Universiteit Utrecht

Master Thesis

Molecular Simulations using CUDA

Gerhard Burger, BSc.
burger.ga@gmail.com

Supervisors: Prof. dr. ir. Marjolein Dijkstra
Prof. dr. Rob Bisseling
Daily supervisor: Anjan Gantapara, MSc.

Utrecht, The Netherlands
September 28, 2013

Abstract

Computer simulations play a vital role in understanding the phase behavior of colloidal dispersions, however, most simulation results suffer from finite-size effects [1]. These finite-size effects can be eliminated by finite-size scaling or by simulating large system sizes. In this thesis we show how to simulate large system sizes efficiently on Graphical Processing Units (GPUs). Whereas efficient GPU Molecular Dynamics implementations are readily available [2], Monte Carlo (MC) simulations have not yet received much attention, mainly due to their serial nature. We present a GPU implementation of the Monte Carlo simulation using the Hybrid Monte Carlo (HMC) method [3]. Our implementations for long and short-ranged potentials are at least one order of magnitude faster than the regular CPU implementations. We show the correctness of our implementation by reproducing the equation of state for the three-dimensional Lennard-Jones system.

Contents

Chapter 1

Introduction

The importance of computers in developing and applying scientific knowledge has increased tremendously over the last decades. More and more, computer simulations are deemed a viable alternative to otherwise complicated experiments. Largely, this increase in importance is due to the seemingly ever increasing computational power available. The trend that the number of transistors on a chip will double every two years, better known as Moore's law [4], turned out to be accurate, especially after chip manufacturers made it their design principle.

The achievement of Moore's law is largely contributed to Dennard scaling. Dennard et al. postulated in 1974 that if transistors are scaled down both performance and energy efficiency improve, and more transistors fit per unit area [5]. Unfortunately, chip designs start to run into physical limits; the insulating layer that separates the gate from the rest of the transistor gets so thin that because of quantum tunneling it starts to leak electrons, which increases its energy consumption.

As a result, power consumption increases at a faster rate than performance and this phenomenon of diminishing returns is the so called 'Power Wall', postulated by Patterson. Besides the Power Wall, Patterson also postulated an 'ILP Wall', the difficulty of finding enough Instruction Level Parallelism, and a 'Memory Wall', the increasing difference between processor and memory speeds [6]. Patterson states that together these Walls form a so called 'Brick Wall',

> Power Wall + Memory Wall + ILP Wall = Brick Wall. [6]

This Brick Wall is the main reason that since 2005 the single core performance of a chip does not increase at the same rate as the transistor count, as Figure 1.1 shows.

To avoid this Brick Wall, chip manufacturers started to produce multi-core chips, according to Intel CEO Craig Barrett " [...] it's the way that the industry is going to continue to follow Moore's Law going forward[...]" (as cited by Fish in [8]). Multi-core processors are a good solution in personal computers, where multiple tasks are often executed at the same time, however, they are less suitable for scientific use, where it is desirable that one lengthy task executes as fast as possible.

For this reason, the focus in Scientific Computing is shifting to parallel computing, first on multi-core machines and supercomputers, but now more and more on General-Purpose Computing on Graphics Processing Units (GPGPU). Currently there are two dominant frameworks for GPGPU: CUDA, a proprietary framework developed by NVIDIA [9], and

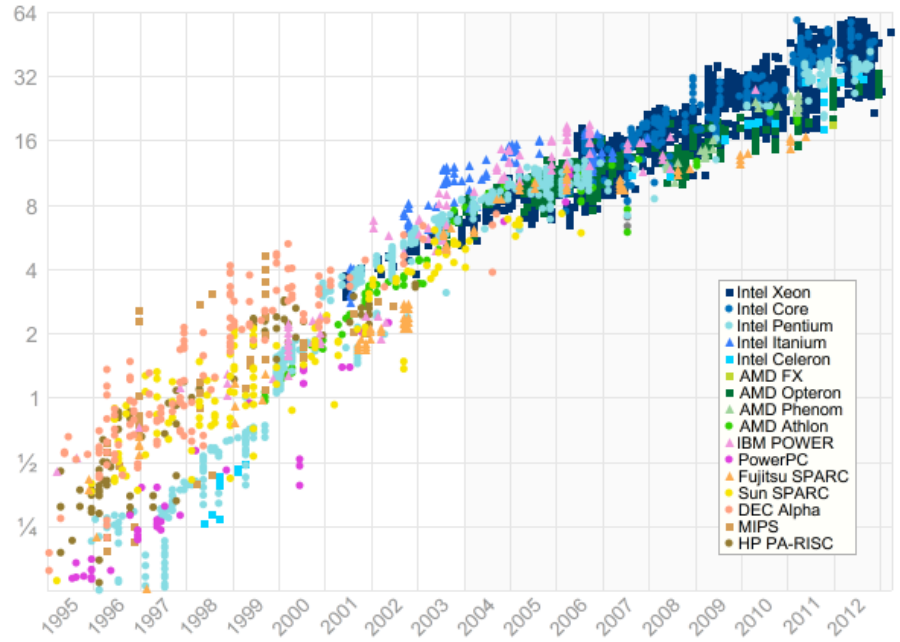


Figure 1.1: Single threaded floating-point performance (image produced using adjusted SPECfp® data [7]).

OpenCL, an open framework maintained by Khronos Group [10]. CUDA is currently the more popular framework because it is more mature and has implemented many popular libraries such as BLAS, FFT, and Thrust. The use of CUDA in supercomputers further illustrates that it is the current framework of choice: Titan, the world's fastest supercomputer until Tianhe-2 took the lead in June 2013, has 18,688 NVIDIA Tesla GPUs, and Tianhe-1A, the predecessor of Tianhe-2, currently ranked tenth in the supercomputer TOP500, has 7,168 NVIDIA Tesla GPUs [11].

1.1 Aim

Our original aim was study the phase behavior of two-dimensional hard-particle systems. In particular, we were interested in regular polygons and the change in phase behavior that would occur when going from the equilateral triangle to approximately a circle by increasing the number of sides, as Figure 1.2 shows.

These regular polygons are of interest because the densest packings and coexistence densities have not yet been studied in detail, and because they are suspected to show large finite-size effects. Bernard and Krauth showed that in hard-disk systems finite-size effects still occur for large system sizes; only for a million particles the finite-size effects disappeared. Figure 1.3 shows these finite-size effects [1]. Recent simulations by Anderson et al. confirmed these results by Bernard and Krauth [12].

We suspect that if there are finite-size effects for hard disks, other hard particles, especially those similar to hard disks, might also suffer from finite-size effects. A paper from 2012 on the phase behavior of rounded squares by Avendaño and Escobedo reinforces

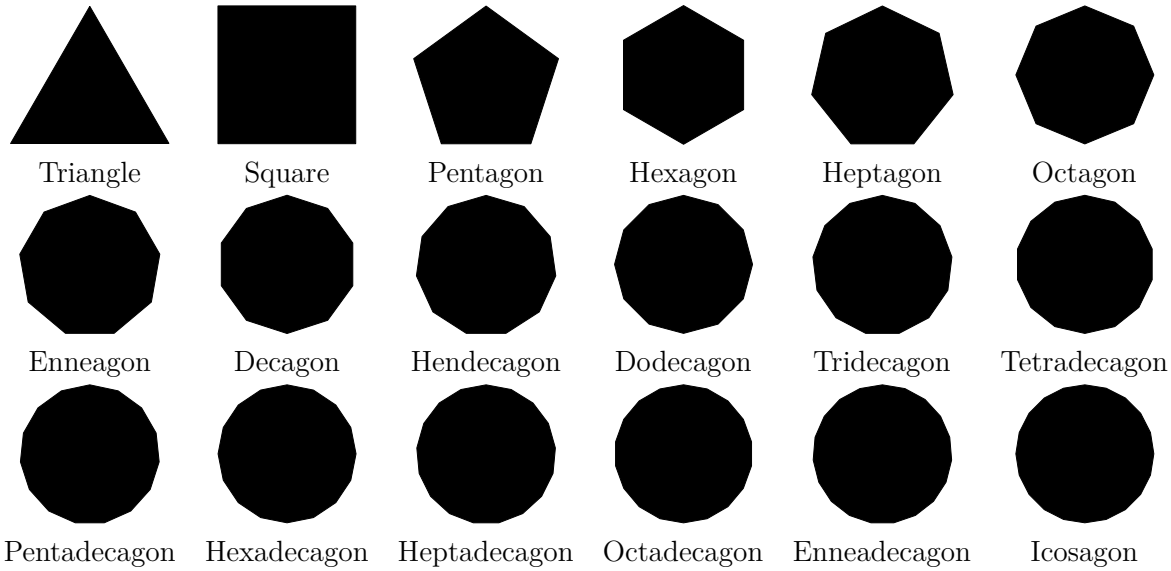


Figure 1.2: The regular polygons from the equilateral triangle to the icosagon.

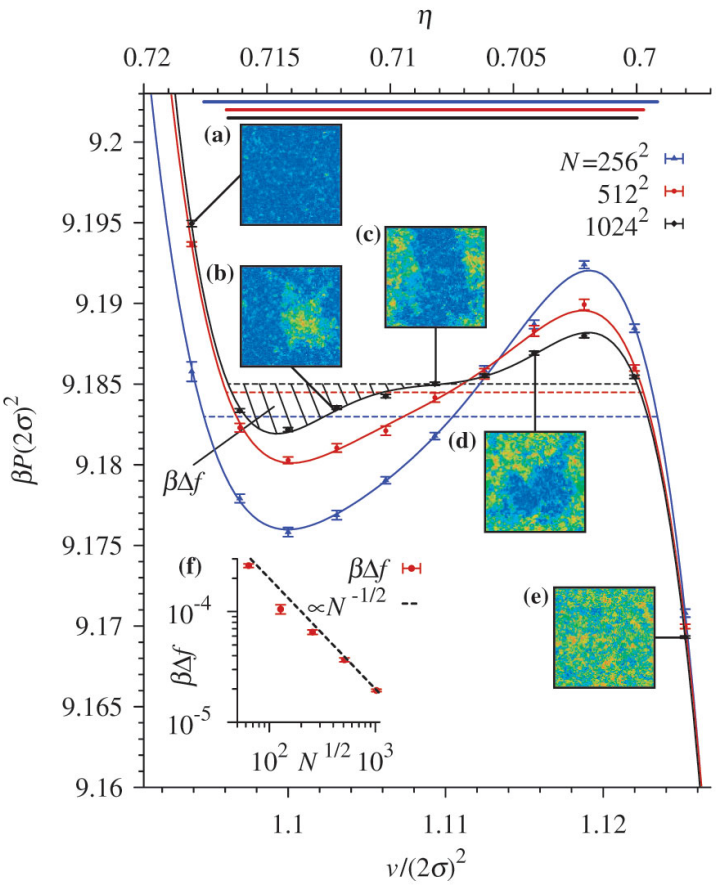


Figure 1.3: The equation of state for hard disks as published by Bernard and Krauth. Note the difference between the curves for 512^2 and 1024^2 particles [1].

this hypothesis[13].

Apart from the physical level, this problem is also of interest from a computational point of view; unlike the Molecular Dynamics (MD) method, the Monte Carlo (MC) method has not yet been successfully parallelized. To parallelize we will use CUDA, currently the most dominant GPGPU framework.

During our research we discovered that our aim was overly ambitious, and we set a more realistic aim. We will still implement the traditional hard-particle MC method, and show some preliminary results of two-dimensional hard-particle systems, but we will only parallelize the Monte Carlo method for Lennard-Jones particles. In our discussion we will discuss the additional work needed to achieve the original aim.

1.2 Outline

Because this thesis will be read by computer physicists, mathematicians and computer scientists, we will show some ideas in more detail. First, we will discuss the 'traditional' implementation of hard-particle Monte Carlo systems on Central Processing Units (CPUs), and show some preliminary equations of state. Next we will explain why, based on the properties of MC simulations, we decided to use Hybrid Monte Carlo (HMC) to parallelize our simulations. Also we explain why we switched from hard-particle systems to a Lennard-Jones (LJ) system. Since HMC also uses MD simulations we will also discuss the CPU implementation of MD and finally HMC. We will show that the results for HMC are consistent with the results we obtained from the regular MC implementation.

Then we will parallelize the MD and HMC simulation using CUDA and show that the results are consistent with our previous results. Finally we will discuss the suitability of CUDA for large scale particle simulations by looking at the performance of the GPU implementation versus the CPU implementation, the limitations of the GPU implementation, how to improve/extend the GPU implementation, and how to switch back from Lennard-Jones to hard-particle systems.

Chapter 2

Two-dimensional hard-particle systems

In this chapter we will discuss two-dimensional hard-particle systems. We will start by showing the basic Monte Carlo algorithm. Technically, Monte Carlo methods are a broad class of algorithms, but in the course of this thesis Monte Carlo denotes the Metropolis-Hastings algorithm, introduced by Metropolis et al. in 1953 [14]. We will then show how to use this algorithm to study systems with hard disks and regular polygons.

The purpose of this is to get a basic implementation which we can use to generate reference data. More importantly, we will use this implementation to get a thorough understanding of the algorithm, see where the performance bottlenecks are, and use this knowledge to design an efficient parallel algorithm for use on CUDA.

2.1 The basic Monte Carlo algorithm

The simplest ensemble for Monte Carlo simulations is the canonical or NVT ensemble. In this ensemble the number of particles N, the volume V and the temperature T are fixed, and the resulting pressure is measured. The system moves from one state to the other by random particle moves, satisfying detailed balance, which ensures convergence to the correct equilibrium distribution, as Algorithm 1 shows.

Algorithm 1 Monte Carlo in Canonical (NVT) ensemble

```
1: initialize
2: for i = 1 → steps do
3:     ParticleMove()          ▷ Includes accept/reject logic
4:     Measure()               ▷ Compute pressure
5: end for
```

We accept or reject these particle moves using the Metropolis acceptance criterion; if a trial move causes an energy change ΔU then the acceptance probability of this move is

$$\min\left(1, \exp(-\beta \Delta U)\right), \tag{2.1}$$

where $\beta = 1/k_B T$ is the thermodynamic temperature of the system.

Another well-known ensemble is the Isobaric-Isothermal or NPT ensemble. In this ensemble the pressure P is constant, and the volume is allowed to fluctuate. Algorithm 2 shows the Monte Carlo algorithm for the NPT ensemble, including the new volume trial move which allows the volume to fluctuate.

Algorithm 2 Monte Carlo in Isobaric-Isothermal (NPT) ensemble

```
1: N ← number of particles
2: initialize
3: for i = 1 → steps do
4:     r = Floor(RandUniform(0,N + 1))
5:     if r < N then
6:         ParticleMove()                    ▷ Includes accept/reject logic
7:     else
8:         VolumeMove()                      ▷ Includes accept/reject logic
9:     end if
10:    Measure()                             ▷ Compute volume/density
11: end for
```

The acceptance rule depends on how the volume V fluctuates; if the volume changes from V to $V + \Delta V$, where ΔV is a random number between $-\Delta V_{\max}$ and $+\Delta V_{\max}$, the acceptance probability is [15]:

$$\min(1, \exp(-\beta[\Delta U - P\Delta V - N\beta^{-1} \ln((V + \Delta V)/V)])). \tag{2.2}$$

It is also possible to change the logarithm of the volume from $\ln(V)$ to $\ln V + \Delta(\ln(V))$; then the acceptance rule changes to

$$\min(1, \exp(-\beta[\Delta U - P\Delta V - (N + 1)\beta^{-1} \ln((V + \Delta V)/V)])). \tag{2.3}$$

A volume move is expensive and to make sure that the system changes sufficiently before trying a new volume move, only one in $N + 1$ moves is a volume move.

2.1.1 Periodic Boundary Conditions and the Minimum Image Convention

Because we use MC simulations to provide information about macroscopic properties we have to choose boundary conditions such that our system mimics an infinite bulk system. We can do this by using Periodic Boundary Conditions (PBC). A system with PBC often uses the Minimum Image Convention (MIC). With the MIC two particles interact only via their nearest image; interactions between particles at longer distances are ignored.[1] Figure 2.1 illustrates how PBC and MIC work.

[1]Note that this introduces an implicit cutoff radius equal to the box size, which is why tail corrections are needed (see Section 3.2.1).

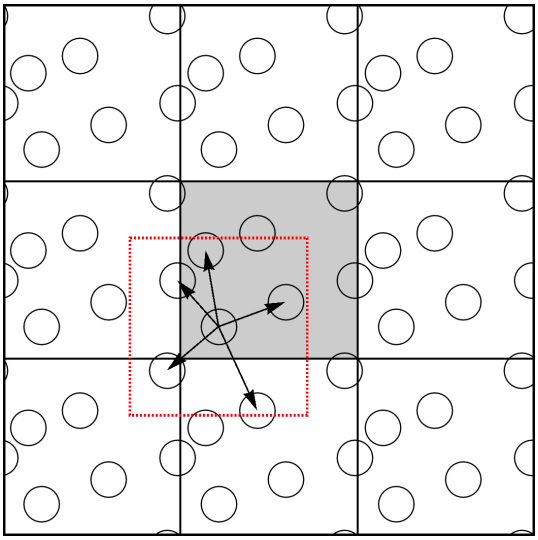


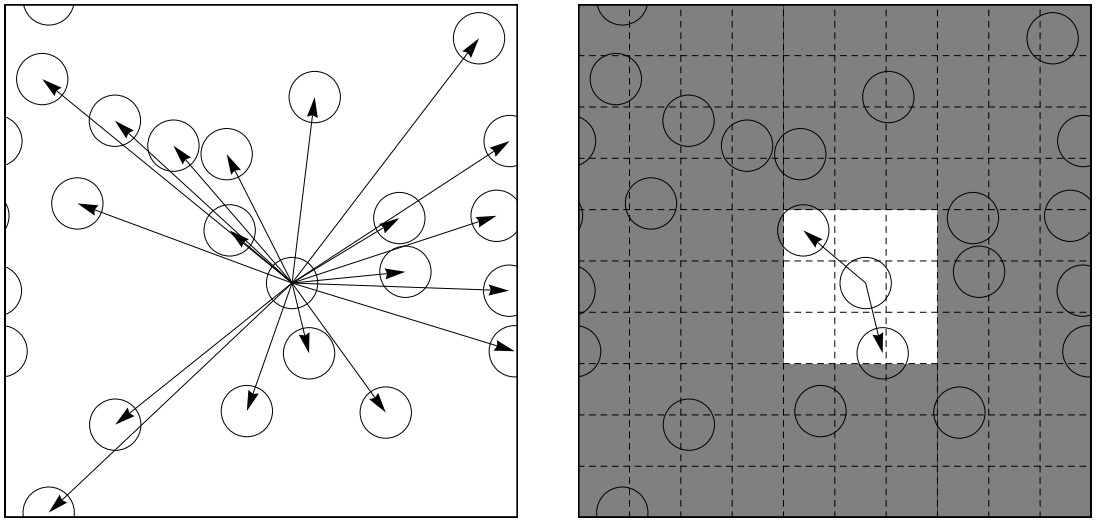
Figure 2.1: Periodic Boundary Conditions and Minimum Image Convention. The dashed line shows the area which contains the nearest neighbors for the left bottom particle.

2.2 Hard disks

With these basic MC techniques covered, we can implement the hard-disk system. Hard-particle systems are particularly easy to implement because the potential energy between hard particles is zero if they do not intersect, and infinity if they do. The acceptance criterion for a trial move given by (2.1) thus reduces to checking for intersections, if there is an intersection the move is immediately rejected. Also volume moves are immediately rejected if there is an intersection, if there is no intersection the move is accepted by (2.2) or (2.3), with $\Delta U = 0$.

