Hermann Günter Kureck, BSc.

Design and Implementation of Massively Parallel Algorithms on GPUs for Particle Simulation

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Telematics

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl-Ing. Dr.techn. Christian Steger

Institute for Technical Informatics

Advisors: Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger
Dr.rer.nat. Charles Radeke (RCPE GmbH)

Graz, May 2016

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis dissertation.

Date

Signature

Kurzfassung

Die Diskrete-Elemente-Methode (DEM) ist eine numerische Methode zur Berechnung von Bewegungsabläufen von Teilchen, wobei im Zuge der Simulation jedes Teilchen separat betrachtet wird. Da üblicherweise sehr große Partikelanzahlen und kleine Zeitschritte notwendig sind, um reale Prozesse in ausreichender Genauigkeit abbilden zu können, werden parallele Implementierungen der DEM verwendet, um in annehmbarer Zeit Ergebnisse liefern zu können.

In erster Linie ist es bei dieser Methode relevant zu wissen, ob sich Partikel berühren, da dies eine Kraftwirkung zur Folge hat. Deshalb ist auch der Algorithmus, der zur Nachbarsuche verwendet wird, kritisch für die Gesamtperformance. Zusätzlich werden Änderungen in der aktuellen Implementierung notwendig sein, um die spezifischen Eigenschaften der Zielplattform (GPUs, CUDA®) auszunutzen und eine gute Gesamtperformance zu erreichen. Im Speziellen läuft dies auf eine Trennung von Nachbarsuche und Kraftberechnung hinaus.

Ziel der Arbeit ist es nun die bestehenden Algorithmen zu verbessern (Kontaktlisten, Verfolgen von Kontakten solange aktiv) und Alternativen (Bounding Volume Hierarchie, BVH) zu evaluieren. Nach Beschreibung der Algorithmen und der notwendigen Schritte zur Implementierung wird mit Hilfe von geeigneten Testfällen gezeigt dass die Simulationsergebnisse statistisch gleichwertig sind und, je nach Anwendungsfall, ein sehr guter Speedup erreicht wird.

Schlüsselwörter

Diskrete-Elemente-Methode, Partikelsimulation, CUDA®, GPU, Nachbarsuche, Kontaktliste, Bounding Volume Hierarchie

Abstract

The Discrete Element Method (DEM) is a numerical simulation method for computing the motion of particle systems, being aware of each single particle throughout the simulation. As there typically are huge amounts of particles needed to accurately model real-world problems, in conjunction with small time steps, massively parallel algorithms are used to make computation times acceptable.

In the first place, the DEM addresses short-range forces during particle contact. Therefore, the used neighbor search algorithm is critical for the overall performance. Additionally, due to parallel algorithms and special platform requirements when targeting graphics cards using CUDA®, major changes in the existing implementation are required to achieve good performance, namely decoupling neighbor search and force calculation.

Objectives of this thesis include optimization of existing algorithms (introducing contact lists and tracking), as well as implementation of promising new approaches (Bounding Volume Hierarchies, BVH). After description of the algorithms and implementation steps it is shown that the accuracy has not changed for designed test cases, but significant speed-ups can be observed.

Keywords

Discrete Element Method, Particle Simulation, CUDA®, GPU, Neighbor Search, Contact List, Bounding Volume Hierarchy

Acknowledgment

This master thesis was carried out during the years 2015/2016 at the Institute for Technical Informatics at Graz University of Technology. I am deeply grateful to Christian Steger for the opportunity to write this thesis and for his support during that period.

This thesis would not have been possible without support of the Research Center of Pharmaceutical Engineering. I would like to express my gratitude to Charles Radeke for introducing me to the fascinating world of particle simulation and for enriching discussions in this matter. Also, I would like to thank all the other colleagues at work for their support, especially Georg Neubauer, who worked on a related thesis in parallel, for the constructive collaboration.

Last, but not least, I would like to thank my family and friends for providing me with continuous encouragement throughout my years of study and through the process of writing this thesis.

Graz, May 2016 Hermann Kureck, BSc.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This master thesis was written in cooperation with the Research Center Pharmaceutical Engineering (RCPE) and the Institute for Technical Informatics of the TU Graz (ITI). The following introduction gives an overview of this thesis' motivation, objectives, and goals.

1.1 Motivation

In industry the possibility of doing simulations before building prototypes often gives a decisive time advantage when optimizing production processes. This is true for many different branches, for example food industry, or cosmetics. For the pharmaceutical industry such typical processes include for example tablet pressing, filling capsules, or coating tablets with active ingredients.

The Discrete Element Method (DEM) is a numerical simulation method for computing the motion of particle systems [CS79], originally developed for rock mechanics problems. Typically a large number of particles is needed to accurately model real-world problems like granular flow or powder mechanics. As the most important non-constant contribution to the particle motion are the forces due to colliding particles, which is kind of a short-range force, the problem itself can be highly parallelized if looking at each particle individually. The used neighbor search algorithm therefore has critical impact on the overall algorithm performance.

At the RCPE the DEM software prototype eXtended Particle System (XPS) is being developed since 2011. It is able to couple with the Computational Fluid Dynamics (CFD) solution AVL FIRE®, extending its possibilities. Currently implemented particle types are spherical, glued-spheres, bi-convex tablets, and polyhedra which can be seen in figure 1.1. Features included are for example arbitrary geometries (e.g. a mixer) via a triangle mesh (optionally moving, e.g. rotating), heat transfer capability, and coating ability via

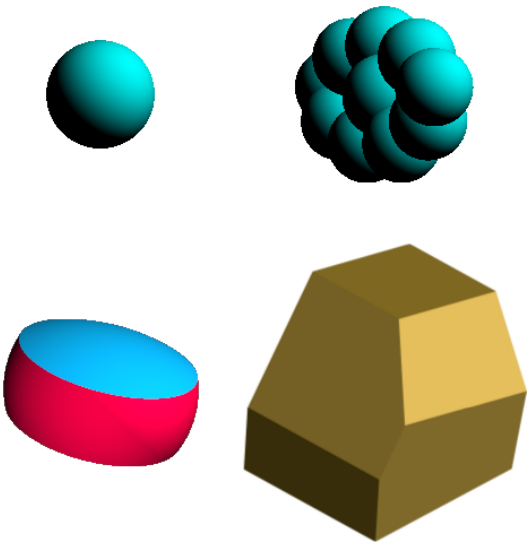

Figure 1.1: Shapes currently implemented in XPS - Sphere, Multi-Sphere (also known as Clump-Sphere or Glued-Sphere), bi-convex tablet, polyhedron. (Source: RCPE)

a ray-tracing method. To make simulations on single workstations feasible the simulation core was written in the parallel computing platform Compute Unified Device Architecture (CUDA®) (see chapter 2.2), to use the massive computing power of NVIDIA graphics cards for general purpose computing. [RGK10]

Examples of simulations done with XPS are shown in figures 1.2 and 1.3. The first one is a DEM-only simulation of a simple mixer device, the other one is a combined CFD-DEM simulation of a so-called Wurster-Coater where particles get fluidized due to an inlet airflow, and coated (i.e. a liquid sprayed onto the particles).

1.2 Objectives

The main objective of this thesis is to analyze different approaches which could improve the overall performance of the simulations. Currently the neighbor search is done using an uniform grid, where each particle in the neighbor cells can be easily accessed and checked for collision. This simple method is very efficient, but may not be suitable for every real-world problem. Two major issues, which already have been discussed in [Kar12b], could deduce algorithm performance:

- Mixing a large number of small particles with big particles, which requires a big cell size and lots of collision checks.

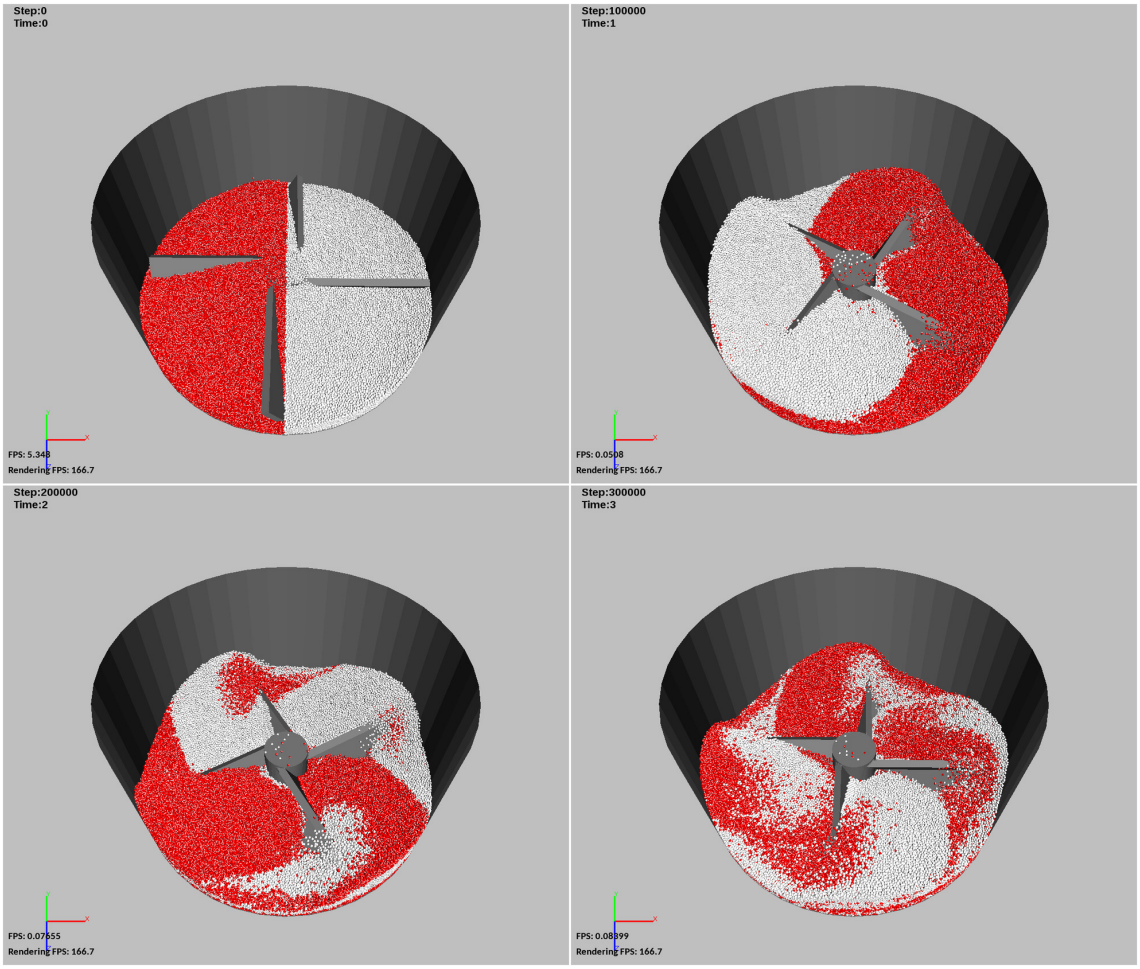

Figure 1.2: DEM-only simulation of a simple mixer device where the blades are rotating with 40 revolutions per minute. The upper-left part is the initial state at $t = 0\,s$, where particles got color-tagged. Upper-right, lower-left and lower-right are snapshots at times $t = 1\,s$, $t = 2\,s$ and $t = 3\,s$, respectively. (Source: RCPE)

- The so-called teapot-in-a-stadium problem, if a big part of the simulation domain does not contain particles at all. In this case lots of memory gets wasted on empty cells.

