int_t cell_num_virt = _n_x
            | ( ( _n_y | ( _n_z <<
            log_num_cells_y ) ) <<
            log_num_cells_x );
        const md::int_t cell_begin_virt =
            cell_begin_virt_global[
            cell_num_virt];
        const md::int_t cell_begin_imag =
            cell_begin_imag_global[
            cell_num_virt];
        const md::int_t logical_addr_virt =
            cell_begin_virt +
            addr_in_cell_idx;
        const md::int_t physical_addr_virt =
            cell_begin_imag +
            addr_in_cell_idx;
        pos_x_global[physical_addr_virt] =
            _pos_x_idx;
        pos_y_global[physical_addr_virt] =
            _pos_y_idx;
        pos_z_global[physical_addr_virt] =
            _pos_z_idx;
        cell_num_global[physical_addr_virt]
            = cell_num_virt;
        logical_addr_virt_global[
            physical_addr_virt] =
            logical_addr_virt;
        physical_addr_global[
            logical_addr_virt] =
            physical_addr_virt;
      } /* end if_x if_y if_z */
    } /* end for mx */
  } /* end for my */
} /* end for mz */

}

// todo
const md::float_t r_c = 2.5;
const md::float_t r_c_pow2 = r_c*r_c;
const md::float_t r_c_rpow2 = 1/r_c_pow2;
const md::float_t r_c_rpow4 = r_c_rpow2 *
    r_c_rpow2;
const md::float_t r_c_rpow6 = r_c_rpow4 *
    r_c_rpow2;
const md::float_t r_c_rpow8 = r_c_rpow4 *
    r_c_rpow4;
const md::float_t r_c_rpow12 = r_c_rpow6 *
    r_c_rpow6;
const md::float_t r_c_rpow14 = r_c_rpow8 *
    r_c_rpow6;
const md::float_t c_1 = 4 * ( -12 * r_c_rpow14
    + 6 * r_c_rpow8 );
const md::float_t c_0 = 4 * ( r_c_rpow12 -
    r_c_rpow6 ) - 0.5 * c_1 * r_c_pow2;

const md::float_t dr_s = 0.5;
const md::float_t r_r = r_c + dr_s;
__device__ static void force_lj_device
(
 const md::float_t pos_x_i,
 const md::float_t pos_y_i,
 const md::float_t pos_z_i,
 const md::float_t pos_x_j,
 const md::float_t pos_y_j,
 const md::float_t pos_z_j,
 md::float_t* const pf_x_i,
 md::float_t* const pf_y_i,
 md::float_t* const pf_z_i
 )
{
  md::float_t& f_x_i = *pf_x_i;
  md::float_t& f_y_i = *pf_y_i;
  md::float_t& f_z_i = *pf_z_i;

  const md::float_t delta_x = pos_x_i -
      pos_x_j;
  const md::float_t delta_y = pos_y_i -
      pos_y_j;
  const md::float_t delta_z = pos_z_i -
      pos_z_j;

  const md::float_t r_pow2 = delta_x*delta_x +
      delta_y*delta_y + delta_z*delta_z;

  if(r_pow2 < r_c_pow2) {

    const md::float_t r_rpow2 = 1.0/r_pow2;
    const md::float_t r_rpow4 = r_rpow2*
        r_rpow2;
    const md::float_t r_rpow6 = r_rpow4*
        r_rpow2;

    const md::float_t r_rpow8 = r_rpow4*
        r_rpow4;
    const md::float_t r_rpow14 = r_rpow8*
        r_rpow6;

    const md::float_t k_lj = -4 * ( -12 *
        r_rpow14 + 6 * r_rpow8 ) + c_1;

    f_x_i += k_lj * delta_x;
    f_y_i += k_lj * delta_y;
    f_z_i += k_lj * delta_z;

  }
}

__device__ static md::float_t e_pot_lj_device
(
 const md::float_t pos_x_i,
```

```
 const md::float_t pos_y_i,
 const md::float_t pos_z_i,
 const md::float_t pos_x_j,
 const md::float_t pos_y_j,
 const md::float_t pos_z_j
 )
{
  const md::float_t delta_x = pos_x_i -
       pos_x_j;
  const md::float_t delta_y = pos_y_i -
       pos_y_j;
  const md::float_t delta_z = pos_z_i -
       pos_z_j;

  const md::float_t r_pow2 = delta_x*delta_x +
       delta_y*delta_y + delta_z*delta_z;

  if(r_pow2 < r_c_pow2) {

    const md::float_t r_rpow2 = 1.0/r_pow2;
    const md::float_t r_rpow4 = r_rpow2*
         r_rpow2;
    const md::float_t r_rpow6 = r_rpow4*
         r_rpow2;
    const md::float_t r_rpow12 = r_rpow6 *
         r_rpow6;

    const md::float_t e = 4 * ( r_rpow12 -
         r_rpow6 ) - 0.5 * c_1 * r_pow2 - c_0;
    return e;

  } else {

    return 0;

  }
}

__global__ static void calc_force_kernel
(
 const md::int_t log_num_cells_x,
 const md::int_t log_num_cells_y,
 const md::int_t log_num_cells_z,
 const md::float_t* const pos_x_global,
 const md::float_t* const pos_y_global,
 const md::float_t* const pos_z_global,
 md::float_t* const f_x_global,
 md::float_t* const f_y_global,
 md::float_t* const f_z_global,
 md::int_t* const cell_num_global,
 md::int_t* const cell_begin_virt_global,
 md::int_t* const physical_addr_global
 )
{
  const uint idx =  blockDim.x * blockIdx.x +
       threadIdx.x;

  const md::int_t num_cells_x = 1<<
       log_num_cells_x;
  const md::int_t num_cells_y = 1<<
       log_num_cells_y;
  const md::int_t num_cells_z = 1<<
       log_num_cells_z;

  const md::float_t pos_x_i = pos_x_global[idx
       ];
  const md::float_t pos_y_i = pos_y_global[idx
       ];
  const md::float_t pos_z_i = pos_z_global[idx
       ];

  md::float_t f_x_i = 0;
  md::float_t f_y_i = 0;
  md::float_t f_z_i = 0;

  const md::int_t cell_num = cell_num_global[
       idx];

  const md::int_t n_x = cell_num & (
       num_cells_x-1);
  const md::int_t n_y = (cell_num >>
       log_num_cells_x) & (num_cells_y-1);
  const md::int_t n_z
    = (cell_num >> (log_num_cells_x+
         log_num_cells_y) ) & (num_cells_z-1);

  for(int dz = -1; dz<=1; ++dz) {

    const md::int_t m_z = n_z + dz;

    for(int dy = -1; dy<=1; ++dy) {

      const md::int_t m_y = n_y + dy;

      for(int dx = -1; dx<=1; ++dx) {

        const md::int_t m_x = n_x + dx;

        const md::int_t cell_num_m = m_x | ( (
             m_y | ( m_z << log_num_cells_y )
             ) << log_num_cells_x );

        const md::int_t cell_begin_m =
             cell_begin_virt_global[cell_num_m
             ];
        const md::int_t cell_end_m =
             cell_begin_virt_global[cell_num_m
             +1];

        if(dx|dy|dz) {

          // software pipelining
          int j0;
          int j1 = physical_addr_global[
               cell_begin_m];

          for(int _j=cell_begin_m; _j<
               cell_end_m; _j++) {

            j0 = j1;
            j1 = physical_addr_global[_j+1];

            const md::float_t pos_x_j0 =
                 pos_x_global[j0];
            const md::float_t pos_y_j0 =
                 pos_y_global[j0];
            const md::float_t pos_z_j0 =
                 pos_z_global[j0];

            force_lj_device(pos_x_i, pos_y_i,
                 pos_z_i,
                           pos_x_j0, pos_y_j0
                                , pos_z_j0,
                           &f_x_i, &f_y_i, &
                                f_z_i);