2.2.1 Cell lists

For every trial move, we need to check if the particles do not overlap. In a naive implementation n^2 checks are needed for a volume move and n checks for a particle move. Domain decomposition allows us to do this more efficiently. We divide the simulation box in cells such that the width of the cell is larger or equal to the cutoff radius r_c of the particle, for hard particles the cutoff radius is equal to the diameter of the particle. In that way a particle can only overlap with particles in the neighboring cells. The number of particles in one cell is on average $m = \rho\ A_{\text{cell}}$, so for one particle we check 9 cells with on average m particles. The complexities of a volume move and particle move are therefore $O(9mn) = O(n)$ and $O(9m) = O(1)$ (!) which are much better than the $O(n^2)$ and $O(n)$ of the naive approach. Figure 2.2 illustrates the impact of using cell lists; the arrows show the pairs of particles that will be checked.



(a) (b)

Figure 2.2: (a) Without cell lists, and (b), with cell lists. The arrows show the pairs of particles that are checked.

2.3 Hard regular polygons

One step up in complexity are systems with hard regular polygons (see Figure 1.2). Apart from a rotation trial move, we also need some additional techniques. Whereas for hard disks collision detection is just calculating the distance between the centers, for convex polygons we need the Separating Axis Theorem[2] to detect collisions. Because some lattice structures do not fit in a square box we also use the floppy-box method, which allows the box to change shape. The following subsections discuss these techniques in more detail.

2.3.1 Rotation move

Including a rotation move is straightforward, we replace ParticleMove() in Algorithms 1 and 2 by Algorithm 3.

Algorithm 3 Replacement for ParticleMove() in Algorithms 1 and 2.

```
1: r = RandUniform(0,1)
2: if r < 0.5 then
3:     ParticleMove()
4: else
5:     RotateMove()
6: end if
```

The acceptance criterion is the same: if there is an intersection, reject the move, otherwise accept. Using the Separating Axis Theorem we can determine whether or not two regular polygons intersect.

[2]Better known to mathematicians as the Hyperplane Separation Theorem.

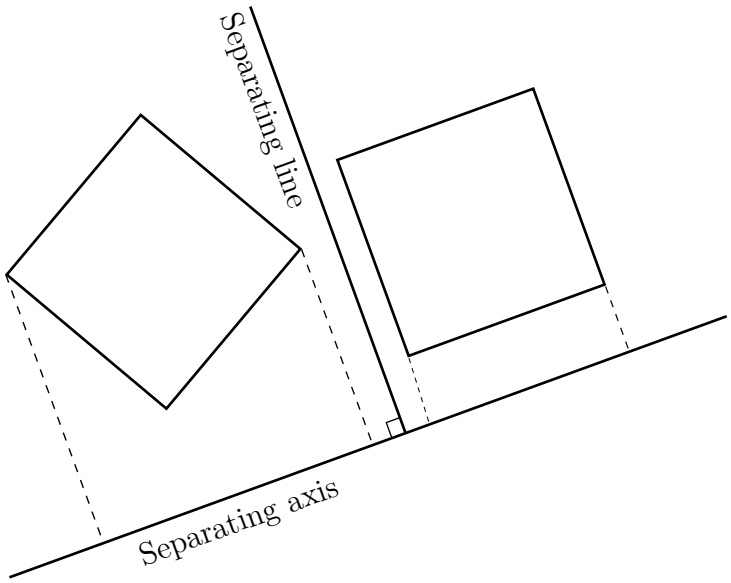


Figure 2.3: Separating Axis Theorem. Because the projections of the squares do not overlap the axis is a separating axis.

2.3.2 Separating Axis Theorem

The Separating Axis Theorem (SAT) states that if two convex objects do not overlap then there exists an axis for which the projections of the objects onto it do not overlap. In case of regular polygons the possible axes are those perpendicular to the edges of either polygon and if none of the axes is separating then there is a collision [16]. Figure 2.3 shows SAT for two non-intersecting convex objects.

SAT is computationally expensive, especially when checking two objects with a large number of edges. The upside about using regular polygons is that they allow some optimizations. For instance, for even-sided regular polygons we only half of the edges need to be checked; two opposite edges correspond to the same axis. We can also use the circumscribed and inscribed circle as a broad-phase collision detection: We first check if the distance between the particles is larger than twice the radius of the circumscribed circle, if so, there is no intersection. Then we check if the distance is smaller than twice the radius of the inscribed circle, if so, there is an intersection. If the distance is somewhere in between we use SAT to check for intersection. Note that the difference between the radii for the circumscribed and inscribed circle decreases when the number of sides increases, as Figure 2.4 shows. This means that although SAT becomes more expensive when the number of sides increases, the probability to use SAT decreases.

2.3.3 Variable box shape simulations

Variable box shape or 'floppy box' for Monte Carlo simulations is a relatively new technique discussed by Filion et al. [17]. In three-dimensional floppy-box simulations the simulation box is a parallelepiped, as Figure 2.5 shows; in two-dimensional systems the simulation box is a parallelogram. In a volume move in a MC simulation, we change either an angle between two edges, or an edge length, as Algorithm 4 shows.

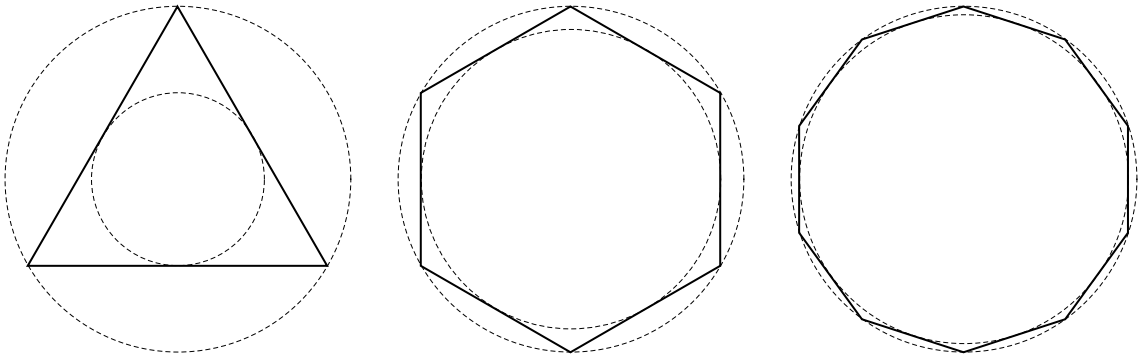


Figure 2.4: Circumscribed and inscribed spheres for the triangle, hexagon, and decagon. Note how the difference in radii between the circumscribed and inscribed sphere decreases as the number of sides increases.

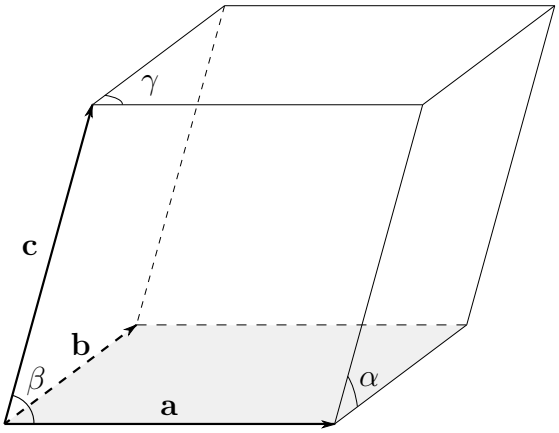


Figure 2.5: The parallelepiped simulation box.

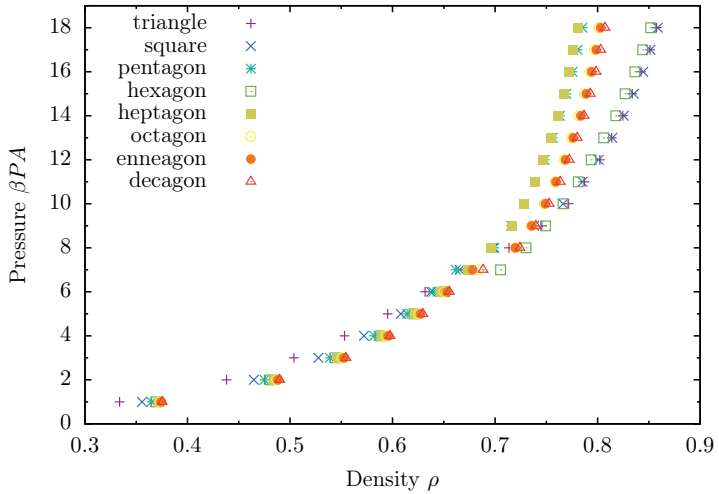
Algorithm 4 Replacement for VolumeMove() in Algorithm 2.

```
r = RandUniform(0,1)
if r < 0.5 then
    VolumeMove()                    ▷ Old volume move, scales the edges
else
    FloppyMove()    ▷ New volume move, changes an angle or an edge length
end if
```

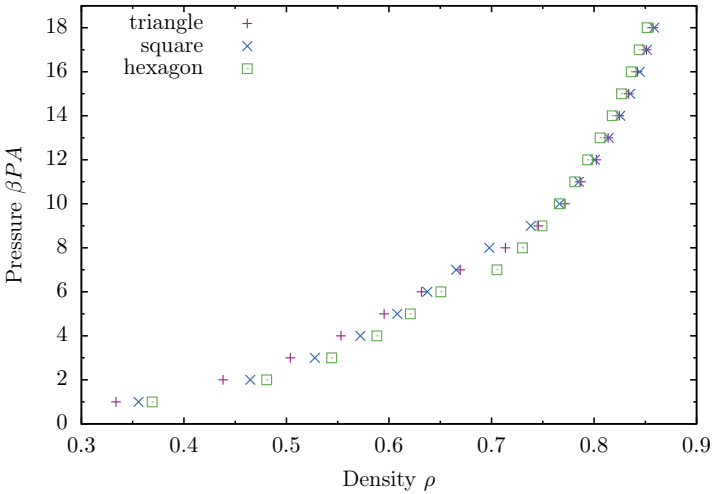
2.4 Results

Using the approach outlined above we made a CPU version for hard regular polygons. The polygons used have unit area, such that the density ρ is equal to the number density n. We defined the reduced pressure as $P^* = \beta PA = \beta P$ where $A = 1$ is the area of the polygons. Figure 2.6 shows some preliminary results. Because these simulations are in the NPT ensemble, the results do not show the Van der Waals loop[3] that is present in Figure 1.3. Instead, in NPT simulations there is a jump in the equation of state, like Figure 2.6d shows for density $\rho = 0.7$.

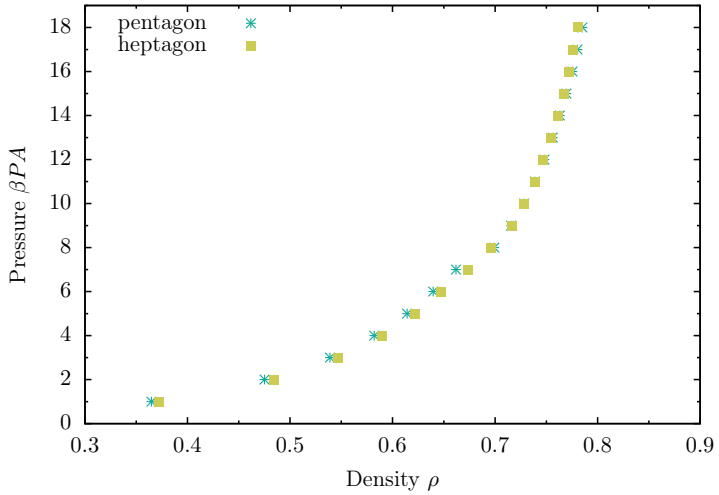
[3]The Van der Waals loop is the oscillating part in the middle. Since a necessary condition for stability in thermodynamic equilibrium is that the pressure does not increase with the volume, the oscillating part is considered 'unphysical' and usually replaced with a straight line, in such a way that the area below the line is equal to the area above the line. These lines, also shown in Figure 1.3, are so-called Maxwell constructions [18].



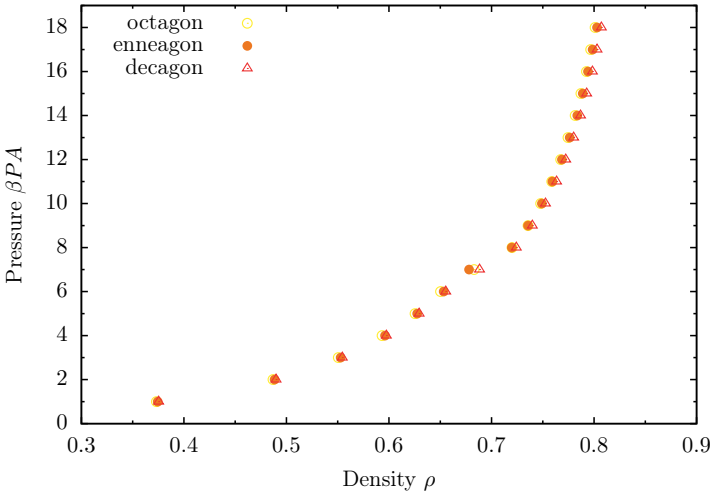
(a) Overview



(b) Spacefilling structures



(c) Pentagon and heptagon



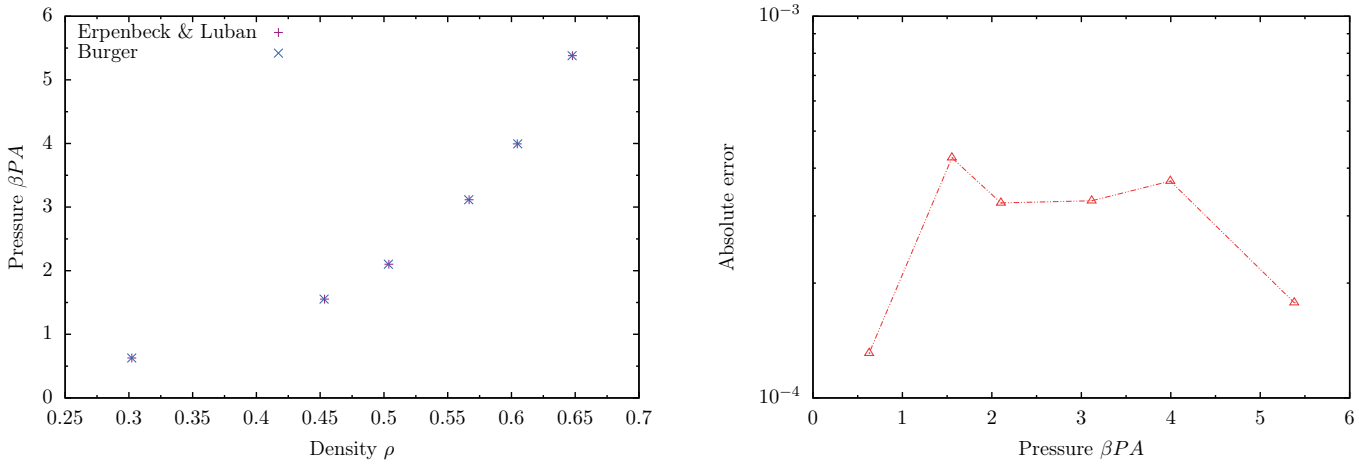
(d) Octagon, enneagon and decagon

Figure 2.6: Some preliminary results for two-dimensional hard-particle systems with size 300-400.

When the number of sides is very large the system reduces to the hard-disk system. With the regular chiliagon (1000 sides) and the system specifications in Table 2.1 we were able to reproduce the results for the hard-disk liquid by Erpenbeck and Luban, which Figure 2.7 shows [19].

2D MC	
Particles	400
Sides	1000
Equilibration cycles	500,000
Measurement cycles	3,000,000

Table 2.1: System specifications for the results in Figure 2.7.



(a) (b)

Figure 2.7: Comparison of our results for the regular chiliagon with the hard-disk equation of state by Erpenbeck and Luban [19]. Note that since the simulation is in the NPT ensemble which is why in (b) the pressure is on the x-axis.

The reason the data in Figure 2.6 is preliminary is because of the small system size. To show just how big the finite-size effects can be, we used the regular chiliagon in attempt to reproduce a data point near the coexistence region in the hard-disk equation of state by Qi, Gantapara, and Dijkstra [20]. Figure 2.8 shows the results. Although we have not picked a value in the coexistence region, the finite-size effects are large, which illustrates the need to look at large systems.

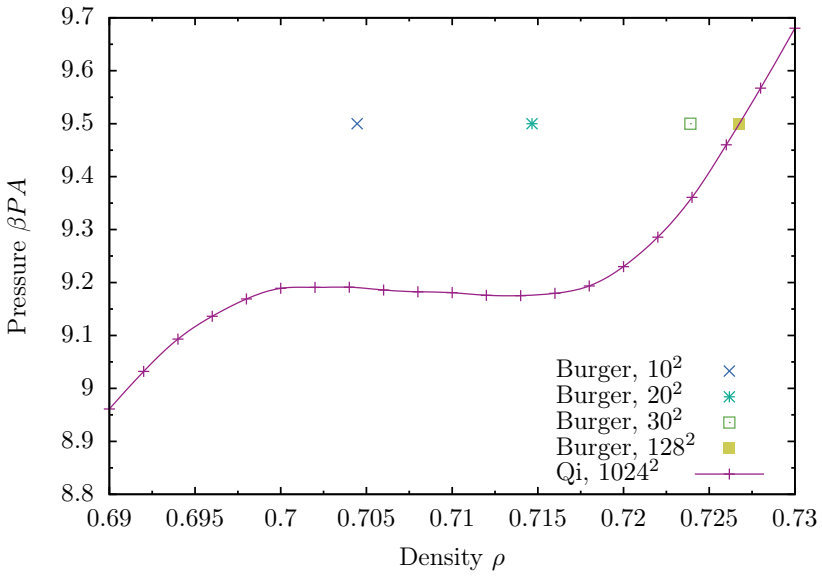


Figure 2.8: Illustration of the finite-size effects near the coexistence region. The reference equation of state is from Qi, Gantapara, and Dijkstra [20]. Note that since since the simulation is in the NPT ensemble the pressure is the input variable and density the measured value.

Chapter 3

Hybrid Monte Carlo simulations

Parallelizing the Monte Carlo method is non-trivial because of the trial particle move. A naive approach would be to randomly move all particles by a small amount. Unfortunately, even with an individual acceptance rate close to one, the acceptance probability for a set of a thousand particles is already close to zero. This means that a move gets rejected even if the particles are essentially standing still.

One way to move all particles at the same time and with a high acceptance rate is to use a small Molecular Dynamics (MD) simulation as trial move. This approach is called the Hybrid Monte Carlo (HMC) method, introduced by Duane et al. in 1987. Mehlig, Heermann, and Forrest showed that this approach is effective for Condensed Matter systems and it combines some of the advantages of Molecular Dynamics methods with those of Monte Carlo methods [21]:

- It is an exact method; the ensemble averages do not depend on the step size used in the MD simulation.
- It allows global moves, unlike MC.
- It is numerically stable, unlike MD.
- Temperature is used in the correct statistical mechanical sense, yielding the canonical probability distributions, unlike isokinetic MD.

