GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN

Fakultät für Physik

Bachelor's Thesis

Mit fraktalen Kurven beschleunigte Studie von granularer Oberflächenspannung

Fractal curve accelerated study of granular surface tension

prepared by

Niklas Bölter

from Celle

at the Max Planck Institute for Dynamics and Self-Organization

Thesis period:	1st May 2015 until 15th September 2015
First referee:	Dr. Marco G. Mazza
Second referee:	Prof. Dr. Stephan Herminghaus

Abstract

In this thesis we look at a performance bottleneck of running molecular dynamics code on GPGPU devices (specifically the CUDA platform), namely random memory access into global graphics card memory. We offer a solution that involves reordering memory blocks in order to enable more memory reads from the local cache (shared memory) instead of the global memory. For determining a memory block order that sufficiently increases performance a space-filling curve was used.

Significant performance increases for higher filling fractions were observed with a molecular dynamics simulation that was written from scratch.

Keywords: Molecular dynamics, CUDA, GPGPU, Space-filling curves, Shared memory

Contents

1. Introduction

The theory of granular gases deals with the flow and structure of macroscopic particles like sand and corn, which are of great practical importance. Examples include industrial processes and agriculture. If we can efficiently simulate such a system, we will gain insights from the detailed information about particle positions that are impossible to extract from an experimental system alone and might be able to offer explanations and optimizations.

The starting point for this research was the work of James Clewett on simulating a system of granular gases under vibration. [1]

In a granular gas, macroscopic particles with many internal degrees of freedom are interacting with each other. Since energy can be deposited into these internal degrees of freedom, the particles will lose kinetic energy during interactions. [2] To keep the system from freezing, kinetic energy has to be added continually.

For this reason the granular gas represents a system that is dissipating energy and is far from thermal equilibrium, and can be used as a model system for non-equilibrium thermodynamics. Interesting effects that are reminiscent from equilibrium thermodynamics have been observed. [1]

To simulate such a granular gas the equations of motion for an N-body problem need to be solved by numerical integration (molecular dynamics simulation).

In the preceding research a molecular dynamics simulation was created that utilizes the unique capabilities of a graphics cards (massive parallelization). To calculate the forces resulting from particle-particle interactions the positions of particles that are close-by in simulation space need to be read from memory - however the memory locations storing these positions are in general widely distributed over the graphics card memory.

These random memory reads (in the sense that memory locations that are not adjacent are accessed) from the graphics card memory are the main bottleneck for the performance of the molecular dynamics simulation, since they cannot be parallelized. Ref. [1] found that "the execution times for code using global memory are solely a function of the number of memory accesses, that is, the instruction execution times are negligible compared to that latency of the memory access."

The main goal of this work now is to reduce the number of these memory accesses with the hope of dramatically increasing the performance of the simulation.

2. The Problem

To calculate the forces acting on the particles in the simulation box, their relative positions and velocities need to be known. Since the overall performance of the program is determined by memory latency, for the purpose of this research a simple spring-dashpot model with an interaction cutoff was used:

$$\vec{F} = \begin{cases} -K\Delta\vec{x} - c_d\Delta\vec{v} & \text{if } |\Delta\vec{x}| < d \\ 0 & \text{if } |\Delta\vec{x}| \geq d \end{cases}$$

Here $\Delta\vec{x}$ is the vector pointing from a particle to its interaction partner, K the spring constant, $\Delta\vec{v}$ their relative velocity, c_d a viscous drag coefficient, d the particle diameter, and $\vec{F}$ the resulting force on the particle. More complicated and realistic formulas can be used if so desired.

Generally a suitable algorithm loops over every particle in memory, determines potential interaction partners and their relative positions and if they overlap ($|\Delta\vec{x}| < d$) calculates the relative velocities and the forces acting on the particle under inspection.

To efficiently determine potential interaction partners, the simulation volume is subdivided into boxes, for each box a list of particles is maintained and only particles in adjacent boxes are considered for interaction (cf. Fig. 2.1).

To determine the relative velocity and position of that potential interaction partner with the particle under inspection the absolute velocity and position of both particles need to be read from memory.

To see why this is a problem, one has to look at the memory layout of a GPGPU architecture (cf. Fig. 2.2): Several execution units are grouped in one block, they all have access to a small (on the order of 32 kilobytes) block of memory ("shared memory" or cache) with very fast access times. However blocks cannot access the shared memory of other blocks. The much larger (on the order of 6 gigabytes) global memory can be accessed from all blocks, but random accesses that cannot be parallelized are painfully slow.

When the algorithm that calculates the forces is initialized, the memory corresponding to the current positions and velocities of all particles is partitioned such that the first block of execution units receives the first N particles (in memory order), the second block the next N particles and so on. This is a very fast operation since the memory is read sequentially.

Now each execution block is working in parallel to calculate the forces on the particles it was assigned to, in general requesting the positions and velocities of their potential interaction partners from global memory, which we term cache miss because it is outside of the cache, unless they happen to be already assigned to an execution unit in the same block, in which case the read can be satisfied from the shared memory, which we term cache hit, because it is inside the cache.

Since shared memory reads are much faster, the goal is to ensure that the particles that are assigned to each execution block are likely interaction partners, which will reduce the number of global memory reads. A measure for this is the cache hit probability, which measures how many memory reads will hit the cache compared to the total number of memory reads.

It is clear that to increase the cache hit probability, particles that are likely interaction partners (particles that are close in simulation space) need to be close in global memory as well. Then they will be assigned together to an execution unit, increasing the likelihood of cache hits.

Two extreme cases of this are shown in Fig. 2.3 (no cache hits) and Fig. 2.4 (no cache misses) respectively.

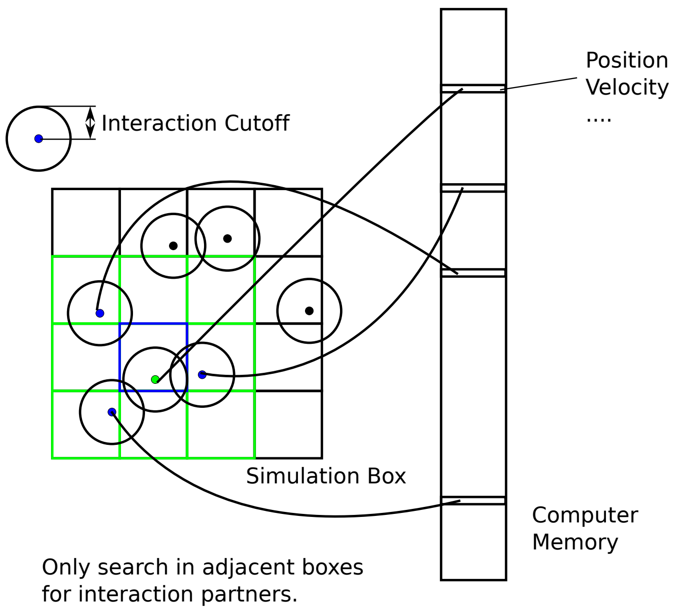

Figure 2.1.: Molecular dynamics with finite interaction cutoff: Only particles in adjacent boxes are considered. The positions and velocities of nearby particles are located at random memory locations since the particles are moving chaotically through space but are fixed in memory.

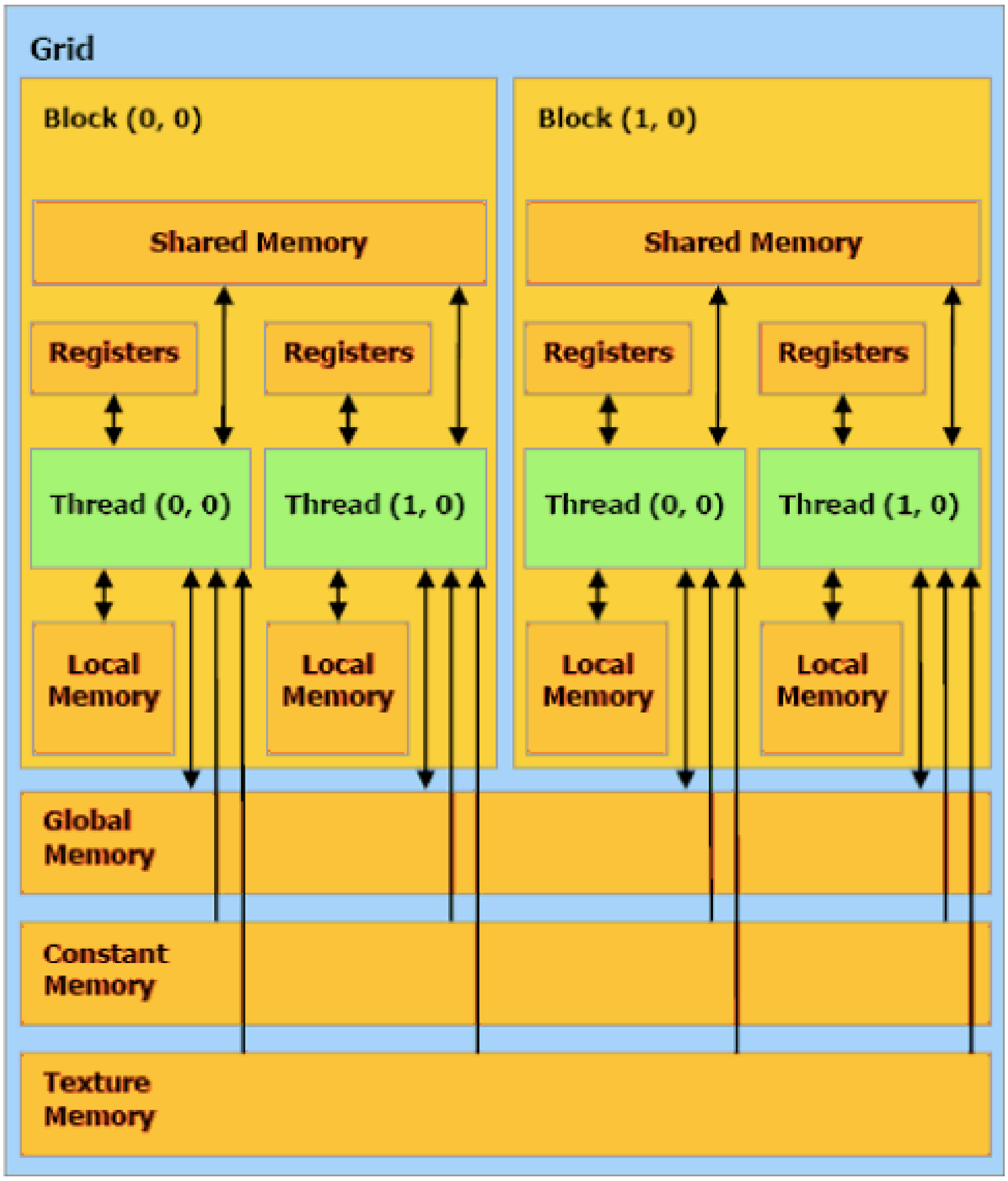

Figure 2.2.: Cuda memory layout, image from Farber [3].

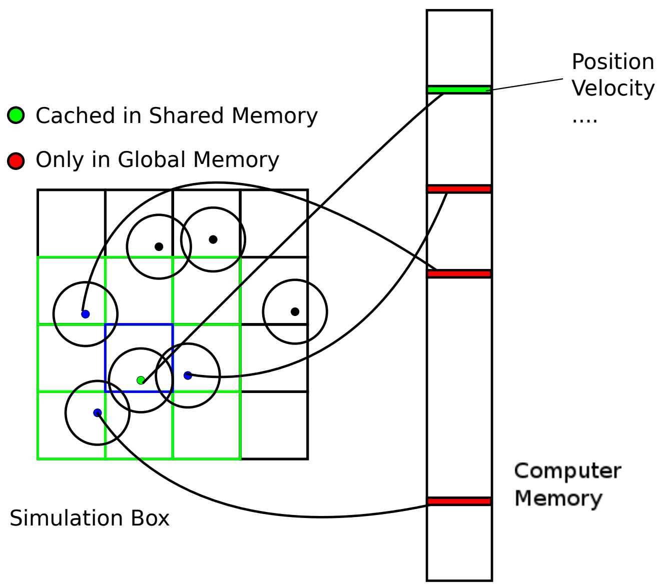

Figure 2.3.: The Problem: Random memory locations have to be read to calculate the forces between particles.