          }
        } else {

          // software pipelining
          int j0;
          int j1 = physical_addr_global[
               cell_begin_m];

          for(int _j=cell_begin_m; _j<
               cell_end_m; ++_j) {

            j0 = j1;
            j1 = physical_addr_global[_j+1];

            if(j0==idx) continue;

            const md::float_t pos_x_j =
                 pos_x_global[j0];
            const md::float_t pos_y_j =
                 pos_y_global[j0];
            const md::float_t pos_z_j =
                 pos_z_global[j0];

            force_lj_device(pos_x_i, pos_y_i,
                 pos_z_i,
                           pos_x_j, pos_y_j,
                                pos_z_j,
                           &f_x_i, &f_y_i, &
                                f_z_i);

          }
        }
      }
    }
  }

  f_x_global[idx] = f_x_i;
  f_y_global[idx] = f_y_i;
  f_z_global[idx] = f_z_i;
}

__global__ static void calc_energy_kernel
(
 const md::int_t log_num_cells_x,
 const md::int_t log_num_cells_y,
 const md::int_t log_num_cells_z,
 const md::float_t* const pos_x_global,
 const md::float_t* const pos_y_global,
 const md::float_t* const pos_z_global,
 const md::float_t* const mom_x_global,
 const md::float_t* const mom_y_global,
 const md::float_t* const mom_z_global,
 md::float_t* const e_kin_global,
 md::float_t* const e_pot_global,
 md::int_t* const cell_num_global,
 md::int_t* const cell_begin_virt_global,
 md::int_t* const physical_addr_global
 )
{
  const uint idx =  blockDim.x * blockIdx.x +
       threadIdx.x;

  const md::int_t num_cells_x = 1<<
       log_num_cells_x;
  const md::int_t num_cells_y = 1<<
       log_num_cells_y;
  const md::int_t num_cells_z = 1<<
       log_num_cells_z;

  const md::float_t mom_x_i = mom_x_global[idx
       ];
  const md::float_t mom_y_i = mom_y_global[idx
       ];
  const md::float_t mom_z_i = mom_z_global[idx
       ];

  const md::float_t pos_x_i = pos_x_global[idx
       ];
  const md::float_t pos_y_i = pos_y_global[idx
       ];
  const md::float_t pos_z_i = pos_z_global[idx
       ];
```

```
  md::float_t e_pot_i = 0;
  const md::int_t cell_num = cell_num_global[
      idx];

  const md::float_t e_kin_i = 0.5
    * (
      mom_x_i * mom_x_i +
      mom_y_i * mom_y_i +
      mom_z_i * mom_z_i
      );

  e_kin_global[idx] = e_kin_i;

  const md::int_t n_x = cell_num & (
      num_cells_x-1);
  const md::int_t n_y = (cell_num >>
      log_num_cells_x) & (num_cells_y-1);
  const md::int_t n_z
    = (cell_num >> (log_num_cells_x+
        log_num_cells_y) ) & (num_cells_z-1);

  for(int dz = -1; dz<=1; ++dz) {
    const md::int_t m_z = n_z + dz;
    for(int dy = -1; dy<=1; ++dy) {
      const md::int_t m_y = n_y + dy;
      for(int dx = -1; dx<=1; ++dx) {
        const md::int_t m_x = n_x + dx;

        const md::int_t cell_num_m = m_x | ( (
            m_y | ( m_z << log_num_cells_y )
            ) << log_num_cells_x );

        const md::int_t cell_begin_m =
            cell_begin_virt_global[cell_num_m
            ];
        const md::int_t cell_end_m =
            cell_begin_virt_global[cell_num_m
            +1];

        for(int _j=cell_begin_m; _j<cell_end_m
            ; ++_j) {

          int j = physical_addr_global[_j];

          if(cell_num_m==cell_num && j==idx)
              continue;
          // if(j>=idx) break;

          const md::float_t pos_x_j =
              pos_x_global[j];
          const md::float_t pos_y_j =
              pos_y_global[j];
          const md::float_t pos_z_j =
              pos_z_global[j];

          e_pot_i += e_pot_lj_device(pos_x_i,
              pos_y_i, pos_z_i,
                        pos_x_j, pos_y_j,
                            pos_z_j);

        }
      }
    }
  }

  e_pot_global[idx] = e_pot_i;
}

__global__ static void update_particle_kernel
(
 const md::float_t cell_size,
 const md::int_t log_num_cells_x,
 const md::int_t log_num_cells_y,
 const md::int_t log_num_cells_z,
 md::float_t* const pos_x_global,
 md::float_t* const pos_y_global,
 md::float_t* const pos_z_global,
 md::float_t* const mom_x_global,
 md::float_t* const mom_y_global,
 md::float_t* const mom_z_global,
 const md::float_t* const f_x_global,
 const md::float_t* const f_y_global,
 const md::float_t* const f_z_global,
 const md::int_t* const cell_num_global,
 const md::int_t* const cell_begin_imag_global
     ,
 const md::int_t* const addr_in_cell_global,
 const md::float_t dt,
 md::float_t* const vel_pow2_global
 )
{
  extern __shared__ md::float_t
      vel_pow2_shared[];

  const uint idx =  blockDim.x * blockIdx.x +
      threadIdx.x;

  const md::int_t addr_in_cell_idx =
      addr_in_cell_global[idx];

  const md::float_t f_x_i = f_x_global[idx];
  const md::float_t f_y_i = f_y_global[idx];
  const md::float_t f_z_i = f_z_global[idx];

  md::float_t mom_x_i = mom_x_global[idx];
  md::float_t mom_y_i = mom_y_global[idx];
  md::float_t mom_z_i = mom_z_global[idx];

  md::float_t pos_x_i = pos_x_global[idx];
  md::float_t pos_y_i = pos_y_global[idx];
  md::float_t pos_z_i = pos_z_global[idx];

  const md::int_t cell_num = cell_num_global[
      idx];

  mom_x_i += f_x_i * dt;
  mom_y_i += f_y_i * dt;
  mom_z_i += f_z_i * dt;

  mom_x_global[idx] = mom_x_i;
  mom_y_global[idx] = mom_y_i;
  mom_z_global[idx] = mom_z_i;

  const md::float_t vel_pow2_i
    = mom_x_i*mom_x_i + mom_y_i*mom_y_i +
        mom_z_i*mom_z_i;

  vel_pow2_shared[threadIdx.x] = vel_pow2_i;

  pos_x_i += mom_x_i * dt;
  pos_y_i += mom_y_i * dt;
  pos_z_i += mom_z_i * dt;

  pos_x_global[idx] = pos_x_i;
  pos_y_global[idx] = pos_y_i;
  pos_z_global[idx] = pos_z_i;

  mylib::func::reduce<md::float_t, mylib::max
      >(vel_pow2_shared);
  if(threadIdx.x==0) {
    vel_pow2_global[blockIdx.x] =
        vel_pow2_shared[0];
  }