Algorithm 5 shows the pseudo code for Hybrid Monte Carlo simulations in the NVT ensemble. Algorithm 5 is essentially Algorithm 1 where ParticleMove() is replaced by MolecularDynamicsSimulation(), and before every new MD simulation new random velocities are drawn from the Maxwell-Boltzmann distribution. Whereas normal MD does not have accept/reject logic, in HMC the MolecularDynamicsSimulation() move is accepted or rejected using the MC acceptance criterion in (2.1).

For the NPT ensemble Algorithm 2 is modified in the same way, resulting in Algorithm 6.

3.1 Molecular Dynamics simulations

To implement the Hybrid Monte Carlo method it is necessary to implement Molecular Dynamics simulations as well. Since the MD simulation serves to replace the particle

Algorithm 5 Hybrid Monte Carlo in canonical (NVT) ensemble

```
initialize
for i = 1 → steps do
    RandomizeVelocities()                ▷ From Maxwell–Boltzmann distribution
    MolecularDynamicsSimulation()        ▷ Includes accept/reject logic
    Measure()                            ▷ Compute pressure
end for
```

Algorithm 6 Hybrid Monte Carlo in isobaric-isothermal (NPT) ensemble

```
N ← number of particles
initialize
for i = 1 → steps do
    r = Floor(RandUniform(0,N + 1))
    if r < N then
        RandomizeVelocities()            ▷ From Maxwell–Boltzmann distribution
        MolecularDynamicsSimulation()    ▷ Includes accept/reject logic
    else
        VolumeMove()                     ▷ Includes accept/reject logic
    end if
    Measure()                            ▷ Compute volume/density
end for
```

trial move, which is accepted or rejected based on the difference in potential energy ΔU, MD is used in the microcanonical or NVE ensemble. This way energy is (approximately) conserved, which leads to high acceptance rates. Algorithm 7 shows the basic MD NVE scheme.

Algorithm 7 Molecular Dynamics in the microcanonical (NVE) ensemble

```
initialize
ComputeAccelerations(x)
for i = 1 → steps do
    Integrate(x, v, a)                   ▷ Includes computing new accelerations
    Measure()
    t ← t + dt
end for
```

The accelerations are computed using $F = -\nabla U$ and $F = ma$, where $m = 1$ is the particle mass, which results in $a = -\nabla U$. To integrate we use the Velocity Verlet integrator.

3.1.1 Velocity Verlet integration

A common choice for the numerical integration in MD simulations is the Velocity Verlet integrator, since it is energy conserving and time reversible [15]. This integrator consists of the basic Verlet integrator, popularized by Verlet in 1967 for use in Molecular Dynamics

simulations [22], plus the explicit calculation of the velocities as done by Swope et al. [23].

The basic Verlet integrator is an explicit central difference approximation for the second derivative of position:

$$a_n = \ddot{r}_n \approx \frac{\frac{r_{n+1}-r_n}{h} - \frac{r_n - r_{n-1}}{h}}{h} = \frac{r_{n+1} - 2r_n + r_{n-1}}{h^2}. \tag{3.1}$$

Treating (3.1) as equality gives the basic Verlet algorithm:

$$r_{n+1} = 2r_n - r_{n-1} + a_n h^2. \tag{3.2}$$

Approximating the velocity by a central difference approximation as well,

$$v_n = \dot{r}_n \approx \frac{r_{n+1} - r_{n-1}}{2h}, \tag{3.3}$$

rewriting (3.2) gives

$$r_{n+1} = r_n + v_n h + \frac{1}{2} a_n h^2, \tag{3.4}$$

and rewriting (3.3) gives

$$v_{n+1} = v_n + \frac{1}{2}\left(a_{n+1} + a_n\right) h, \tag{3.5}$$

which is the Velocity Verlet scheme [23].

This scheme is simple to implement using Algorithm 8:

Algorithm 8 Velocity Verlet integration

Require: Accelerations a are computed

1: $x \leftarrow x + vh$
2: $v \leftarrow v + ah/2$
3: $a \leftarrow \textsc{ComputeAcceleration}(x)$
4: $v \leftarrow v + ah/2$

The Velocity Verlet scheme has a Local Truncation Error (LTE) of order $O(h^4)$ for the position and $O(h^2)$ for velocity (easy to show by Taylor expansion), and a Global Truncation Error (GTE) of $O(h^2)$ for both (determined from the LTE). This makes the Velocity Verlet scheme one order more precise than the symplectic Euler method.

Algorithm 9 shows Algorithm 7 including the Velocity Verlet integration method of Algorithm 8 (using $h = dt$).

The accuracy of the MD simulation, and thus the acceptance of the trial move, is determined by the number of steps in the MD simulation and the time step dt; more steps and a smaller time step result in a better energy conservation, and thus a higher acceptance rate, but both increase the computational cost. The optimal values for these parameters are system dependent.

Algorithm 9 Molecular Dynamics in the microcanonical (NVE) ensemble, including Velecity Verlet integration.

```
1: initialize
2: ComputeAccelerations(x)
3: for i = 1 → steps do
4:     x ← x + v dt
5:     v ← v + a dt/2
6:     a ← ComputeAcceleration(x)
7:     v ← v + a dt/2
8:     Measure()
9:     t ← t + dt
10: end for
```

3.2 The Lennard-Jones potential

For hard-disk systems there exists an efficient CPU algorithm called Event Driven Molecular Dynamics by Rapaport [24]. This algorithm is fast and uses a binary tree to store events. It is not possible to use the binary tree data structure on GPUs (yet) because the binary tree structure cannot be updated in parallel (yet). This makes parallelizing EDMD a challenge in itself.

An alternative approach is to use free-flight Molecular Dynamics simulations (time driven), where the simulation is rewinded as soon as a collision is detected, as discussed by Rebertus and Sando [25]. We will first implement the Lennard-Jones MD simulation in three-dimensions, which is considerably easier, and discuss later how to switch back to hard particles in two dimensions using free-flight Molecular Dynamics.

The Lennard-Jones potential, given by (3.6), is a mathematically simple approximation of the pair potential between molecules.

$$u_{\mathrm{lj}}(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{3.6}$$

In this formula r is the separation between two Lennard-Jones particles and σ is diameter of the particle, in simulations usually set to one. Figure 3.1 shows the Lennard-Jones potential.

The Lennard-Jones potential, introduced in 1924 by Jones, is a commonly used potential in molecular simulations [26]. The r^{-12} term approximates the short-range Pauli repulsion and the r^{-6} term approximates the long-range Van der Waals attraction.

According to Ercolessi, the Lennard-Jones potential is

> ...the standard potential to use for all the investigations where the focus is on fundamental issues, rather than studying the properties of a specific material. The simulation work done on LJ systems helped us (and still does) to understand basic points in many areas of condensed matter physics, and for this reason the importance of LJ cannot be underestimated. [27]

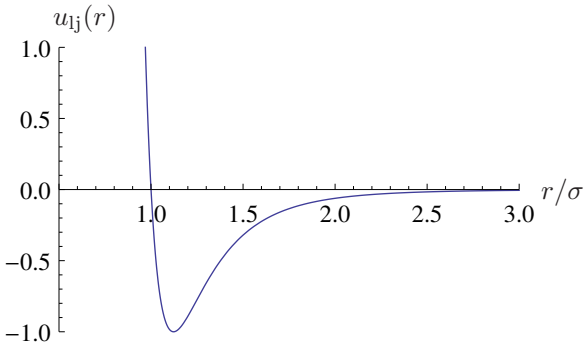


Figure 3.1: The Lennard-Jones potential: the potential between two Lennard-Jones particles, as a function of the separation r between them.

3.2.1 Truncation

For computational efficiency the LJ potential is often truncated at cutoff radius r_c:

$$u_{\mathrm{trunc}}(r) = \begin{cases} u_{\mathrm{lj}}(r) & r \leq r_c \\ 0 & r > r_c \end{cases}. \tag{3.7}$$

A common choice for this truncation is $r_c = 2.5\sigma$. This truncation allows the use of cell lists, which can speed up simulations significantly. Unfortunately, because the potential is not exactly zero at the cutoff radius we introduce truncation errors. We can correct for these truncations errors by using the given tail corrections for the pressure and the potential energy in (3.8) and (3.9).

$$P_{\mathrm{tail}} = \frac{16}{3}\pi\rho^2\epsilon\sigma^3\left[\frac{2}{3}\left(\frac{\sigma}{r_c}\right)^9 - \left(\frac{\sigma}{r_c}\right)^3\right] \tag{3.8}$$

$$u_{\mathrm{tail}} = \frac{8}{3}\pi\rho\epsilon\sigma^3\left[\frac{1}{3}\left(\frac{\sigma}{r_c}\right)^9 - \left(\frac{\sigma}{r_c}\right)^3\right] \tag{3.9}$$

Because of the popularity of the truncated (and uncorrected) Lennard-Jones potential it has become a reference model on its own, however, we include the tail corrections to compare our data with the data of Johnson, Zollweg, and Gubbins [28, 29].

3.2.2 Truncation and shift

Using the truncated Lennard-Jones potential in Molecular Dynamics simulations creates a new problem: whenever a particle crosses the cutoff radius, there is a sudden jump in its potential energy contribution, if this happens for a large number of particles the energy in the system is not conserved [15, 27]. Hence, for MD simulations it is required to use the truncated and shifted Lennard-Jones potential; shifting the entire potential by $u_{lj}(r_c)$ removes the discontinuity at $r = r_c$. Equation 3.10 shows the truncated and shifted LJ potential.

$$u_{\text{tr-sh}}(r) = \begin{cases} u_{\mathrm{lj}}(r) - u_{\mathrm{lj}}(r_c) & r \leq r_c \\ 0 & r > r_c \end{cases}, \tag{3.10}$$

For this truncated and shifted potential we can use the same tail corrections, (3.8) and (3.9), and an additional correction for the potential energy, equal to the average number of particles that are within distance r_c from a given particle, multiplied by half the value of $u_{lj}(r_c)$ [15]. For a three-dimensional system this results in an additional correction of

$$u_{\text{add}} = \frac{4}{3}\pi {r_c}^3 \rho \frac{1}{2} u_{lj}(r_c) = \frac{8}{3}\pi\rho\epsilon {r_c}^3 \left[\left(\frac{\sigma}{r_c}\right)^{12} - \left(\frac{\sigma}{r_c}\right)^{6} \right]. \tag{3.11}$$

3.3 Results

Using the above approach we implemented the HMC method in the NVT and NPT ensembles. For both we needed MD in the NVE, which we also implemented separately to make performance comparisons later on. For the same reason, we also implemented the MC method in the NPT ensemble. We tested the MD NVT ensemble by adding simple velocity scaling to the MD NVE implementation; this is not the best way to do NVT in MD, but it gives reasonable results. Table 3.1 shows the system specifications we chose to test these implementations:

HMC NVT	
Particles	512
Potential	LJ tr-sh
Temperature	2.0
MD time step	0.003
MD steps	10
Equilibration cycles	100,000
Measurement cycles	100,000

HMC NPT	
Particles	128
Potential	LJ tr-sh
Temperature	2.0
MD time step	0.003
MD steps	10
Equilibration cycles	300,000
Measurement cycles	500,000

MD NPT	
Particles	512
Potential	LJ tr-sh
Temperature	2.0
MD time step	0.003
Equilibration cycles	100,000
Measurement cycles	100,000

MC NPT	
Particles	512
Potential	LJ trunc
Temperature	2.0
Equilibration cycles	1,000,000
Measurement cycles	3,000,000

Table 3.1: System specifications for the results in Figure 3.2.

Figure 3.2 shows the equations of state obtained, together with the reference equation of state by Johnson, Zollweg, and Gubbins. Figure 3.3 shows the difference with the reference equation of state [28]. Because the HMC CPU implementations were just a proof of principle and not intended as production code, we did not optimize them and chose small system sizes and a small number of cycles.

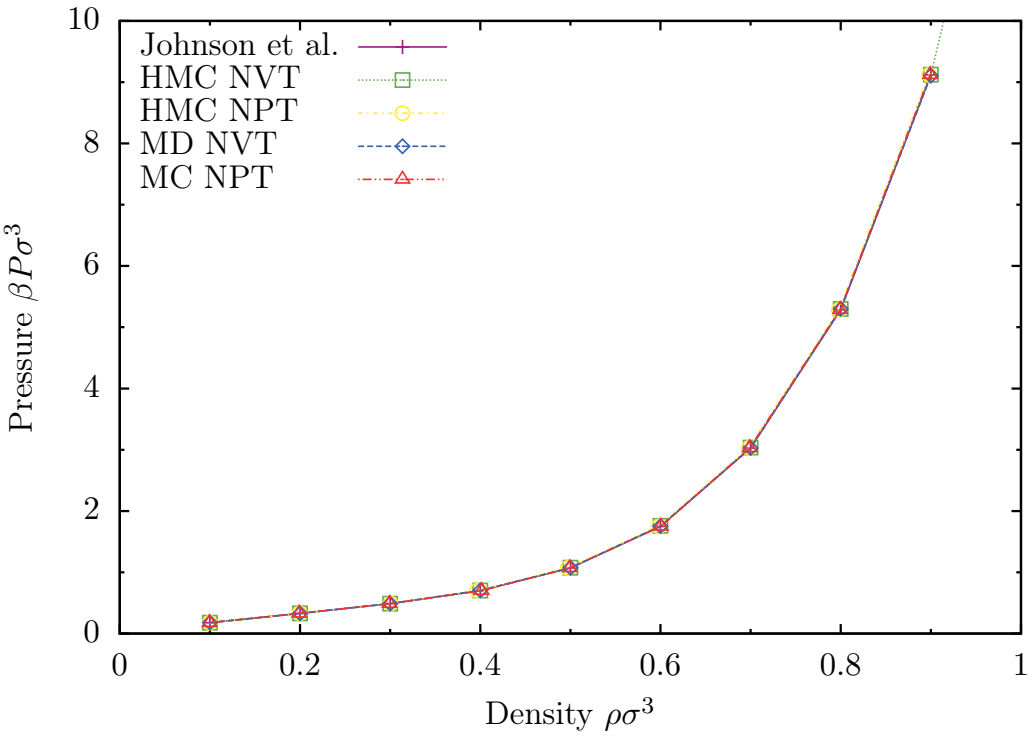
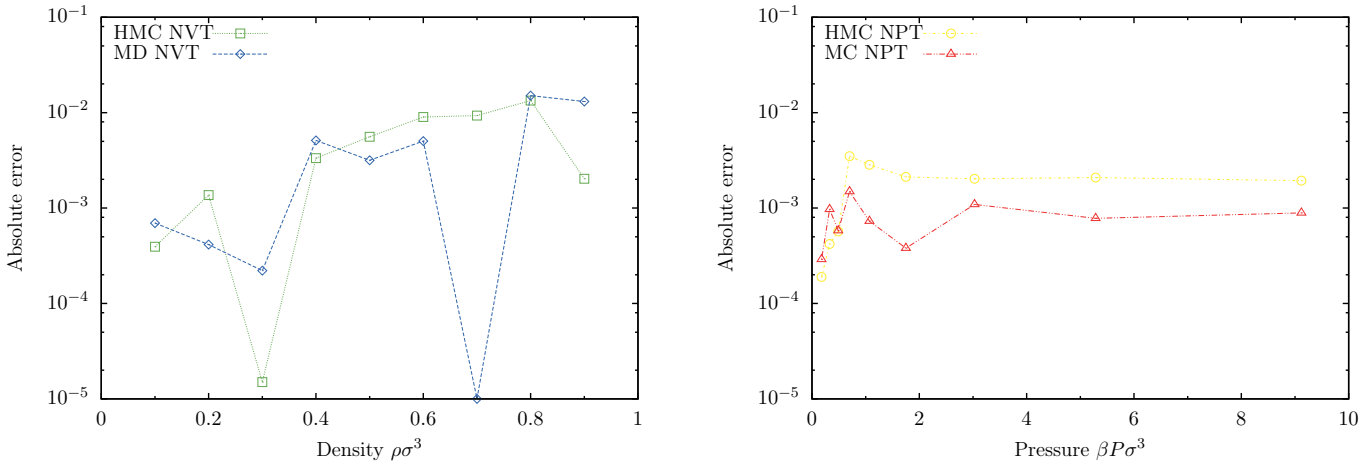


Figure 3.2: Equations of state obtained using the systems specified in Table 3.1. All simulations are done on a CPU.



(a) NVT (b) NPT

Figure 3.3: Absolute error for the equations of state in Figure 3.2.

In the NPT implementations we also computed the virial pressure, and as Table 3.2 shows the imposed and measured (virial) pressure agree, which is a good indicator our implementation is correct.

$\beta P\sigma^3$	$\langle \beta P_{\mathrm{HMC}}\sigma^3 \rangle$	$\langle \beta P_{\mathrm{MC}}\sigma^3 \rangle$
0.1776	0.17783	0.17766
0.329	0.32886	0.32833
0.489	0.48958	0.48685
0.700	0.69890	0.70124
1.071	1.07123	1.06370
1.75	1.75181	1.74574
3.028	3.02506	3.02088
5.285	5.28552	5.24481
9.12	9.11574	9.06023

Table 3.2: Comparison of the imposed and measured (virial) pressure for the HMC and MC implementation in the NPT ensemble.

3.3.1 Profiling

Before parallelizing our code it is useful to have a look at what the current bottlenecks and the theoretical limits of parallelization are.

To have a look at the current bottlenecks we have profiled the applications above using `gprof`, a utility that shows how much time a program spends inside each function. Figure 3.4 shows these profiles.

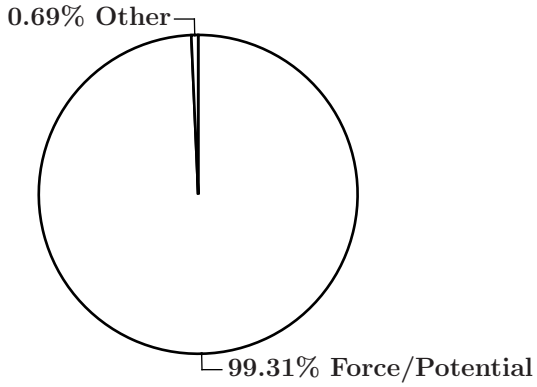
Figure 3.4a and b show that the force/potential calculations dominate the execution time of the HMC algorithms, which is expected, considering that they use a small MD simulation as trial move. The HMC NPT simulation also includes volume moves, but also volume moves consist mainly of potential calculations. Figure 3.4c shows that even for an optimized MD implementation using cell lists, the force/potential calculation still dominates the execution time. These profiles show that if we want an efficient HMC implementation we should focus our efforts on an efficient MD implementation, or more specifically, an efficient force/potential calculation.

Figure 3.4d shows that our cell-list MC implementation is well optimized: 72.51% of its time is spent on partial potential calculations, which are done for every trial move, and only 24.04% of its time on full potential calculations. In Section 2.2.1 we showed that trial moves have $O(1)$ complexity, which makes improvement by parallelization limited, but because they are executed often (the probability of a trial move is $N/(N+1)$, see Algorithm 2) they still take up most of the time.

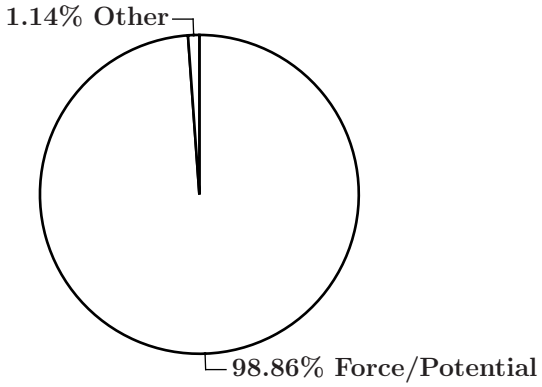
Based on these profiles we can make theoretical predictions about the maximum parallelization speedup.