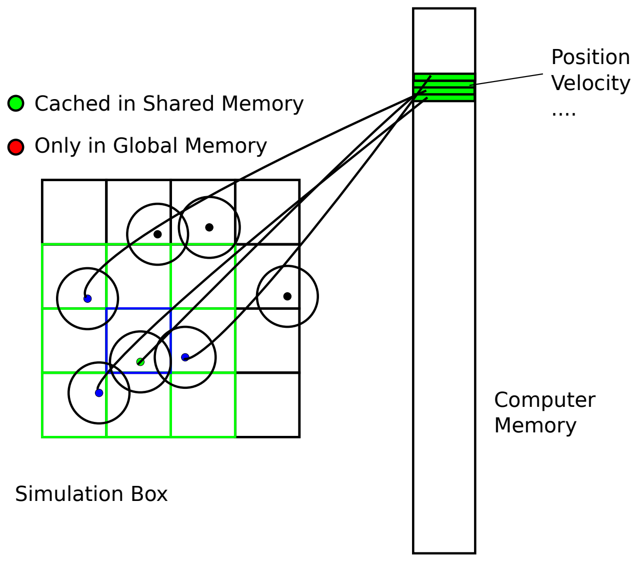

Figure 2.4.: The Goal: Reorder memory to convert random memory reads into reads from the shared memory (cache).

3. The Solution

As we saw in the previous chapter, the declared goal is to reorder memory blocks such that parameters of particles that are close together in simulation space need to be close together in memory as well.

Naturally, since the particles are only loosely confined [1], they will move away from each other in simulation space, which will break the correlation between memory position and simulation position such that a reordering of memory blocks is required periodically. Since computer memory is linear, the requirement stated in the first paragraph is equivalent to imposing an order on the particles, i.e. assigning them a one-dimensional index where points mapped close in 3D space have close indices as well.

This property is a well-known property of space-filling curves [4] described in the literature as clustering [5]. A space-filling curve for our purposes is a continuous map from the unit interval to a subset with a finite area (2D curve) or volume (3D curve) of Euclidean space.

We will subdivide our simulation box into many boxes of linear size $s \simeq d$, where d is the particle diameter, and then find a space-filling curve that traverses each box once. The order in which the curve visits a box will then establish the sort order (cf. Fig. 3.1 and Fig. 3.2). In general a space-filling curve cannot be injective [4], however the finite iterations of many of them are self-avoiding and therefore establish a unique sort order. We will not consider curves that are not self-avoiding.

Since the boxes are stationary, their sorting index can be calculated before starting the simulation and reused during the simulation - as such the generation of the curves themselves is not performance critical and no effort has been spent on the performance of the curve generation.

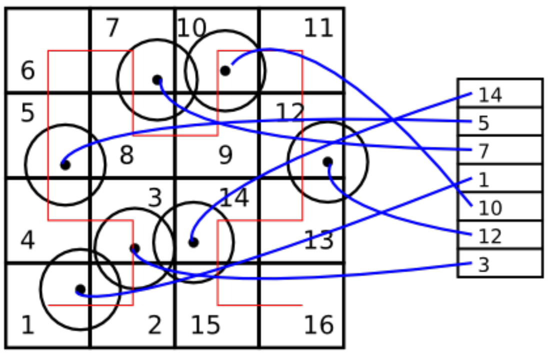

Figure 3.1.: Particles in a two-dimensional simulation box with an inscribed finite iteration of the HILBERT space-filling curve. The memory locations are unordered.

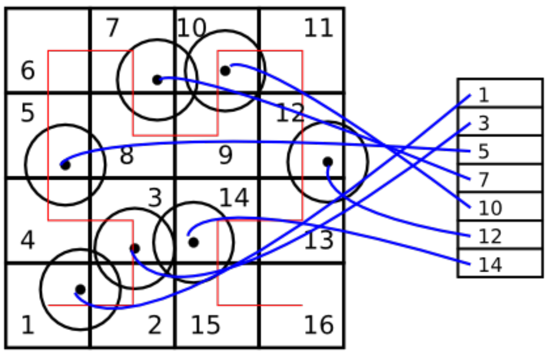

Figure 3.2.: The same particles as in Fig. 3.1 with reordered memory locations.

4. Space-Filling Curves

In this chapter we will present the space-filling curves that were used for our simulations. The first space-filling curve was published by PEANO [6], which is why space-filling curves in general are also sometimes called PEANO curves. PEANO provided an arithmetic definition of his curve using the ternary number system, we shall however skip right to the later geometric description due to HILBERT.

We will only look at curves that fill a square or a cube, since these are much easier to work with in computer simulations, for example when imposing boundary conditions (e.g. periodic boundary conditions).

The first iteration steps for the HILBERT curve are shown in Fig. 4.1 [7]. Each square is divided into sub-squares of equal size, and each vertex of the previous iteration step is replaced with the same horseshoe-pattern, which might have been rotated or mirrored. If this process was repeated ad infinitum, the result would be the HILBERT curve.

If only the first iteration step is changed to a square instead of a horseshoe, then a very close relative to the HILBERT curve, the MOORE curve emerges instead. It looks locally exactly like a HILBERT curve, just that its end point and start point coincide.

Since we are looking for a traversal of a number of boxes with finite size, we will generally only take a finite number k of iteration steps to fill a square with 2^k boxes on a side.

Using HILBERT's approach one can also now describe the PEANO curve with the same scheme [4], as can be seen in Fig. 4.2. The underlying pattern is a serpentine line, and each square is replaced with 9 instead of 4 subsquares, but the construction principle is very similar.

These two curves also share the property that squares that are neighbors in the ordering imposed by the curve are also adjacent in Euclidean space - however this is not mandatory, as can be seen from the Z curve in Fig. 4.3.

The Z curve, which is closely related to the LEBESGUE curve, is commonly employed in computer science, since the mapping can be very easily calculated in the binary representation [5]. For example taking the binary representation of the two-dimensional Z curve parameter and cutting out all the digits at even positions will produce the y-coordinate and vice versa.

Since we are not looking for a fast generation of the curve and can afford to spend a few milliseconds generating it once (when the simulation is started), the simplicity of the Z curve is not important here. Also since it *jumps* around, we naively wouldn't expect it to outperform the other candidates.

To represent these curves in computer memory we chose a simple scheme where the small boxes that need to be ordered are each identified with an integer value calculated from their integral coordinates in Euclidean space:

$$i = x + y \cdot N_x + z \cdot N_x \cdot N_y \tag{4.1}$$

Here N_x, N_y, N_z are the number of boxes along the x,y, and z axis, respectively. The curve is then just an array of length $N_x \cdot N_y \cdot N_z$ filled with box indices.

To construct these arrays a replacement scheme called a Lindenmayer system [8] was used. This closely mirrors HILBERT's construction principle, for example given the following alphabet:

- F Move one step forward
- > Rotate 90° counter-clockwise
- < Rotate 90° clockwise

A horseshoe pattern can be written as "F<F<F", and the HILBERT curve can be constructed by starting with a horseshoe pattern ">BF<AFA<FB>" and replacing each occurrence of A with the pattern itself and B with a chirality reversed version of the pattern where all clockwise rotations or A have been changed to counter-clockwise rotations or B and vice versa k times (this represents the replacement of the square with four subsquares). This process is visualized in Fig. 4.4. Then the A & B are deleted and the string representation of the curve can be evaluated with a simple computer program that keeps track of the current position and direction ("Turtle Graphics") [8].

Recruiting a turtle equipped with rockets, it is easy to generalize this procedure to three dimensions as well, using a different alphabet:

- F Move one step forward
- $-$ Turn right
- $+$ Turn left
- ^ Pitch up
- & Pitch down
- $<$ Roll left
- $>$ Roll right

One of the basic patterns that can be used to construct a 3D Hilbert curve is shown in Fig 4.5. Because of time limitations and because the 2D Hilbert curve was giving the best results for 2D simulations, no other 3D curves have been tested as part of this project.

Since the experimental system is not a cube but a rather shallow rectangular cuboid with a square base, additionally a Hilbert curve had to be found that parameterizes this geometry.

Assuming for simplicity a system with a size of 64x64x8 boxes, first a 2D curve was generated that parametrizes a 8x8 square and then on each subsquare a 3D curve of size 8x8x8 was put, giving a total dimension of 64x64x8.

This was implemented using a somewhat more complicated Lindenmayer system that switches from a 2D replacement pattern to a 3D replacement pattern a few steps into the iteration. We named this curve "Hilbert pancake" and it can be seen in Fig. 4.6.

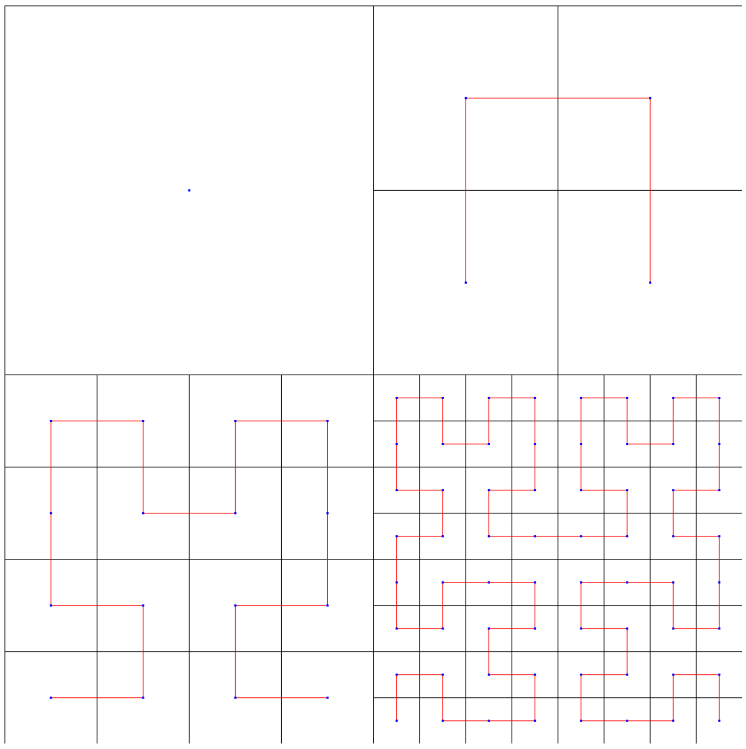
Figure 4.1.: First four iterations of a HILBERT Curve.

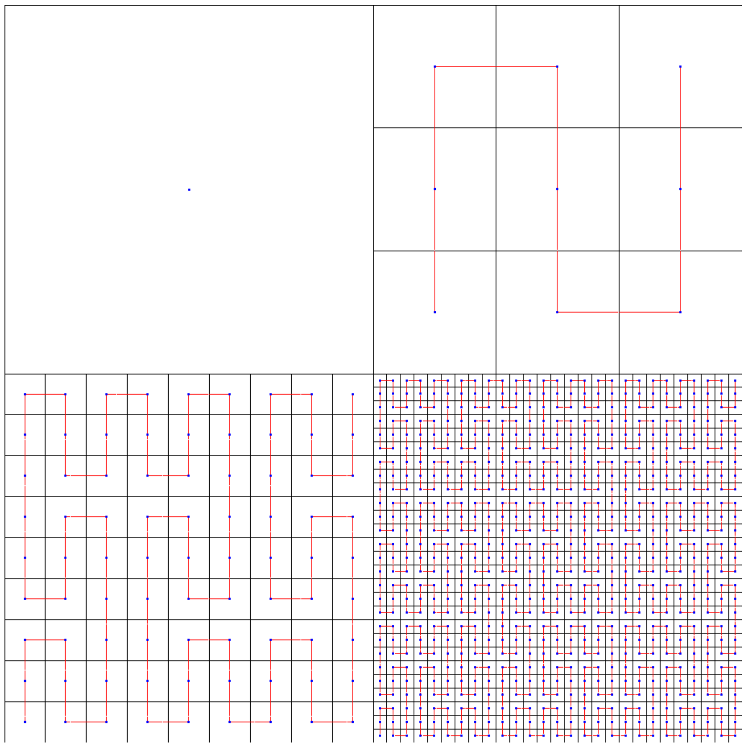
Figure 4.2.: First four iterations of a PEANO Curve.