To overcome these issues one part of this thesis should be dedicated to evaluate tree structures, the so-called Bounding Volume Hierarchy (BVH), and to compare with uniform grid spatial subdivision already implemented in XPS.

On the other hand, it makes a difference how complex the collision detection and force calculation is. In case of really simple particle shapes like spheres, one kernel which executes both steps together is fine. Recently more complex shapes were implemented in XPS, for example bi-convex tablets and so-called polyhedra, but performance decreased hugely. Due to execution divergence (see chapter 2.2) it would be a good idea to divide the collision step into broad- and narrow-phase collision detection, which means collecting neighbors in one step and calculating forces separately.

Another important point is to make contact lists available to optionally track contacts over time, which can for example be used to implement history-dependent tangential forces needed in some models, or to visualize force chains. As for splitting the collision step some kind of neighbor list is necessary anyway to store the potential collision partners, this might be a good starting point.

Last but not least the performance and correctness of the algorithms should be analyzed with suitable test cases.

1.3 Structure

The following chapter 2 explains the basics of the DEM and the CUDA® developing platform. Subsequently an analysis of the current implementation in XPS is given, as well as alternative approaches for collision detection will be discussed in chapter 3 and 4. In chapter 5 implementation details of the chosen algorithms are presented. Finally, in chapter 6 appropriate test cases are chosen, along with the performance measures carried out, and a subsequent discussion is given. Validation of the new algorithms with standard test cases is done as well, to show that the simulation results have not changed in statistical manner. Last but not least, chapter 7 gives a conclusion of what has been done as well as an outlook for possible improvements in the future.

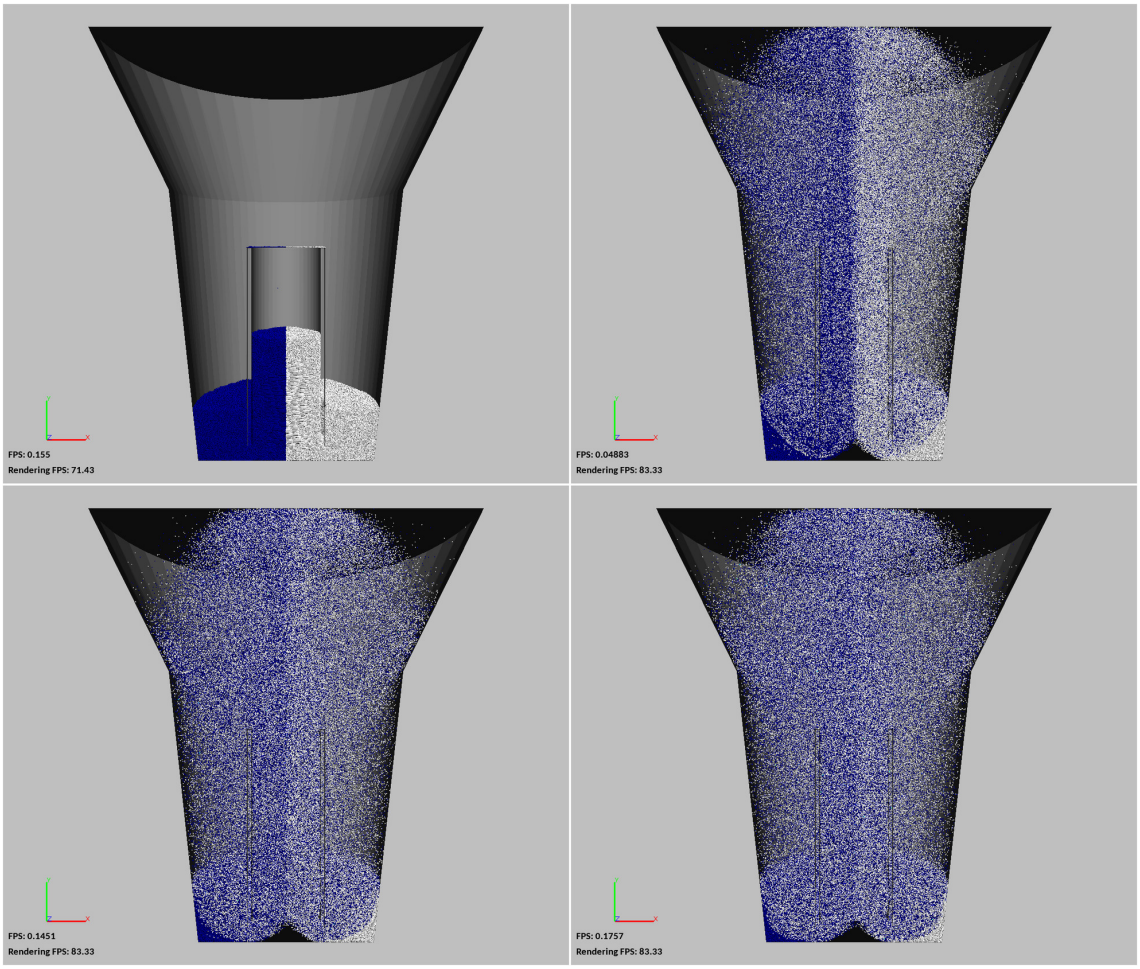

Figure 1.3: Combined CFD-DEM simulation of a so-called Wurster-Coater where particles get fluidized. The upper-left part is the initial state at $t = 0\,s$, where particles got color-tagged. Upper-right, lower-left and lower-right are different snapshots as time progresses. (Source: RCPE)

Chapter 2

Theory

2.1 Discrete Element Method (DEM)

The method, invented by Cundall in 1971 ([Cun71], [CS79]), is a numerical method for computing the motion of a large number of particles. Firstly, the particles are positioned and given an initial velocity. Then this time-step based method calculates all kind of forces the particles are exposed to and then changes position and velocity of each particle according to some numerical integration scheme to solve Newton's equations of motion

$$\vec{F_i} = \frac{\mathrm{d}\vec{p_i}}{\mathrm{d}t} \tag{2.1}$$

$$\vec{M_i} = \frac{\mathrm{d}\vec{L_i}}{\mathrm{d}t} \tag{2.2}$$

where $\vec{F_i}$ is the overall force, $\vec{M_i}$ is the overall torque acting on particle i, $\vec{p_i}$ is the momentum, and $\vec{L_i}$ is the angular momentum of particle i. Calculating forces followed by numerical integration is done in a loop until a specified end time is reached. What is used in XPS is a so called soft-sphere model, where particles can overlap, and depending on the specific overlap volume the collision force direction and magnitude is calculated. Possible values for the time step depend on various particle properties, maximum velocities and material parameters. To keep the overlap between particles low usually really small time step values have to be chosen, for example $t_{\mathrm{step}} = 10^{-5}$ seconds is a commonly used value. So in this example for only a second of process time to be simulated 10^5 time steps have to be calculated, including at least search of neighbors, calculation of forces, and numerical integration. If a feasible simulation with millions of particles and maybe minutes of process time is desired, the neighbor search has to be fast, as the trivial algorithm would, for each particle, loop over all other particles, which results in time complexity $\Theta(n^2)$. What is currently done to avoid this is explained in 2.3.

2.2 Graphics Processing Units and CUDA

Graphics Processing Units (GPUs) are specialized for graphics rendering and therefore are highly parallel, multi-threaded processors with many cores and have massive computation power along with a very high memory bandwidth. In 2006 NVIDIA invented CUDA® with the aim of getting GPUs widely used also for general-purpose programming, an approach which is widely known as General Purpose Computation on Graphics Processing Units (GPGPU). [NVI15]

2.2.1 Programming Model

CUDA® extends the C language by letting the programmer define functions which should be executed N times in parallel by N different threads. These functions are called kernels and are declared with the *__global__* specifier to let the compiler know that this code should be compiled for the GPU. Such a kernel is invoked on the Central Processing Unit (CPU)-side by not only a function call, but also specifying how much threads should be created via the so-called execution configuration syntax *<<< number of blocks, threads per block >>>*. So the caller has to define how many blocks and how many threads per block should be created. A short example code which adds two arrays A and B element-wise and stores the result in array C could look as follows:

```
1  // Kernel definition
2  __global__ void VecAdd(float* A, float* B, float* C) {
3    int i = threadIdx.x;
4    C[i] = A[i] + B[i];
5  }
6  int main() {
7    ...
8    // Kernel invocation with N threads
9    VecAdd <<< 1, N >>> (A, B, C);
10   ...
11 }
```

The number of blocks is also called grid-size, the number of threads per block is also known as block-size.

Both grid- and block-size can be two- or three-dimensional types as well, if this suits the problem better or is more convenient to use (e.g. for matrices or images given as multi-dimensional arrays). Each thread has access to its own thread index (range [0, block-size - 1]) and block index (range [0, grid-size - 1]) in each dimension. A configuration of having 6 blocks with 12 threads each is illustrated in figure 2.1.

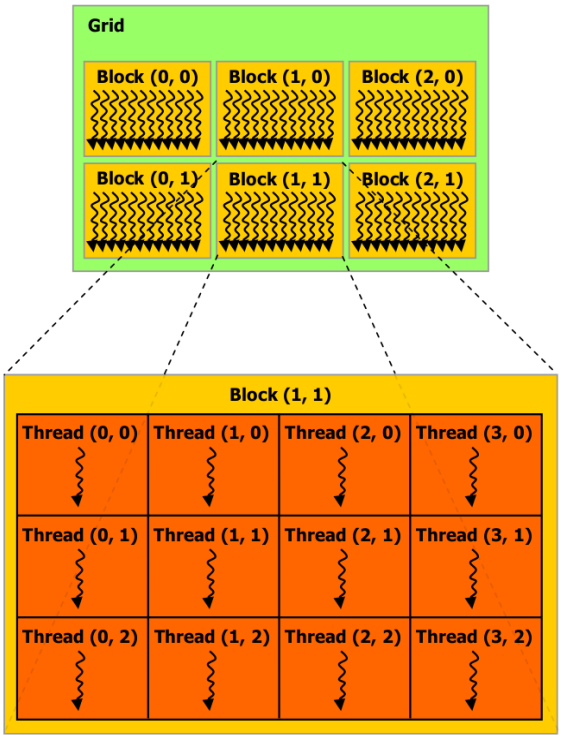

Figure 2.1: Grid of 6 thread blocks with 12 threads per block. (Source: [NVI15])

Besides the *__global__* specifier there exist also *__host__* and *__device__*, where *host* always refers to the CPU and *device* refers to the GPU. This means for example that a *device* function can only be called from another *device* function or from a *global* function because it is only compiled for the *device* and not for *host*. [Wil13], [NVI15]

2.2.2 Hardware Implementation

When a CUDA® kernel is invoked the blocks of the grid are distributed to so-called Streaming Multiprocessors (SM, SMX). Such a multiprocessor is designed to execute hundreds of threads concurrently. Each multiprocessor is assigned multiple thread blocks (if there are enough), and the threads of a block are executed concurrently. Different GPUs have a different number of SMs, but the programming model is scalable as less multiprocessors only lead to more blocks, and therefore more work, assigned to each SM, see figure 2.2.