Amdahl's law

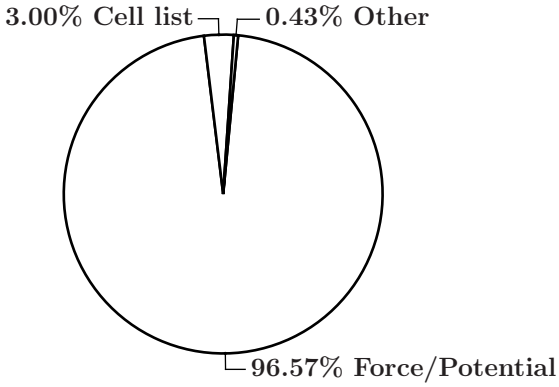
There are two laws that limit the speedup: The first is Amdahl's law which describes strong scaling. Strong scaling describes the influence of adding extra cores/threads to a



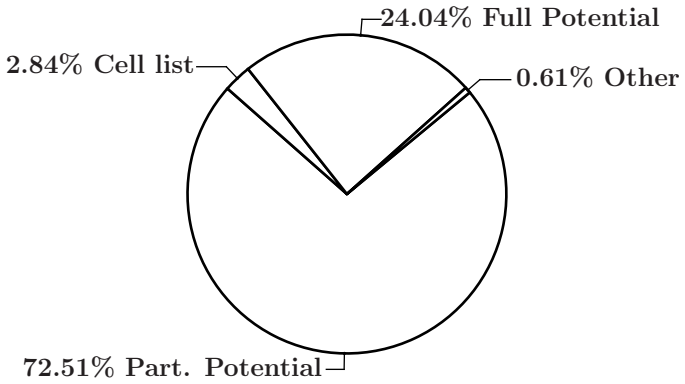
(a) HMC NVT



(b) HMC NPT



(c) MD (cell list)



(d) MC (cell list)

Figure 3.4: Profiles for the CPU implementations obtained with `gprof`. The charts show the percentage of execution time the program spends in each function category.

problem with fixed size. Amdahl's law (3.12) then gives the maximum speedup S:

$$S = \frac{1}{(1-P) + P/N}, \tag{3.12}$$

where P is the fraction of the problem that can be parallelized, and N the number of cores/threads [30]. Suppose that there are infinitely many threads available, such that the P/N term goes to zero, then the maximum speedup is $1/(1-P)$. That means that if 90% of the code can be parallelized (P=0.9), the maximum speedup is only 10, even with an infinite(!) number of cores/threads. The profiling results in Figure 3.4 show that for the HMC implementations we have $P \approx 0.99$, which gives a maximum theoretical speedup $S \approx 100$.

Gustafson's law

There is also Gustafson's law which describes weak scaling. Weak scaling describes the maximum speedup if more cores/threads are added which all do an equal amount of work. The problem size is thus dependent on the number of cores/threads that are available. Gustafson's law (3.13) gives as maximum speedup S:

$$S = (1-P) + N \cdot P = N + (1-N)(1-P). \tag{3.13}$$

For a problem where P is close to one, Gustafson's law gives a speedup equal to the number of cores/threads [31]. An example of Gustafson's law is adding more particles or iterations. You can see that for molecular simulations this will increase the portion of the code that can be parallelized (which will give a higher speedup in Amdahl's law). Both Amdahl's and Gustafson's law show that MD and HMC simulations are suitable for parallelization.

Chapter 4

CUDA

In this chapter we will explain how to parallelize the Hybrid Monte Carlo algorithms introduced in the previous chapter and make them suitable for CUDA. NVIDIA introduced CUDA (Compute Unified Device Architecture) in 2006 as a "general purpose parallel computing architecture", and it is currently the most dominant framework for General Purpose Computing on Graphics Processing Units (GPGPU). Because CUDA is a proprietary framework, CUDA is only implemented by the GPUs of NVIDIA. CUDA is supported in many programming languages and there are many libraries that support CUDA, or are implemented by CUDA, for example:

- cuFFT, the CUDA Fast Fourier Transform, which is up to 10 times faster than the Intel Math Kernel Library (MKL) [32],
- cuBLAS, the CUDA Basic Linear Algebra Subroutines, which is up to 6 times to 17 times faster than MKL BLAS [32],
- Thrust, resembles the C++ Standard Template Library (STL), but is suitable for parallel frameworks such as CUDA and OpenMP [33].

CUDA, and other GPGPU frameworks, use the GPU as a coprocessor for compute intensive tasks. GPUs can process many similar tasks in parallel, in contrast to CPUs which can process one task very fast, but has to do tasks after each other. The rest of this chapter will discuss in more detail how CUDA works.

4.1 Hardware

First we will give an overview of the hardware used. On a local machine we have an Intel Core i7 2600K and two NVIDIA GT520's. We also have access to the Little Green Machine (LGM) in Leiden which has 20 nodes, each containing two graphics cards. The graphics cards in most nodes are NVIDIA GTX480s, but there is also a node with GTX580s and a test node with a NVIDIA Tesla C2075, a professional High Performance Computing (HPC) product. We mainly used the GTX480s in this machine. Figure 4.1 shows an overview photo of the LGM. Table 4.1 shows the characteristics of the available hardware.

Figure 4.1: The Little Green Machine in Leiden. It contains 20 nodes with each two graphics cards [34].

				
Name	i7-2600K	GT520	GTX480	Tesla C2075
Price†	€260	€40	€205	€2200
Clock speed	3.4 GHz			
CUDA cores (SP)		48	448	448*

† Prices retrieved/estimated from the Tweakers.net Pricewatch database.
∗ The difference between the GTX480 and the Tesla is double precision performance and the custom HPC driver.

Table 4.1: Overview of the available hardware.

4.2 Programming: Thrust

We use Thrust to show some basic and easy CUDA examples. CUDA interoperability with Thrust is one of the biggest benefits of CUDA. For portions of code that are straightforward to parallelize, or 'embarrassingly parallel', Thrust allows one to write the parallel code in a few lines without having to worry about the low level implementation. Examples of such embarrassingly parallel functions are the Velocity Verlet updates from Section 3.1.1, where each particle can be updated independently. Thrust also has efficient implementations for less embarrassingly parallel functions, such as adding and sorting. Listings 4.1 and 4.2 show how easy it is to switch from sorting on a CPU to sorting on a GPU using Thrust and CUDA.

Listing 4.1: Sorting in C++

```
//allocate space for n floats
thrust::host_vector<float> h_vec(n);
//fill the vector with random floats
thrust::generate(h_vec.begin(), h_vec.end(), random_float());
//sort the vector using the sort function from the C++ Standard Library
std::sort(h_vec.begin(),h_vec.end());
```

Listing 4.2: Sorting using Thrust (on CUDA)

```
//allocate space for n floats, both on the host and the device
thrust::host_vector<float> h_vec(n);
thrust::device_vector<float> d_vec(n);
//fill the host vector with random floats
thrust::generate(h_vec.begin(), h_vec.end(), random_float());
//copy the random floats from the host to the device
thrust::copy(h_vec.begin(), h_vec.end(), d_vec.begin());
//sort the vector using the sort function from Thrust
thrust::sort(d_vec.begin(), d_vec.end());
//copy the result back to host
thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
```

Table 4.2 compares the performance of the above sort implementations.

n	i7 2600K	GT520	GTX480	GT520 transfer	GTX480 transfer
10^2	0.003	0.353	0.512	0.024	0.043
10^4	0.488	0.820	0.772	0.049	0.262
10^6	68.300	26.378	3.044	1.695	2.651
10^8	8389.533	2231.09	173.854	134.290	229.979

Table 4.2: Sorting performance CPU vs GPU in milliseconds

From these results we can make some interesting observations. The first is that using a GPU is only beneficial for a large problem size, because the transfer between host and device is expensive; as Table 4.2 shows, the sorting takes less time than the transfers to and from the device for 10^8 values. The second is the large overhead on the GPU which increases with the complexity of the devices; this is why the slower GT520 outperforms the GTX480 on the smaller problems.

The CUDA Programming Guide recommends to use the GPU only if the computation is of a higher complexity than the transfer costs [35]. For instance, it is not recommended to do matrix additions on the GPU; the cost for transferring is $3N^2$ whereas the computation cost is only N^2, which gives a complexity ratio of $3N^2/N^2 = O(1)$. Matrix multiplication is more suitable since its computation cost is N^3, which gives a ratio of $O(N)$.

4.2.1 Functors

One step up in complexity is using Thrust algorithms on functors. A functor is a function object, i.e. a function with a state. Thrust already provides many basic functors, and for embarrassingly parallel algorithms writing functors is often straightforward; Listing 4.3 shows the functor for the second step of the Velocity Verlet algorithm (Algorithm 8). The functor describes the necessary operations for one particle, the `thrust::for_each` function then applies this code to all particles in parallel. For more information see the documentation in Appendix A.

Listing 4.3: Second step of Velocity Verlet using a functor.

```
struct integrate2_functor{
        float _dt;
        //constructor (same as constructor for classes)
        __host__ __device__ integrate2_functor(float dt) : _dt(dt) {}
        //definition of the () operator which allows this struct to be used
            as function
        template <typename Tuple>
        __host__ __device__ void operator()(Tuple t){
                //extract particle velocity and acceleration from tuple
                volatile float4 v4 = thrust::get<0>(t);
                volatile float4 a4 = thrust::get<1>(t);
                float3 v3 = make_float3(v4.x, v4.y, v4.z);
                float3 a3 = make_float3(a4.x, a4.y, a4.z);
                //the second velocity verlet step: v += 0.5*a*dt
                v3 += 0.5f * a3 * _dt;
                //store the new values back into tuple
                thrust::get<0>(t) = make_float4( v3, 0.0f);
                thrust::get<2>(t) = dot(v3,v3);
        }
};

//create a new instance of the functor
integrate2_functor integrate2_op(deltaTime);
//wrap raw C pointer to thrust::device_ptr, so thrust can handle them
thrust::device_ptr<float4> d_vel4((float4 *) vel);
thrust::device_ptr<float4> d_acc4((float4 *) acc);
thrust::device_ptr<float> d_tkin(kin);
//this for_each statement applies the integrate2_op instance of the
   integrate2_functor to every element in the range given by the two
   iterators.
//In this case we pass zip_iterators which 'zip' together multiple
   properties (velocity, acceleration, kinetic energy) of particles in a
   memory-coalesced way.
thrust::for_each(
```

```
        thrust::make_zip_iterator(thrust::make_tuple(d_vel4, d_acc4, d_tkin
            )),
        thrust::make_zip_iterator(thrust::make_tuple(d_vel4+numParticles,
            d_acc4+numParticles, d_tkin+numParticles)),
        integrate2_op
);
```

If the problem is not embarrassingly parallel, or Thrust is too slow, one needs to program so-called kernels. Programming kernels is much more complicated than using Thrust and requires a more detailed understanding of how CUDA works.

4.3 How CUDA works

As mentioned before, CUDA works by sending a parallel portion of the code to the GPU, which then processes it and returns the result. Figure 4.2 shows this schematically.

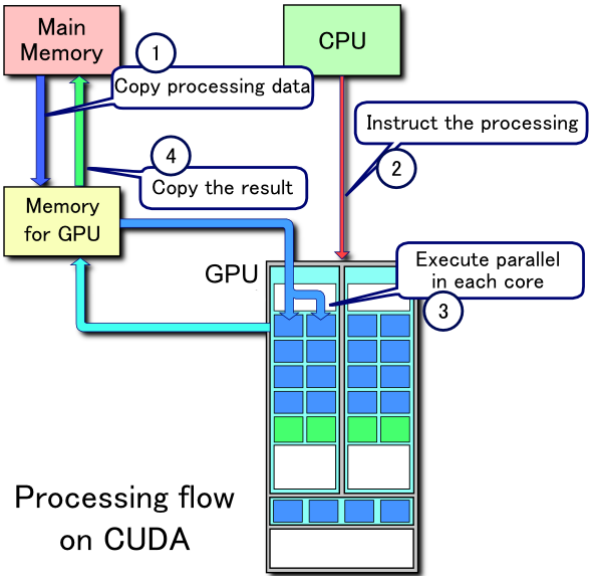


Figure 4.2: CUDA processing flow (modified from [36]).

The exact working of CUDA is very complex, but the following is a basic summary.[1]

4.3.1 Block-level parallelism

There are two levels of parallelism in CUDA: block-level parallelism (coarse-grained) and thread-level parallelism (fine-grained). Block-level parallelism is the easiest to explain: The code that is sent to the device is subdivided into blocks, which are then distributed among the Streaming Multiprocessors (SMs) of the device. Each device has a fixed number of SMs, where expensive cards have more SMs than cheap cards, and every SM can process one block at the time. This block-level parallelism also enables multiple cards to work on the same problem. Figure 4.3 shows this scalability.

[1]For more detail, have a look at the CUDA Programming Guide [35].

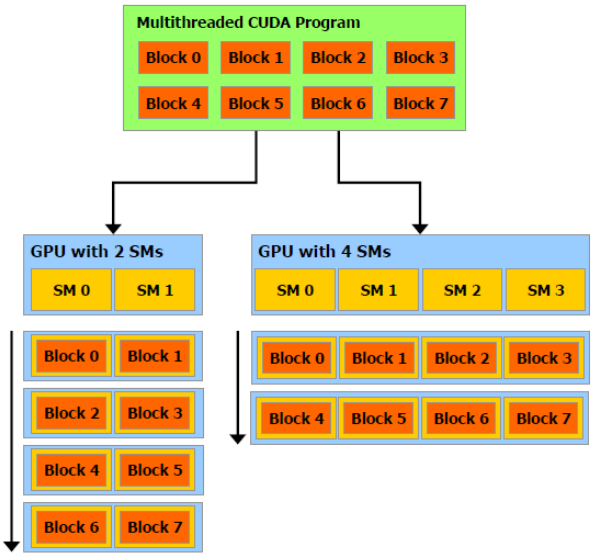


Figure 4.3: Automatic Scalability. The code is subdivided into 8 blocks, the first card has 2 SMs, so it can do 2 blocks at the time. The second card has 4 SMs so it can do 4 at the time. If they work together they can do 6 at the time [35].

4.3.2 Thread-level parallelism

Thread-level parallelism, the parallelism that happens within the block, is somewhat more complex. Each block consists of a number of threads, usually referred to as threads per block (TPB). This number, which must be set by the programmer, is application specific, but almost always a multiple of 32.[2] The number of threads per block determines the number of blocks. If you have for example 1024 particles, and you pick 256 threads per block you will have 4 blocks, if you pick 64 threads per block, you will have 16 blocks (assuming one thread per particle).

One of the reasons that the threads per block are almost always a multiple of 32, is that 32 is the so called warp size. The SM subdivides each block into warps. Then the warp scheduler(s) of the SM schedule(s) the warps for execution. How fast the SM processes the warp depends on the number of Streaming Processors (SPs) per SM. The total number of Streaming Processors, better known as CUDA cores, is often the most prominent property of the card advertised.[3]

[2]One can determine the TPB by trial and error, or (technical) by using the NVIDIA CUDA Occupancy Calculator (`http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`), which shows what percentage of the available processing power can be used with a given compute capability (the compute capability distinguishes different versions of the hardware architecture, see `http://stackoverflow.com/a/10961845/1439843` for more details), threads per block, shared memory per block, and registers per thread.

[3]The exact scheduling and execution is incredibly complex, and is described/discussed in more detail on the CUDA forums and Stack Overflow. Fortunately, detailed understanding of this is not necessary to write a reasonably fast CUDA program.

4.3.3 Memory

In terms of memory management, CUDA is also more complex than CPUs. In the CUDA device there are different kinds of memory with different properties, Table 4.3 and Figure 4.4 show these.

Memory	Location	Cached	Access	Scope	Lifetime
Register	On chip	n/a	r/w	1 thread	Thread
Local	Off chip	Yes†	r/w	1 thread	Thread
Shared	On chip	n/a	r/w	All threads in block	Block
Global	Off chip	Yes†	r/w	All threads + host	Host allocation
Constant	Off chip	Yes	r	All threads + host	Host allocation
Texture	Off chip	Yes	r	All threads + host	Host allocation

† Cached only on devices of compute capability > 2.0.

Table 4.3: Features of CUDA device memory [37].

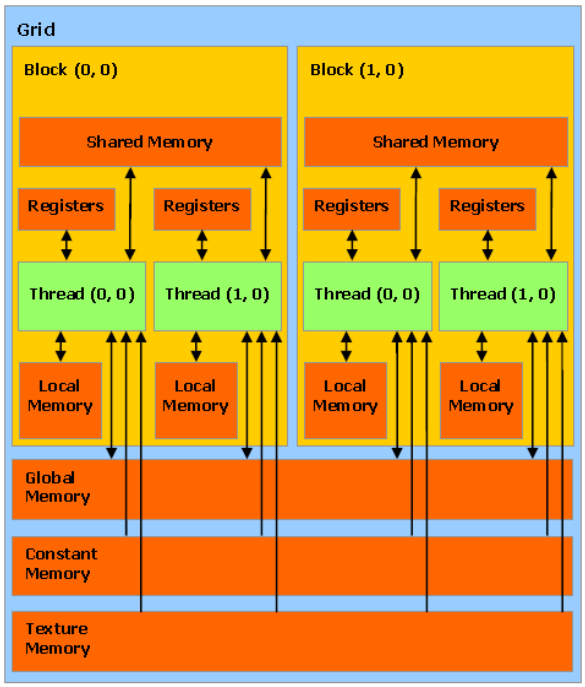


Figure 4.4: Schematic overview of the CUDA memory types [35].

The memory types that deserve special mention are global and shared memory. Global memory is the main memory of the device and since it is off-chip memory its performance and access latency are worse than that of shared memory (on-chip memory). When reading from global memory we need to make sure to coalesce memory access. Coalesced memory access means that each thread accesses a different memory location, and that the accessed memory region is aligned to the cache lines. According to the CUDA Best Practices [37], coalescing is the "single most important performance consideration in programming for CUDA-capable GPU architectures."

Figure 4.5 shows a coalesced access; all threads in the warp access a `float` (4-bytes) that is within one 128-byte cache line. Figure 4.6 shows a misaligned access: the threads in the warp now access values that are in two 128 byte cache lines. Both warps need to get 128 bytes of data, however, the second warp needs to load 256 bytes of data (two cache lines), cutting memory performance in half.

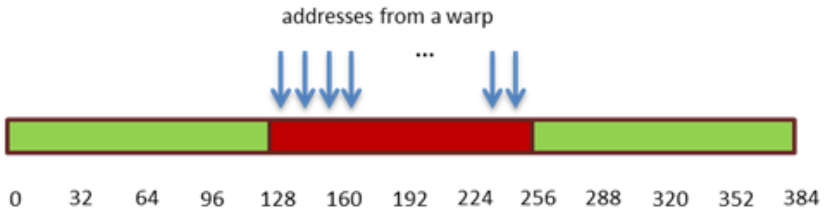


Figure 4.5: Aligned memory access [37].

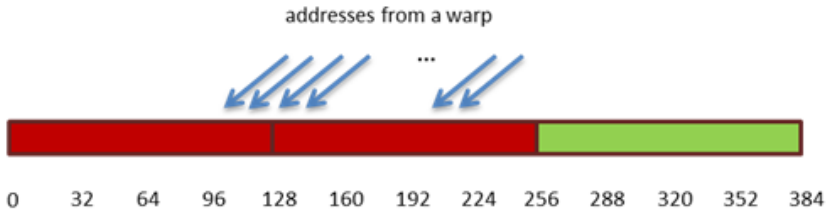


Figure 4.6: Misaligned memory access [37].