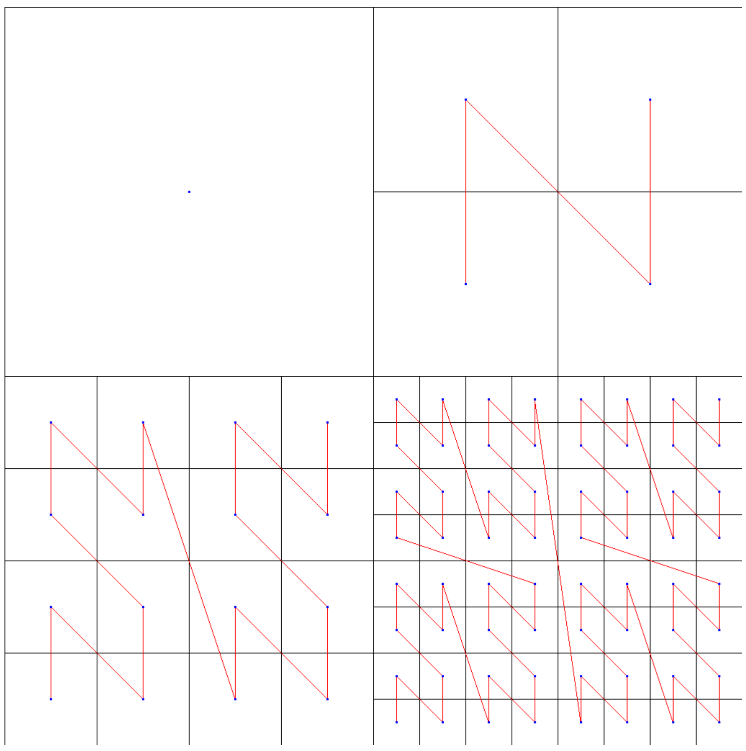
Figure 4.3.: First four iterations of a Z Curve.

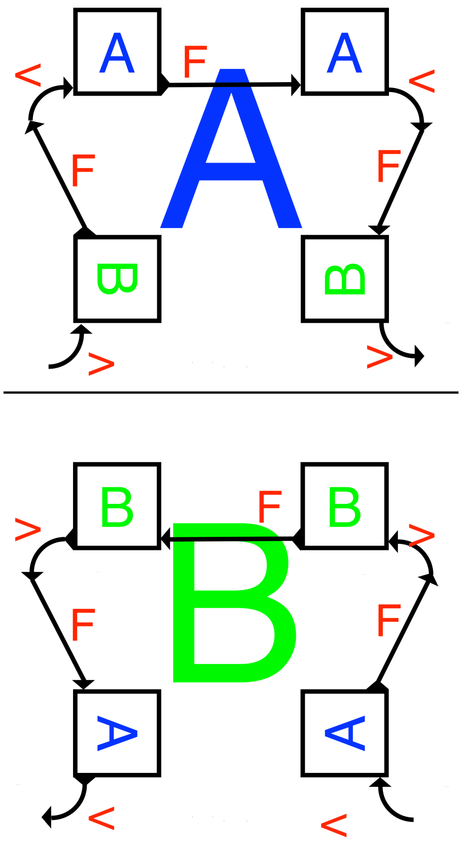

Figure 4.4.: Visualization of a simple two dimensional Lindenmayer replacement system [A → >BF<AFA<FB>, B → <AF>BFB>FA<]. The upper half represents the A pattern, the lower half the B pattern. During one iteration step, the four small boxes (A and B) are replaced by the entire A/B patterns, appropriately rotated. Removing the boxes in the A or B patterns, the base horseshoe pattern of the HILBERT curve emerges.

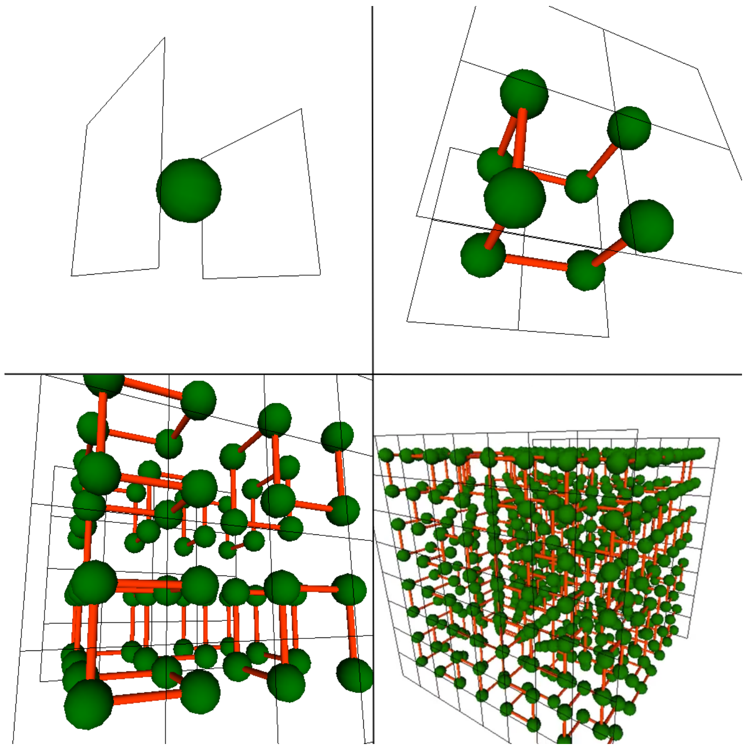

Figure 4.5.: First four iterations of a three-dimensional HILBERT curve.

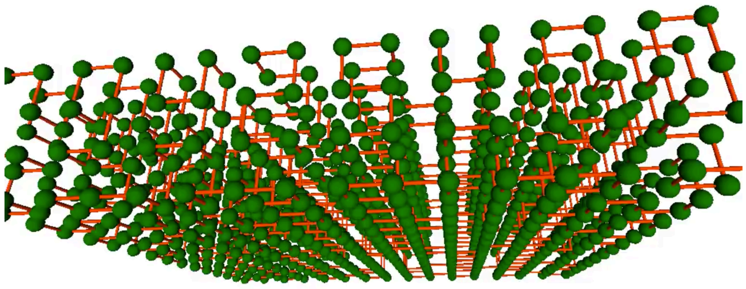

Figure 4.6.: A two-dimensional HILBERT curve where each vertex has been replaced with an appropriately rotated/mirrored three-dimensional hilbert curve "HILBERT pancake".

5. Theoretical Results

Since runtime debugging is much easier on the CPU, a molecular dynamics algorithm (in the C programming language) was first written as CPU code. Also the space-filling curve generation and the reordering algorithms were only implemented in CPU code. The generation of the space-filling curves is not relevant to the total performance and can as such be implemented in CPU code which is easier to write.

The reordering algorithm that we used is rather naive since it copies each particle's data individually and uses quite a lot of random-memory accesses that are particularly ill-suited for the GPU. Since the particle data has to be transferred to the CPU memory periodically anyway in order to write it to disk and save particle trajectories, we decided to use these occasions to also reorder the particles on the CPU before continuing the simulation on the GPU. Since the particles will in general be already partially ordered, we expect subsequent reordering steps to be accelerated by the large CPU cache.

We tested the reordering code for random particles as well as already ordered particles with a naive single-core implementation as well as with an OpenMP based multi-threaded algorithm that better utilizes modern CPUs. The results are shown in Fig. 5.1 and as expected the CPU cache accelerates the "reordering" of already ordered particles quite a bit and the multi-threaded algorithm is significantly faster.

We also implemented an algorithm that predicts the cache-hit ratio a GPU based code would encounter. For this we partitioned the memory in blocks that have the size of the GPU cache (local memory) and then checked during simulation if the two interacting particles share the same box and so would be cached together. This enabled us to compare different types of space-filling curves in theory and to get a first impression on possible performance improvements.

We started our work with a two-dimensional system because, for example, the generation of the space-filling curves and the visualization are easier to do. In order to test the two-dimensional space-filling curves, two different geometries were used, one with dimensions $N \times N \times 1$ where $1 \times 1 \times 1$ cubes were put on a $N \times N$ square, one with dimensions $N \times N \times N$ where $1 \times 1 \times N$ square prisms are put on the same $N \times N$ square. These geometries are pictured Fig. 5.2. In the second case the actual simulation was three-dimensional, although the space-filling curves was only two-dimensional.

The results have been plotted in Fig. 5.3 & Fig. 5.4 for the first system and in Fig. 5.5 & Fig. 5.6 for the second system. For the entirely two-dimensional system the cache-hit ratios are quite excellent, ranging above 90% for the regime we are interested in. In the second case, where a 2D curve was used in a 3D system, the cache-hit ratios drop off much more precipitously - so a three-dimensional space-filling-curve has to be investigated.

These simulations were run with different types of interactions including ideal gas (no interactions) and a simple granular gas (interactions only when spherical particles intersect, inelastic collisions with a spring-dashpot model).

One important aspect of the performance is not only the cache-hit ratio right after a reorder, but also how it develops over time between reorders as the particles move around and the correlation between particle positions in simulation and memory space break down. It is natural to assume that the higher the velocities of the particles are, the faster they will become uncorrelated. This decay can be seen for a system without interactions in Fig. 5.7 & Fig. 5.8 for a filling fraction of 10% and 50% respectively. As we expected, higher velocities lead to faster decorrelation of particle positions and a faster decay of the cache-hit ratio.

Because the HILBERT curve showed the best cache-hit ratios in the two dimensions and because of time constraints we choose only to implement the HILBERT curve in three dimensions.

The instantaneous cache-hit ratios (directly after reorder) for a cubic system of different dimensions and different filling fractions have been plotted in Fig. 5.9. For small system sizes the cache size becomes large in comparison with the particle number and the cache-hit ratio becomes artificially inflated, since significant portions of the system fit in the cache. On the other side, for larger system sizes the cache-hit ratio stays constant over a large range, as one would expect from the self-similar nature of the used fractal curve.

The decay of the cache-hit ratio after a reorder has also been studied for the three-dimensional system, and the results are plotted in Fig. 5.10 & Fig. 5.11. It is clear that at a filling fraction of 25% the confinement due to collisions slows down the decay of the cache-hit ratio quite significantly. Also for inelastic collisions the system freezes and the decay slowly comes to a halt. The system size (apart from the expected artefacts once the system size becomes comparable to the cache size as discussed above) does not seem to have an affect on the decay, as shown in Fig. 5.12.

To visualize the difficulty associated with the small cache-size of the GPU, a map of the system was overlaid with the average cache-hit ratio to see how different memory reordering strategies affect the cache-hit ratio. As can be seen from Fig. 5.13, for a typical GPU cache a naive reordering scheme without space-filling curves that exhibit the clustering property is not sufficient.