  /* mirroring */

  md::float_t _pos_x_i;
  md::float_t _pos_y_i;
  md::float_t _pos_z_i;

  const md::int_t num_cells_x = 1<<
      log_num_cells_x;
  const md::int_t num_cells_y = 1<<
      log_num_cells_y;
  const md::int_t num_cells_z = 1<<
      log_num_cells_z;

  const md::int_t n_x = cell_num & (
      num_cells_x-1);
  const md::int_t n_y = (cell_num >>
      log_num_cells_x) & (num_cells_y-1);
  const md::int_t n_z
    = (cell_num >> (log_num_cells_x+
        log_num_cells_y) ) & (num_cells_z-1);

  const md::float_t periodic_mod_x = (
      num_cells_x-2)*cell_size;
  const md::float_t periodic_mod_y = (
      num_cells_y-2)*cell_size;
  const md::float_t periodic_mod_z = (
      num_cells_z-2)*cell_size;

  for(int mz=0; mz<3; ++mz) {

    bool if_z;
    md::int_t _n_z;

    switch(mz) {

    case 0:
      if(n_z==1) {
        if_z = true;
        _n_z = num_cells_z - 1;
        _pos_z_i = pos_z_i + periodic_mod_z;
        break;
      } else {
        continue;
      }

    case 1:
      if_z = false;
      _n_z = n_z;
      _pos_z_i = pos_z_i;
      break;

    case 2:
      if(n_z==num_cells_z-2){
        if_z = true;
        _n_z = 0;
        _pos_z_i = pos_z_i - periodic_mod_z;
        break;
      } else {
        continue;
      }
    }

    for(int my=0; my<3; ++my) {

      md::int_t if_y;
      md::int_t _n_y;
      switch(my) {

      case 0:
        if(n_y==1) {
          if_y = true;
```

```
        _n_y = num_cells_y - 1;
        _pos_y_i = pos_y_i + periodic_mod_y;
        break;
      } else {
        continue;
      }

    case 1:
      if_y = false;
      _n_y = n_y;
      _pos_y_i = pos_y_i;
      break;

    case 2:
      if(n_y==num_cells_x-2) {
        if_y = true;
        _n_y = 0;
        _pos_y_i = pos_y_i - periodic_mod_y;
        break;
      } else {
        continue;
      }
    }

    for(int mx=0; mx<3; ++mx) {

      md::int_t if_x;
      md::int_t _n_x;
      switch(mx) {

      case 0:
        if(n_x==1){
          if_x = true;
          _n_x = num_cells_x - 1;
          _pos_x_i = pos_x_i +
                periodic_mod_x;
          break;
        }else{
          continue;
        }

      case 1:
        if_x = false;
        _n_x = n_x;
        _pos_x_i = pos_x_i;
        break;

      case 2:
        if(n_x==num_cells_x-2){
          if_x = true;
          _n_x = 0;
          _pos_x_i = pos_x_i -
                periodic_mod_x;
          break;
        } else {
          continue;
        }
      }

      if(if_x||if_y||if_z) {
        const md::int_t cell_num_virt = _n_x
              | ( ( _n_y | ( _n_z <<
              log_num_cells_y ) ) <<
              log_num_cells_x );

        const md::int_t cell_begin_imag =
              cell_begin_imag_global[
              cell_num_virt];

        const md::int_t physical_addr_virt =
               cell_begin_imag +
              addr_in_cell_idx;

        pos_x_global[physical_addr_virt] =
              _pos_x_i;
        pos_y_global[physical_addr_virt] =
              _pos_y_i;
        pos_z_global[physical_addr_virt] =
              _pos_z_i;

      } /* end if_x if_y if_z */

    } /* end for mx */

  } /* end for my */

 } /* end for mz */

 /* end mirroring */

}


int main(const int argc, char* const argv[])
{
  std::vector<md::int_t> block_size_array;
  block_size_array.push_back(256);
  block_size_array.push_back(256);

  const md::int_t num_reduction_steps =
        block_size_array.size();

  std::vector<md::int_t> num_blocks_array(
        num_reduction_steps);
  md::int_t num_particles;
  md::int_t reduction_wmem_size = 0;

  {
    md::int_t num_blocks_i = 1;
    for(int i=num_reduction_steps-1; i>=0; --i
        ) {
      num_blocks_array[i] = num_blocks_i;
      reduction_wmem_size += num_blocks_i;
      num_blocks_i *= block_size_array[i];
    }
    num_particles = num_blocks_i;
  }

  const md::int_t num_particles_virt = 2 *
        num_particles;

  const md::int_t log_num_cells_1 = 5;
  const md::int_t log_num_cells_x =
        log_num_cells_1;
  const md::int_t log_num_cells_y =
        log_num_cells_1;
  const md::int_t log_num_cells_z =
        log_num_cells_1;

  const md::int_t num_cells_1 = 1<<
        log_num_cells_1;
  const md::int_t num_cells_x = 1<<
        log_num_cells_x;
  const md::int_t num_cells_y = 1<<
        log_num_cells_y;
  const md::int_t num_cells_z = 1<<
        log_num_cells_z;

  const md::int_t num_cells = num_cells_x *
        num_cells_y * num_cells_z;

  const md::float_t cell_size = r_r;

  const md::float_t L = cell_size *
        num_cells_1;
  const md::float_t L_x = cell_size *
        num_cells_x;
  const md::float_t L_y = cell_size *
        num_cells_y;
  const md::float_t L_z = cell_size *
        num_cells_z;
  fprintf(stderr, "[info] L = %g\n", L);

  const md::float_t dt = 0.0001;

  md::float_t rem_dist = -1;

  signal(SIGINT, sigint_catch);

  thrust::host_vector<md::float_t> pos_x_host(
        num_particles);
  thrust::host_vector<md::float_t> pos_y_host(
        num_particles);
  thrust::host_vector<md::float_t> pos_z_host(
        num_particles);

  thrust::device_vector<md::float_t>
        pos_x_0_device(num_particles_virt);
  thrust::device_vector<md::float_t>
        pos_y_0_device(num_particles_virt);
  thrust::device_vector<md::float_t>
        pos_z_0_device(num_particles_virt);

  thrust::device_vector<md::float_t>
        mom_x_0_device(num_particles);
  thrust::device_vector<md::float_t>
        mom_y_0_device(num_particles);
  thrust::device_vector<md::float_t>
        mom_z_0_device(num_particles);

  thrust::device_vector<md::int_t>
        cell_num_0_device(num_particles_virt);
  thrust::device_vector<md::int_t>
        addr_in_cell_0_device(num_particles);