The architecture used is Single-Instruction Multiple-Thread, what means that the SM partitions blocks into warps which then are managed, scheduled, and executed. A warp is a group of 32 threads and always executes one common instruction (i.e. common for all the threads in a warp). However, each thread still has its own register state and instruction address counter, which allows for independent execution. Still, full efficiency

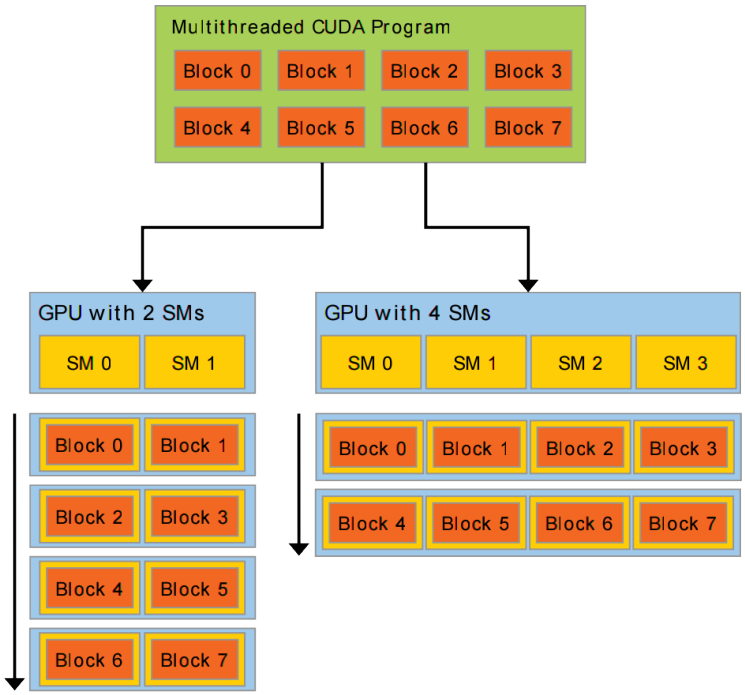

Figure 2.2: Scalability when using a different number of Streaming Multiprocessors (Source: [NVI15])

is only reached if each thread in a warp executes the same instruction at a time. If that is not the case serialization of the different branch paths happens, with some threads being inactive. During the lifetime of a warp the execution context is stored on-chip, so a context switch has no cost.

Available resources should always be kept in mind as the amount of registers per multiprocessor is limited and the number of blocks and warps being processed on a multiprocessor depend on register usage and requested shared memory (see next chapter 2.2.3). Depending on the compute capability of the device there is also a maximum number of resident warps/blocks per multiprocessor. [Wil13], [NVI15], [Kre11]

2.2.3 Memory Hierarchy

In figure 2.3 the memory hierarchy is illustrated. Most of the data used for processing on the GPU is usually stored in the so-called global memory, which is located in device memory and accessible from each thread. On the other hand each thread has some private local memory, which for example is used if register spilling occurs, i.e. if there are not enough registers available for kernel execution. This should be avoided because this kind of memory also resides in device memory and therefore accesses have comparably low bandwidth and high latency.

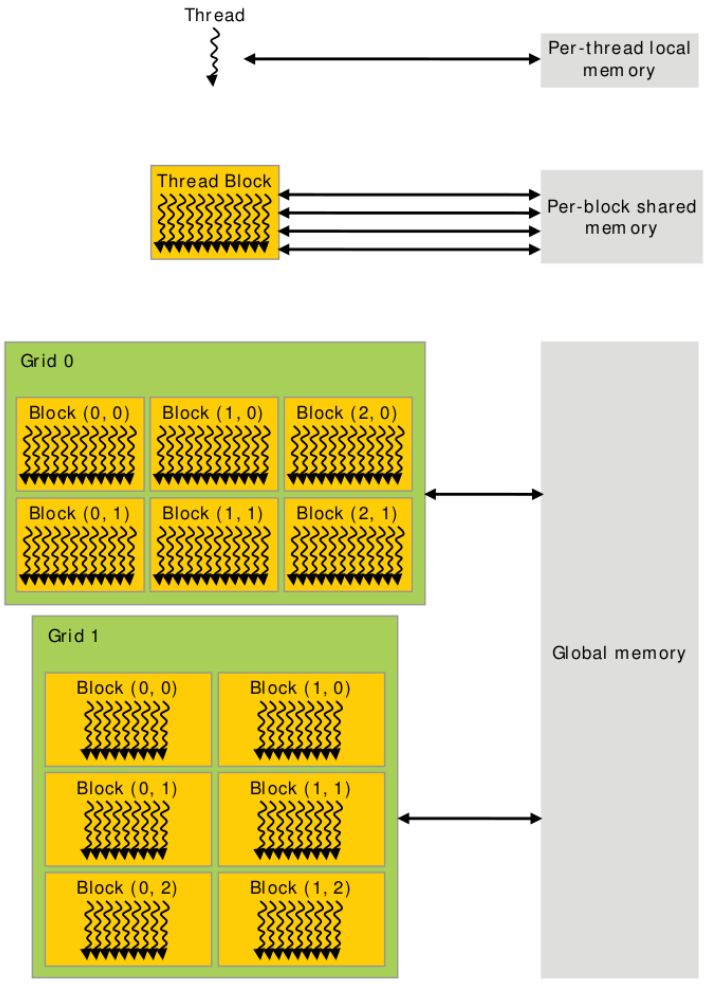

Figure 2.3: Memory hierarchy. (Source: [NVI15])

When it comes to optimizing memory throughput it is important to know that a warp coalesces accesses into one or more memory transactions, depending on word sizes and addresses. Usually it is best to only use words of size 1, 2, 4, 8 or 16 bytes which are not scattered, i.e. thread 5 reads the 5^{th} element of an array and thread 8 reads the 8^{th} one. Depending on the architecture different accessing modes (e.g. strides) lead to a different memory throughput. A general rule is that more scattered memory accesses lead to a lower memory throughput.

Minimizing global memory accesses can be achieved using different on-chip caches (e.g. L1 and L2 cache available depending on device architecture), or using shared-memory which can be seen as a user-managed cache. It is possible to have a limited amount of this on-chip shared memory per thread block, accessible with higher bandwidth and lower latency compared to global memory. The exact amount is specified via the execution configuration syntax. If used, synchronization primitives are available for a safe communication between threads of a block (*__syncthreads()*).

The L1 and L2 cache are traditional hardware-managed caches. Usually the L1 cache uses the same on-chip memory as shared memory, so the available resources should be always kept in mind. Additional cache types are the so-called constant cache, which is used for the constant memory space (resided in device memory, readable only), and the texture cache which can have benefits over direct global memory reads if the access patterns are not optimal. Additionally texture reads feature interpolation modes, and address calculation by dedicated units. [Wil13], [NVI15]

2.2.4 Warp Intrinsics

With the Kepler^{TM} architecture NVIDIA introduced warp shuffle intrinsics for exchanging data within a warp more efficiently than possible via shared memory. These functions either return data from specific threads or shuffle elements up or down within a warp. Also available are so-called warp voting functions which evaluate predicates for all active threads of a warp and then broadcast the result. [Wil13], [NVI15]

2.2.5 Atomic Functions

Atomic functions are available on current architectures to enable atomic read-modify-write operations on global or shared memory. For example, discussed later in chapter 5, for processing the neighbor lists *atomicAdd(address, value)* is used to atomically add forces if more than one thread could do so concurrently. [Wil13], [NVI15]

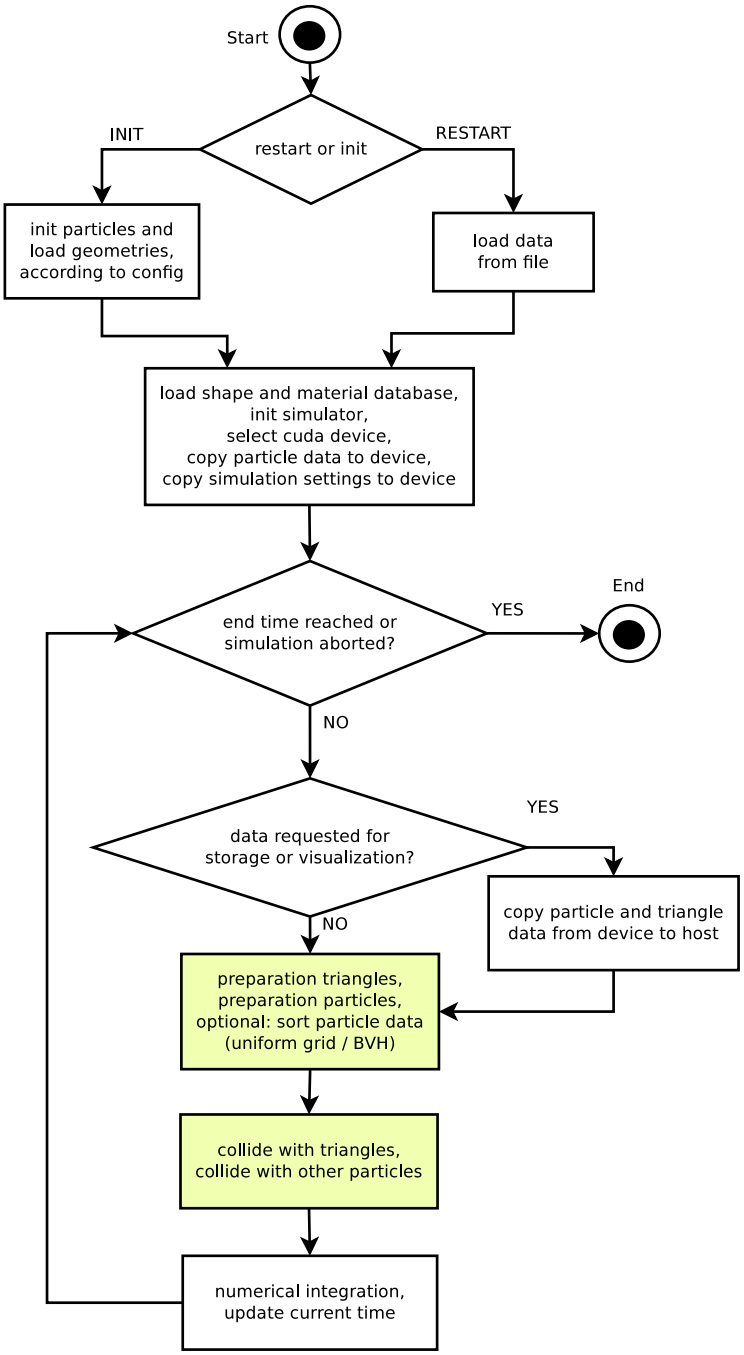

Figure 2.4: XPS program flow. Colored steps are discussed in detail. (Source: RCPE)

Figure 2.5: Objects assigned to uniform grid cells - how to find potential collision partners. (Source: [Kar12b])

2.3 XPS - Current Implementation

The simplified XPS program flow, where only the basic steps necessary for a pure DEM-only simulation are given, is illustrated in figure 2.4. Particles and geometries are either loaded from a restart file or freshly initialized according to input files. A configuration file has to be given, along with shape and material databases where detailed information about the particles to be simulated is given. After selection of the CUDA® device to use, all the needed data is copied to the device, followed by the simulation loop.