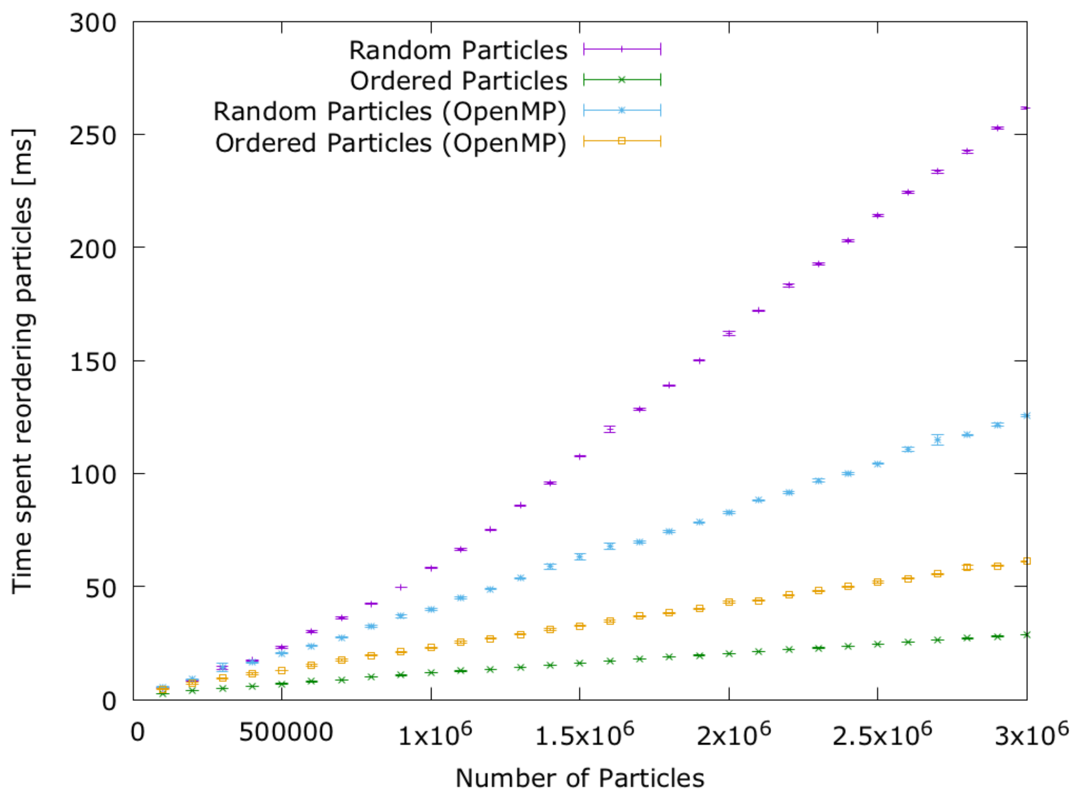

Figure 5.1.: Performance of particle reordering using a naive single-threaded implementation and a multi-threaded OpenMP based implementation. Each algorithm is also run on previously ordered particles as well to estimate how the CPU cache affects reordering performance of (partially) ordered particles.

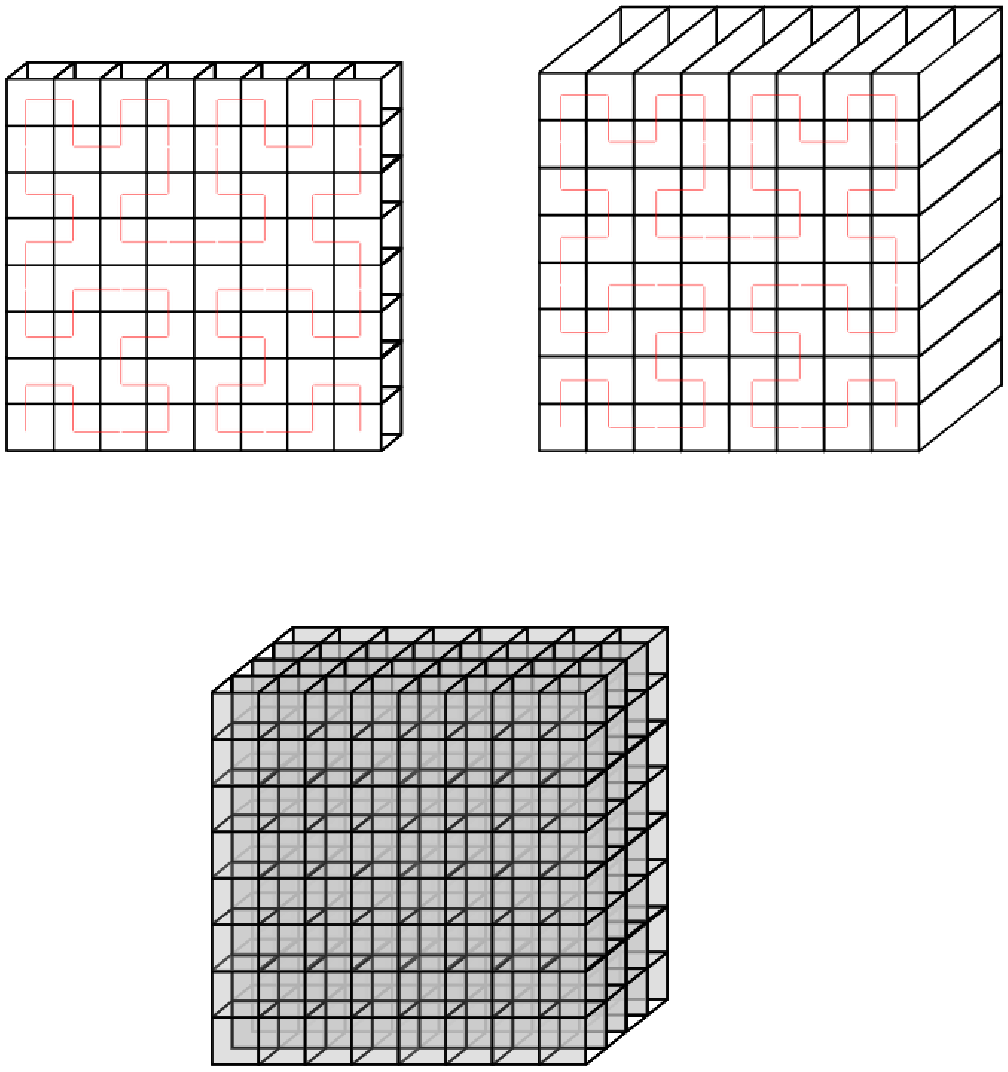

Figure 5.2.: Different geometries used for testing: 2D array of square boxes (top left), 2D array of cuboid boxes (top right), 3D array of square boxes (bottom). The box square bases were 1x1 in size in units of particle diameter and the cuboids had a height of 10.

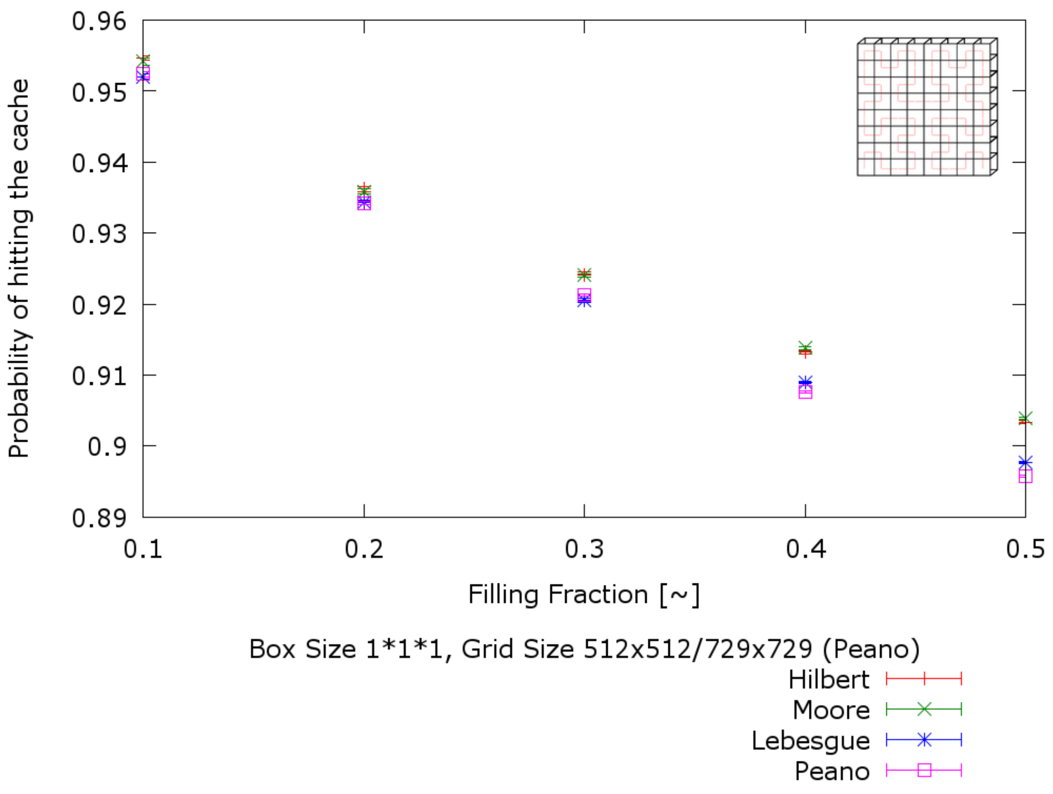

Figure 5.3.: Simulated cache hit probabilities for two-dimensional space-filling curves directly after reorder (no decay). Because the PEANO curve has a 3x3 base pattern a power of three was used for its grid size. A cache size of 320 particles was used and the geometry is a 2D array of **cubes**.

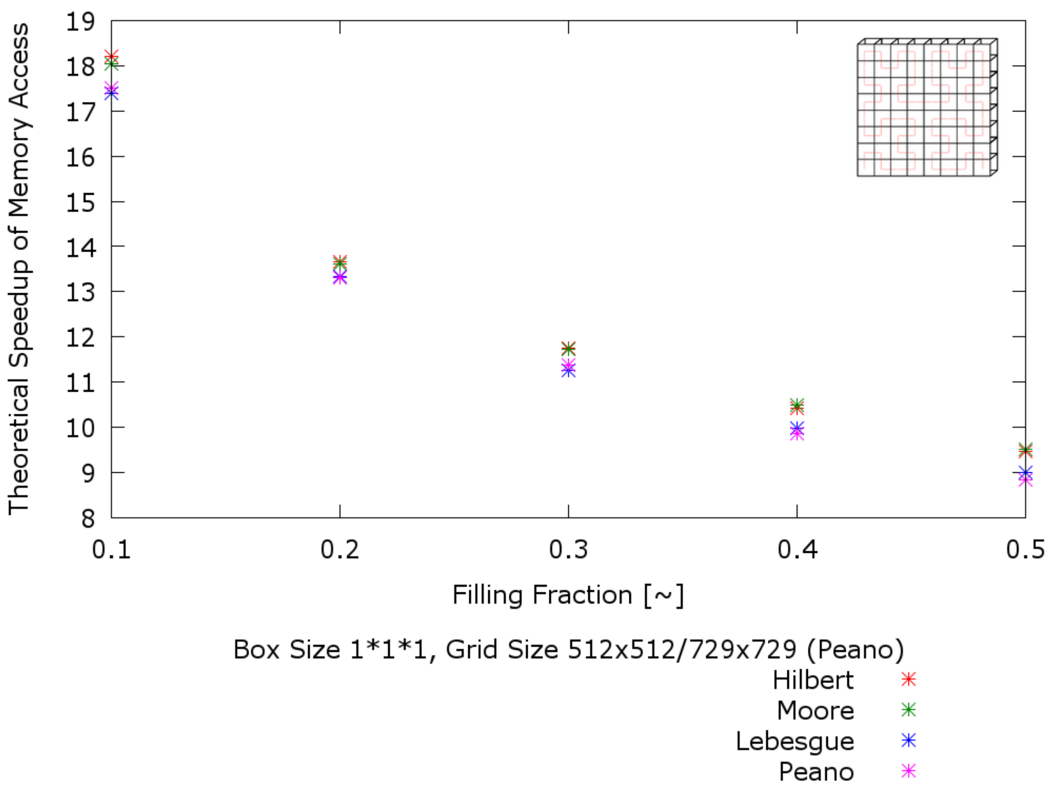

Figure 5.4.: Simulated speedup of the memory access due to cache hits for the two-dimensional space-filling curves directly after reorder (**cubic** boxes). A factor of 100 was used for the difference in access time between cache hit (served from local memory) and cache miss (served from global memory).