Some data structures introduced in CUDA are automatically aligned, such as `float2` and `float4`. This explains the common practice to store xyz-coordinates for three-dimensional simulations in `float4` instead of `float3`. A more in depth discussion of memory coalescing falls outside the scope of this thesis, and we refer the reader to the CUDA C Best practices guide [37].

Global memory also serializes simultaneous requests to the same memory location. So if a block has 32 threads, and all threads want to access the same value in global memory, 32 reads are done consecutively. In these cases it can be more efficient to transfer a portion of global memory to shared memory; shared memory is faster on-chip memory, and can broadcast the contents of a memory location to all threads in a block at the same time. The downside of shared memory is its limited size. We will see in Section 4.5.1 how we can use shared memory to avoid serialized global memory access in the N-body approach.

4.4 Programming: Kernels

Now that we understand the basics of CUDA we can discuss kernels. Listing 4.4 shows a kernel for computing the hash of all particles. Because this function is embarrassingly parallel it is an easy example of a kernel. In Section 4.5.1 and Section 4.5.2 on the N-body approach and the cell-list approach, we will see more complicated kernels.

A kernel can be recognized by the `__global__` specifier in front of it. A kernel is launched using the syntax `kernelName<<< numBlocks, numThreads, sharedMemSize >>>(parameter` The `numBlocks` and `numThreads` are the number of blocks and the number of threads per

block as discussed in Section 4.3. The `sharedMemSize` parameter is optional and needs to be specified if shared memory is used. A main difference between kernels and functors is that kernels are thread dependent; in a kernel every thread first calculates the particle it is responsible for and loads the corresponding particle data from memory, whereas Thrust passes the particle data to the functor using the `for_each` function. Listing 4.4 shows how a thread first computes its index, and then loads the position data at that index from the array containing the particle positions. The `blockIdx.x` is the number of the block, `blockDim.x` is the number of threads per block, and `threadIdx.x` is the number of the thread in the block. For example, if we have a thousand particles, divided in 4 blocks of 256 particles, then thread 17 of block 2 is responsible for particle $2 * 256 + 17 = 529$, and thread 255 of block 3 is responsible for particle 1023 which is ≥ 1000, so it not responsible for a real particle and the thread will do nothing.

Listing 4.4: Calculates and stores hash for all particles.

```
__global__ void calcHashD(uint   *gridParticleHash,
           uint   *gridParticleIndex,
           float4 *pos,
           uint    numParticles) {
        uint index = blockIdx.x * blockDim.x + threadIdx.x;

        if (index >= numParticles) return;

        //retrieve position of particle that the thread is responsible for
        volatile float4 p = pos[index];

        // get address in grid
        int3 gridPos = calcGridPos(make_float3(p.x, p.y, p.z));
        uint hash = calcGridHash(gridPos);

        // store grid hash and particle index
        gridParticleHash[index] = hash;
        gridParticleIndex[index] = index;
}

calcHashD<<< numBlocks , numThreads >>>(gridParticleHash,gridParticleIndex,
    (float4 *) pos, numParticles);
```

4.4.1 Device functions

Device functions, recognizable by `__device__`, can be used to store reusable pieces of code that are part of a kernel. In Listing 4.4 there are two of these functions, `calcGridPos()` and `calcGridHash()` (documented in Appendix A). If device functions have no CUDA specific code they can also be used on the host; these are recognizable by the `__host__ __device__` specifier.

4.5 Force calculation implementation

Section 3 showed that the force/potential calculations are the most computationally expensive part of the HMC and MD implementations. This section shows how we can

implement the force calculation more efficiently by using CUDA. First we will show the implementation for the N-body approach, suitable for long ranged interactions. Then we will show the cell-list approach which is more suitable for short-ranged interactions. The code fits in the framework that NVIDIA provides in their 'Particles' example code, and we reused a portion of this framework [38].

4.5.1 N-Body approach

The N-body system is very suitable for parallelization. This section gives a short overview of the algorithm introduced by Nyland, Harris, and Prins [39], and discusses some of our modifications.

Because we cannot exploit the symmetry of the force matrix on CUDA it is necessary to evaluate the full $N \times N$ matrix. A naive approach is calculating these values row-by-row. The rationale behind this is that if the data of particles is accessed simultaneously by multiple threads, CUDA will serialize these calls and there will be no performance improvement (see Section 4.3.3). Also, a row-by-row approach is easy to implement using the `for_each` statement in Thrust. This approach gives some performance improvement but the occupancy (the percentage of the available processing power used) is generally low, even for a large number of particles; computation on one row has to be finished before computation on the next row starts, even if the data needed for the first elements on the next row are not being accessed by threads working on the previous row.

The solution is to use shared memory, because shared memory can do a so called 'broadcast' in which it sends one value to all threads in the block. If we denote the number of threads in a block by p, instead of p sequential reads from global memory, only one thread reads the value from global memory and puts it in local memory. Because all threads can do this in parallel for one particle, we can load p particle values. After loading these values in shared memory, each row is done sequentially by one thread in the block. Note that all threads work on the same column in parallel because the value for that column is broadcast by the shared memory. Figure 4.7 visualizes this approach as a $p \times p$ computational tile. Each entry in the tile corresponds to an entry in the full $N \times N$ matrix.

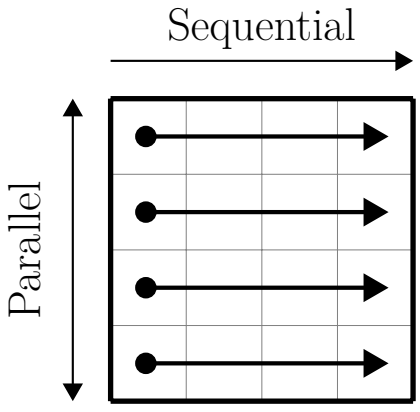


Figure 4.7: Schematic figure of computational tile (image reproduced from [39]).

Because there are N particles in total and p threads can load p values at a time, there

are N/p points where the threads need to load new particle data into shared memory, as Figure 4.8 shows.

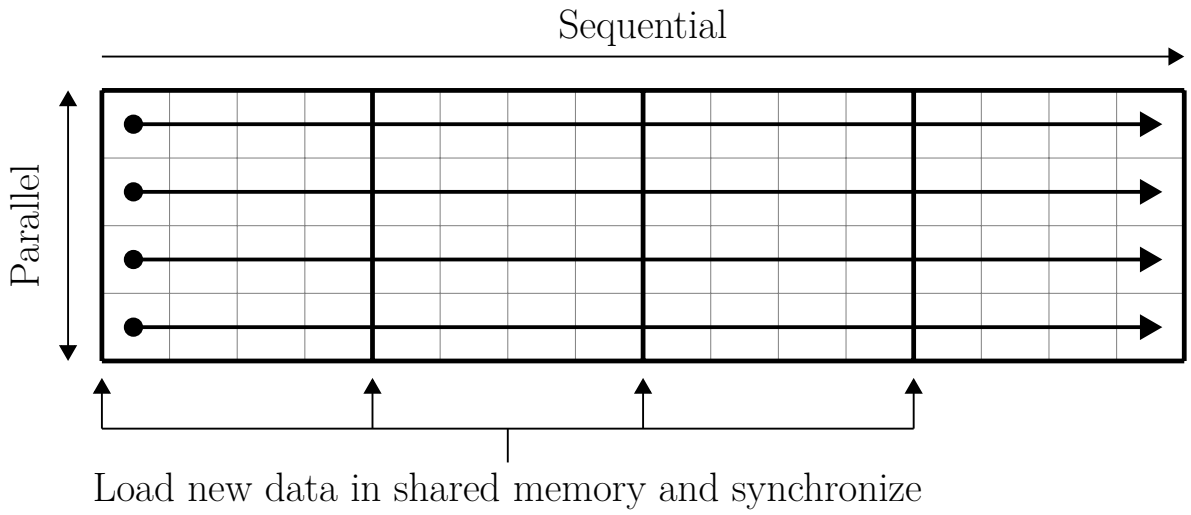


Figure 4.8: Computational row (image reproduced from [39]).

At these N/p points first all threads in the block synchronize, i.e. they wait until all threads have finished. The threads then load the values for the particles in the new tile, and synchronize again. Only then they start computing the entries in the next tile. The first synchronization step is necessary to avoid that threads overwrite particle data from the old computational tile before other threads finish computation on that data, and the second synchronization is necessary to avoid computation of the values in the new tile using data of the old tile.

Besides all threads in a block working in parallel (thread-level parallelism), there are also multiple blocks working in parallel (block-level parallelism), as Figure 4.9 shows.

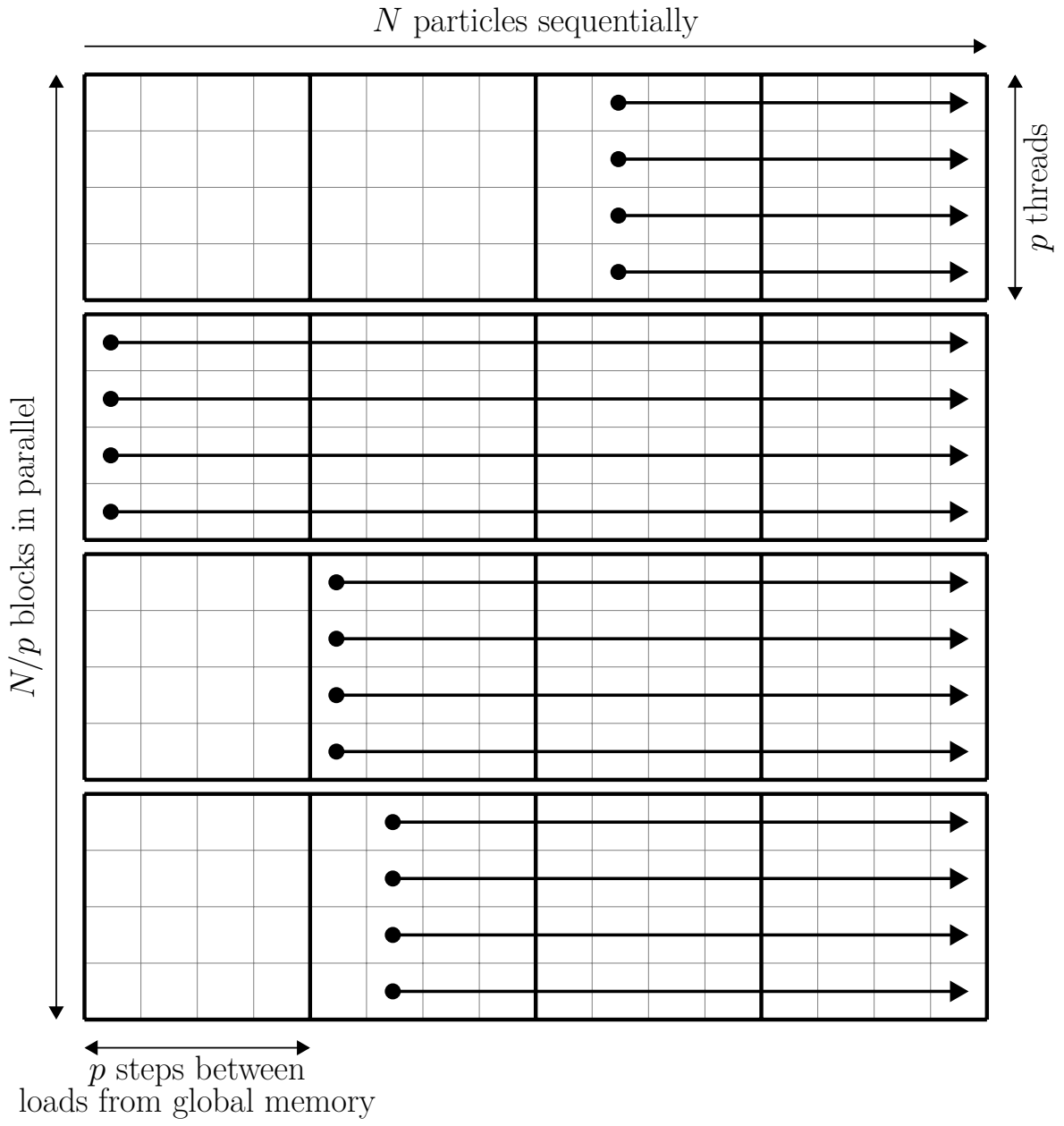


Figure 4.9: Computational grid (image reproduced from [39]).

Modifications

The above algorithm only works if the number of particles is a multiple of the number of threads per block, however, for some shapes the number of particles needs to be a multiple of a given number to avoid geometrical frustrations. For example, for triangles in a cubic box we need the number of particle to be a multiple of six. As a result we want to be able to pick an arbitrary number of particles. Our approach introduces dummy particles at infinite distance; that way they do not influence measurements, and no additional decision logic (overhead) is needed.

Implementation

Listing 4.5 shows the implementation for the N-body approach. The device function `tile_calculation()` does the calculation Figure 4.7 shows. The device function `ppInteraction()` computes the interaction between two particles based on their position, and returns the resulting acceleration. For dummy particles at distance infinity this acceleration will be zero.

In the for-loop in the kernel `calculate_forces_nbody()` you can see how the computation progresses tile by tile, and how in between tiles the threads load new values into shared memory as Figure 4.8 illustrates. Also the dummy values, discussed in Section 4.5.1, are set here. Note that the device function `tile_calculation()` does not know if a particle is a dummy particle, which saves time on decision logic and possible warp-divergence.[4]

Listing 4.5: Kernel for the force calculation using the N-body approach.

```
__device__ float4 tile_calculation(float4 myPos, float4 accel){
        extern __shared__ float4 shPos[];
        for( int i = 0; i < blockDim.x; ++i){
                accel += ppInteraction(myPos, shPos[i]);
        }
        return accel;
}

 __global__ void calculate_forces_nbody(float4 *d_x4, float4 *d_a4, float *
    virial, int numParticles){
        //compute global thread id
        int gtid = blockIdx.x * blockDim.x + threadIdx.x;
        //reserve some space in shared memory to store information that
           must be accessed by all threads at the same time
        extern __shared__ float4 shPos[];
        //reserve space to store own position (size is set in launch
           configuration)
        float4 myPos;
        //set acceleration to zero
```

[4]Technical: warps execute according to the SIMD model. SIMD stands for Single Instruction Multiple Data, meaning that all 32 threads in a warp execute the same instruction on different data. It is similar to SIMT (Single Instruction Multiple Threads), but different in how it handles conditionals; if the kernel includes if/else logic, some threads are temporarily disabled. An example is `if(ThreadIdx.x < 16) ... else ...` . In this case, the first half of the warp executes first while the second half waits, and then the second half executes while the first half waits. This is called warp-divergence, and slows down the program significantly (a factor two in this example).

```
        float4 acc = make_float4(0.0f);
        //if the thread is responsible for a real (not dummy) particle load
            its position
        if(gtid < numParticles) myPos = d_x4[gtid];
        //do calculation on a tile (the size of which is determined by the
           launch configuration)
        for( int tile = 0; tile < gridDim.x ; ++tile){
                //see which particle I am responsible for in this tile
                int idx = tile * blockDim.x + threadIdx.x;
                //if it is a real particle store its position in shared
                   memory
                if(idx < numParticles)  shPos[threadIdx.x] = d_x4[idx];
                //if it is a dummy set the particles position in shared
                   memory to inf
                else shPos[threadIdx.x] = make_float4(logf(0));
                //synchronize to make sure that we're al at the same point
                   of execution, needed for memory broadcast
                __syncthreads();
                //if responsible for a real particle calculate acceleration
                    due to the particles in the tile
                if(gtid < numParticles) acc = tile_calculation(myPos, acc);
                //sync again to ensure everyone is done with calculation
                   before storing new values in shared memory
                __syncthreads();
        }
        //if real particle store acceleration and virial
        if(gtid < numParticles) {
                d_a4[gtid] = acc;
                virial[gtid] = acc.w;
        }
}

//calls the kernel with the launch configuration
calculate_forces_nbody <<< numBlocks, numThreads, numThreads*sizeof(float4)
    >>>((float4 *) pos, (float4 *) acc, virial, numParticles);
```

4.5.2 Cell-list approach

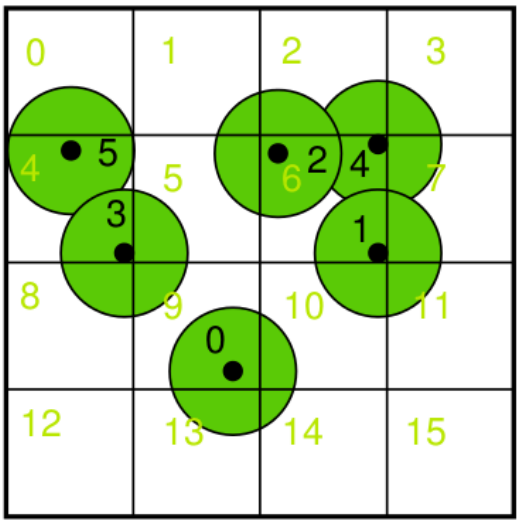
In case of short-range potentials many particle-particle interactions are negligible, and it is more efficient to use the cell-list approach.

Building the cell list

Fortunately, building a cell list is very efficient on a GPU thanks to Radix sort. Radix sort is a non-comparative integer sorting method with a complexity of $O(n)$ whereas traditional (sequential) sorting methods are all $O(n \log n)$.

We store the particles as id-hash pairs and then sort them on their hashes. The hash of the particle denotes in which cell the particle is, in the simplest case we use the linear cell id (Figure 6.4 shows an alternative hash, using the Z-order curve).

To get a functional cell list we need to store the begin and end for each cell, which can also be done in parallel by comparing every particle hash in the sorted list to the hash of the previous particle. If they are different there is a cell boundary. Figure 4.10 shows how to build the cell list based on the particle hashes.



Index	Unsorted list (cell id, particle id)	List sorted by cell id	Cell start
0	(9, 0)	(4, 3)	
1	(6, 1)	(4, 5)	
2	(6, 2)	(6, 1)	
3	(4, 3)	(6, 2)	
4	(6, 4)	(6, 4)	0
5	(4, 5)	(9, 0)	
6			2
7			
8			
9			5
10			
11			
12			
13			
14			
15			

Figure 4.10: Building the cell list using sorted hashes. The last column shows for each cell the index of its first particle in the sorted list [40].

Implementation

The implementation for the cell-list approach is similar to the CPU cell-list implementation. The main differences are that we sort the particles on their hash to increase memory coalescence, and that we do everything in parallel.

Listing 4.6 shows the sorting and reordering which we do after calculating the hash for all particles (Listing 4.4). Listing 4.7 shows how the force calculation then happens in parallel. Note that the function `ppInteraction` is the same device function we used in the N-body approach in Listing 4.5. The implementation for the potential energy calculation needed in NPT MC is almost identical to the force calculation implementation.