The algorithms used in XPS for broad-phase collision detection will be presented in this section. Basically, an uniform grid is used and particles are assigned to cells, illustrated in figure 2.5. For the collision detection itself an easy access to the particles of a cell is needed. The cells have to be at least the size of the biggest particle diameter to assure that all potential collision partners are located inside the $3^3 = 27$ cells around the cell containing the particle of interest. How particles are assigned to cells and how the data structures for looping over particles in a cell are created is described in the following sub-sections. [RGK10]

2.3.1 Uniform Grid Sorting

Figure 2.6 presents a 2D-simplified version of the used algorithm to make easy access to particles in a cell possible. In XPS this step is also known as preparation step. Firstly, each particle is assigned a cell-hash-value, which is a simple mapping from the three-dimensional cell-position to one number:

$$\begin{aligned} \text{cellPos}.xyz &= \left\lfloor \frac{\text{pos}.xyz - \text{worldOrigin}.xyz}{\text{cellSize}.xyz} \right\rfloor \\ \text{hash} &= \text{cellPos}.z \cdot \text{gridSize}.y \cdot \text{gridSize}.x + \text{cellPos}.y \cdot \text{gridSize}.x + \text{cellPos}.x \end{aligned} \tag{2.3}$$

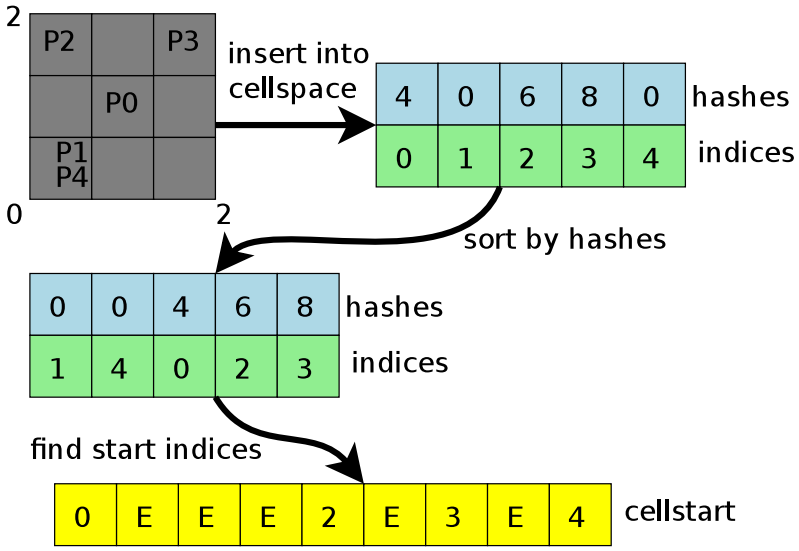

Figure 2.6: Uniform grid algorithm - assign particles to cells, sort, and retrieve cell-start indices. (Source: RCPE)

Then the particle indices are sorted with hash-values as keys, leading to particle-indices of particles in one cell being contiguous in memory. Now the cell-start vector is introduced with length equal to the number of cells in the simulation domain. This vector is then filled with a magic number equal to the maximum integer value, representing empty cells (**E** in the figure). Afterwards, the cell-start vector is filled with the starting indices for each cell. That is done by looking at the indices where the hash-values, and therefore the cell positions, change and storing these indices in the appropriate positions in the cell-start vector. In pseudo-code that is essentially just those two lines, ignoring index checks:

```
1 if(hashes[i] != hashes[i-1])
2   cellstart[hashes[i]] = i;
```

Here i is the thread index, where one thread is started for each hash value in the sorted hash-vector. A more sophisticated approach is used in XPS to avoid loading two hash values per thread, that is using shared memory and sharing the i^{th} value with the thread i+1.

How the cell-start vector is used can be seen in the next section where the collision kernel is explained. In principle, each cell-start value points to the first particle of the according cell.

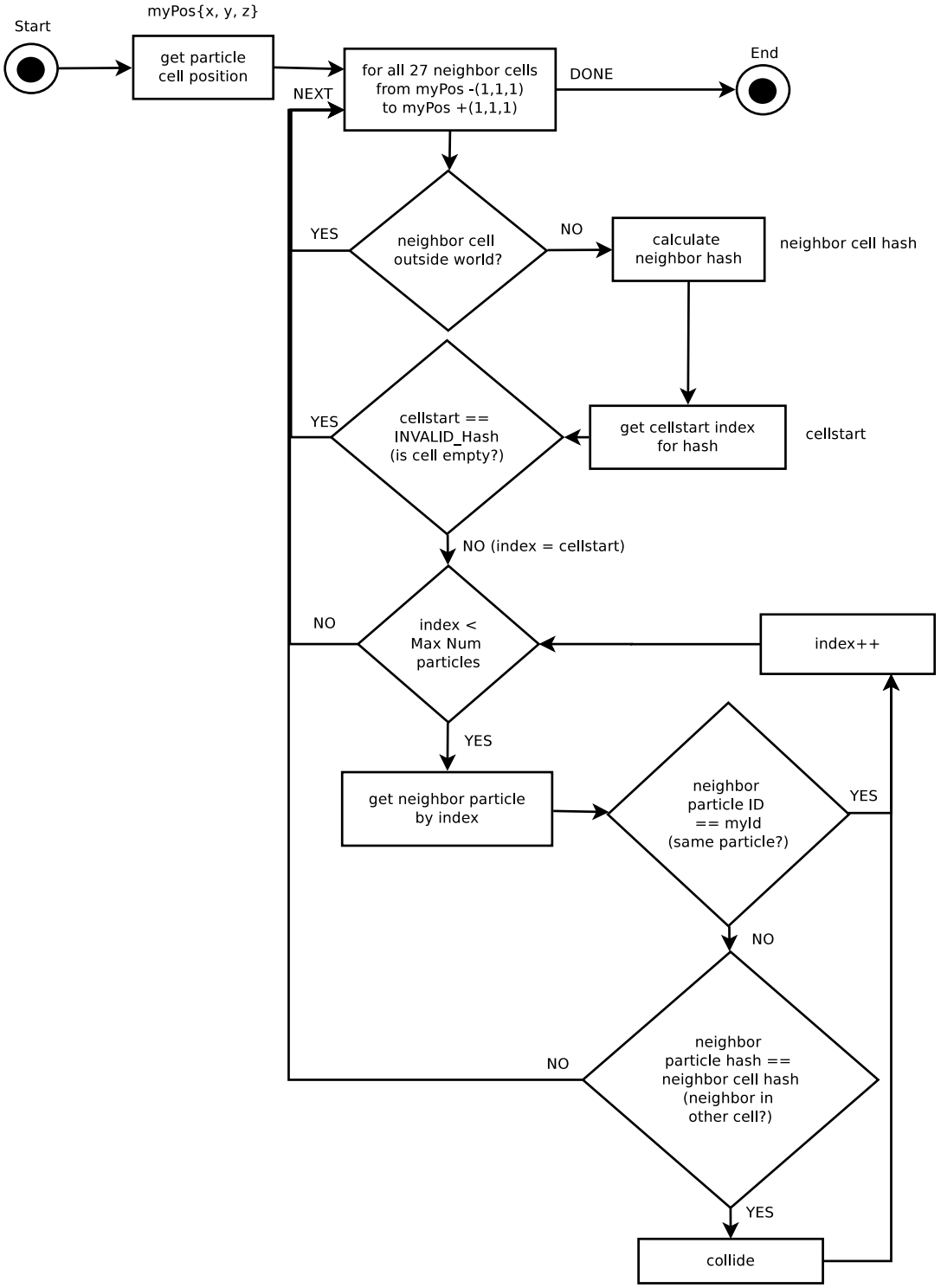

Figure 2.7: Collision kernel - neighbor search via uniform grid. (Source: RCPE)

2.3.2 Collision Kernel

In figure 2.7 a flow-chart is given of how looping through potential collision partners is done using the uniform grid. At first, the particle position has to be read from global memory, followed by calculation of the cell position introduced in equation 2.3. With proper cell sizes, as already discussed, for each particle looping over the 27 neighbor cells is sufficient. From each cell position the cell hash then is evaluated, which allows for looping over the particles in a specific cell, if the cell is not empty. This is achieved using the pair of vectors introduced as hashes and particle indices in figure 2.6. For each neighbor particle it is then decided if a collision can occur or not. The end of this loop is reached when the particle hash changes, i.e. a different cell is accessed. [RGK10]

2.3.3 Handling Particle-Geometry Collisions

For the purpose of geometry input, XPS is capable to read STereoLithography (STL) files, giving an arbitrarily detailed triangulated surface of the real-world geometry. Collision detection is also based on the uniform grid, what means the triangles have to be inserted into the uniform grid as well. The main difference is that a single triangle is located in more than one cell, in general. Therefore, the triangle preparation step, as it is called in XPS, roughly consists of the following parts:

1. For each triangle, count number of occupied cells.
2. Allocate memory for indices and hashes vector for all those cells.
3. Do step 1 again, instead of counting, fill indices and hashes vector with appropriate values.
4. Similar as for particles (figure 2.6), sort indices by hash values.
5. Find cell-start indices by looking at changing hash values, indicating a change in cell position.

In the collision step, similar as for particle neighbor search, the 27 surrounding cells are looped through, where all the triangles located in a cell can be accessed by looking at the cell-start vector.

2.3.4 Memory Management

All the dynamic data of the objects in the simulation is usually kept on the GPU, only updating the data on host-side if needed for visualization or storing results. Read-only values often loaded from memory such as shape parameters, material parameters, gravity, simulation domain, or size of uniform grid, are stored in constant memory, and therefore enable fast access via the constant cache (see chapter 2.2.3).

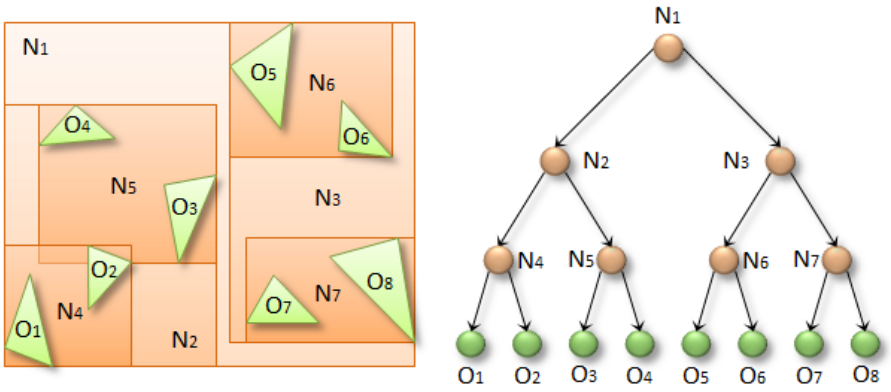

Figure 2.8: The structure of a Bounding Volume Hierarchy. (Source: [Kar12b])

2.4 Alternative Approaches

2.4.1 Neighbor Search

As already mentioned in section 1.2, using an uniform grid for spatial subdivision is efficient, but has some issues like differently sized particles or largely empty areas in the simulation domain. [Kar12b]

There are different approaches to address either polydisperse particles or the teapot-in-a-stadium issue. For example a hierarchical grid could be used, that means grouping particles into sets with different grid sizes, discussed by Ogarko et al. ([OL12]) recently. But, this approach does not address large empty areas in the simulation domain at all. Additionally, at particle samples with a continuous distribution, it is difficult to select the number of hierarchies and the optimal size ranges covered by each hierarchy.

On the other hand, to address the sparse cell occupation, it could be beneficial to have an alternative to the huge cell-start vector, storing the start-index for each cell in the simulation area, occupied or not. One possibility would be hash tables which are filled with the occupied cells as keys and the cell's particles as value. There exist algorithms for creating fast and memory efficient hash tables on the GPU (see [ASA+09]), and research is ongoing regarding this matter.