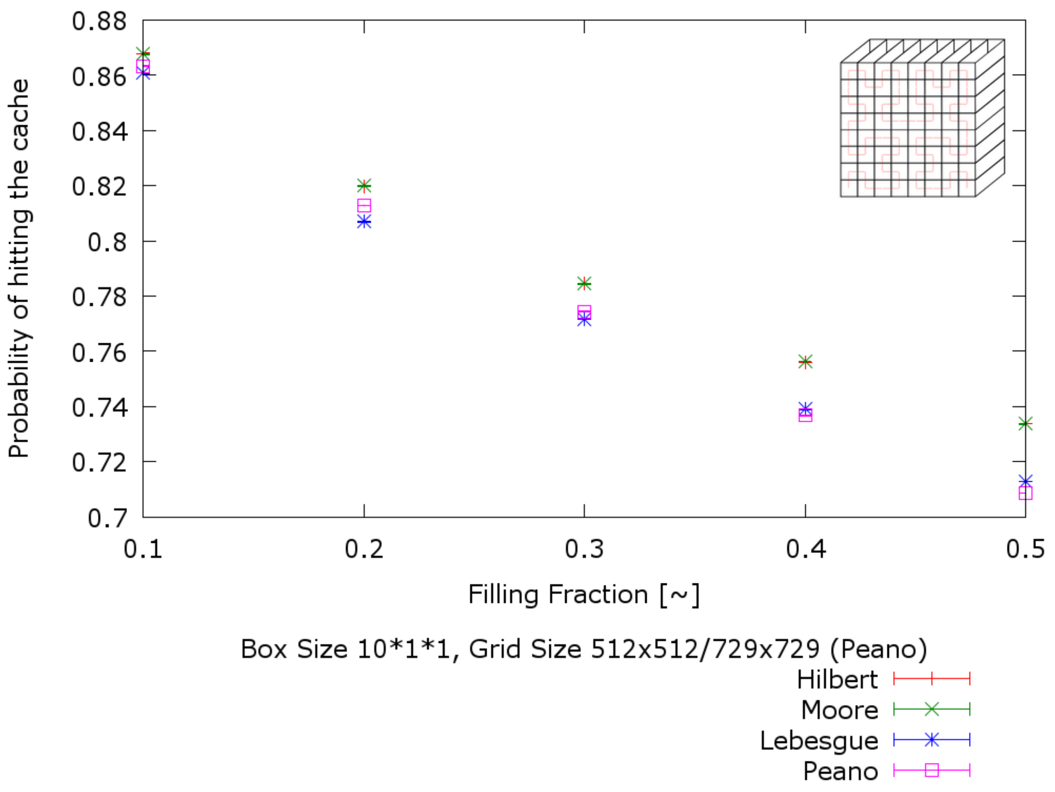

Figure 5.5.: Simulated cache hit probabilities for two-dimensional space-filling curves directly after reorder (no decay). Because the PEANO curve has a 3x3 base pattern a power of three was used for its grid size. A cache size of 320 particles was used and the geometry is a 2D array of **cuboids**.

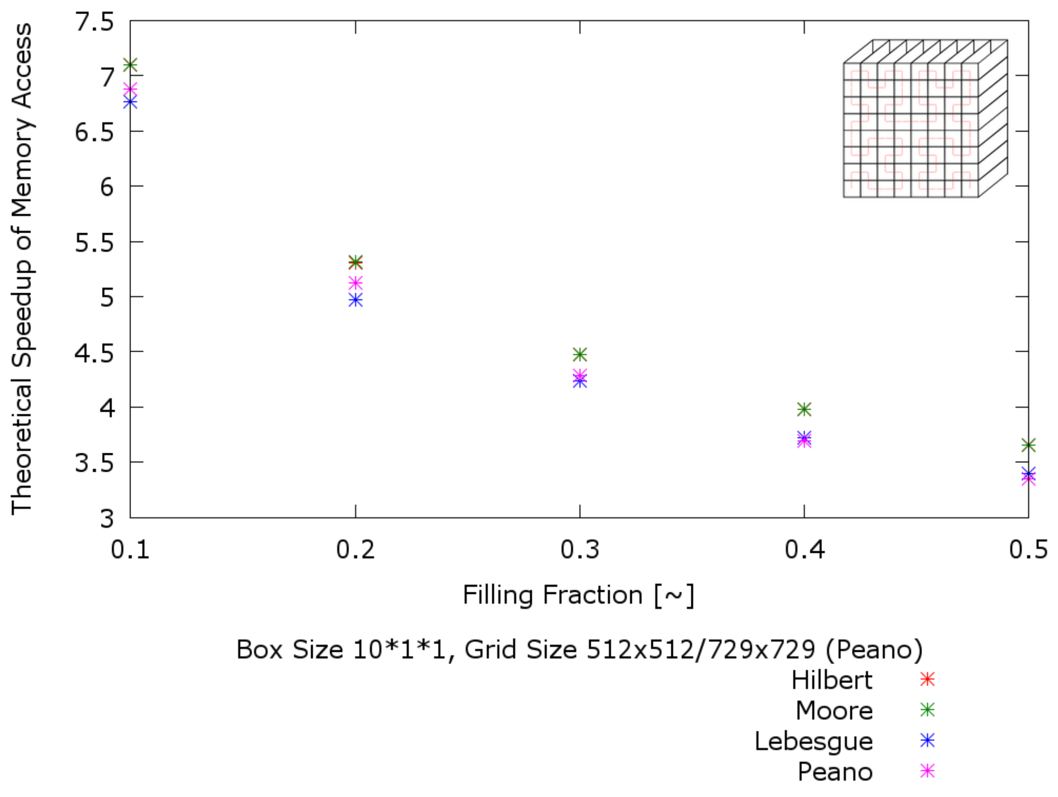

Figure 5.6.: Simulated speedup of the memory access due to cache hits for the two-dimensional space-filling curves directly after reorder (**cuboid** boxes). A factor of 100 was used for the difference in access time between cache hit (served from local memory) and cache miss (served from global memory).

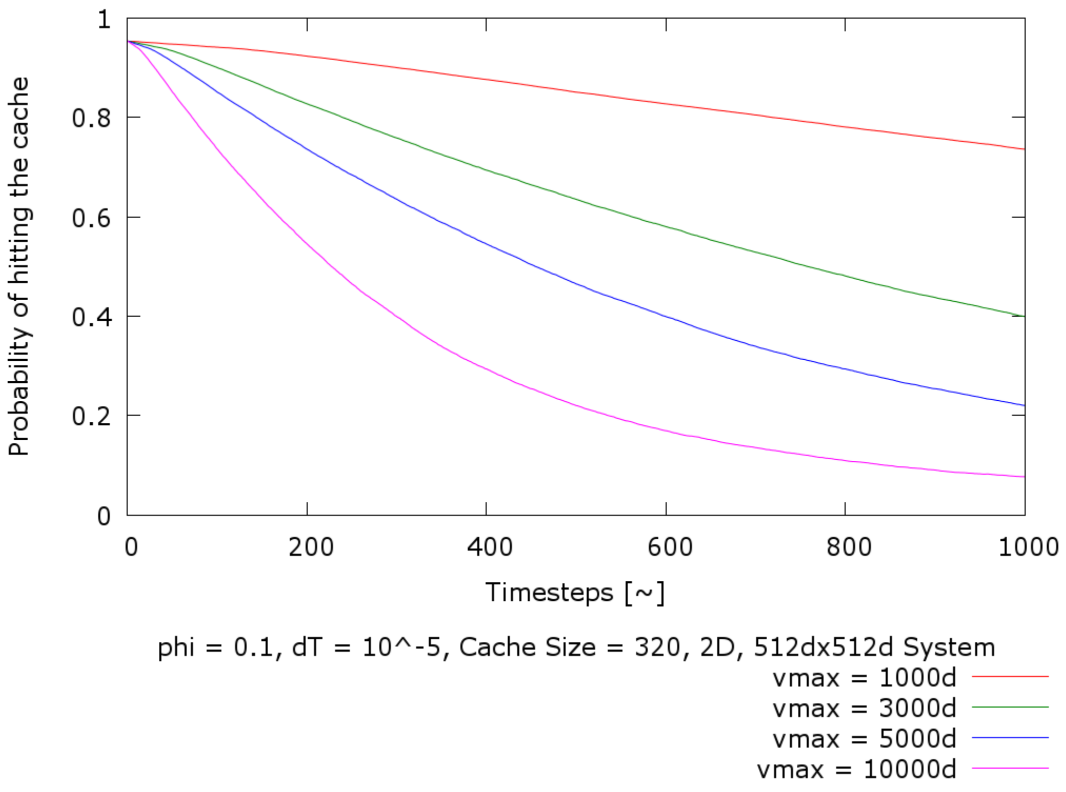

Figure 5.7.: Decay of the cache hit ratio over simulation steps for different initial velocities (temperature) for a dilute system (10% volume filling fraction). The temperature is measured in particle diameters per simulation step (d).

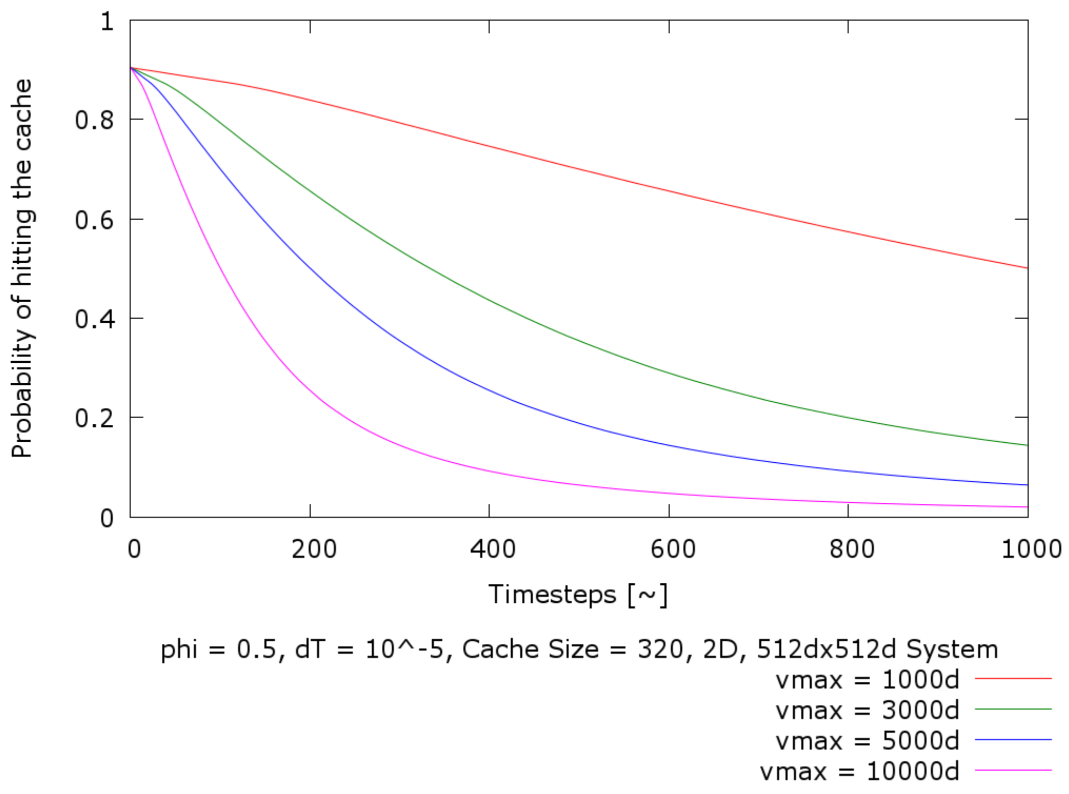

Figure 5.8.: Decay of the cache hit ratio over simulation steps for different initial velocities (temperature) for a dense system (50% volume filling fraction). The temperature is measured in particle diameters per simulation step (d).