Listing 4.6: Sorting and reordering the particle data.

```
//stores the permutation that would make the particles sorted
thrust::sort_by_key(thrust::device_ptr<uint>(dGridParticleHash), thrust::
    device_ptr<uint>(dGridParticleHash + numParticles), thrust::device_ptr<
    uint>(dGridParticleIndex));

//the actual reordering including finding the cell boundaries

__global__
void reorderDataAndFindCellStartD(uint *cellStart, uint *cellEnd, float4 *
    sortedPos, uint *gridParticleHash, uint *gridParticleIndex, float4 *
    oldPos, uint numParticles) {

        // blockSize + 1 elements extern __shared__ uint sharedHash[];
    uint index = blockIdx.x * blockDim.x + threadIdx.x;
    uint hash;
    // handle case when no. of particles not multiple of block size
    if (index < numParticles)
    {
        hash = gridParticleHash[index];
        // Load hash data into shared memory so that we can look
        // at neighboring particle's hash value without loading
        // two hash values per thread
        sharedHash[threadIdx.x+1] = hash;
        if (index > 0 && threadIdx.x == 0)
        {
            // first thread in block must load neighbor particle hash
            sharedHash[0] = gridParticleHash[index-1];
        }
    }
    __syncthreads();
    if (index < numParticles)
    {
        // If this particle has a different cell index to the previous
        // particle then it must be the first particle in the cell,
        // so store the index of this particle in the cell.
        // As it isn't the first particle, it must also be the cell end of
        // the previous particle's cell
        if (index == 0 || hash != sharedHash[threadIdx.x])
        {
            cellStart[hash] = index;
            if (index > 0)
                cellEnd[sharedHash[threadIdx.x]] = index;
        }
        //last hash is always the end of a cell
        if (index == numParticles - 1)
        {
            cellEnd[hash] = index + 1;
        }
        // Now use the sorted index to reorder the pos data
        uint sortedIndex = gridParticleIndex[index];
        float4 pos = oldPos[sortedIndex];
        sortedPos[index] = pos;
    }
```

```
}

//calls the kernel
uint smemSize = sizeof(uint)*(numThreads+1);
reorderDataAndFindCellStartD<<< numBlocks, numThreads, smemSize>>>(
    cellStart, cellEnd, (float4 *) sortedPos, gridParticleHash,
    gridParticleIndex, (float4 *) oldPos, numParticles);
```

Listing 4.7: Kernel for the force calculation using N-body approach.

```
__device__ float4 calculateForcesCell(int3 gridPos, uint index, float4 pos,
    float4 *sortedPos, uint *cellStart, uint *cellEnd) {

    uint gridHash = calcGridHash(gridPos);
    //get start of bucket for this cell
    uint startIndex = cellStart[gridHash];
    float4 acc = make_float4(0.0f);

    if(startIndex != 0xffffffff) { //cell is not empty
            //get end of bucket for this cell
            uint endIndex = cellEnd[gridHash];
            //iterate over particles in this cell
            for( uint j = startIndex; j < endIndex; ++j){
                    if( j != index ) { //avoid self
                            float4 pos2 = sortedPos[j];
                            acc += ppInteraction(pos, pos2);
                    }
            }
    }
    return acc;
}
 __global__ void calculate_forces_celllist (float4 *d_a4, float *virial,
    float4 *d_x4, uint *gridParticleIndex, uint *cellStart, uint *cellEnd,
    uint numParticles) {

    uint index = blockIdx.x * blockDim.x + threadIdx.x;
    if(index >= numParticles) return;
    //read particle data from sorted arrays
    float4 pos = d_x4[index];
    float4 acc = make_float4(0.0f);
    int3 gridPos = calcGridPos(make_float3(pos));
    //examine neighboring cells
    for(int z = -1; z <= 1; ++z) {
            for(int y = -1; y <= 1; ++y) {
                    for(int x = -1; x <= 1; ++x) {
                            int3 neighborPos = gridPos + make_int3(x,y,z);
                            acc += calculateForcesCell(neighborPos, index,
                                pos, d_x4, cellStart, cellEnd);
                    }
            }
    }
    //write new acceleration back to original unsorted location
    uint originalIndex = gridParticleIndex[index];
    d_a4[originalIndex] = acc;
```

```
    virial[originalIndex] = acc.w;
}

//calls the kernel with the right launch configuration
calculate_forces_nbody <<< numBlocks, numThreads, numThreads*sizeof(float4)
    >>>((float4 *) pos, (float4 *) acc, virial, numParticles);
```

4.6 Additional implementation details

There are some other aspects of the CUDA implementation that deserve special mention.

The first is the random number generation. Although CUDA random number generators exist, we chose CPU random number generation for simplicity. Every cycle we generate a new set of numbers on the CPU and then copy it to the device.

The second is what should happen if we reject a move. In the CPU implementation, it is common practice that if a move is rejected, the equal but opposite move is applied, e.g. if in a volume move all particle positions are multiplied by a scaling factor a, then if the move is rejected the particle positions are divided by that same scaling factor a. This does not work on CUDA! Because CUDA works in single precision there is no guarantee that $x * a/a = x$. So while you intend to undo the move and restore the old situation you will still end up with a new situation. This will break the simulation, make a backup of your configuration before every move, so that it can be restored if the move is rejected.

Chapter 5

Results

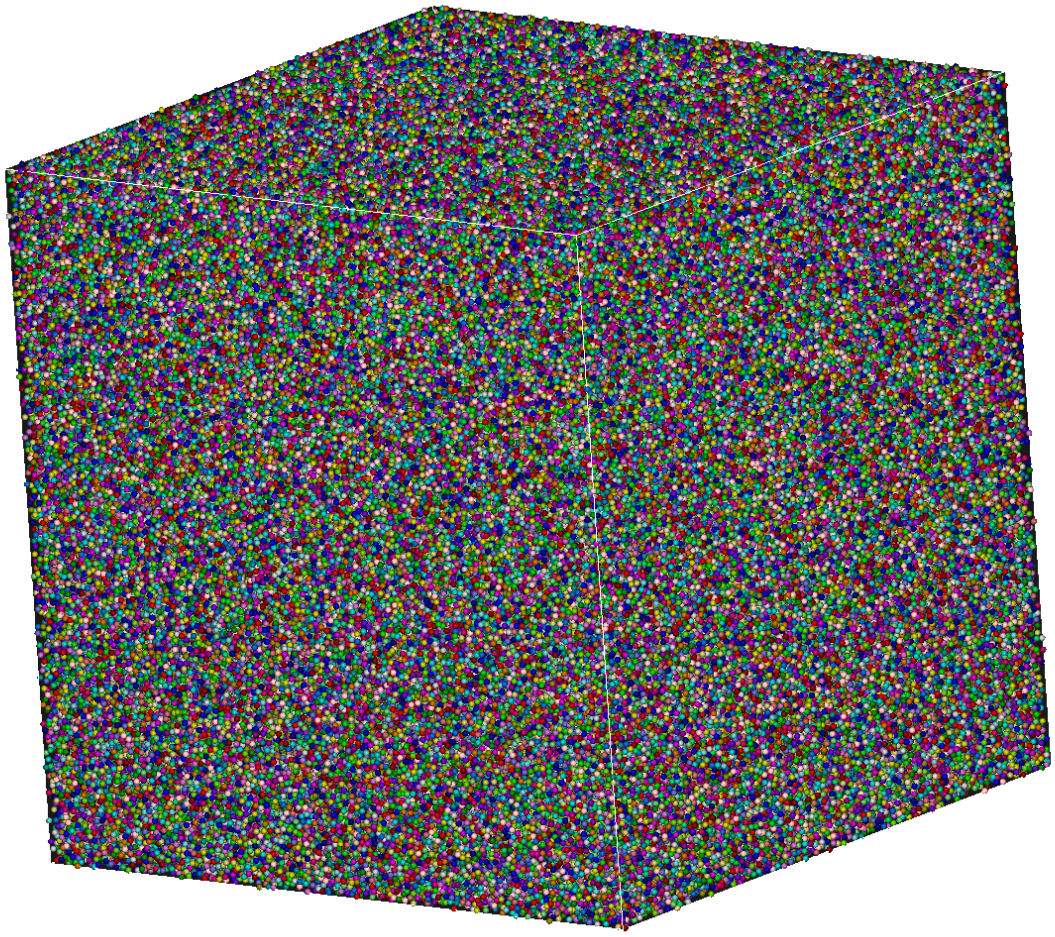


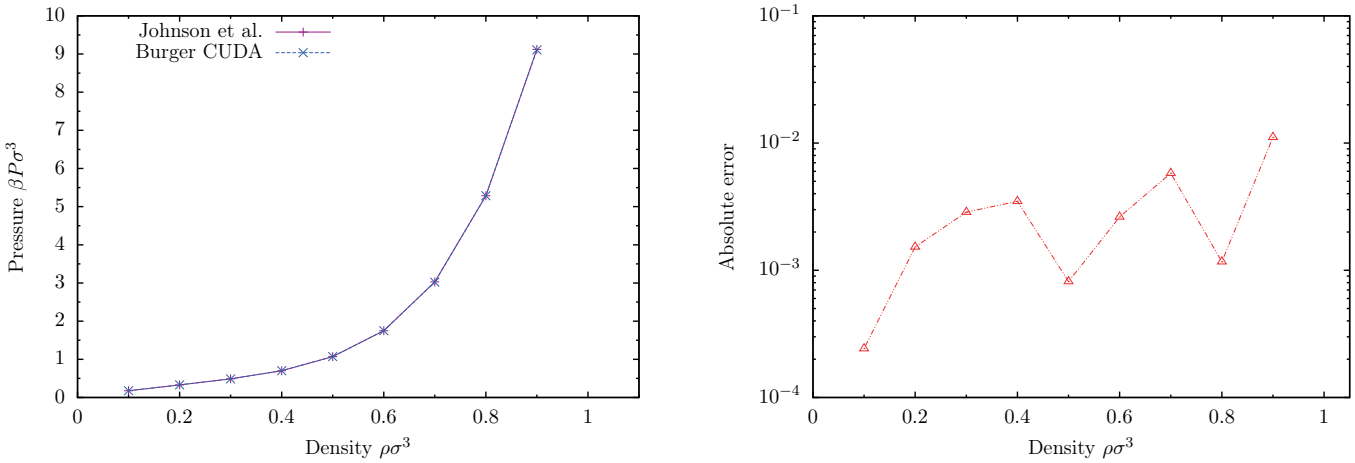
Figure 5.1: Snapshot of a Lennard-Jones MD simulation in the NVT ensemble with a million particles.

5.1 Molecular Dynamics

To test the correctness of our implementation we try to reproduce the equation of state by Johnson, Zollweg, and Gubbins [28]. We start with the MD implementation using the system specifications in Table 5.1, and as Figure 5.2 shows, the results are consistent.

CUDA MD NVT	
Particles	1024
Potential	LJ tr-sh
Temperature	2.0
MD time step	0.003
Equilibration cycles	15,000
Measurement cycles	15,000

Table 5.1: System specifications for the results in Figure 5.2.



(a) Equation of state

(b) Absolute error

Figure 5.2: Comparison of the equation of state obtained by our CUDA MD NVT and the equation of state by Johnson, Zollweg, and Gubbins [28].

5.1.1 Performance

Force calculation: N-body

MD simulations on CUDA improves performance dramatically; Figure 5.3 shows that for a single force computation using the N-body approach, the GT520, a cheap CUDA card, outperforms the CPU by an order of magnitude for a system size of only a hundred particles.[1][2] CUDA performance gets even better when the system size increases; for a million particles the GTX480 is almost two orders of magnitude faster.

[1] See Section 4.1 for the hardware overview.

[2] Implementation note: do not time CUDA functions using CPU timers; CUDA kernels launch asynchronously, i.e. control is immediately passed back to the CPU so that it can continue executing the

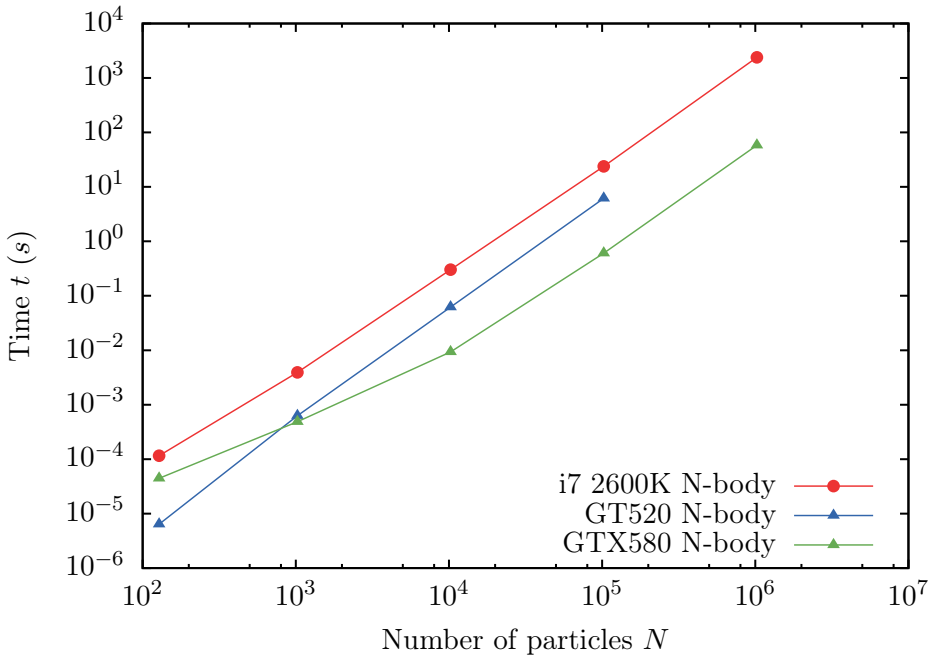


Figure 5.3: Performance for the force calculation in the CUDA MD N-body implementation. For a million particles the GT520 ran out of memory.

Force calculation: Cell lists

Figure 5.4 shows that the cell-list implementation is even several orders of magnitude faster. In our CPU implementation for a system size of a million particles, cell lists made the force calculation a thousand times faster. This is still a prudent estimate; we tested the cell-list code at density $\rho\sigma^3 = 0.9$, setting the density lower makes the cell-list code even faster. Although the CPU code is already fast, our CUDA implementation is forty times faster. Also it is clear from the figure that the cell-list implementation is only useful for larger system sizes because of the additional overhead.

First Velocity Verlet step

The force computation is the most computationally intensive part of the implementation, which is why the implementation benefits most from an improvement to the force calculation. Nevertheless, it pays to improve other functions as well. Figure 5.5 shows the increased performance for the first Velocity Verlet integration step. For a million particles, the GTX580 GPU beats the Core i7 CPU by almost a factor thirty.

General performance

To illustrate just how fast our CUDA implementation is, assume a system size of a million particles. Then it is possible to

rest of the code (asynchronous execution). While the CUDA simulation is still running the CPU will stop the timer and it will appear that the CUDA code is extremely fast. Time using `CudaEventRecords` instead, see Appendix A for an implementation.

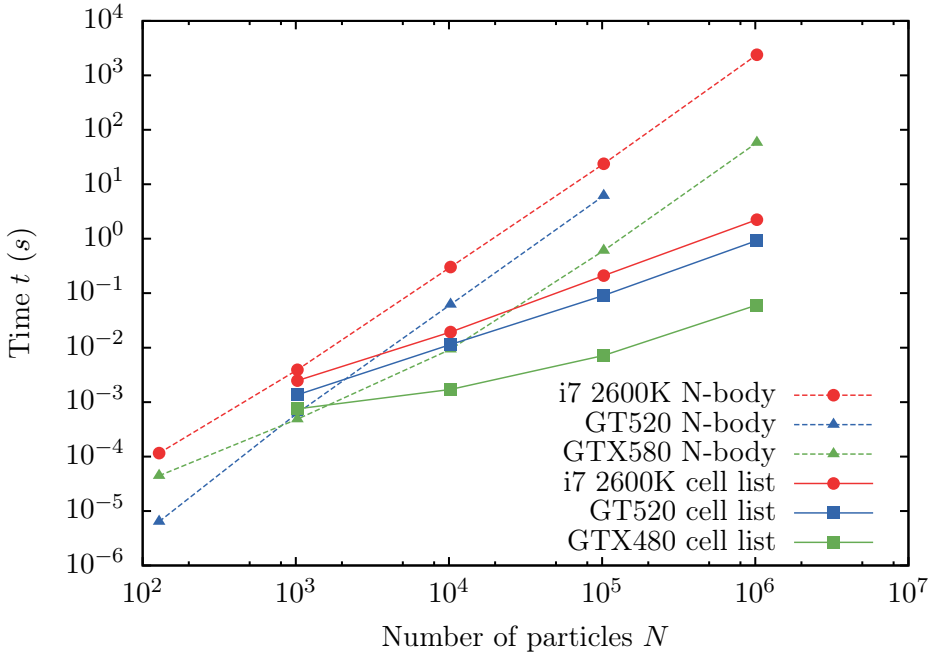


Figure 5.4: Performance for the force calculation in the CUDA MD Cell-list implementation.

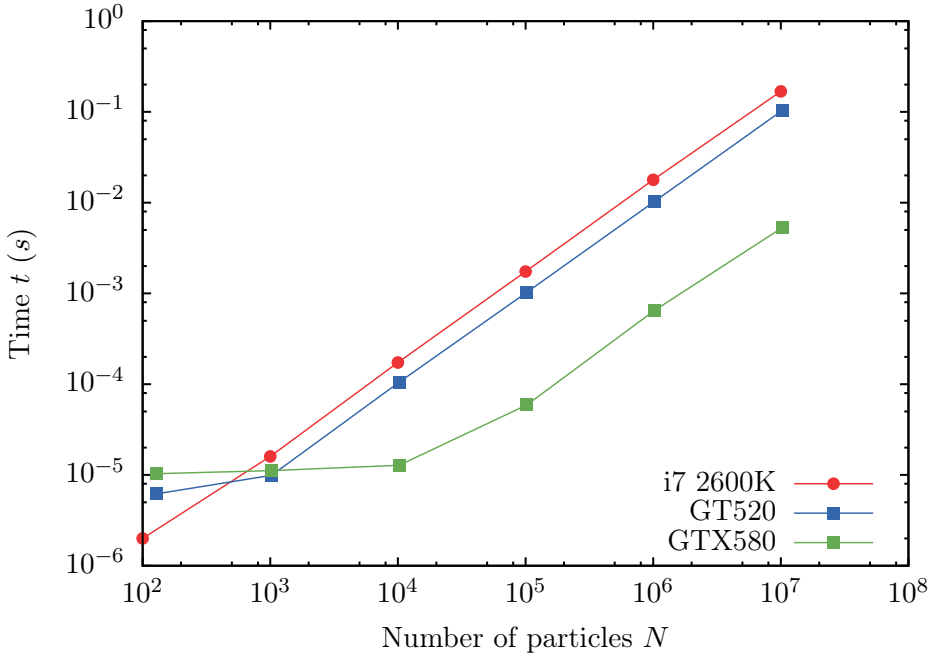


Figure 5.5: Performance for the first Velocity Verlet integration step in the CUDA MD implementation.

- do one iteration at high density in one minute using the N-body approach,
- do a thousand iterations at high density in 1 to 1.5 minutes using the cell-list approach,
- create an accurate equation of state, using 10 data points with 30,000 cycles each, in approximately 6 hours.