  thrust::device_ptr<md::float_t> pos_x_device
        = pos_x_0_device.data();
  thrust::device_ptr<md::float_t> pos_y_device
        = pos_y_0_device.data();
  thrust::device_ptr<md::float_t> pos_z_device
        = pos_z_0_device.data();

  thrust::device_ptr<md::float_t> mom_x_device
        = mom_x_0_device.data();
  thrust::device_ptr<md::float_t> mom_y_device
        = mom_y_0_device.data();
  thrust::device_ptr<md::float_t> mom_z_device
        = mom_z_0_device.data();

  thrust::device_ptr<md::int_t>
        cell_num_device = cell_num_0_device.
        data();
  thrust::device_ptr<md::int_t>
        addr_in_cell_device =
        addr_in_cell_0_device.data();

  thrust::device_vector<md::float_t>
        pos_x_1_device(num_particles_virt);
  thrust::device_vector<md::float_t>
        pos_y_1_device(num_particles_virt);
  thrust::device_vector<md::float_t>
        pos_z_1_device(num_particles_virt);

  thrust::device_vector<md::float_t>
        mom_x_1_device(num_particles);
  thrust::device_vector<md::float_t>
        mom_y_1_device(num_particles);
  thrust::device_vector<md::float_t>
        mom_z_1_device(num_particles);

  thrust::device_vector<md::int_t>
        cell_num_1_device(num_particles_virt);
```

```
thrust::device_vector<md::int_t>
      addr_in_cell_1_device(num_particles);

thrust::device_ptr<md::float_t>
      pos_x_sub_device = pos_x_1_device.data
      ();
thrust::device_ptr<md::float_t>
      pos_y_sub_device = pos_y_1_device.data
      ();
thrust::device_ptr<md::float_t>
      pos_z_sub_device = pos_z_1_device.data
      ();

thrust::device_ptr<md::float_t>
      mom_x_sub_device = mom_x_1_device.data
      ();
thrust::device_ptr<md::float_t>
      mom_y_sub_device = mom_y_1_device.data
      ();
thrust::device_ptr<md::float_t>
      mom_z_sub_device = mom_z_1_device.data
      ();

thrust::device_ptr<md::int_t>
      cell_num_sub_device = cell_num_1_device
      .data();
thrust::device_ptr<md::int_t>
      addr_in_cell_sub_device =
      addr_in_cell_1_device.data();

thrust::device_vector<md::float_t>
      e_kin_device(num_particles);
thrust::device_vector<md::float_t>
      e_pot_device(num_particles);

thrust::device_vector<md::float_t>
      vel_pow2_device(reduction_wmem_size);

thrust::device_vector<md::float_t>
      f_x_device(num_particles);
thrust::device_vector<md::float_t>
      f_y_device(num_particles);
thrust::device_vector<md::float_t>
      f_z_device(num_particles);

thrust::device_vector<md::int_t>
      num_particles_cell_device(num_cells+1);
thrust::device_vector<md::int_t>
      num_particles_cell_virt_device(
      num_cells+1);
thrust::device_vector<md::int_t>
      num_particles_cell_imag_device(
      num_cells+1);

thrust::device_vector<md::int_t>
      cell_begin_device(num_cells+1);
thrust::device_vector<md::int_t>
      cell_begin_virt_device(num_cells+1);
thrust::device_vector<md::int_t>
      cell_begin_imag_device(num_cells+1);

thrust::device_vector<md::int_t>
      logical_addr_virt_device(
      num_particles_virt);
thrust::device_vector<md::int_t>
      physical_addr_device(num_particles_virt
      );

const md::float_t particle_interval = 1.5;
const md::int_t num_particles_line = floor((
      L-2.5*cell_size) / particle_interval) ;

init_kernel<<<num_blocks_array[0],
      block_size_array[0]>>>
  (particle_interval, num_particles_line,
    cell_size * 1.5,
    cell_size * 1.5,
    cell_size * 1.5,
    pos_x_device.get(),
    pos_y_device.get(),
    pos_z_device.get(),
    mom_x_device.get(),
    mom_y_device.get(),
    mom_z_device.get(),
    f_x_device.data().get(),
    f_y_device.data().get(),
    f_z_device.data().get()
    );

CHECK_CUDA_KERNEL_ERROR();

const int output_interval = 1000;
const int num_output = -1;

md::float_t md_time = 0;

cudaEvent_t start_ev, end_ev;
cudaEventCreate(&start_ev);
cudaEventCreate(&end_ev);

cudaEventRecord(start_ev, 0);

for(int o=0; o!=num_output; ++o) {
  for(int i=0; i!=output_interval; ++i) {
    if(rem_dist<0) {

      const bool if_sort = true;

fprintf(stderr, "[info]_rebuilding_
      cell_structure_md_time=%.5g_o=%d_
      i=%d\n", md_time, o, i);

thrust::fill
  (num_particles_cell_device.begin(),
   num_particles_cell_device.end(),
   0.0);

thrust::fill
  (num_particles_cell_virt_device.
       begin(),
   num_particles_cell_virt_device.end
       (),
   0.0);

thrust::fill
  (num_particles_cell_imag_device.
       begin(),
   num_particles_cell_imag_device.end
       (),
   0.0);

detect_cell_num_kernel
  <<<num_blocks_array[0],
       block_size_array[0]>>>
  (cell_size, 1.0/cell_size,
   log_num_cells_x, log_num_cells_y,
       log_num_cells_z,
   pos_x_device.get(),
   pos_y_device.get(),
   pos_z_device.get(),
   cell_num_device.get(),
   num_particles_cell_device.data().
       get(),
   num_particles_cell_virt_device.data
       ().get(),
   num_particles_cell_imag_device.data
       ().get(),
   addr_in_cell_device.get()
   );

CHECK_CUDA_KERNEL_ERROR();

if(if_sort) {

  thrust::exclusive_scan
    (num_particles_cell_device.begin()
       ,
     num_particles_cell_device.end(),
     cell_begin_device.begin()
     );
}
thrust::exclusive_scan
  (num_particles_cell_virt_device.
       begin(),
   num_particles_cell_virt_device.end
       (),
   cell_begin_virt_device.begin()
   );

thrust::exclusive_scan
  (num_particles_cell_imag_device.
       begin(),
   num_particles_cell_imag_device.end
       (),
   cell_begin_imag_device.begin(),
   num_particles
   );

if(if_sort) {

  sort_and_update_addr_kernel
    <<<num_blocks_array[0],
         block_size_array[0]>>>
    (
     cell_size,
     log_num_cells_x, log_num_cells_y,
         log_num_cells_z,

     pos_x_device.get(),
     pos_y_device.get(),
     pos_z_device.get(),
     mom_x_device.get(),
     mom_y_device.get(),
     mom_z_device.get(),
     cell_num_device.get(),
     addr_in_cell_device.get(),

     pos_x_sub_device.get(),
     pos_y_sub_device.get(),
     pos_z_sub_device.get(),
     mom_x_sub_device.get(),
     mom_y_sub_device.get(),
     mom_z_sub_device.get(),
     cell_num_sub_device.get(),
     addr_in_cell_sub_device.get(),

     cell_begin_device.data().get(),
     cell_begin_virt_device.data().get
         (),
     cell_begin_imag_device.data().get
         (),
```