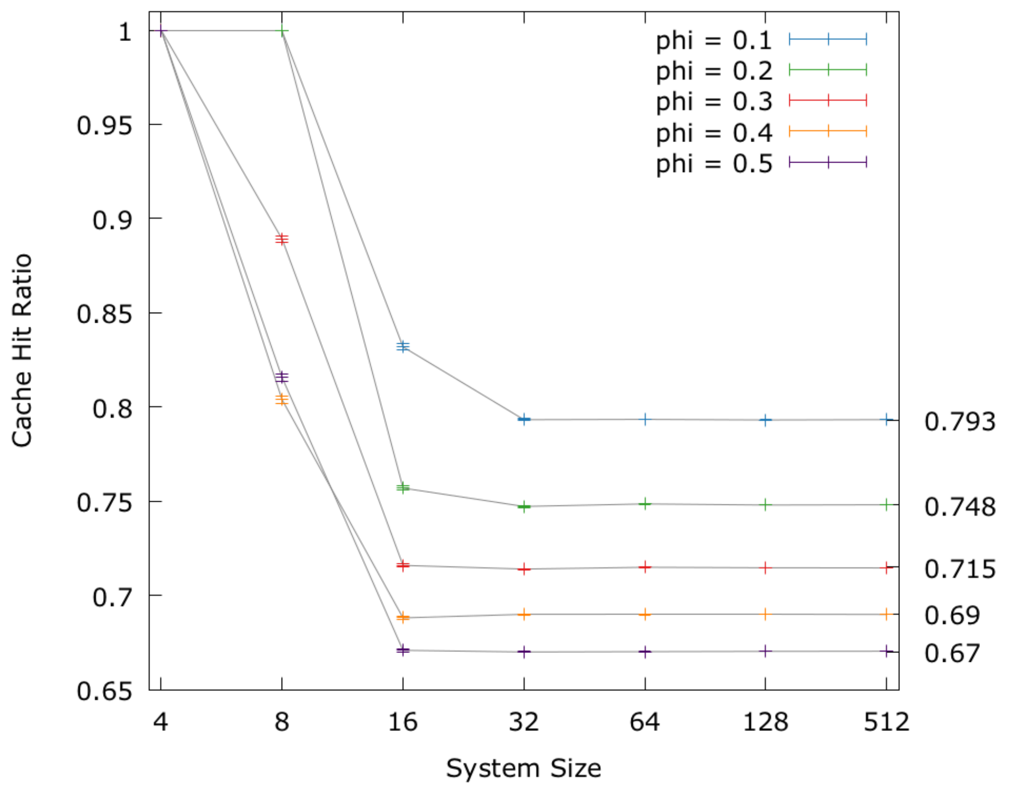

Figure 5.9.: Cache hit ratio with fixed cache size for three-dimensional Hilbert curve directly after reorder. The system geometry is cubic with *System Size* boxes in each direction. Once the number of particles becomes large compared to the cache size, the cache hit ratios stay constant over a wide range.

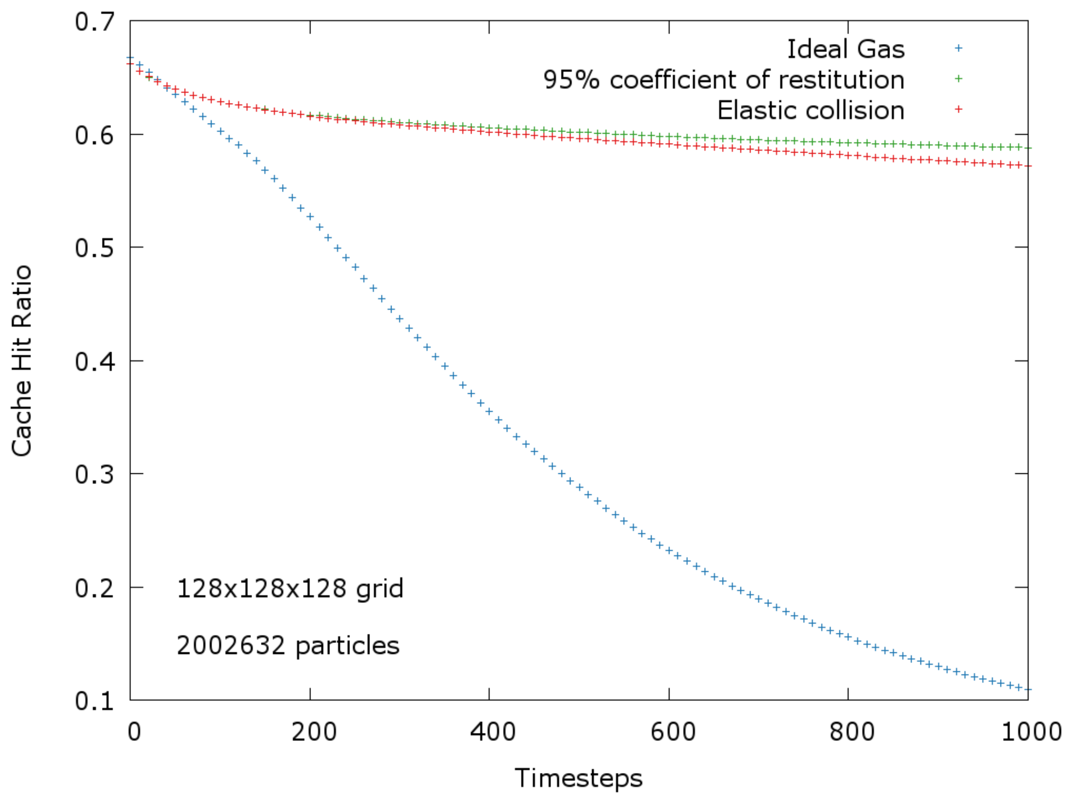

Figure 5.10.: Decay of the cache hit ratio over simulation steps for different interaction types. Reorder at time step 0 using a three-dimensional Hilbert curve, filling fraction 25%. The non-interacting particle cache hit ratio decays much faster since there is no confinement of particle positions due to collisions.

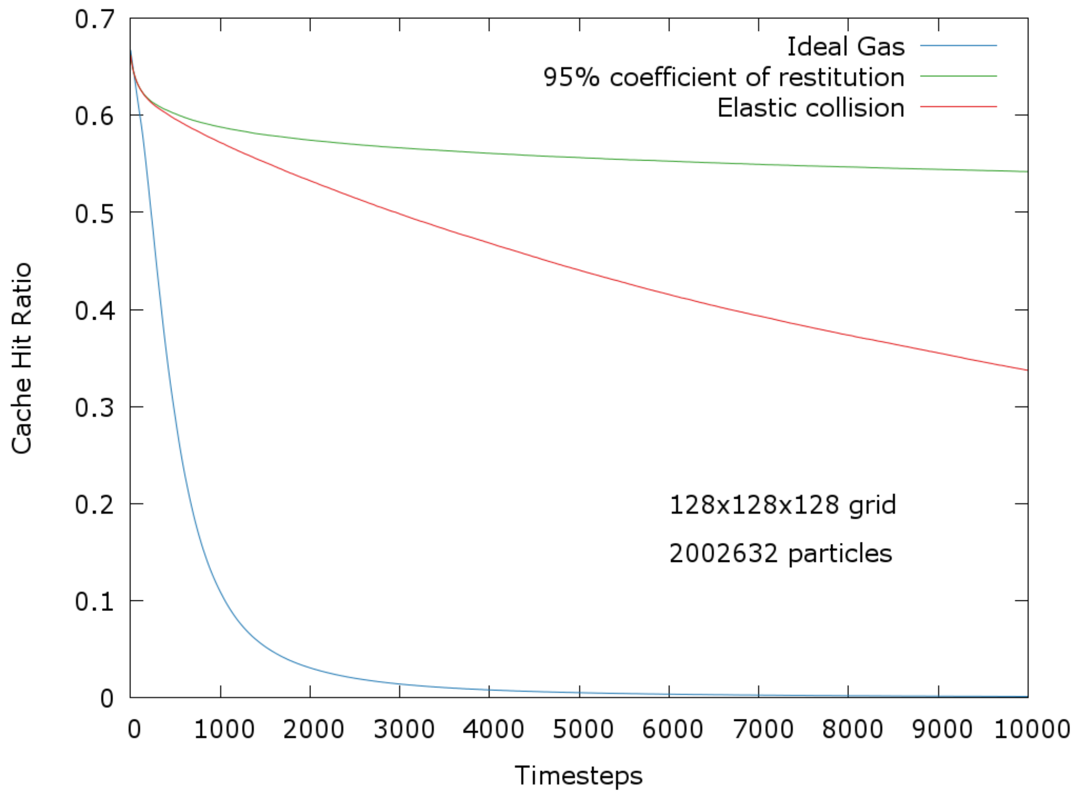

Figure 5.11.: The same plot as before with 10x more time steps. The cache hit ratio for the inelastically colliding particles decays much slower because the system freezes.

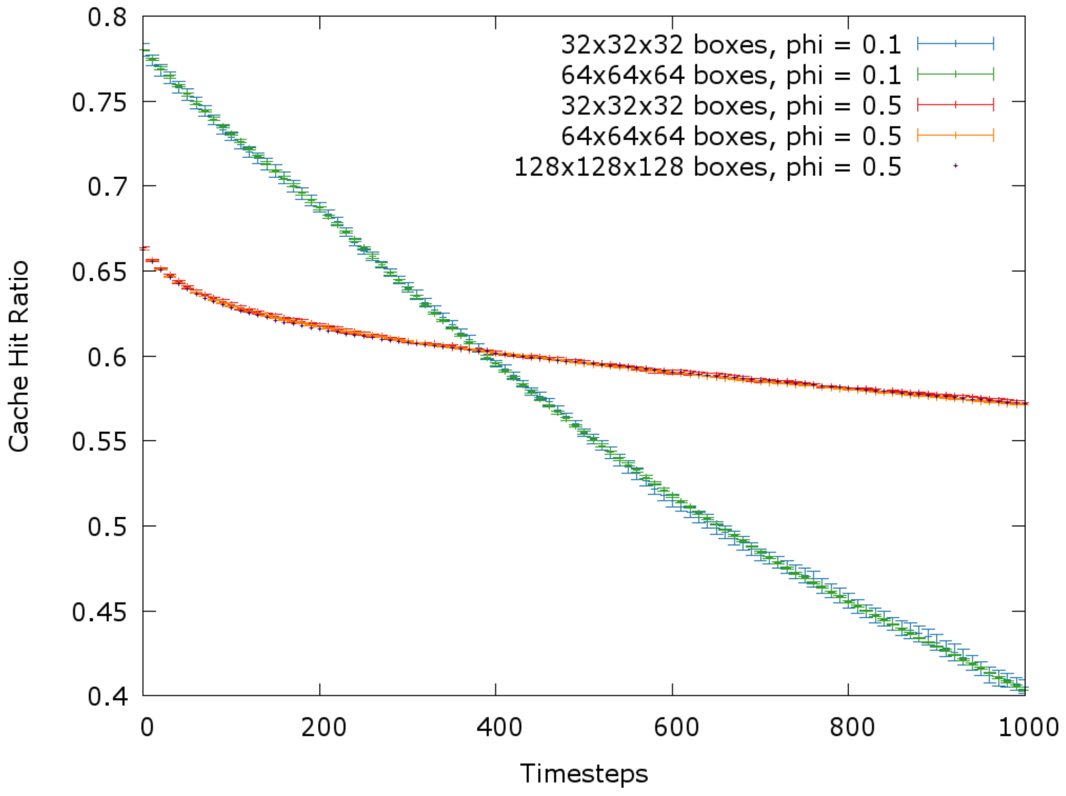

Figure 5.12.: Decay of the cache hit ratio over simulation steps for a system of elastically colliding particles. The decay does not seem to depend on the system size as long as the filling fraction is kept constant.