Addressing the two issues of interest at once, different authors have discussed fast parallel BVH construction in the past. The BVH is a hierarchy where bounding volumes contain other bounding volumes or elements, organized as a tree, see figure 2.8. Each node in the tree has its Axis Aligned Bounding Box (AABB) or some other representative for the bounding volume - other possibilities would be bounding spheres or oriented bounding boxes. Such trees are often used as accelerating structures for ray-tracing or broad-phase collision detection. If the tree contains all the elements to collide with, collision detection is equivalent to traversing the tree, starting at the root node on the very top level. Via

each node's bounding box it can be decided if a collision with children nodes is possible or not.

Depending on the application it is often difficult to find the right quality (i.e. traversal performance) vs. construction speed tradeoff. For particle simulation in every time step the tree has to be re-built or updated as the positions change constantly, so certainly also the tree construction time matters. A very simple approach which reduces the construction problem to a simple sorting problem is called Linear Bounding Volume Hierarchy (LBVH) which is focused on quick construction but still produces good-quality BVHs. To achieve this, a so-called space-filling curve is used for ordering the input elements. Elements that are close to each other then effectively end up close to each other in the resulting tree. Algorithms, including construction and traversal, were presented in [LGS+09] and, more recently, [Kar12a].

2.4.2 Neighbor Lists

One important implementation detail to consider is warp divergence. As already discussed in chapter 2.2.2 all the 32 threads of a warp execute the same instruction at a time, with different execution paths being serialized. Taking a look at the collision kernel (chapter 2.3.2) this happens at the following places:

1. Looping over a different number of neighbor cells for threads handling particles near simulation domain boundary.
2. Looping over a different number of particles in a cell.
3. If distance of particles small enough or bounding boxes overlap: do fine collision detection and/or calculate force and torque.

The first two points are inherent for this kind of simulation as each particle has its individual number of neighbors, the same applies for traversing trees in a similar way. But, doing fine collision detection right after the coarse check leads to a lot of inactive threads for some time. If complex shapes are involved there is a lot more work to do to even decide if collision occurs or not. Global memory loads additional to positions are involved as well, e.g. velocity, angular orientation, and angular velocity of the neighbor particle, which are usually needed to calculate the resulting force.

What is often done to avoid this is known as two-phase collision detection. That means, after the coarse collision check, the potential collision partner is added to a list, also known as broad-phase collision detection. In a second step the list is processed, which is often called narrow-phase collision detection. For example this approach was

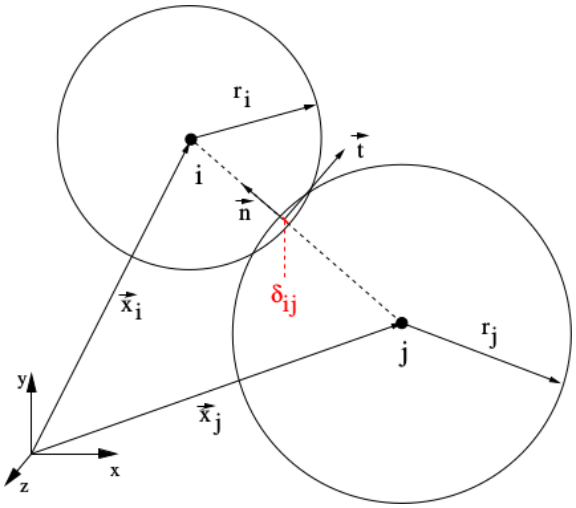

Figure 2.9: Overlapping particles. $\vec{n}$ is the normal direction, $\vec{t}$ lies in the tangential plane of the contact. (Source: [Rad06, page 12])

implemented by Govender et al. ([GWKE14]) for storing neighbors in their state-of-the-art DEM-implementation featuring polyhedral shapes, using an array of neighbors with fixed size.

There are different storage possibilities for neighbors. A static list could store a fixed number of neighbors per particle. Also being discussed could be a dynamic list of variable length, which may be resized if needed. In this case writing to the array has to be atomic, as the neighbor search algorithm is running massively parallel. A similar approach was discussed in [Wan10], where also a hybrid version combining static and dynamic list was suggested.

Such a contact list may also be extremely useful for particle-triangle contacts. Imagining particles in a geometry, only a fraction of them may be in contact with the geometry at all, leading to high execution divergence.

A great advantage of having contact lists available is that they do not have to be rebuilt every time step, but can be kept from one to another, enabling to track the contact and to know something about its history. If it is possible to modify the contact list directly, append new contacts, and delete vanishing contacts, additional values can be stored for each contact which can be accessed or modified whenever needed.

2.4.3 Force History

Figure 2.9 shows two particles during a contact, and in figure 2.10 the DEM contact model is illustrated. When a spring/damper model is used in tangential direction as well, tracking the contact over time becomes necessary to get an idea about the tangential spring displacement. It can be seen that the normal spring displacement δ_{ij} is known in every time

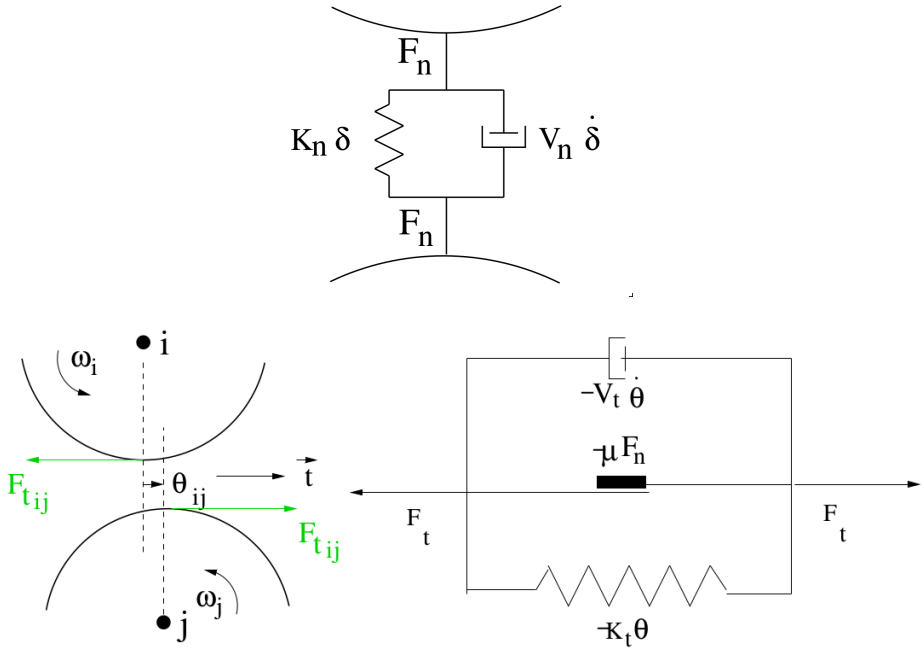

Figure 2.10: DEM contact model in normal (top) and tangential (bottom) direction. (Source: [Rad06, page 13])

step, being equal to the overlap distance, what is not the case for the tangential direction, where the contact history is needed. The tangential spring displacement analytically is defined as the time-integral over the tangential velocity v_t

$$\theta_{ij}(t) = \int_{t_0}^{t} v_t(\tau) \cdot \mathrm{d}\tau \tag{2.4}$$

where the contact first occured at $t = t_0$. Initialized with 0 for each new contact, it is then numerically approximated by adding $v_t \cdot t_{\text{step}}$ to the current displacement value in each force calculation step. For physical correctness additional steps are necessary, including a projection of the already existing tangential displacement onto the current tangential plane and limiting the force according to the so-called Coulomb condition. Various approaches are reviewed by Kruggel-Emden et al. ([KEWS08]), also discussing methods of how the tangential spring could be constrained for physical correctness. Still, managing to get a full history working within a GPU implementation is challenging, as the number of contacts is not known in advance. Having a working neighbor list is the first step, enabling force history then is just a matter of having an additional storage element for each contact, and to find a method how to in-place modify the neighbor list instead of creating everything from scratch.

Chapter 3

Design and Implementation of Neighbor Search

As discussed in 2.4.1, the LBVH is a good implementation candidate as a more flexible acceleration structure compared to the original uniform grid approach. This chapter is dedicated to the needed algorithms, and additionally introduces possible approaches for further optimization of the uniform grid neighbor search.

3.1 Linear BVH

This section is supposed to show possible algorithm designs for binary radix tree construction, creation of the BVH by assigning bounding boxes, as well as how to traverse it in the broad-phase neighbor search step to collect potential collision partners. The most recent publication dealing with these subjects under the aim of maximizing parallelism is [Kar12a], showing that their implementation outperforms previously published methods. In contrast to processing each level separately, leading to limited parallelism in the higher levels of the tree, as these layers may consist of a few nodes only, their algorithm parallelizes over the entire tree.

3.1.1 Morton Code

The main idea of the LBVH is to order the input elements along a space-filling curve. During generation of the node hierarchy, internal nodes are assigned subtrees, each of them equivalent to a linear range of sorted input elements. Used for ordering elements is the Z-order, which is often referred to as Morton Code. In figure 3.1 it is illustrated how the one-dimensional value is constructed bit-wise from the three-dimensional coordinates by interleaving bit patterns.

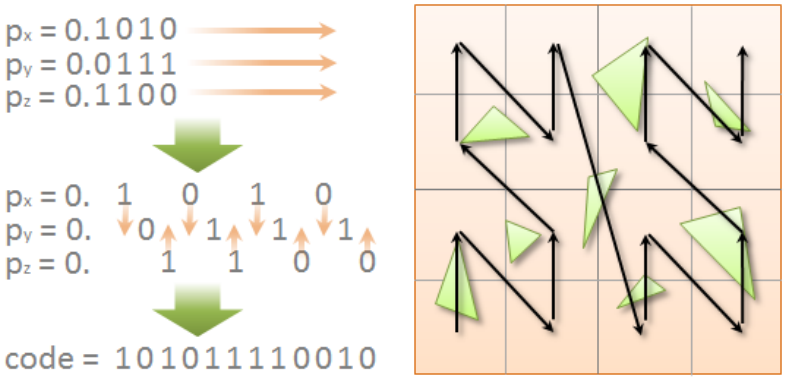

Figure 3.1: Morton Code (Z-order). (Source: [Kar12b])

This automatically results in high locality, i.e. elements close to each other in space also being close to each other in the tree. Coordinates used for calculation of Morton Codes are normalized values of the AABB's centroid. How to normalize depends on the Morton Code's storage type. In practice integer types of 32 or 64 bit width can be used, resulting in a Morton Code of $3 \cdot 10 = 30$ bit or $3 \cdot 31 = 63$ bit width, cutting the simulation domain into 2^{10} or 2^{31} parts along each axis, respectively. In the case of a 30-bit Morton Code the normalization is a mapping of each coordinate to an integer between 0 and $2^{10} - 1$, which can be accomplished as follows:

$$\text{normalizedPos}.xyz = \min\left(\left\lfloor \frac{\text{pos}.xyz - \text{worldOrigin}.xyz}{\text{worldSize}.xyz} \cdot 2^{10} \right\rfloor, 2^{10} - 1\right) \tag{3.1}$$