```
        logical_addr_virt_device.data().
            get(),
        physical_addr_device.data().get()
        );
    CHECK_CUDA_KERNEL_ERROR();
    std::swap(pos_x_device,
        pos_x_sub_device);
    std::swap(pos_y_device,
        pos_y_sub_device);
    std::swap(pos_z_device,
        pos_z_sub_device);
    std::swap(mom_x_device,
        mom_x_sub_device);
    std::swap(mom_y_device,
        mom_y_sub_device);
    std::swap(mom_z_device,
        mom_z_sub_device);
    std::swap(cell_num_device,
        cell_num_sub_device);
    std::swap(addr_in_cell_device,
        addr_in_cell_sub_device);
  } else {
    update_addr_kernel<<<
        num_blocks_array[0],
        block_size_array[0]>>>
      (
        cell_size,
        log_num_cells_x, log_num_cells_y,
            log_num_cells_z,
        pos_x_device.get(),
        pos_y_device.get(),
        pos_z_device.get(),
        cell_num_device.get(),
        addr_in_cell_device.get(),
        cell_begin_virt_device.data().get
            (),
        cell_begin_imag_device.data().get
            (),
        logical_addr_virt_device.data().
            get(),
        physical_addr_device.data().get()
        );
    CHECK_CUDA_KERNEL_ERROR();
  }
  rem_dist = dr_s * 0.5;
}
if(i==0 || sigint_status>1) {
  cudaEventRecord(end_ev, 0);
  cudaEventSynchronize(end_ev);
  float cudatime;
  cudaEventElapsedTime(&cudatime,
      start_ev ,end_ev);
  const double mups = num_particles *
    ( (i==0)? output_interval : i )
    * 1e-3 / cudatime;
  // output energy
  calc_energy_kernel<<<num_blocks_array
      [0], block_size_array[0]>>>
    (log_num_cells_x, log_num_cells_y,
        log_num_cells_z,
      pos_x_device.get(),
      pos_y_device.get(),
      pos_z_device.get(),
      mom_x_device.get(),
      mom_y_device.get(),
      mom_z_device.get(),
      e_kin_device.data().get(),
      e_pot_device.data().get(),
      cell_num_device.get(),
      cell_begin_virt_device.data().get()
          ,
      physical_addr_device.data().get() )
          ;
  CHECK_CUDA_KERNEL_ERROR();
  const md::float_t e_kin_total = thrust
      ::reduce(e_kin_device.begin(),
      e_kin_device.end());
  const md::float_t temperature =
      (2.0/3.0) * e_kin_total /
      num_particles ;
  const md::float_t e_pot_total = thrust
      ::reduce(e_pot_device.begin(),
      e_pot_device.end()) * 0.5;
  const md::float_t e_pot_density =
      e_pot_total / num_particles;
  const md::float_t e_total =
      e_kin_total + e_pot_total;
  const md::float_t e_density = e_total
      / num_particles;
  fprintf(stdout, "%.5g_%.17g_%.17g_%.17
      g_%g\n", md_time, temperature,
      e_pot_density, e_density, mups);
  fflush(stdout);
  cudaEventRecord(start_ev, 0);
  // sigint break
  if(sigint_status) {
    ++sigint_status;
    break;
  }
}
calc_force_kernel
  <<<num_blocks_array[0],
      block_size_array[0]>>>
  (log_num_cells_x, log_num_cells_y,
      log_num_cells_z,
    pos_x_device.get(),
    pos_y_device.get(),
    pos_z_device.get(),
    f_x_device.data().get(),
    f_y_device.data().get(),
    f_z_device.data().get(),
    cell_num_device.get(),
    cell_begin_virt_device.data().get(),
    physical_addr_device.data().get()
    );
CHECK_CUDA_KERNEL_ERROR();
update_particle_kernel
  <<<num_blocks_array[0],
      block_size_array[0],
  block_size_array[0] * sizeof(md::
      float_t)>>>
  (
    cell_size,
    log_num_cells_x, log_num_cells_y,
        log_num_cells_z,
    pos_x_device.get(),
    pos_y_device.get(),
    pos_z_device.get(),
    mom_x_device.get(),
    mom_y_device.get(),
    mom_z_device.get(),
    f_x_device.data().get(),
    f_y_device.data().get(),
    f_z_device.data().get(),
    cell_num_device.get(),
    cell_begin_imag_device.data().get(),
    addr_in_cell_device.get(),
    dt,
    vel_pow2_device.data().get()
    );
CHECK_CUDA_KERNEL_ERROR();
md::float_t vel_pow2_max;
{
  thrust::device_ptr<md::float_t>
      vel_pow2_i_device
    = vel_pow2_device.data();
  for(int i=1; i<num_reduction_steps; ++
      i) {
    thrust::device_ptr<md::float_t>
        vel_pow2_i1_device
      = vel_pow2_i_device +
          num_blocks_array[i-1];
    mylib::kernel::reduce<md::float_t,
        mylib::max>
      <<<num_blocks_array[i],
          block_size_array[i],
      block_size_array[i]*sizeof(md::
          float_t)>>>
      (vel_pow2_i_device.get(),
          vel_pow2_i1_device.get());
    CHECK_CUDA_KERNEL_ERROR();
    vel_pow2_i_device =
        vel_pow2_i1_device;
  }
  vel_pow2_max = vel_pow2_i_device[0];
}
const md::float_t dr_max = sqrt(
    vel_pow2_max) * dt;
rem_dist -= dr_max;
md_time += dt;
}
if(sigint_status>1) break;
} /* end for num output */
```

```
  cudaEventDestroy(start_ev);
  cudaEventDestroy(end_ev);
```

```
}
```

## 5.2 Implementation Code for HMF Simulation

Only the important part is shown. Template library thrust [50] is used in the code.

```
/* ---- reduction library ---- */
template<bool if_sync, typename T1, typename
    T2>
__device__ int reduce_half_func
(
  int active_threads,
  T1* const sum1_shared,
  T1* const sum2_shared
)
{
  T1 sum1_self = sum1_shared[threadIdx.x];
  T2 sum2_self = sum2_shared[threadIdx.x];
  const int half_active_threads =
      active_threads >> 1;
  active_threads = (active_threads+1) >> 1;
  if(if_sync) { __syncthreads(); }
  // __syncthreads();
  if(threadIdx.x < half_active_threads){
    T1 sum1_read = sum1_shared[threadIdx.x+
        active_threads];
    T2 sum2_read = sum2_shared[threadIdx.x+
        active_threads];
    sum1_shared[threadIdx.x] = sum1_self +
        sum1_read;
    sum2_shared[threadIdx.x] = sum2_self +
        sum2_read;
  }
  return active_threads;
}