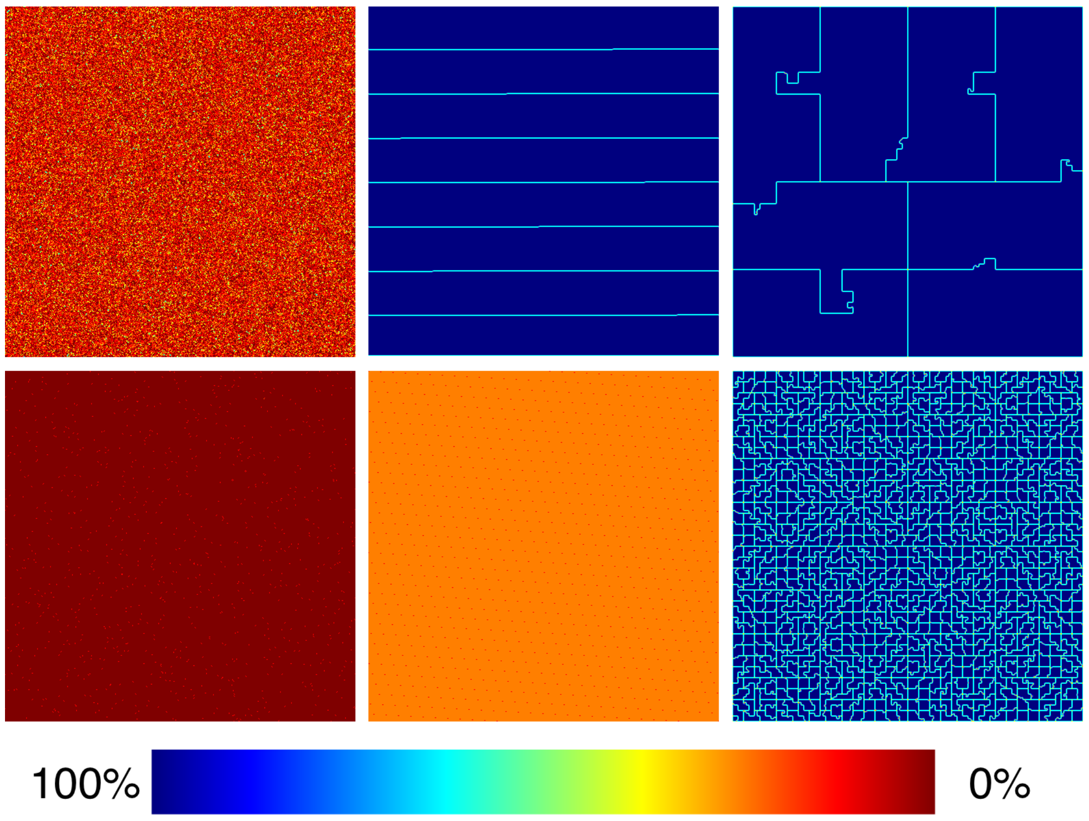

Figure 5.13.: Map of the cache hit ratio for different cache sizes (top row: CPU with a cache size of 33125 particles, bottom row: GPU with a cache size of 256 particles) with different memory layouts: unordered on the left, striped in the center and HILBERT curve on the right. Contiguous blue areas are cached together. The small cache of the GPU requires a more sophisticated reordering algorithm to be effective.

6. Real-world Performance

As a final step, the molecular dynamics simulation itself (calculation of the forces, Verlet integration, calculating the appropriate box for each particle, data transfer between CPU and GPU memories) was ported to CUDA [9] C++ code in order to be executed on the GPU. It was then run on a computer with a Tesla K20 dedicated CUDA card.

We chose a system that was well studied in recent work [1] to better compare and characterize our results. We included hard walls at the top and bottom of the system that are oscillating in phase as well as periodic boundaries in the other dimensions and spring-dashpot interactions with a realistic coefficient of restitution.

The simulations were carried out with different particle densities (parametrized by the volume filling fraction Φ) with both the reorder turned on and off to compare the performance. We study only a dilute gas, that is, only filling fractions up to 26% were tested.

As can be seen in Table 6.1, the performance increase is much more pronounced for larger particle densities, which is expected because particles will be more strongly confined in their relative positions due to the higher occurrence of collisions, which slows down the decorrelation between particle positions in simulation space and memory space.

Also it is clear that the force calculation is responsible for most of the runtime reduction since the total performance improvements closely tracks the performance of the force calculation alone, which is exactly the bottleneck we expected. This has also been verified by profiling the algorithm - the results of which are pictured in Fig. 6.1.

Because of the decay of the cache-hit ratio, the reorder should happen as often as possible to speed up the actual simulation. However, since reordering also takes time, there should be an optimum of simulation steps between reorders such that increasing the frequency of the reorder will decrease performance due to the additional time taken up by the reorder and decreasing the frequency of the reorder will decrease performance due to the additional time taken up by memory reads that missed the cache on the GPU.

The absolute and relative performance of the algorithm have been plotted in Fig. 6.2 and Fig. 6.3 respectively. As can be seen the molecular dynamics simulation already runs twice as fast with a filling fraction of 5% and increases to more than three times as fast with higher filling fractions, which in line with the expectations from the theoretical results.

Computation	Filling Fraction		
	5%	20%	40%
Force calculation	146.8%	297.3%	313.6%
Other	101.8%	124.7%	119.3%
Total	137.5%	273.9%	301.3%

Table 6.1.: Relative performance of reorder with a three-dimensional HILBERT curved compared to no reorder. *Other* includes all operations that include the GPU except the force calculation (cf. Fig. 6.1). Since most of the random memory accesses are done in the Force calculation, it is the only computation that runs significantly faster.

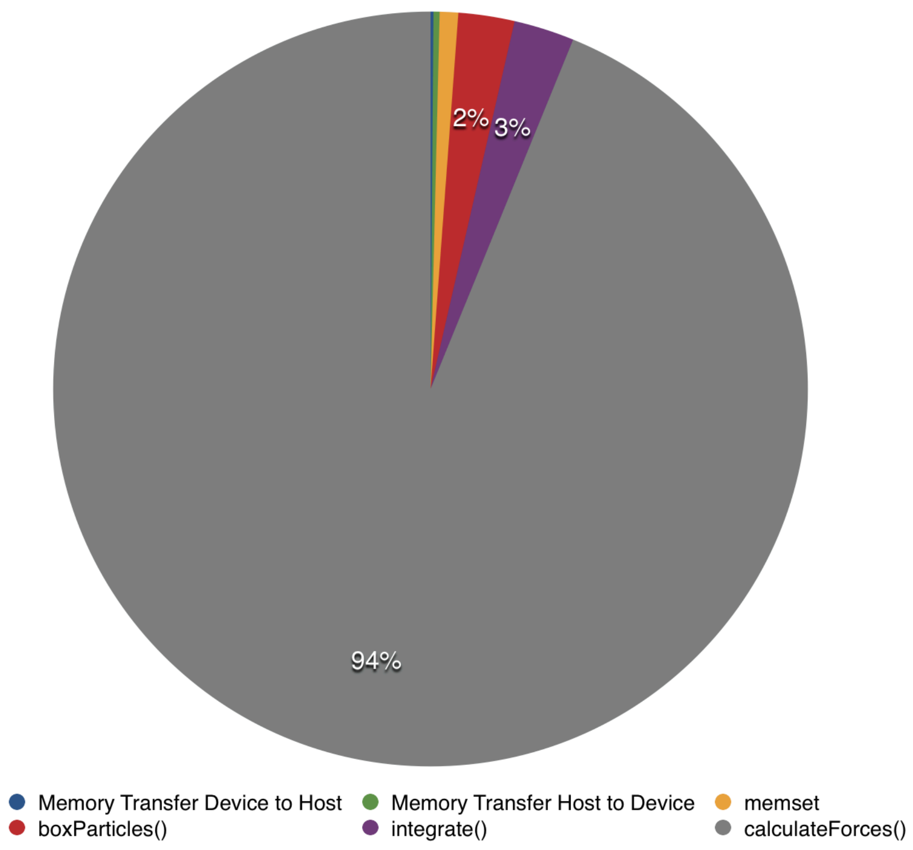

Figure 6.1.: Time spent in different parts of the molecular dynamics simulation when no reorder is used, at a filling fraction of 20%. boxParticles() is the algorithm keeping track of the boxes in which particle currently resides and integrate() is performing a position Verlet integration. Clearly the force calculation is the performance bottleneck in this case, as expected.

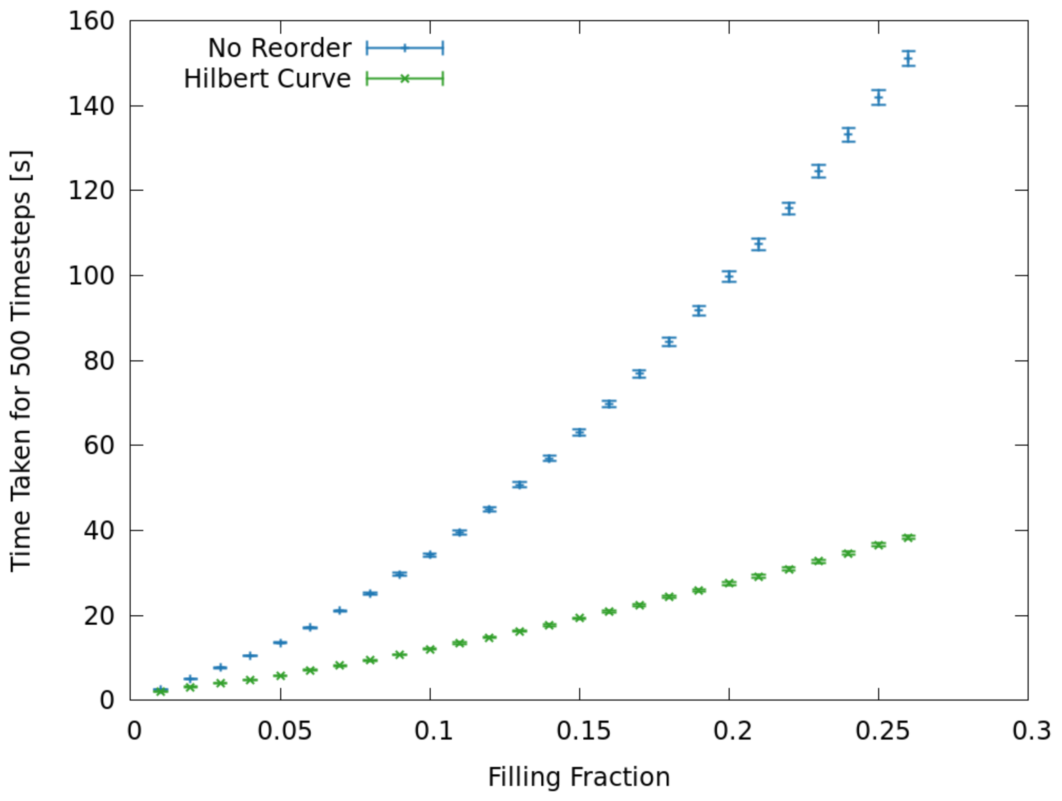

Figure 6.2.: Runtime of molecular dynamics simulation in a system of dimensions 512x512x10 (in units of particle diameter) with walls on the top and bottom that are shaken vertically with frequency 50 Hz and Amplitude 1.8d. spring-dashpot interaction with coefficients of restitution 80% for particle-particle and 90% for particle-wall interactions. The two curves are a) simulation with no reorder and b) simulation with HILBERT pancake reorder every 500 simulation steps.

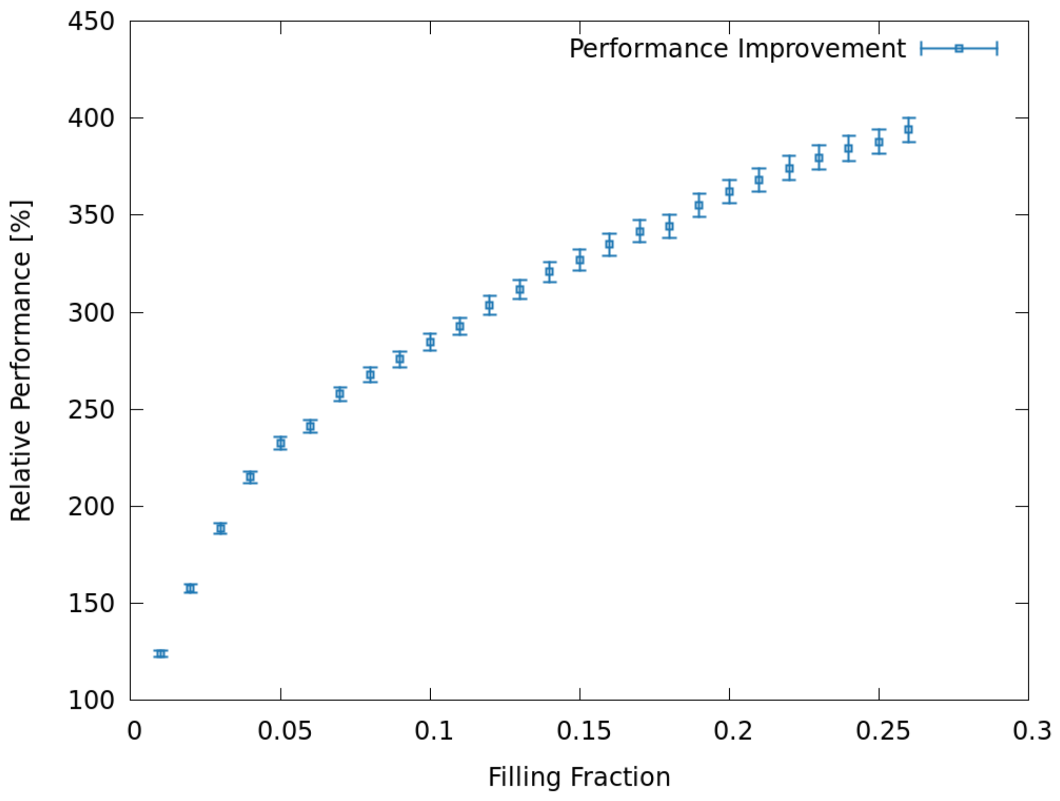

Figure 6.3.: Relative performance of molecular dynamics simulation with the same parameters as in Fig. 6.2.