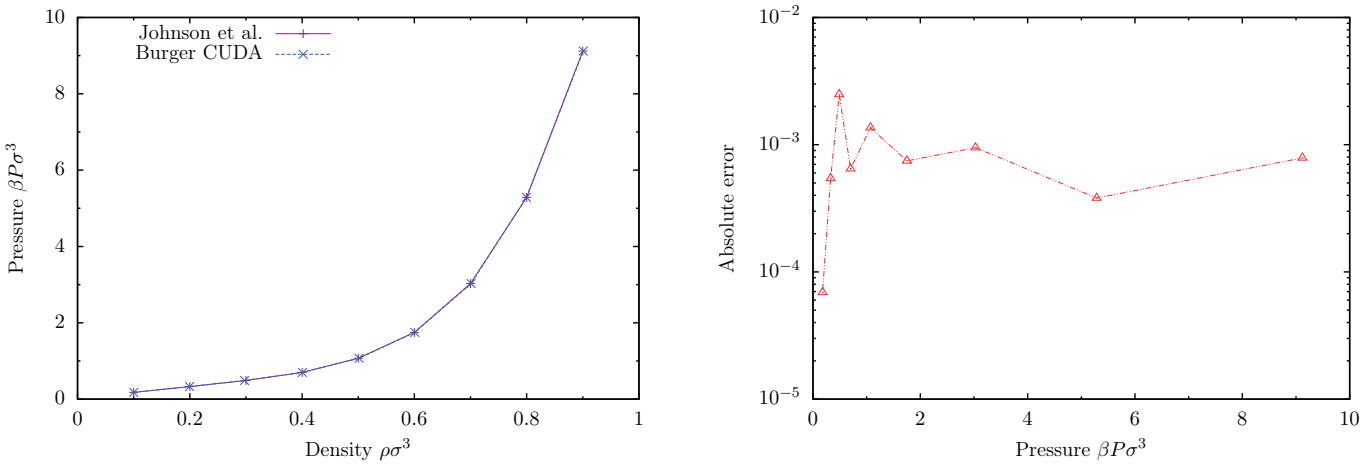
5.2 Hybrid Monte Carlo

We also tested our CUDA HMC implementation using the system specifications in Table 5.2. Using this system we obtained the equation of state in Figure 5.6 which also agrees with the reference equation of state by Johnson, Zollweg, and Gubbins [28].

CUDA MC NPT	
Particles	1024
Potential	LJ tr-sh
Temperature	2.0
MD time step	0.003
MD steps	10
Volume move prob.†	0.9
Equilibration cycles	15,000
Measurement cycles	15,000

† See Section 5.2.1.

Table 5.2: System specifications for the results in Figure 5.6.



(a) Equation of state

(b) Absolute error

Figure 5.6: Comparison of the equation of state by Johnson, Zollweg, and Gubbins and the CUDA MC NPT implementation [28].

As before, we also computed the virial pressure and as Table 5.3 shows, the imposed and virial (measured) pressures align.

5.2.1 Performance

We measured the performance of our MD code by comparing the force calculation in both the CPU and the CUDA implementation. For Hybrid Monte Carlo it is more difficult to compare performance, because we need to compare it to regular NPT MC, which has a

$\beta P\sigma^3$	$\langle \beta P_{\text{HMC\_CUDA}} \sigma^3 \rangle$
0.1776	0.177739
0.329	0.328452
0.489	0.487456
0.700	0.707204
1.071	1.081920
1.75	1.754475
3.028	3.047251
5.285	5.287418
9.12	9.126866

Table 5.3: Comparison imposed and measured pressure for the CUDA NPT implementation.

fundamentally different trial move. As Section 3 explains, in Hybrid Monte Carlo a trial move consists of several MD moves, whereas in regular NPT a trial move displaces just one particle. Because of this difference we have timed the complete moves instead of just the force/potential calculation, with the system specified in Table 5.4.

CUDA MC NPT	
Potential	LJ tr-sh
Temperature	2.0
Pressure	10
Starting density	0.5
MD time step	0.003
MD steps	10
Equilibration cycles	0
Measurement cycles	1,000†

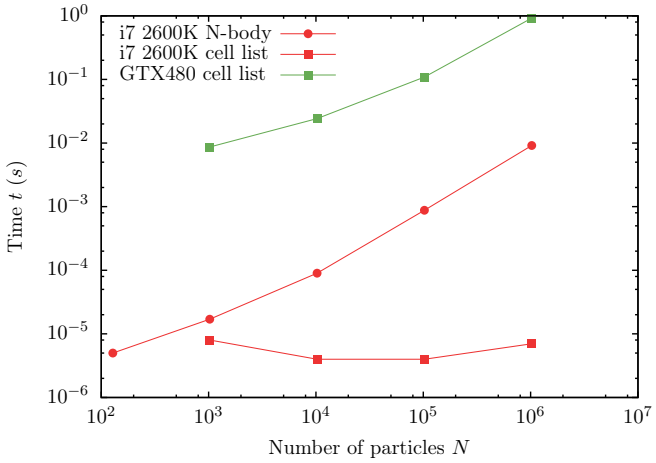
† Unless this resulted in prohibitively long running times.

Table 5.4: System specifications for the results in Figure 5.7.

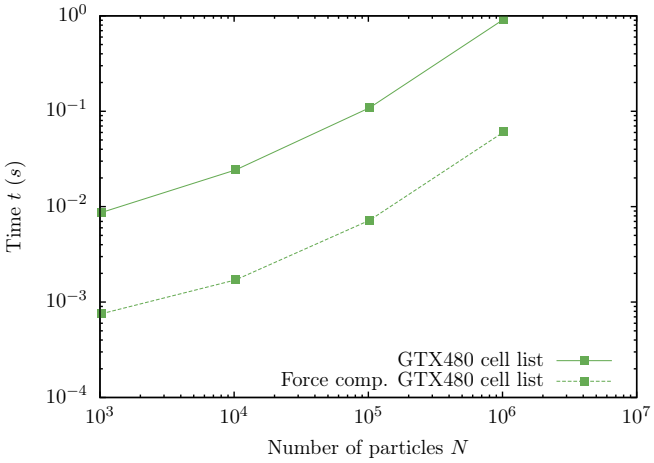
As Figure 5.7a shows, the CUDA trial move is much slower than the CPU trial move. Considering the results for the force computation in Figure 5.4 these results are in line with our expectations; one trial move in CUDA consists of 10 MD steps plus a potential energy calculation, and as Figure 5.7b shows, one iteration takes approximately 11 times the time required for a force/potential computation. In contrast, Figure 5.7c shows that volume moves are faster on CUDA, since for both the CPU and CUDA implementation it consists of a single potential calculation.

To get a more accurate idea of the performance, Figure 5.7d shows the weighted average of an iteration in both CPU MC and CUDA HMC NPT.

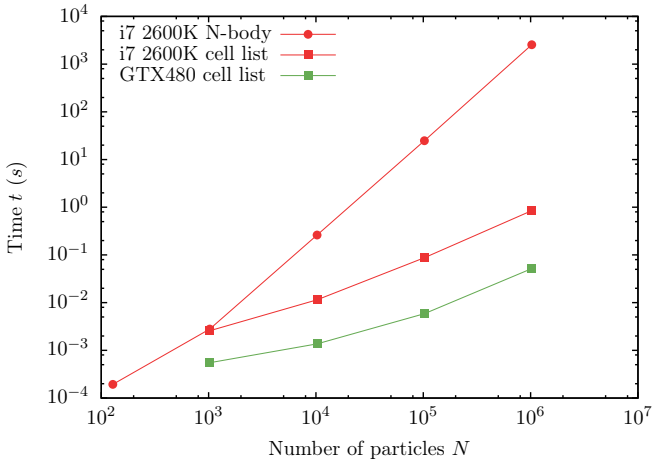
This looks bad: the average iteration on the CPU is more than three orders of magnitude faster, but actually it is not that bad. Figure 5.8 shows that the weights for trial



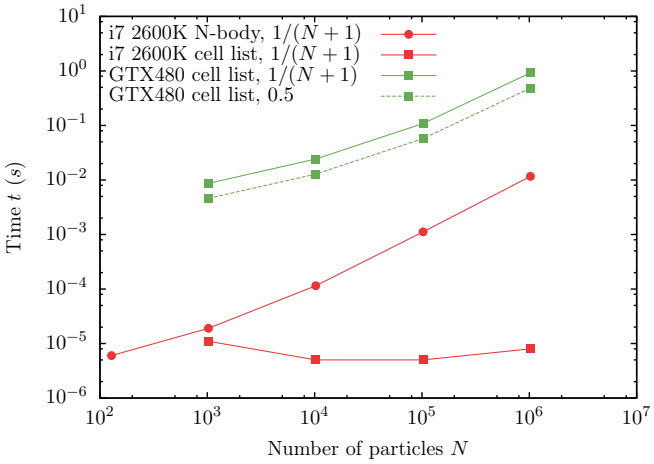
(a) Trial move



(b) Trial move vs. force computation



(c) Volume move



(d) Weighted move

Figure 5.7: Performance results for the CUDA HMC NPT implementation. (d) shows the weighted move depending on the volume move probability.

and volume moves are reversed on CUDA, and this allows us to fundamentally change the Monte Carlo algorithm.

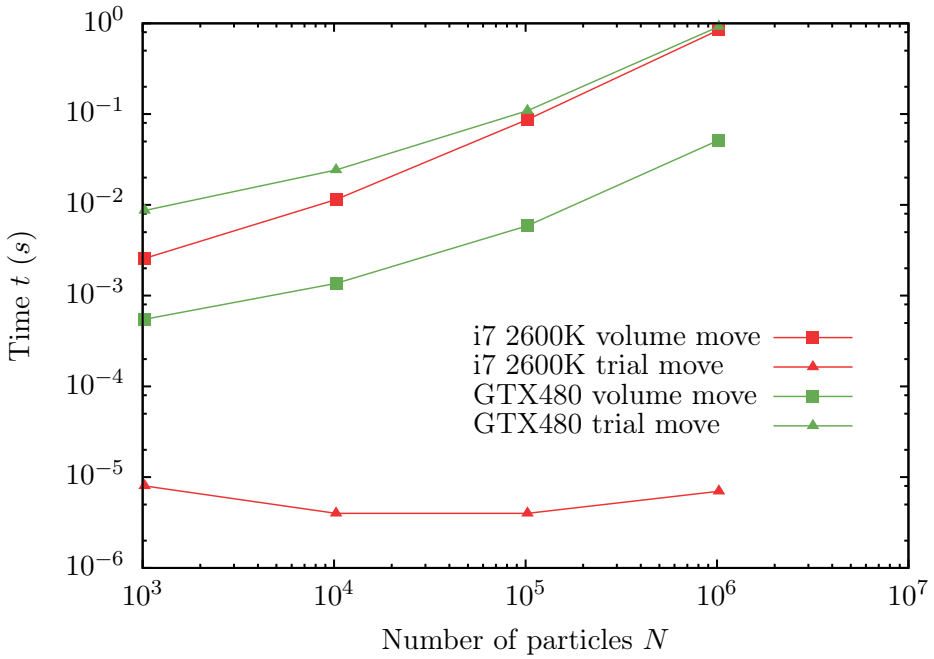


Figure 5.8: Comparison trial and volume moves.

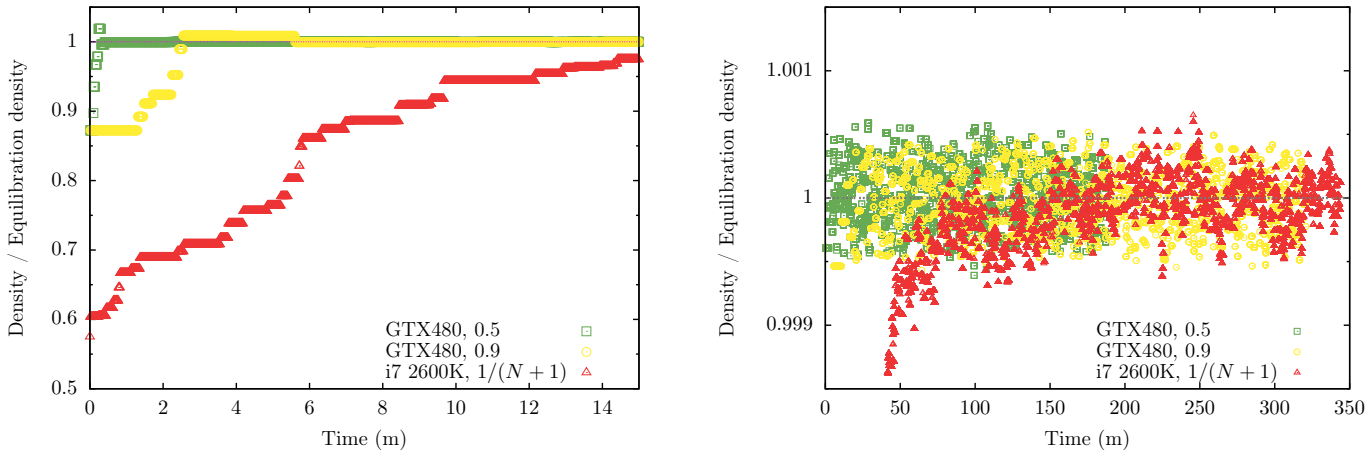
In a regular NPT MC implementation the probability of a volume move is recommended to be $1/(N+1)$; after a failed volume move, there needs to be a large number of trial moves before it is likely that the particles that are responsible for the failed move are updated. It is undesirable to do a volume move too soon because it is expensive.

In contrast, the HMC implementation updates all particles at the same time in the MD steps. This allows us to increase the probability of volume moves without decreasing the acceptance ratio. Unfortunately, even with the extreme volume move probability of 0.5 this does not significantly improve the weighted average in Figure 5.7d, but it makes equilibration in HMC much faster. Figure 5.9 shows how fast the systems equilibrate for different volume move probabilities. Table 5.5 shows the specifications of the used systems.

As Figure 5.9 shows, the CPU implementation has trouble compressing the system as the density increases. The closeup in Figure 5.9b shows that the CPU implementation only reaches its equilibration value after approximately 2.5 hours, whereas the CUDA implementation equilibrates in less than 10 minutes. Using the increased volume probabilities we can reduce the number of cycles by several orders of magnitude. As a consequence the CUDA implementation is at least an order of magnitude faster. Using compression to avoid the long equilibration for new data points can reduce the time needed for an accurate equation of state even further.

CUDA HMC NPT		CPU MC NPT	
Particles	1,024,000	Particles	1,024,000
Potential	LJ tr-sh	Potential	LJ trunc
Temperature	2.0	Temperature	2.0
MD time step	0.003	Cycles	2 10^9
MD steps	10		
Cycles	20,000		

Table 5.5: System specifications for the results in Figure 5.9.



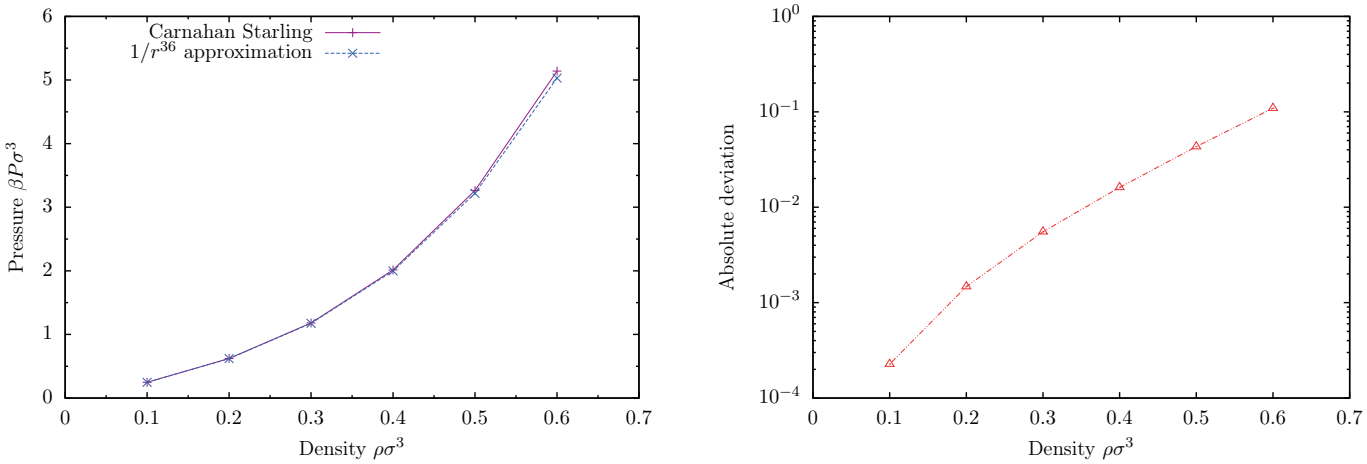
(a) Equilibration (b) Close up

Figure 5.9: Equilibration for HMC compared with normal NPT MC.

Chapter 6

Discussion

Following our original aim, we tried to approximate the hard-disk system using the $1/r^{36}$ potential by Sanz and Marenduzzo [41]. This gives reasonable results for the fluid phase, but the results deviate from those of the hard spheres as the density of the system increases, as Figure 6.1 shows.



(a) Equation of state

(b) Deviation

Figure 6.1: Comparison of the equation of state obtained with our CUDA code using the $1/r^{36}$ potential and the hard-sphere equation of state (Carnahan Starling).

6.1 To two-dimensional hard-particle systems

From the current state of the code, we need to take the following steps to get to get to our original aim: a functional parallel two-dimensional hard-particle Monte Carlo implementation.

- Switch to two dimensions:
  The current code models a three-dimensional system because going one dimension down is easier than going one up. Alternatively, use spheres in a monolayer, as done by Qi, Gantapara, and Dijkstra [20].

- Implement hard-particle MD simulation:
  Implementing parallel hard-particle MD simulation is a challenge in itself. It is hard to beat the very efficient CPU implementation by Rapaport, an implementation which is particularly hard to parallelize because it uses binary trees. Yet we do need to use the GPU for the MD part because transferring the state from and to the GPU to perform the MD part on a CPU is expensive and negates all performance improvements achieved by doing some steps in parallel. An alternative implementation which is also easier to parallelize is the free-flight MD. If a collision occurs in free-flight MD the system rewinds to the point when the collision began [25].

  The following items need to be implemented:

  - Rewind on collision (free-flight MD, most work),
  - Include rotation.

- Implement floppy-box HMC:
  Whereas the floppy box method is difficult to include in a MD simulation, it is actually not that difficult to include in HMC because it is inserted in the MC part and not in the MD part.

6.2 Optimizations

It is also still possible to improve on the existing MD code by at least one order of magnitude. We know this because HOOMD (Highly Optimized Object-oriented Many-particle Dynamics) by Glotzer et al. is approximately a factor five faster, as Figure 6.2 shows [42].

Any improvements in the MD implementation will directly translate to an improvement in the HMC implementation, because MD constitutes most of the HMC code.

Figure 6.3 shows the breakdown of the running time for the HMC code with the system specifications in Table 6.1.

CUDA MC NPT	
Particles	1024
Potential	LJ tr-sh
Temperature	2.0
Pressure	10
MD time step	0.003
MD steps	10
Cycles	10,000

Table 6.1: Configuration for the profile in Figure 6.3.

If the system size increases the force/potential calculation (the MD part) will take up even a bigger percentage, so any improvement will be significant; force/potential calculation are done on average 11 times for each trial move!