template<typename T1, typename T2>
__device__ void reduce_per_block_func
(T1* const sum1_shared, T2* const sum2_shared)
{
  int active_threads = blockDim.x;
  //for(; active_threads > 32 ;) {
  while (active_threads > 32) {
    active_threads = reduce_half_func<true, T1
        , T2>
      (active_threads, sum1_shared,
          sum2_shared);
  }
  active_threads = reduce_half_func<false, T1,
      T2>
    (active_threads, sum1_shared, sum2_shared)
        ;
  active_threads = reduce_half_func<false, T1,
      T2>
    (active_threads, sum1_shared, sum2_shared)
        ;
  active_threads = reduce_half_func<false, T1,
      T2>
    (active_threads, sum1_shared, sum2_shared)
        ;
  active_threads = reduce_half_func<false, T1,
      T2>
    (active_threads, sum1_shared, sum2_shared)
        ;
  active_threads = reduce_half_func<false, T1,
      T2>
    (active_threads, sum1_shared, sum2_shared)
        ;
}

template<typename T1, typename I1, typename O1
    , typename T2, typename I2, typename O2>
__global__ void reduce_per_block_kernel
(I1 input1, I2 input2, O1 output1, O2 output2)
{
  extern __shared__ char shared[];
  T1* const sum1_shared = (T1*)shared;
  T2* const sum2_shared = (T2*)&shared[
      blockDim.x*sizeof(T1)];
  const int idx = blockDim.x * blockIdx.x +
      threadIdx.x;
  sum1_shared[threadIdx.x] = input1[idx];
  sum2_shared[threadIdx.x] = input2[idx];
  reduce_per_block_func<T1, T2>(sum1_shared,
      sum2_shared);
  if(threadIdx.x==0) {
    output1[blockIdx.x] = sum1_shared[0];
    output2[blockIdx.x] = sum2_shared[0];
  }
}

template<typename T1, typename I1, typename O1
    , typename T2, typename I2, typename O2>
void reduce_per_block
(
  const size_t num_blocks, const size_t
      block_size,
  const I1 input1, const I2 input2,
  const O1 output1, const O2 output2
)
{
  reduce_per_block_kernel<T1, I1, O1, T2, I2,
      O2>
    <<<num_blocks, block_size,
    (sizeof(T1)+sizeof(T2))*block_size>>>
    (input1, input2, output1, output2);
  DEVICE_CHECK_ERROR_KERNEL();
}

template<typename T1, typename I1, typename O1
    , typename T2, typename I2, typename O2
    >
void reduce_impl(
  const size_t num_degree,
  const size_t* const num_blocks_vec,
  const size_t* const block_size_vec,
  const I1 input1, const I2 input2,
  const O1 wmem01, const O2 wmem02,
  const O1 wmem11, const O2 wmem12,
  const O1 output1, const O2 output2
)
{
  reduce_per_block<T1, I1, O1, T2, I2, O2>
    (num_blocks_vec[0], block_size_vec[0],
      input1, input2,
      ( (1==num_degree)? output1: wmem01 ),
      ( (1==num_degree)? output2: wmem02 )
    );
  for(size_t d=1; d<num_degree; ++d) {
    reduce_per_block<T1, O1, O1, T2, O2, O2>(
      num_blocks_vec[d], block_size_vec[d],
      ( (d&1)? wmem01 : wmem11 ),
      ( (d&1)? wmem02 : wmem12 ),
      ( (d==num_degree-1)? output1: (d&1)?
          wmem11 : wmem01 ),
      ( (d==num_degree-1)? output2: (d&1)?
          wmem12 : wmem02 )
    );
  }
  // LOG_PRINTF("called\n");
}

template<typename I1, typename O1, typename I2
    , typename O2>
void reduce
(
  const size_t num_degree,
```

```
  const size_t* const num_blocks_vec,
  const size_t* const block_size_vec,
  const I1 input1, const I2 input2,
  const O1 wmem01, const O2 wmem02,
  const O1 wmem11, const O2 wmem12,
  const O1 output1, const O2 output2
)
{
  reduce_impl
    <typename metafunc::deref_type<O1>::type,
        I1, O1,
     typename metafunc::deref_type<O2>::type,
        I2, O2>
    (num_degree, num_blocks_vec,
        block_size_vec,
      input1, input2,
      wmem01, wmem02,
      wmem11, wmem12,
      output1, output2);
}

/* ---- update kernel ---- */
__global__ void advance_state_kernel
(
  md::float_t num_particles_inv,
  md::float_t dt_c,
  md::float_t dt_d,
  md::float_t* theta_global,
  md::float_t* omega_global,
  md::float_t* pos_x_global,
  md::float_t* pos_y_global,
  md::float_t* mag_x_global,
  md::float_t* mag_y_global,
  md::float_t* wmem_mag_x_global,
  md::float_t* wmem_mag_y_global
)
{
  extern __shared__ md::float_t sum_x_shared
      [];
  md::float_t* const sum_y_shared =
      sum_x_shared+blockDim.x;

  const int idx = blockDim.x * blockIdx.x +
      threadIdx.x;
#ifdef READ_POS
  const md::float_t mag_x_cached =
      mag_x_global[0];
  const md::float_t mag_y_cached =
      mag_y_global[0];

  md::float_t pos_x_idx = pos_x_global[idx];
  md::float_t pos_y_idx = pos_y_global[idx];

  md::float_t omega_idx = omega_global[idx];
  md::float_t theta_idx = theta_global[idx];
#else
  // read in advance for sincos
  md::float_t theta_idx = theta_global[idx];
  // conceal memory access time
  const md::float_t mag_x_cached =
      mag_x_global[0];
  const md::float_t mag_y_cached =
      mag_y_global[0];
  md::float_t omega_idx = omega_global[idx];

  md::float_t pos_x_idx;
  md::float_t pos_y_idx;
  sincos(theta_idx, &pos_y_idx, &pos_x_idx);
#endif
  // Advance Omega
  omega_idx =
    omega_idx +
    (
      mag_y_cached * pos_x_idx
    - mag_x_cached * pos_y_idx
    ) * dt_c;

  omega_global[idx] = omega_idx;

  // Advance Theta
  theta_idx += + omega_idx * dt_d;
  theta_global[idx] = theta_idx;

  // Renew Position
  sincos(theta_idx, &pos_y_idx, &pos_x_idx);

  pos_x_global[idx] = pos_x_idx;
  pos_y_global[idx] = pos_y_idx;

  sum_x_shared[threadIdx.x] = pos_x_idx *
      num_particles_inv;
  sum_y_shared[threadIdx.x] = pos_y_idx *
      num_particles_inv;