7. Conclusions

We demonstrated how to use two- and three-dimensional space-filling curves using Lindenmayer replacement systems and constructed a HILBERT space-filling curve that fills a cuboid. When we analyzed the clustering properties of the different curves when reordering memory positions in the two-dimensional case, the HILBERT curve outperformed the others.

We found out that for higher filling fractions the confinement of particles due to collisions significantly helps with the decay of the cache-hit ratio, so our approach is well suited for that scenario. This optimization is quite generally applicable for molecular dynamics simulations of a granular gas as long as the density is not too low.

We checked the correctness of the code by verifying that it correctly reproduces HAFF's law of homogeneous cooling [10].

In general we are quite satisfied with the performance improvements that we have achieved, especially for higher filling fractions. When doing a parameter study with different filling fractions and a system with fixed geometry, naturally the cases with higher filling fractions have a longer runtime because they are dealing with more particles - so this optimization could be quite helpful in dealing with that issue.

Since the only requirement for this type of optimization are short-ranged interactions we expect that they can be generalized to other systems as well.

Because of time-constraints there have been several questions that we did not tackle in this research, these include:

We only assummed that reordering particles on the CPU would be faster due to the large cache (cf. Chapter 5), we would have liked to write an algorithm that reordered the particles on the GPU and see how actual performance compares. Implementing other types of three-dimensional space-filling curves and compare real-life performance would also have been insightful.

There is also a practical result that one would like to know, i.e. given your favourite system, how many simulation steps would have to be executed between reorders to hit the sweet spot between reorders taking too much time and cache-hit ratios decaying too much. In the system we studied, 500 simulation steps was a good compromise, however no rigorous analysis has been performed.

Bibliography

[1] James Clewett. *Emergent Surface Tension in Boiling Granular Media.* PhD thesis, University of Nottingham, 2013.

[2] Nikolai V. Brilliantov; Thorsten Pöschel. *Kinetic Theory of Granular Gases.* Oxford University Press, 2004.

[3] Rob Farber. CUDA, Supercomputing for the Masses: Part 11, March 2009. URL `http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/215900921`.

[4] Hans Sagan. *Space-Filling Curves.* Springer-Verlag, 1994.

[5] Michael Bader. *Space-Filling Curves Space-Filling Curves - An Introduction with Applications in Scientific Computing.* Springer-Verlag, 2013.

[6] Giuseppe Peano. Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen*, 36:157–160, 1890.

[7] David Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, 1891.

[8] Heinz-Otto Peitgen, Hartmut Jürgens, and Dietmar Saupe. *Chaos and Fractals.* Springer-Verlag, 2004.

[9] NVIDIA Corporation. Cuda toolkit, 2015. URL `https://developer.nvidia.com/cuda-toolkit`.

[10] P. K. Haff. Grain flow as a fluid-mechanical phenomenon. *Journal of Fluid*

Mechanics, 134:401–430, 1983.

Acknowledgements

I want to thank Marco Mazza for providing guidance and feedback over the course of my project and bringing many potential ideas to broaden the research to my attention, even though many of them couldn't be realized due to time constraints.

Also many thanks to James Clewett for providing me with the idea and lots of support with the technical aspects, including a crash course on visualization with OpenGL and CUDA programming.

Thanks to Stephan Herminghaus for examining my thesis and to the whole NESM group for providing a great work atmosphere.

A. Appendix

Code to generate 3D Hilbert curve

```c
typedef enum direction3_enum
{
    RIGHT, REAR, UP, LEFT, FRONT, DOWN
} direction3_t;
// Lookup Table for Rotations: new = rotations[axis of rotation][old direction]
static direction3_t rotations[6][6] =
{
  {RIGHT, UP, FRONT, LEFT, DOWN, REAR}, // RIGHT
  {DOWN, REAR, RIGHT, UP, FRONT, LEFT}, // REAR
  {REAR, LEFT, UP, FRONT, RIGHT, DOWN}, // UP
  {RIGHT, DOWN, REAR, LEFT, UP, FRONT}, // LEFT
  {UP, REAR, LEFT, DOWN, FRONT, RIGHT}, // FRONT
  {FRONT, RIGHT, UP, REAR, LEFT, DOWN}  // DOWN
};
// Lookup Table for opposite direction new = opposite[old direction]
static direction3_t opposite[6] = {LEFT, FRONT, DOWN, RIGHT, REAR, UP};

void roll_left(void)
{
  up = rotations[front][up];
}
void roll_right(void)
{
  up = rotations[opposite[front]][up];
}
```

```
void turn_left(void)
{
  front = rotations[up][front];
}
void turn_right(void)
{
  front = rotations[opposite[up]][front];
}
void pitch_down(void)
{
  direction3_t tmp;
  tmp = front;
  front = opposite[up];
  up = tmp;
}
void pitch_up(void)
{
  direction3_t tmp;
  tmp = front;
  front = up;
  up = opposite[tmp];
}
void turn_around(void)
{
  front = opposite[front];
}
void move_forward(void)
{
  switch(front)
  {
    case RIGHT: x++;  break;
    case LEFT:  x--;  break;
    case UP:    y++;  break;
    case DOWN:  y--;  break;
    case REAR:  z++;  break;
    case FRONT: z--;  break;
  }
}
void curve3D_iterate(int depth, int system)
{
  // [...]
  static char lindenmayer[11][100] = {
    // Hilbert Curve 3D - there are four different chiral versions
    "B-#+C#C+#-D&#^D-#+&&C#C+#+B<<",      // A system == 0
    "A&#^C#B^#^D^^-#-D^|#^B|#C^#^A<<",    // B system == 1
    "|D^|#^B-#+C^#^A&&#A&#^C+#+B^#^D<<",  // C system == 2
    "|C#B-#+B|#A&#^A&&#B-#+B|#C<<"        // D system == 3
  };
  for(C = lindenmayer[system]; *C != '\0'; C++)
  {
    switch(*C)
    {
      case '#': move_forward(); break;

      case '+': turn_left();    break;
      case '-': turn_right();   break;

      case '&': pitch_down();   break;
      case '^': pitch_up();     break;

      case '<': roll_left();    break;
      case '>': roll_right();   break;

      case '|': turn_around();  break;

      default:  curve3D_iterate(depth, *C - 'A');
    }
  }
}
```

CUDA Force Calculation

```
void calculateForcesGPU(void)
{
  cudaMemset(d_fx, 0x00, nparticles*sizeof(double));
  cudaMemset(d_fy, 0x00, nparticles*sizeof(double));
  cudaMemset(d_fz, 0x00, nparticles*sizeof(double));

  calculateForcesKernel<<<nBlocks, blockSize>>>(
                                        wall_position, mu, mu_wall,
                                        d_rootNodes, d_nextParticle,
                                        d_x,  d_y,  d_z,
                                        d_vx, d_vy, d_vz,
                                        d_fx, d_fy, d_fz,
                                        gridsize, nparticles);
  checkCudaError(cudaGetLastError(), __FILE__, __LINE__);
  checkCudaError(cudaDeviceSynchronize(), __FILE__, __LINE__);
}
__global__ void calculateForcesKernel(double wall_position,
                                      double mu, double mu_wall,
                                      int *rootNodes, int *nextParticle,
                                      double *x,  double *y,  double *z,
                                      double *vx, double *vy, double *vz,
                                      double *fx, double *fy, double *fz,
                                      int gridsize, int nparticles)
{
  int i,j,k,l,m;
  int rows[3], columns[3], layers[3]; // row/column of current box.
  double dx, dy, dz;        // Relative Position.
  double dv_x, dv_y, dv_z; // Relative Velocity.
  double distance;          // Distance between Particles.
  double speed;             // Relative tangential Speed.
  double F;
  double l_x, l_y, l_z;
  double l_fx = 0.0, l_fy = 0.0, l_fz = 0.0;
#ifdef SHAKE
```

```
  double dz1, dz2;
#endif

  i = blockDim.x * blockIdx.x + threadIdx.x;

  if(i >= nparticles)
  {
    return;
  }

  l_x = x[i];
  l_y = y[i];
  l_z = z[i];

  #ifdef SHAKE
  dz1 = wall_position + 0.5 - l_z;
  dz2 = SHAKE_BOXHEIGHT + wall_position - 0.5 - l_z;

  if(dz1 > 0.0)
  {
    l_fz =  K*dz1 - mu_wall*vz[i];
  }
  else if(dz2 < 0.0)
  {
    l_fz =  K*dz2 - mu_wall*vz[i];
  }
  #endif

  columns[0] = (int) l_x;
  columns[1] = (columns[0]+1)%gridsize;
  columns[2] = (columns[0]-1+gridsize)%gridsize;

  rows[0] = (int) l_y;
  rows[1] = (rows[0]+1)%gridsize;
  rows[2] = (rows[0]-1+gridsize)%gridsize;
```

```
layers[0] = (int) l_z;
layers[1] = (layers[0]+1)%gridsize;
layers[2] = (layers[0]-1+gridsize)%gridsize;

for(k = 0; k < 3; k++)
{
  for(l = 0; l < 3; l++)
  {
    for(m = 0; m < 3; m++)
    {
      j = rootNodes[ columns[k] + gridsize * (rows[l] + gridsize*layers[m]) ];

      // Follow linked list.
      while(j != -1)
      {
        if(i == j)
        {
          j = nextParticle[j];
          continue;
        }
        // If |dx|,|dy| or |dz| > 2,
        // the particle must be behind the periodic boundary.

        dx = x[j] - l_x;
        if(dx >  2.0)
        {
          dx -= gridsize;
        }
        if(dx < -2.0)
        {
          dx += gridsize;
        }

        dy = y[j] - l_y;
```

```
if(dy >  2.0)
{
  dy -= gridsize;
}
if(dy < -2.0)
{
  dy += gridsize;
}

dz = z[j] - l_z;
if(dz >  2.0)
{
  dz -= gridsize;
}
if(dz < -2.0)
{
  dz += gridsize;
}

distance = dx*dx + dy*dy + dz*dz;
if(distance > 1.0)
{
  j = nextParticle[j];
  continue;
}
distance = sqrt(distance);

dx    = dx/distance;
dy    = dy/distance;
dz    = dz/distance;

dv_x  = vx[j] - vx[i];
dv_y  = vy[j] - vy[i];
dv_z  = vz[j] - vz[i];
```

```
          speed = dv_x*dx + dv_y*dy + dv_z*dz;
          F = -K*(1.0-distance) + mu*speed;

          // F is oriented along (dx,dy,dz) by construction.
          l_fx += dx*F;
          l_fy += dy*F;
          l_fz += dz*F;

          j = nextParticle[j];
        } // while(j != -1)
      } // for(m = 0; m < 3; m++)
    } // for(l = 0; l < 3; l++)
  } // for(k = 0; k < 3; k++)

  fx[i] = l_fx;
  fy[i] = l_fy;
  fz[i] = l_fz;
}
```

Erklärung nach §13(8) der Prüfungsordnung für den Bachelor-Studiengang Physik und den Master-Studiengang Physik an der Universität Göttingen:

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe.
Darüberhinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, im Rahmen einer nichtbestandenen Prüfung an dieser oder einer anderen Hochschule eingereicht wurde.

Göttingen, May 22, 2015

(Niklas Bölter)