Afterwards, the Morton Code is constructed out of the normalized position in each dimension, by expanding the bits (i.e. insert 2 zeroes between each two consecutive bits) and interleaving the resulting bit patterns. Expanding the normalized positions can be done with magic number operations, and interleaving can be reduced to bit-shifting and addition. For 30-bit Morton Codes the lines of code for expanding the bit patterns are:

```
unsigned int expandBits(unsigned int v) {
  v = (v * 0x00010001u) & 0xFF0000FFu;
  v = (v * 0x00000101u) & 0x0F00F00Fu;
  v = (v * 0x00000011u) & 0xC30C30C3u;
  v = (v * 0x00000005u) & 0x49249249u;
  return v;
}
```

The final result for the Morton Code $m[i]$ of element i then is computed out of the normalized positions $p[i]$ as follows:

$$m[i] = 4 \cdot \text{expandBits}(p_x[i]) + 2 \cdot \text{expandBits}(p_y[i]) + \text{expandBits}(p_z[i]) \tag{3.2}$$

3.1.2 Common Prefix

The common prefix between two Morton Codes is defined as the matching leading bits and is interesting for creation of the node hierarchy, as splits occur at changes in the common prefix. Therefore the common prefix length is the value of interest. The following equation is used to calculate the common prefix length between two nodes i and j:

$$\delta(i,j) = \begin{cases} -1, & \text{for } j \notin [0, n-1] \\ \_\_\text{clz}(m[i] \oplus m[j]), & \text{for } m[i] \neq m[j] \\ 32 + \_\_\text{clz}(i \oplus j), & \text{for } m[i] = m[j] \end{cases} \tag{3.3}$$

Here, $m[i]$ is the Morton Code of the i-th element, *__clz()* (available as CUDA® intrinsic) counts leading zeroes, and $\oplus$ stands for the bit-wise XOR operation. This definition handles non-unique Morton Codes as well, using the index as a fallback value when two Morton Codes are equal, as suggested in [Kar12a]. In this case the leading 32 bits are equal (as 32-bit integers are used), therefore 32 is added to get an even higher value for those pairs, as they have more in common.

3.1.3 Tree Properties

For simplicity Morton Codes of the elements to be inserted into the tree are called keys in the following sections. Figure 3.2 illustrates an example binary radix tree, which is basically a hierarchical representation of the keys' common prefixes, omitting nodes with only one child. When n is the number of keys, the resulting tree consists of $n-1$ internal nodes. As the tree is ordered, an internal node's range $[i, j]$ implies that common prefix lengths within this range are greater than or equal the common prefix of keys i and j, i.e. $\delta(k_1, k_2) \geq \delta(i, j) \,\forall\, k_1, k_2 \in [i, j]$. The first differing bit, following the common prefix length $\delta(i, j)$ of the internal node's range of keys, determines where to split into sub-trees. The last key having a zero bit at this position is denoted as split position γ. The common prefix length at the split location $\delta(\gamma, \gamma + 1)$ then equals $\delta(i, j)$, with the sub-ranges $[i, \gamma]$ and $[\gamma + 1, j]$ each having a strictly greater common prefix length than $\delta(i, j)$. This is because each sub-range has at least one additional common bit, 0 in the left and 1 in the right sub-range, respectively.

3.1.4 Tree Construction

From the previous section a sequential algorithm directly follows by starting at the whole range, splitting at the first differing bit, and continue in a recursive fashion with the two child nodes. As shown in [Kar12a], a special numbering of internal nodes (see figure 3.2, bottom) allows to process each node independently. If the internal and leaf nodes are

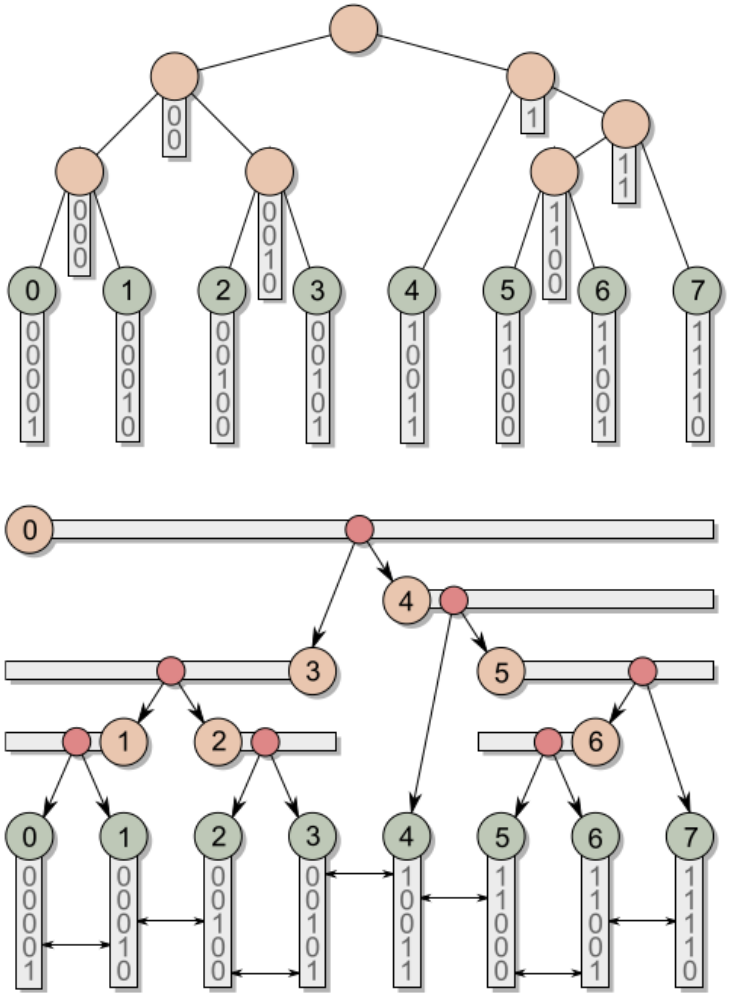

Figure 3.2: Hierarchy generation: How to determine split positions. Top: Binary radix tree. Bottom: Special Ordering of internal nodes for independent construction. Leaf nodes are green. (Source: [Kar12a])

stored in separate arrays, I and L, with the first internal node located at I_0, covering the whole range $[0, n-1]$ of keys, the children's indices can be assigned according to the split position as follows:

- The children's indices of a split at the position γ become γ and $\gamma + 1$
- If the range covers only one key, it has to be a leaf (L_γ or $L_{\gamma+1}$), and an internal node otherwise (I_γ or $I_{\gamma+1}$)

This layout has the property that each internal node's index is equal either to its first or last key due to this particular construction method: The root node is located at the beginning of its range, each left child is located at the end of range $[i, \gamma]$, and each right child is located at the beginning of range $[\gamma + 1, j]$. It should be kept in mind here that from a bottom-down view when a node is splitted into sub-ranges, the split position alone, in addition to its already known own range, determines the sub-ranges. But, as each internal node should be processed independently and fully parallel, the relationships are incomplete yet, and therefore the range of parent nodes can not simply be looked up. Hence, a separate step to determine the range is necessary. From this point of view the algorithm consists of following steps for node i:

1. Determine direction of range, i.e. is key i the last or first key of its range
2. Determine other end of range
3. Determine split position
4. Assign child/parent relationship

This construction algorithm does more work, compared to the recursive approach, which only needs one binary search per node. It would not make sense at all if used for a serial implementation, as the dependency between nodes is not a problem at all when processing the levels serially.

Determine Direction and Minimum Common Prefix

The direction must be chosen so that the own range has a greater common prefix length than the parent node, as follows from previous considerations. For this the neighbor keys' common prefix lengths have to be examined, $\delta(i, i-1)$ and $\delta(i, i+1)$. In the 'wrong' direction there is actually the splitting partner located, from one level above, as follows by construction rules. Subsequently, the common prefix is as long as for the parent node range. As the own node's range must have a greater common prefix length, the direction, as well as the minimum common prefix length given by the parent's split (i.e. the 'wrong' direction), can be determined by following lines of code:

```
int d = sign(delta(i, i + 1) - delta(i, i - 1));
int deltamin = delta(i, i - d);
```

By definition (see equation 3.3), $\delta(0, -1) = -1$, and the direction of the root node becomes $+1$, which is required by the construction algorithm.

Determine Range of Node

The idea is to find the largest l that satisfies $\delta(i, i + l \cdot d) > \delta_{\text{min}}$, based on the property that keys covered by a node are also part of the parent node, and therefore the lower bound common prefix length is δ_{min}, as determined in the previous step, which equals the common prefix length of the parent. Two steps are done to find this range of keys. In the direction to look, firstly an upper bound l_{max} is resolved so that $\delta(i, i + l_{\text{max}} \cdot d) > \delta_{\text{min}}$, using power-of-two exponential increase:

```
int lmax = 2;

while (delta(i, i + d * lmax) > deltamin) {
  lmax *= 2;
}
```

Afterwards a binary search is done in the range $l \in [0, l_{\text{max}} - 1]$, to find the largest l that satisfies $\delta(i, i + l \cdot d) > \delta_{\text{min}}$:

```
int l = 0;
int t = lmax;

do {
  t /= 2;
  if ( delta(i, i + d * (l + t)) > deltamin)
    l += t;
}
while (t > 1);

int j = i + l * d;
```

Here, l_{max} is a power-of-two integer, and starting from the highest bit the corresponding values are added to the range length, as long as the condition for a greater common prefix length is satisfied. Finally, the other end of the range $j = i + l \cdot d$ can be fixed, and $\delta(i, j)$ can be calculated.

Determine Split Position

Looking for the split position is basically the same binary search as previous when the range was scanned - the largest s that satisfies $\delta(i, i + s \cdot d) > \delta(i, j)$ gives the last key to

be in the range starting at i. Now, l is not a power-of-two integer, therefore rounding up has to be ensured in each halving step:

```
int commonPrefix = delta(i, j);
int s = 0;

do {
  l = (l + 1) >> 1;
  if ( delta(i, i + d * (s + l)) > commonPrefix)
    s += l;
}
while (l > 1);
split = i + s * d + min(d, 0);
```

When $d = +1$, the range starting with key i corresponds to the left child node with range $[i, \gamma]$, and the according split position equals $\gamma = i + s$. For $d = -1$ when the range goes left (i.e. to lower indices), the situation is different as those keys are covered by the right child node *ending* at key i, with a range starting at $\gamma + 1$, so $\gamma + 1$ equals $i - s$ in this case. The general equation is then given by $\gamma = i + s \cdot d + \min(d, 0)$, being valid for both directions.