  // Reduce Partly
  reduce_per_block_func(sum_x_shared,
      sum_y_shared);

  if(threadIdx.x==0) {
    wmem_mag_x_global[blockIdx.x] =
        sum_x_shared[0];
    wmem_mag_y_global[blockIdx.x] =
        sum_y_shared[0];
  }
}
/* ---- excerpt from main flow ---- */
void excerpt()
{
  ... // certain code
  advance_omega_kernel
    <<<num_blocks_vec[0], block_size_vec[0]>>>
    (
      -symp::pre*dt, omega_device.get(),
      pos_x_device.get(), pos_y_device.get(),
      mag_x_device.get(), mag_y_device.get()
  );
  for(size_t t=0; t<num_advance_steps; ++t){
    for(size_t s=0; s<symp::order-1; ++s) {
      advance_state(
        num_blocks_vec[0], block_size_vec[0],
        num_particles_inv,
        symp::c[s]*dt, symp::d[s]*dt,
        theta_device.get(), omega_device.get()
            ,
        pos_x_device.get(), pos_y_device.get()
            ,
        mag_x_device.get(), mag_y_device.get()
            ,
        wmem_mag_x_device.get(),
            wmem_mag_y_device.get()
      );
      if(conf.update_mag_step>0 && step_count%
          conf.update_mag_step==0) {
        reduce
          ( num_degree-1, &num_blocks_vec[1],
              &block_size_vec[1],
            wmem_mag_x_device.get(),
                wmem_mag_y_device.get(),
            wmem_mag_x_sub_device.get(),
                wmem_mag_y_sub_device.get(),
            wmem_mag_x_device.get(),
                wmem_mag_y_device.get(),
            mag_x_device.get(), mag_y_device.
                get()
          );
      }
    }
    simu_time += dt;
    step_count++;
  }
  advance_omega_kernel<<<num_blocks_vec[0],
      block_size_vec[0]>>>
    (+symp::pre*dt,
      omega_device.get(),
      pos_x_device.get(), pos_y_device.get(),
      mag_x_device.get(), mag_y_device.get());
  .... // certain code
}
```

## 5.3 Supplementaries for HMF model

### 5.3.1 Generating $(\Delta q, \Delta p)$ from $(e, M_0)$

$e$ denotes the energy density $e = \frac{H}{N}$. $M_0$ denotes the initial magnetization $M$.

On initial state, following conditions hold:

$$M_0 = \left| \int \mathbf{s}(q) f(q,p) \, \mathrm{d}q\mathrm{d}p \right| \quad , \tag{5.1}$$

$$e = \int \frac{p^2}{2} f(q,p) \, \mathrm{d}q\mathrm{d}p + \frac{1 - {M_0}^2}{2} \quad . \tag{5.2}$$

Utilizing (3.10), the equations are solved as

$$\Delta q = \mathrm{sinc}^{-1} M_0, \tag{5.3}$$

$$\Delta p = \sqrt{6\left(e - \frac{1 - {M_0}^2}{2}\right)} \quad , \tag{5.4}$$

where $\mathrm{sinc}^{-1} x$ is the inverse function of $\dfrac{\sin x}{x}$ $(0 \leq x \leq \pi)$.

### 5.3.2 Density of States and Action Integral

Density of states for HMF model is

$$D(h; \mathbf{M}) = \int \delta(h(q,p) - h) \, \mathrm{d}q\mathrm{d}p \tag{5.5}$$

$$= 4 \int_0^{q_{\max}} \frac{\mathrm{d}q}{\sqrt{2h + 2M\cos q}} \tag{5.6}$$

$$= \begin{cases} \dfrac{4}{\sqrt{M}} K\left(\sqrt{\dfrac{h+M}{2M}}\right) & (h < M) \\ 4\sqrt{\dfrac{2}{M+h}} K\left(\sqrt{\dfrac{2M}{h+M}}\right) & (h > M) \end{cases} , \tag{5.7}$$

where $K(k)$ is complete elliptic integral of the first kind

$$K(k) = \int_0^{\pi/2} \frac{\mathrm{d}\theta}{\sqrt{1 - k^2 \sin^2 \theta}} \, . \tag{5.8}$$

Action integral is

$$
\begin{aligned}
S\left(h;\mathbf{M}\right) &= \int \chi\left(h-h\left(q,p\right)\right)\mathrm{d}q\mathrm{d}p && (5.9)\\
&= 4\int_{-M}^{h} D\left(h;\mathbf{M}\right)\mathrm{d}h && (5.10)\\
&= \begin{cases} \dfrac{4}{\sqrt{M}}\left\{2\left(h-M\right)K\left(\sqrt{\dfrac{h+M}{2M}}\right)\right. & \\ \quad \left.+4ME\left(\sqrt{\dfrac{h+M}{2M}}\right)\right\} \quad (h<M) & \\ 4\sqrt{2M+2h}E\left(\sqrt{\dfrac{2M}{h+M}}\right) \quad (h>M) & \end{cases} , && (5.11)
\end{aligned}
$$

where $E\left(k\right)$ is complete elliptic integral of the second kind

$$E\left(k\right)=\int_{0}^{\pi/2}\sqrt{1-k^{2}\sin^{2}\theta}\,\mathrm{d}\theta\,. \qquad (5.12)$$

# Acknowledgment

I am deeply grateful to the members and the ex-members of Ito group.

Prof. Nobuyasu Ito gave me a lot of advices not only about research but also other things such as the course of life. The chances to use many core accelerators were valuable experiences.

Dr. Takashi Shimada also gave me a lot of advices with his keen way of thinking.

Dr. Yohsuke Murase, Dr. Naoki Yoshioka and Dr. Hajime Inaoka Dr. Teruhisa Komatsu, Dr Tomoaki Nogawa, Dr. Fumiko Takagi, Dr. Masamichi Miyama, Dr. Yuuki Izumida gave a lot of advices.

Dr. Shigenori Matsumoto had many meaningful discussions and influenced me a lot.

Mr. Koji Oishi, Mr. Takayuki Hiraoka and the other members and ex-members of Ito group supported my life as a student.

I am also deeply grateful to Prof. Masao Doi. The first period I began my research activity, I learned basis of physics in his laboratory.

Finally, I would like to thank my families and my friends for many supports.

# Bibliography


[1] Vito Latora, Andrea Rapisarda, and Constantino Tsallis. Non-gaussian equilibrium in a long-range hamiltonian system. *Phys. Rev. E*, 64:056134, Oct 2001.

[2] Alessandro Campa and Pierre-Henri Chavanis. A dynamical stability criterion for inhomogeneous quasi-stationary states in long-range systems. *Journal of Statistical Mechanics: Theory and Experiment*, 2010(06):P06001, 2010.

[3] Yoshiyuki Y. Yamaguchi, Julien Barr, Freddy Bouchet, Thierry Dauxois, and Stefano Ruffo. Stability criteria of the vlasov equation and quasi-stationary states of the hmf model. *Physica A: Statistical Mechanics and its Applications*, 337(12):36 – 66, 2004.

[4] Shun Ogawa. Spectral and formal stability criteria of spatially inhomogeneous stationary solutions to the vlasov equation for the hamiltonian mean-field model. *Phys. Rev. E*, 87:062107, Jun 2013.

[5] Andrea Antoniazzi, Duccio Fanelli, Julien Barré, Pierre-Henri Chavanis, Thierry Dauxois, and Stefano Ruffo. Maximum entropy principle explains quasistationary states in systems with long-range interactions: The example of the hamiltonian mean-field model. *Phys. Rev. E*, 75:011112, Jan 2007.

[6] Andrea Antoniazzi, Duccio Fanelli, Stefano Ruffo, and Yoshiyuki Y. Yamaguchi. Nonequilibrium tricritical point in a system with long-range interactions. *Phys. Rev. Lett.*, 99:040601, Jul 2007.