Examples of possible improvements:

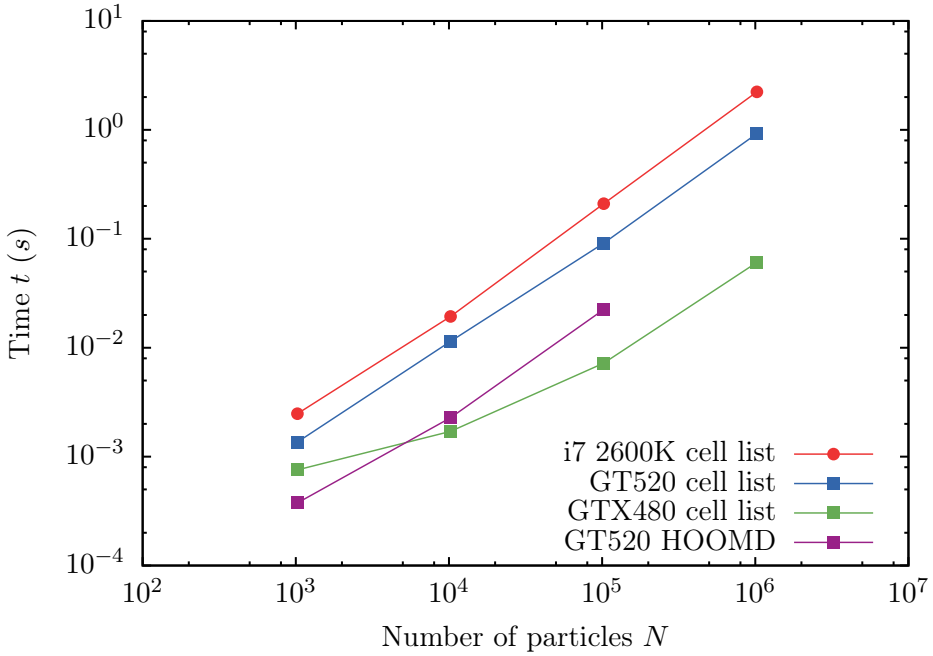


Figure 6.2: Performance compared with HOOMD [42].

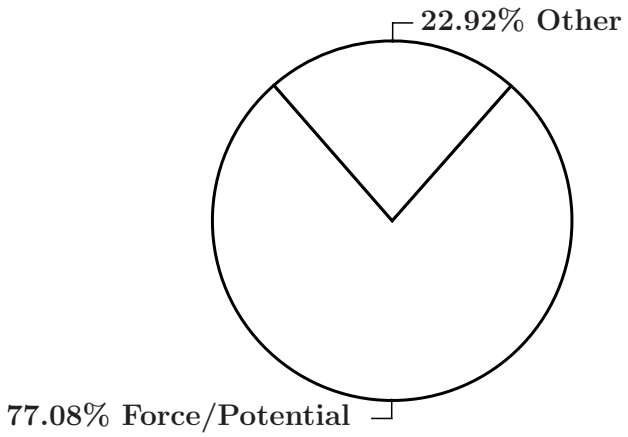
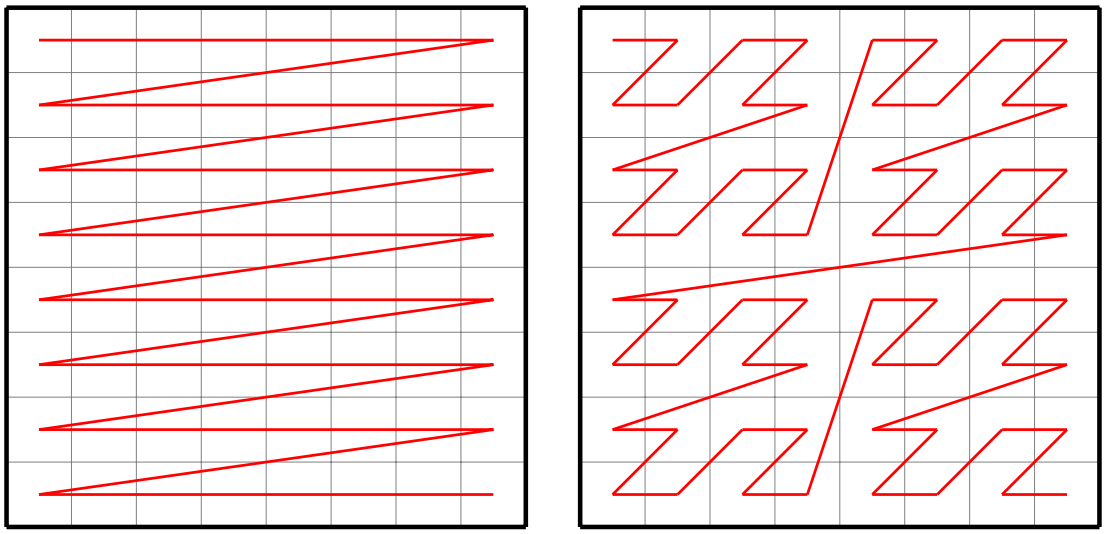


Figure 6.3: Profile of the CUDA HMC implementation.

- Asynchronous concurrent execution:
  Use the GPU and CPU at the same time. For example, currently the code still relies on the CPU to create the random numbers for the new velocities in the MD code. Currently we do this sequentially; the GPU first finishes its operations, then the CPU starts generating random numbers and sends them to the GPU. Using asynchronous concurrent execution it is possible to generate and send these random numbers to the GPU while it is still processing. Alternatively, generating these random numbers on the GPU could also be an improvement.

- Texture fetch
  Because memory access for the neighboring cells in the cell-list approach is non-coalesced, it is beneficial to bind the particle data in global memory to texture memory and use texture lookups. Because these texture lookups are cached, there will be an expected performance increase of 45% [35, 40].

- Z-order curve hashing
  For more coherent memory access it might be more beneficial to use for example a Z-order curve (such that the particles that are close together in the simulation are also close together in memory). Figure 6.4 shows the difference between a hash using linear cell id and using a Z-order curve for the two-dimensional case. As Figure 6.4 shows the cells that are close together, or also closer on the line, and therefore closer in memory. This means that on average we need to load fewer cache lines, which improves the performance of the program.[1]



(a) (b)

Figure 6.4: Hashing using linear cell id (a), or Z-order curve (b).

[1] See Section 4.3.3 on memory coalescing.

6.3 Possible extensions/applications

It is possible to extend the code to study, for example, wetting phenomena and critical Casimir forces [43]. The code is flexible but because HMC uses MD there are limitations on the potentials that can be used. For instance, the regular Lennard-Jones potential is not differentiable, and you have to use the truncated and shifted potential to have energy conservation [15]. Implementing these extensions should be straightforward but debugging CUDA applications can be troublesome, as the code is quite complex.

6.4 Alternatives

Instead of using HMC to parallelize there are some other options available to speed up particle simulations.

The first is the Event Chain Monte Carlo method by Bernard, Krauth, and Wilson [44]. In this method a chain of particles is displaced in a rejection-free manner, instead of randomly displacing a single particle. This is achieved by moving the particle until it collides with another particle, then this other particle is moved until it collides, et cetera. There are two variants of this approach: the Straight Event Chain (SEC) where all particles in a chain move in the same direction, and the Reflected Event Chain (REC) where the refracted angle is equal to the incident angle. Figure 6.5 shows these variants.

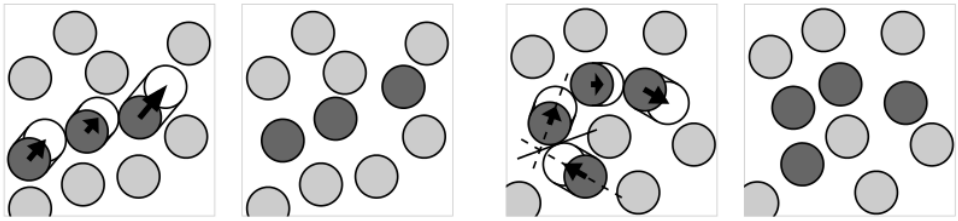


Figure 6.5: Left two panels: straight event-chain (SEC) move. Right two panels: Reflected event-chain (REC) move [44].

The results in Figure 1.3 were created using this approach and, according to Bernard and Krauth, their implementation is two orders of magnitude faster than regular MC implementations [1].

Another option is the CUDA MC algorithm by Anderson et al. [12]. This new algorithm does parallel trial moves by using a checkerboard pattern, as Figure 6.6 shows.

Cells that are active simultaneously are at least one cell apart so their moves do not interact, because the particles are not allowed to leave the cells. One Monte Carlo sweep consists of doing moves in all 4 subsets $\{a, b, c, d\}$. After every sweep the cell boundaries are randomly redrawn to maintain ergodicity.

This approach was used by Anderson et al. to confirm the results by Bernard and Krauth [12]. According to Anderson et al. this approach is 95 times faster than their implementation of this approach on a CPU, however, when compared with the above approach by Bernard, Krauth, and Wilson the approach by Anderson et al. is at most one order of magnitude faster.

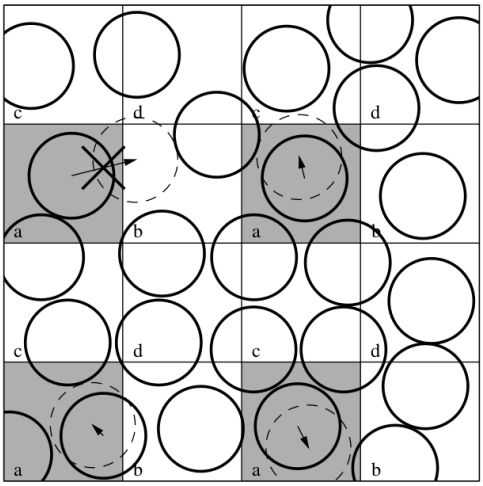


Figure 6.6: Checkerboard for the parallel Monte Carlo move by Anderson et al. [12].

Chapter 7

Conclusion

Recent publications have shown that even for simulations with a relatively large system size, finite-size effects can still occur. A system size of a million particles was needed to eliminate these finite-size effects for hard-disk systems [1, 12]. Because finite-size effects are also suspected for other hard-particle shapes, we decided to study hard regular polygons using Monte Carlo simulations on Graphical Processing Units (GPUs).

In Chapter 2 we implemented the Monte Carlo (MC) method for regular polygons on the CPU and showed some preliminary results, which showed that the finite-size effects for hard disks are indeed large near the coexistence density. To show the correctness of our implementation we reproduced the fluid part of the hard-disk equation of state using the regular chiliagon (1000 sides).

Chapter 3 shows how the inherently serial trial moves of the regular Monte Carlo algorithm can be successfully parallelized using a short Molecular Dynamics (MD) simulation. This approach is better known as Hybrid Monte Carlo (HMC). Because implementing HMC for hard particles is complex, we decided to implement the easier Lennard-Jones (LJ) system first, with the intention of switching to hard particles later. This would also give us the opportunity to test the correctness of our HMC implementation using the reference equation of state by Johnson, Zollweg, and Gubbins [28]. We implemented the HMC algorithm in both the Canonical (NVT) and Isobaric-Isothermal (NPT) ensemble, and compared them to regular MC and MD. They all correctly reproduced the LJ reference equation of state.

Chapter 4 gives a basic introduction to CUDA and how to write CUDA programs, and then shows how to implement the force/potential calculation on CUDA, using two approaches: the N-body approach, which is suitable for all potentials, and the cell-list approach, which is only suitable for short-ranged or hard potentials.

Chapter 5 then shows the results of the implementation of HMC on CUDA. Since HMC uses a MD simulation as trial move, we also implemented the MD simulation separately. As we have seen the MD simulation shows up to 2 orders of magnitude improvement over the CPU, especially for the N-body implementation. The results of the CUDA MD implementation agree with the reference equation of state. For the CUDA HMC NPT implementation, it is more difficult to make a direct performance comparison; one iteration in HMC takes much longer than one in regular MC, but we need less of them to get precise results. Based on the equilibration, we estimate that the HMC implementation is at least an order of magnitude faster. Also, its results are consistent with the reference

equation of state.

In Chapter 6 we discuss the possible applications of the current code such as studying critical Casimir forces, and wetting phenomena. We have to keep in mind that although the code is flexible, it is also complex, and the choice for potentials is limited because they have to be differentiable to preserve energy in the MD part. We also outlined the steps that need to be taken to study hard-particle systems using this code. We advise against this; it is a lot of work and there is no guarantee it will be faster than the available CPU implementations. The same is argued by Anderson et al. [12]: "Hybrid approaches that employ MD trajectories to create trial configurations can be an effective way to exploit GPU parallelism, but require substantial additional programming and are not guaranteed to evolve any faster than MD alone."

In summary, our implementation for massively parallel Monte Carlo is very fast and offers considerably speedups over a CPU implementation. Our implementations are suitable to simulate any differentiable short-ranged and long-ranged potential. Unfortunately, it is unlikely to be effective for hard-particle systems for which the ECMC algorithm by Bernard, Krauth, and Wilson or the CUDA MC implementation by Anderson et al. might be better choices [44, 12].

Bibliography

[1] E. P. Bernard and W. Krauth. "Two-step melting in two dimensions: First-order liquid-hexatic transition". In: *Physical Review Letters* 107.15 (2011), p. 155704.

[2] J. A. Anderson, C. D. Lorenz, and A. Travesset. "General purpose molecular dynamics simulations fully implemented on graphics processing units". In: *Journal of Computational Physics* 227.10 (2008), pp. 5342–5359.

[3] S. Duane et al. "Hybrid Monte Carlo". In: *Physics Letters B* 195.2 (1987), pp. 216–222.

[4] G. E. Moore. *Cramming more components onto integrated circuits*. `http://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf`. Apr. 1965.

[5] R. H. Dennard et al. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268.

[6] D. A. Patterson. *Future of computer architecture*. `www.eecs.berkeley.edu/BEARS/presentations/06/Patterson.ppt`. Presentation. Feb. 2006.

[7] Standard Performance Evaluation Corporation. `http://www.spec.org/`.

[8] R. Fish. *The future of computers*. `http://www.edn.com/design/systems-design/4368705/The-future-of-computers`. Nov. 2011.

[9] NVIDIA. *CUDA home page*. `http://www.nvidia.com/object/cuda_home_new.html`.

[10] Khronos Group. *OpenCL home page*. `https://www.khronos.org/opencl/`.

[11] *TOP500 Supercomputers*. `http://www.top500.org/`.

[12] J. A. Anderson et al. "Massively parallel Monte Carlo for many-particle simulations on GPUs". In: *arXiv preprint arXiv:1211.1646* (2012).

[13] C. Avendaño and F. A. Escobedo. "Phase behavior of rounded hard-squares". In: *Soft Matter* 8.17 (2012), pp. 4675–4681.

[14] N. Metropolis et al. "Equation of state calculations by fast computing machines". In: *The Journal of Chemical Physics* 21 (1953), p. 1087.

[15] D. Frenkel and B. Smit. *Understanding Molecular Simulation: From Algorithms to Applications*. Computational science. Elsevier Science, 2001.

[16] C. Ericson. *Real-time Collision Detection*. Morgan Kaufmann series in interactive 3D technology v. 1. Elsevier, 2005. ISBN: 9781558607323. URL: `http://books.google.nl/books?id=WGpL6Sk9qNAC`.

[17] L. Filion et al. “Efficient method for predicting crystal structures at finite temperature: Variable box shape simulations”. In: *Physical Review Letters* 103.18 (2009), p. 188302.

[18] L. E. Reichl. *A Modern Course in Statistical Physics*. Physics Textbook. Wiley, 2009. ISBN: 9783527407828. URL: http://books.google.nl/books?id=H_4qxPazAYEC.

[19] J. J. Erpenbeck and M. Luban. “Equation of state of the classical hard-disk fluid”. In: *Physical Review A* 32.5 (1985), p. 2920.

[20] W. Qi, A. P. Gantapara, and M. Dijkstra. “Two-stage melting induced by dislocations and grain boundaries in Monolayers of Hard Spheres”. In: *arXiv preprint arXiv:1307.1311* (2013).

[21] B. Mehlig, D. W. Heermann, and B. M. Forrest. “Hybrid Monte Carlo method for condensed-matter systems”. In: *Physical Review B* 45.2 (1992), p. 679.

[22] L. Verlet. “Computer “experiments” on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules”. In: *Physical Review* 159.1 (1967), p. 98.

[23] W. C. Swope et al. “A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters”. In: *The Journal of Chemical Physics* 76.1 (1982), pp. 637–649.

[24] D. C. Rapaport. “The event scheduling problem in molecular dynamic simulation”. In: *Journal of Computational Physics* 34.2 (1980), pp. 184–201.

[25] D. W. Rebertus and K. M. Sando. “Molecular dynamics simulation of a fluid of hard spherocylinders”. In: *The Journal of Chemical Physics* 67.6 (1999), p. 2585.

[26] J. E. Jones. “On the determination of molecular fields. II. From the equation of state of a gas”. In: *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character* 106.738 (1924), pp. 463–477.

[27] F. Ercolessi. *A molecular dynamics primer: The Lennard-Jones potential*. http://www.fisica.uniud.it/~ercolessi/md/md/node15.html. 1997.

[28] J. K. Johnson, J. A. Zollweg, and K. E. Gubbins. “The Lennard-Jones equation of state revisited”. In: *Molecular Physics* 78.3 (1993), pp. 591–618.

[29] F. Ercolessi. *A molecular dynamics primer: Potential truncation and long-range corrections*. http://www.fisica.uniud.it/~ercolessi/md/md/node16.html. 1997.

[30] G. 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*. ACM. 1967, pp. 483–485.

[31] J. L. Gustafson. “Reevaluating Amdahl’s Law”. In: *Communications of the ACM* 31 (1988), pp. 532–533.

[32] NVIDIA. *CUDA GPU-accelerated libraries*. http://developer.nvidia.com/gpu-accelerated-libraries.

[33] *Thrust homepage*. https://code.google.com/p/thrust/.

[34] V. Govers. *LGM overview photo*. pic 1. Email to Gerhard Burger. Jan. 2013.

[35] NVIDIA. *CUDA C Programming Guide*. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[36] Tosaka. *CUDA processing flow*. Wikimedia Commons. https://en.wikipedia.org/wiki/File:CUDA_processing_flow_%28En%29.PNG. Nov. 2008.

[37] NVIDIA. *CUDA C Best Practices Guide*. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[38] NVIDIA. *Particles*. CUDA 2.2 release 3/2009. Source Code. http://docs.nvidia.com/cuda/cuda-samples/index.html#cuda-n-body-simulation.

[39] L. Nyland, M. Harris, and J. Prins. *GPU Gems 3: Fast N-Body Simulation with CUDA*. http://http.developer.nvidia.com/GPUGems3/gpugems3_ch31.html.

[40] S. Green. *Particle Simulation using CUDA*. http://docs.nvidia.com/cuda/samples/5_Simulations/particles/doc/particles.pdf. July 2012.

[41] E. Sanz and D. Marenduzzo. "Dynamic Monte Carlo versus Brownian dynamics: A comparison for self-diffusion and crystallization in colloidal fluids". In: *The Journal of Chemical Physics* 132 (2010), p. 194102.

[42] S. Glotzer et al. *HOOMD Homepage*. http://codeblue.umich.edu/hoomd-blue/citing.html.

[43] H. B. G. Casimir. "On the attraction between two perfectly conducting plates". In: *Proc. K. Ned. Akad. Wet.* Vol. 51. 793. 1948, p. 150.

[44] E. P. Bernard, W. Krauth, and D. B. Wilson. "Event-chain Monte Carlo algorithms for hard-sphere systems". In: *Physical Review E* 80.5 (2009), p. 056704.

Appendices

Appendix A

Source code documentation

The source code is hosted in a git repository on `http://www.bitbucket.org`, please contact me if you would like access. Using the `Makefile` in the `doxygen` directory creates an easy to use HTML documentation which can be accessed by opening `doxygen/html/index.html` in a browser.

I also attached the documentation here: 
Right-click the icon to save the attachment, and use `tar --xz -xf mt_gerhard_doc.tar.xz` to extract.