Assign Node Relationships

Finally the sub-ranges are known to be $[\min(i, j), \gamma]$ and $[\gamma + 1, \max(i, j)]$. As previously discussed, if a range consists of only one key, a leaf node is placed. This is the case if for the left sub-range $\min(i, j) = \gamma$, and for the right sub-range $\max(i, j) = \gamma + 1$, respectively. Therefore, the code used to assign node relationships looks as follows:

```
node.childAIdx = split;
node.childBIdx = split + 1;

if (split == min(i, j)) {
  leafNodes[split].parentID = index;
  node.childAType = TYPE_LEAF;
}
else {
  internalNodes[split].parentID = index;
  node.childAType = TYPE_INTERNAL;
}
if (split + 1 == max(i, j)) {
  leafNodes[split+1].parentID = index;
  node.childBType = TYPE_LEAF;
}
else {
  internalNodes[split+1].parentID = index;
  node.childBType = TYPE_INTERNAL;
}
```

3.1.5 Assigning Bounding Boxes

After tree creation there is still this single step left to make it useful for collision detection. As suggested by [Kar12a], one thread is created per leaf node, walking up towards the root node. It is necessary that both child nodes already have been processed (i.e. an accurate bounding box has been assigned) to go on with the parent. This can be accomplished by having a counter for each internal node, which is initialized with 0 and incremented atomically by each child. The first child just exits, whereas the second child calculates the node's bounding box, which then is safe, followed by continuing the tree-walk. This is achieved by the following lines of code:

```
1  // start with a leaf node's parent
2  uint parent_id = leafNodes.parentNodeID[index];
3  // while root node not reached
4  while (parent_id != INVALID_HASH) {
5    // if we are the first one to call CalcAABB(.), we're finished
6    if (internalNodes.CalcAABB(parent_id))
7      return;
8    // otherwise continue walk-up
9    parent_id = internalNodes.nodes[parent_id].parentNodeID;
10 }
```

where *CalcAABB(.)* roughly does following:

```
1  int old = atomicAdd(&aabb[internalNodeIdx].aabb_processed, 1);
2  if (old == 0)
3    return true;
4  // otherwise calculate and assign bounding box
```

Calculating an axis-aligned bounding box from the child node's boxes is rather simple. It is defined by having a *start* and an *end* point, where the *start* coordinates always have lower values in each axis. That means a per-coordinate minimum of the children's *start* points give the parent's *start* point, followed by calculating the new *end* point by a per-coordinate maximum of its child node's counterparts, giving a new bounding box which contains both child node's bounding boxes.

3.1.6 Tree Traversal

A simple recursive implementation of independent traversal (i.e. one thread launched for each query object) would be to start at the root node and recursively process child nodes if the bounding boxes overlap. This can be done as follows:

```
void traverseRecursive(int particleIndex, AABB& queryAABB,
                       int nodeidx, bool isLeaf) {
  if (!isLeaf && checkOverlap(queryAABB, internalNodes.GetAABB(nodeidx))) {
    // if bounding box overlaps, recurse over children
    InternalNode currentnode = *internalNodes.GetInternalNode(nodeidx);
    traverseRecursive(particleIndex, queryAABB, currentnode.childAIdx,
                      currentnode.childAType == TYPE_LEAF);
    traverseRecursive(particleIndex, queryAABB, currentnode.childBIdx,
                      currentnode.childBType == TYPE_LEAF);
  }
  // for a leaf, do necessary collide steps and return
  if(isLeaf)
    collide(particleIndex, leafNodes.GetObjectIdx(nodeIdx));
}
```

This approach works, but has the problem of high execution divergence. Each thread makes its own decision of recursing over its children, or not, i.e. skipping the node. Nearby threads can easily go out of sync if they process a different path - and two distinct nodes will never be processed parallel in one single warp. A possible solution is to process the recursion stack manually, by defining an explicit stack which stores yet unhandled nodes:

```
uint stack[64];
uint* stackPtr = stack;
*stackPtr++ = INVALID_HASH; // push end condition
uint nodeidx = 0; // start with root node

do {
  bool traverseR = false, traverseL = false;
  uint internL = 0, internR = 0;

  const InternalNode currentnode = *internalNodes.GetInternalNode(nodeidx);

  // ... do overlap checks ...

  if (!traverseL && !traverseR)
    nodeidx = *--stackPtr; // pop
  else {
    nodeidx = traverseL ? internL : internR;
    if (traverseL && traverseR)
      *stackPtr++ = internR; // push
  }
} while (nodeidx != INVALID_HASH);
```

If there is no need to lookup one of the child nodes, the next node index is popped from the stack, with the iterative algorithm finishing when there is no node left. If there is just one candidate for continuing traversal this one is selected as next node. In case this applies to both child nodes, one of them has to be pushed onto the stack for processing in one of

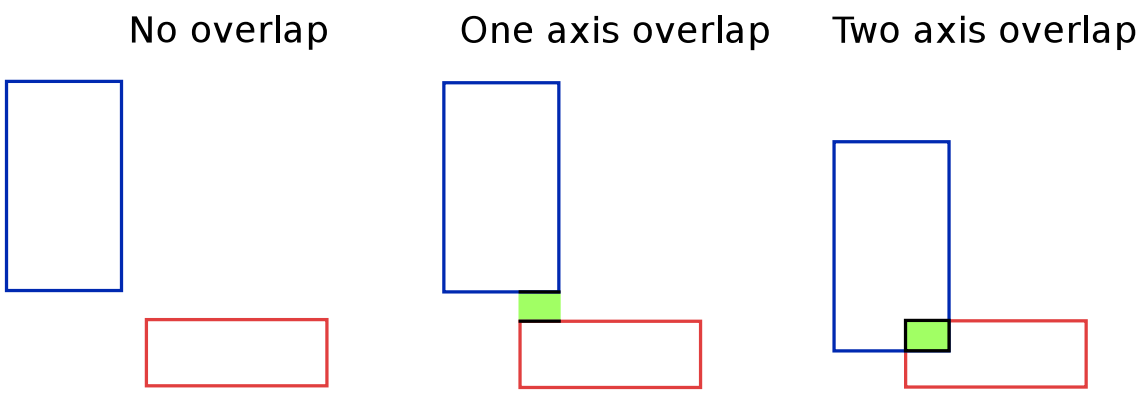

Figure 3.3: Overlap of Axis-Aligned Bounding Boxes, illustrated in 2D. For a real overlap all axes overlap. The same applies in three dimensions - overlaps in all three axis are necessary.

the subsequent iterations. This implementation benefits from much better convergence, as every node is processed by the same part of the code, regardless of where the nodes are located in the tree - each iteration is in sync between threads in a warp, unless some have finished already. Overlap and collision checks are done for each child node separately:

```
1 if (currentnode.childAType == TYPE_LEAF) {
2  collide(idx1, leafNodes.GetObjectIdx(currentnode.childAIdx));
3 }
4 else {
5  internL = currentnode.childAIdx;
6  traverseL = checkOverlap(queryAABB, internalNodes.GetAABB(currentnode.childAIdx));
7 }
8 // the same for child node B
```

The overlap check itself is as simple as constructing the AABBs, with an overlap occuring exactly when there is an intersection in all of the three dimensions, as is illustrated in figure 3.3. The overlap in one coordinate is checked by looking at start- and end-coordinates - if one range starts after the end of the other range, there is no intersection. Another important matter is the order in which elements are processed - when nearby threads process spatially close objects, also data convergence increases because nearby objects will take similar paths for traversal, as they potentially collide with the same leafs. This is applicable to the uniform grid algorithm as well because objects close to each other access partly the same cells, and therefore also the particles contained by these cells.

3.1.7 Exploit Symmetry

If the BVH is used for collision detection of a set of objects against themselves (e.g. for particle-particle collision), each pair of objects is reported twice in the traverse step. In case a symmetric behavior is wanted, e.g. a collision pair being inserted into the neighbor list only once, there is a simple solution by only take leafs into consideration with an index greater than the query object. Then, it is possible to lower the costs of traversing the tree by storing the rightmost index a internal node covers (i.e. the maximum index which can be reached through this node), and not traversing nodes at all if it is impossible to reach leafs with an index greater than the index of the query object:

```
if (internalNodes.rightmostIndices[internR] > query_particle_idx &&
    checkOverlap(queryAABB, internalNodes.GetAABB(currentnode.childBIdx))) {
  traverseR = true;
}
```

Setting up the array of rightmost indices can be done simultaneuously with assigning the bounding boxes, for each internal node it is equal to the maximum of its children's rightmost indices. [Kar12a]

3.1.8 BVH Data Structures

Now, as algorithms for tree creation and traversal were already discussed, some CUDA® specific matters are still open, regarding memory alignment. The data structures for the node type (i.e. is the child an internal node or a leaf) and for the internal nodes are defined in the following way, storing the full relationship between internal nodes (children's indices, types, and parent node index):

```
struct NodeType {
  uint16_t typeA, typeB;
};

struct InternalNode {
  uint parentNodeID;
  uint childAIdx;
  uint childBIdx;
  NodeType childNodeTypes;
};
```

InternalNode elements will be stored in an array. According to this definition, one internal node is a struct containing 16 bytes, following the rules of memory alignment perfectly. Still missing is the data structure for a AABB. It contains start/end point of the box, as well as the counter needed for assigning the bounding boxes:

```
struct __align__(16) AABB {
  float3 start;
  float3 end;
  int aabb_processed;
};
```

There should be an array of AABBs, but the proper alignment is not given automatically. One AABB needs 6 single-precision floating point values (two 3D coordinates), as well as an integer value, consuming 28 bytes of memory. With the *__align__* specifier, padding is introduced so that each bounding box element starts at a 16-byte boundary, which is necessary to fulfill memory alignment rules. Last but not least, device structs containing all needed raw pointers for use in kernels are defined as follows:

```
struct InternalNodes {
  AABB* aabb;
  uint* rightmostIndices;
  InternalNode* nodes;
};

struct LeafNodes {
  uint* parentNodeID;
  uint* objectidx;
};

struct BVH {
  uint numLeafs;
  LeafNodes leafNodes;
  InternalNodes internalNodes;
};
```

What can be seen here is that Structure of Arrays (SoA) are used, which is in general preferred over the so-called Array of Structures (AoS), because of better memory alignment - single elements of an array should be limited in size according programming guide recommendations. [NVI15]

The memory management itself is done by Thrust, a C++ template library for CUDA. Instantiations of *thrust::device_vector<T>* are used for high-level abstraction of arrays resided in device memory. These vectors can be copied, resized, etc. as easy as standard vectors implemented in the Standard Template Library (STL).

3.2 Optimizing Neighbor Search via Uniform Grid

The same symmetry exploitation as for BVHs can be applied to the uniform grid neighbor search as well, if it is sufficient to report collision pairs only once. For this purpose, when looping over the hashes/indices arrays, only elements after the query particle are

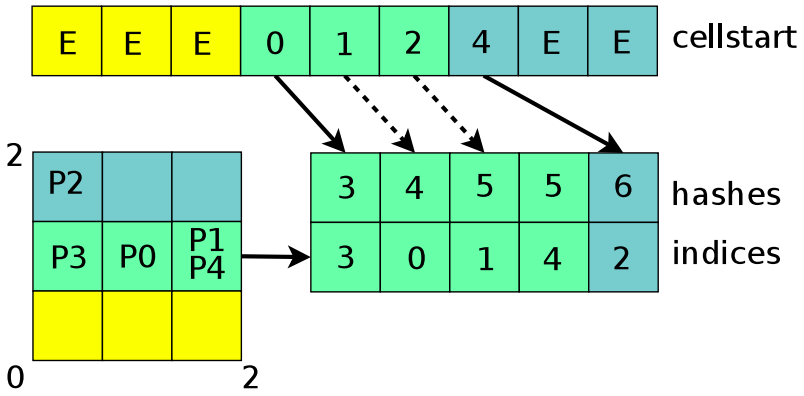

Figure 3.4: Optimized neighbor search via uniform grid. Each row can be processed in a single loop.

considered. Additionally, a recall of the equation used for computing the cell hashes $\text{hash} = \text{cellPos}.z \cdot \text{gridSize}.y \cdot \text{gridSize}.x + \text{cellPos}.y \cdot \text{gridSize}.x + \text{cellPos}.x$, together with a look at figure 3.4 gives insight for further optimizations:

1. Because elements are ordered in x-direction first, then y and z, a relative cell position of $z = -1$ leads to elements being located before the query particle. Thus, those cells have not to be considered at all.