[7] Renato Pakter and Yan Levin. Core-halo distribution in the hamiltonian mean-field model. *Phys. Rev. Lett.*, 106:200603, May 2011.

[8] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[9] Nobuyasu Ito. *BUTSURI*, 67(7):478, 2012. in Japanese.

[10] J.G. Koomey, S. Berard, M. Sanchez, and H. Wong. Implications of historical trends in the electrical efficiency of computing. *Annals of the History of Computing, IEEE*, 33(3):46–54, March 2011.

[11] NVIDIA. `http://www.nvidia.co.jp/object/tesla_c1060_jp.html`.

[12] NVIDIA. `http://www.nvidia.co.jp/object/product_tesla_C2050_C2070_jp.html`.

[13] NVIDIA. `http://www.elsa-jp.co.jp/products/products-top/gpu_computing/tesla_ws/tesla_ws_list/tesla_k20/` Peak FLOPS is (core frequency) × (number of SMXs) × 128.

[14] NVIDIA. `http://www.nvidia.com/content/PDF/kepler/Tesla-K20X-BD-06397-001-v07.pdf` Peak FLOPS is (core frequency) × (number of SMXs) × 128.

[15] NVIDIA. `http://www.elsa-jp.co.jp/products/products-top/gpu_computing/tesla_ws/tesla_ws_list/tesla_k40/` Peak FLOPS is (core frequency) × (number of SMXs) × 128.

[16] NVIDIA. `http://www.elsa-jp.co.jp/products/products-top/gpu_computing/tesla_server/tesla_k/nvidia-tesla-k80/` Peak FLOPS is (boosted core frequency) × (number of SMXs) × 128.

[17] Intel. Xeon W3580. `http://ark.intel.com/products/39723/` Peak FLOPS is (core frequency) × (number of cores) × 4.

[18] Intel. Xeon X3480. `http://ark.intel.com/products/48501` Peak FLOPS is (core frequency) × (number of cores) × 4.

[19] Intel. Xeon X5690. `http://ark.intel.com/products/52576/` Peak FLOPS is (core frequency) × (number of cores) × 4.

[20] Intel. Xeon E5-2687W. `http://ark.intel.com/products/64582/` Peak FLOPS is (core frequency) × (number of cores) × 8.

[21] Intel. Xeon E5-1680 v2. `http://ark.intel.com/products/77912/` Peak FLOPS is (core frequency) × (number of cores) × 8.

[22] Intel. Xeon E5-2697 v2. `http://ark.intel.com/products/75283` Peak FLOPS is (core frequency) × (number of cores) × 8.

[23] Intel. Xeon E7-2890 v2. `http://ark.intel.com/products/75242/` Peak FLOPS is (core frequency) × (number of cores) × 8.

[24] Intel. Xeon E5-2699 v3. `http://ark.intel.com/products/81061` Peak FLOPS is (core frequency) × (number of cores) × 16.

[25] Masuo Suzuki. Fractal decomposition of exponential operators with applications to many-body theories and monte carlo simulations. *Physics Letters A*, 146(6):319 – 323, 1990.

[26] Masuo Suzuki. General theory of higher-order decomposition of exponential operators and symplectic integrators. *Physics Letters A*, 165(56):387 – 395, 1992.

[27] Haruo Yoshida. Recent progress in the theory and application of symplectic integrators. *Celestial Mechanics and Dynamical Astronomy*, 56(1-2):27–43, 1993.

[28] Hiroshi Watanabe, Nobuyasu Ito, and Chin-Kun Hu. Phase diagram and universality of the lennard-jones gas-liquid system. *The Journal of Chemical Physics*, 136(20), 2012.

[29] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with cuda. *GPU Gems*, 3(39):851–876, 2007.

[30] Mark Harris et al. Optimizing parallel reduction in cuda. *NVIDIA Developer Technology*, 2(4), 2007.

[31] NVIDIA Corporation. CUPTI User's Guide. `http://docs.nvidia.com/cuda/cupti/`.

[32] H. Watanabe, M. Suzuki, and N. Ito. Efficient Implementations of Molecular Dynamics Simulations for Lennard-Jones Systems. *Progress of Theoretical Physics*, 126(2):203–235, August 2011.

[33] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[34] Hiroko Koyama and Tetsuro Konishi. Emergence of power-law correlation in 1-dimensional self-gravitating system. *Physics Letters A*, 279(34):226 – 230, 2001.

[35] Julien Barré, Thierry Dauxois, Giovanni De Ninno, Duccio Fanelli, and Stefano Ruffo. Statistical theory of high-gain free-electron laser saturation. *Phys. Rev. E*, 69:045501, Apr 2004.

[36] Christian Beck, Gregory S. Lewis, and Harry L. Swinney. Measuring nonextensitivity parameters in a turbulent couette-taylor flow. *Phys. Rev. E*, 63:035303, Feb 2001.

[37] David Ruelle and Statistical Mechanics. *Rigorous results*. World Scientific, 1969.

[38] Toshio Tsuchiya, Tetsuro Konishi, and Naoteru Gouda. Quasiequilibria in one-dimensional self-gravitating many-body systems. *Phys. Rev. E*, 50:2607–2615, Oct 1994.

[39] Alessandro Torcini and Mickaël Antoni. Equilibrium and dynamical properties of two-dimensional n-body systems with long-range attractive interactions. *Phys. Rev. E*, 59:2746–2763, Mar 1999.

[40] D. Lynden-Bell. Negative Specific Heat in Astronomy, Physics and Chemistry. *Physica A Statistical Mechanics and its Applications*, 263:293–304, February 1999.

[41] Mickael Antoni and Stefano Ruffo. Clustering and relaxation in hamiltonian long-range dynamics. *Phys. Rev. E*, 52:2361–2374, Sep 1995.

[42] Yoshiyuki Y. Yamaguchi. One-dimensional self-gravitating sheet model and lynden-bell statistics. *Phys. Rev. E*, 78:041114, Oct 2008.

[43] T Konishi and K Kaneko. Clustered motion in symplectic coupled map systems. *Journal of Physics A: Mathematical and General*, 25(23):6283, 1992.

[44] S. Inagaki and T. Konishi. Dynamical stability of a simple model similar to self-gravitating systems. *Publications of the Astronomical Society of Japan*, 45:733–735, October 1993.

[45] D. Lynden-Bell and R. Wood. The gravo-thermal catastrophe in isothermal spheres and the onset of red-giant structure for stellar systems. *Monthly Notices of the Royal Astronomical Society*, 138:495, 1968.

[46] Julien Barré and Yoshiyuki Y. Yamaguchi. Small traveling clusters in attractive and repulsive hamiltonian mean-field models. *Phys. Rev. E*, 79:036208, Mar 2009.

[47] Yoshiyuki Y. Yamaguchi. Construction of traveling clusters in the hamiltonian mean-field model by nonequilibrium statistical mechanics and Bernstein-Greene-Kruskal waves. *Phys. Rev. E*, 84:016211, Jul 2011.

[48] W. Ettoumi and M.-C. Firpo. Action diffusion and lifetimes of quasistationary states in the hamiltonian mean-field model. *Phys. Rev. E*, 87:030102, Mar 2013.

[49] Hidetoshi Morita and Kunihiko Kaneko. Collective oscillation in a hamiltonian system. *Phys. Rev. Lett.*, 96:050602, Feb 2006.

[50] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for cuda. *GPU Computing Gems*, 7, 2011.