2. The same applies for $z = 0$ and $y = -1$, the yellow 'x-row' below particle 0.

3. The same applies for $z = 0$, $y = 0$, and $x = -1$, the 'left' neighbor cell of particle 0.

4. For a 'x-row' (cells uniformly colored) it is not necessary to read 3 cell-start values in all cases, because they appear successively anyway. For example, in figure 3.4, the two cell-start values for cells 4 and 5 haven't to be fetched from memory at all. For this optimization, the range for each row has to be accordingly set:

   - Range starts at hash $H = hash(x-1, y, z)$, or $H+1$ if cell H is empty, or $H+2$ if cell $H+1$ is empty, too.
   - Range ends at hash $H + 2$.

In the best case, only 4 cell-start values are read instead of 27:

- 3 'x-row's at relative positions $z = 1$.
- 1 'x-row' at relative position $z = 0$ and $y = 1$.
- The 'own' row contains the particle of interest itself, so loop starts at index+1.

Additionally, there is much better execution divergence, because without optimization all threads in a warp are processing only one cell simultaneously, but in the optimized version all the particles of 3 cells are processed in one loop.

3.3 Memory Consumption Comparison

When the uniform grid algorithm is used, memory consumption depends on the number of particles and the number of grid cells, too. In tables 3.1 and 3.2 the requirements for uniform grid and BVH structures are given. For comparison, the typical memory consumption of dynamic particle data is estimated in table 3.3. As the number of internal nodes is approximately equal to the number of leaf nodes, the LBVH implementation consumes about 64 bytes per leaf node. If it is used for particle neighbor search, this is a non-negligible amount, nearly two-thirds of particle data demands. But, it does not depend on a grid at all, which, in terms of memory consumption, can be an advantage for sparse particle distributions.

Name	**Type**	**Bytes**	
Hashes	uint32	4	per particle
Indices	uint32	4	per particle
Cell-Start	uint32	4	per cell
Total		8	per particle
		4	per cell

Table 3.1: Uniform grid: GPU memory consumption

Name	**Type**	**Bytes**	
Morton codes	uint32	4	per leaf node
Indices	uint32	4	per leaf node
Leaf parent node	uint32	4	per leaf node
Right-most index	uint32	4	per internal node
AABB	*AABB*	32	per internal node
Internal node relationship	*InternalNode*	16	per internal node
Total		12	per leaf node
		52	per internal node

Table 3.2: BVH: GPU memory consumption

Name	**Type**	**Bytes**	
ID	uint32	4	unique particle id
Species	uint32	4	shape type and material
Position	float4	16	current position and shape size
Velocity	float4	16	current velocity
Force	float4	16	current force acting on the particle
Quaternion	float4	16	quaternion giving angular orientation
Angular velocity	float4	16	current angular velocity
Torque	float4	16	current torque acting on the particle
Total		104	

Table 3.3: Typical GPU memory consumption per particle. A *float4* is a struct containing 4 floats.

Chapter 4

Neighbor List Algorithm Design

As already discussed in 2.4.2, the main reason to decouple neighbor search and force calculation is better execution convergence. This chapter is about the different approaches existing regarding data structures, differences in filling and creating the lists, and memory consumption. In figure 4.1 two possibilities are shown. One approach is static and has a fixed amount of neighbors for each particle. The other approach to be discussed is a fully dynamic version, storing index-pairs of potentially colliding objects.

4.1 Static Neighbor List

Within this approach, a fixed-size array stores the indices of potential collision partners. The position of an array entry determines the first involved object index, whereas the content points to the second element involved. In figure 4.1 (bottom) this version of the neighbor list is illustrated for a fixed number of 8 entries per particle. The neighbors of particle 0 start at array-index 0 (first row), the neighbor indices of particle 1 at array-index 8 (second row), and so on.

4.1.1 Memory Consumption/Management

The memory consumption of this static neighbor list is fixed as the size of the array is given in the beginning (number of particles times number of entries). Allocating device memory is done on host-side, once at simulation start. There are, however, some problems with this approach. In a typical simulation each particle has its individual number of neighbors, so some memory is wasted when using the static list. This applies, even more significant, to particle-wall collisions, as a particle could be in contact with more than one wall at a time, while most of the particles have no contact at all.

Another problem is, that there could be too few entries available. It is difficult to determine the appropriate size, and, if the size is sufficient, even more memory gets wasted,

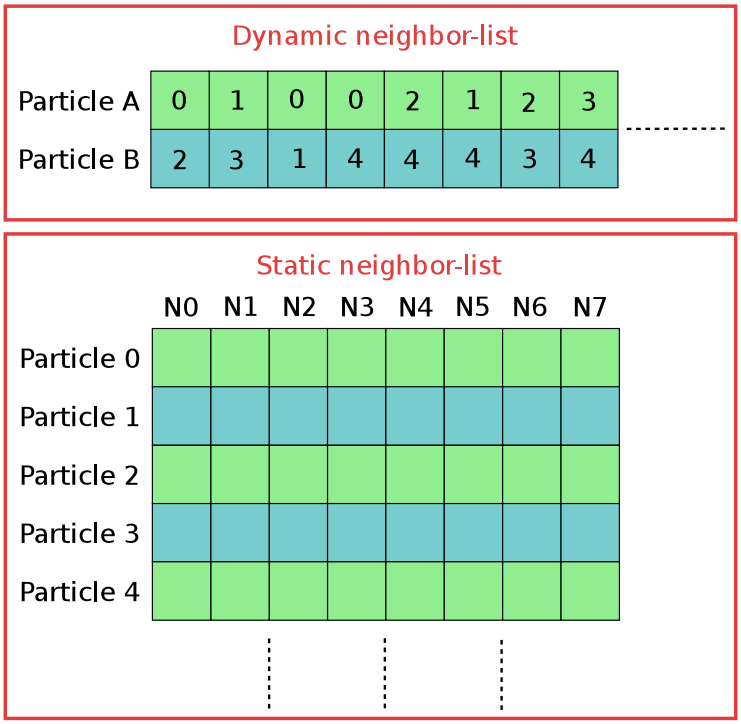

Figure 4.1: Example dynamic neighbor list (top) vs. a static version (bottom) with a fixed number of 8 entries per particle.

as only a few particles may be in touch with the maximum number of neighbors. For example, if uniformly sized spheres are being packed, a maximum number of 12 neighbors is possible (the *Kissing number*, see [CS13, p.21]). But, in general, the maximum number can not be determined in advance, especially when considering non-uniform spheres or non-spherical particles.

4.1.2 Building the List

Filling the array with entries is easy. All of the available neighbor search algorithms work on a per-particle basis. That means the loop, looking for neighbors of one particle, is done by one thread in a serial fashion (i.e. traversing a tree or looking at neighbor cells). For each particle, the offset pointing to its first neighbor can be easily determined, and a local counter can be used to count neighbors, which then points to the next free array position. If the static neighbor list size per particle is N_S, adding a neighbor is as simple as:

```
1 // neigh_nr < N_S!
2 static_list[partidx * N_S + neigh_nr] = neighboridx;
3 neigh_nr = neigh_nr + 1;
```

4.1.3 Processing the List

After the neighbor search step the list is complete and can be processed for fine collision detection and force calculation. There are various possibilities to process the list, depending on symmetry (each collision pair contained once or twice), or on parallelism granularity, i.e. if a thread-per-particle kernel is launched, to process all neighbors of one particle in serial, or if a thread-per-neighbor implementation is desired, processing one entry per thread. Details are given in section 5.1.

4.2 Dynamic Neighbor List

In figure 4.1 (top) this version of the neighbor list is illustrated. In contrast to the static version, there are two arrays, storing pairs of potential collision partners.

4.2.1 Memory Consumption/Management

Twice the memory is needed per entry ('own' and 'neighbor' index), but each entry is valid, opposed to the static list, where lots of entries could be unused. The memory management has to be done on host-side. For this purpose some initial size of the arrays has to be chosen, and resizing could be necessary if more neighbors are found than can be stored. However, when a resize is done, it may be better to have the arrays a bit oversized, to avoid permanent memory re-allocation.

4.2.2 Building the List

Building up the dynamic neighbor list is a bit more complicated, as it is common for all the particles. Entries have to be appended atomically, which can be done for example by using a global counter, which is atomically increased by one, followed by writing the collision pair information. If the current maximum length of the dynamic array (size allocated) is N_{MAX}, adding an entry looks as follows:

```
1 // atomicAdd returns the 'old' value
2 uint neigh_nr = atomicAdd(count, 1);
3 if(neigh_nr < N_MAX) {
4   pids[neigh_nr] = idxA;
5   nbids[neigh_nr] = idxB;
6 }
```

Additionally there has to be some error handling if the maximum number of entries is reached, i.e. there is no more memory space for additional entries. If this happens, the dynamic list has to be resized on host-side after the neighbor search threads have finished. Then, the kernel is launched again, now with a sufficient dynamic list length. Due to

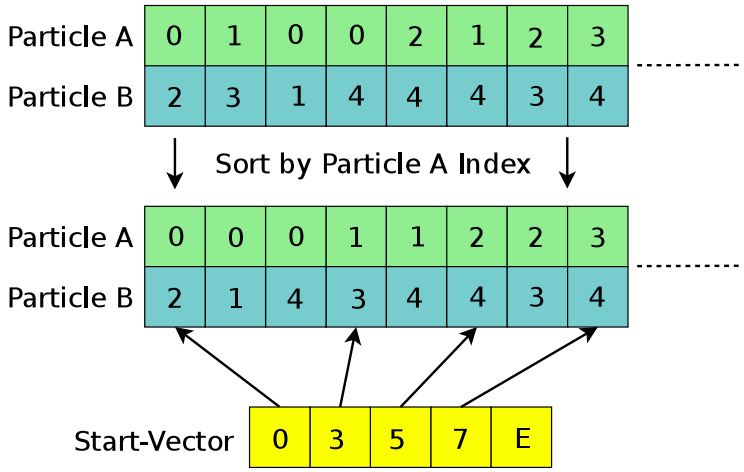

Figure 4.2: Compacted version of dynamic neighbor list.

elements being added atomically, the list is in a random state after the neighbor search. Sorting the list might be necessary, optionally followed by stream compaction, to make kernels on a per-particle basis possible. This procedure is illustrated in figure 4.2.

4.2.3 Processing the List

There are various possibilities to process the list, similar as it is the case for the static neighbor list. For example, the list could be processed on a per-particle basis if the compact variant from figure 4.2 is available. But, processing the unsorted neighbor list in a thread-per-neighbor fashion would also be possible. Details are given in section 5.2.

4.3 Hybrid Neighbor List

Figure 4.3 illustrates a possible structure of a hybrid version of the neighbor list. A great benefit here is that the static neighbor list size can be kept small, because neighbors which can not be stored in the static array will be appended to the dynamic array, allowing to keep the static array size reasonable while still having a working algorithm.

The idea behind hybrid neighbor lists to combine benefits from static and dynamic lists was discussed in [Wan10]. Besides the already given arguments there is a strong benefit using static arrays: They are in a sorted state by definition. With the arrays containing as much entries as some multiple of the particle count, sorting a pure dynamic neighbor list comes at a high cost. But, to modify the contents in-place efficiently, a sorted state is inevitable. Processing sorted lists makes it also easier to use warp reduction to lower the costs of atomically adding particle forces. Details are given in sections 5.1 and 5.2, for the static and dynamic version, respectively.

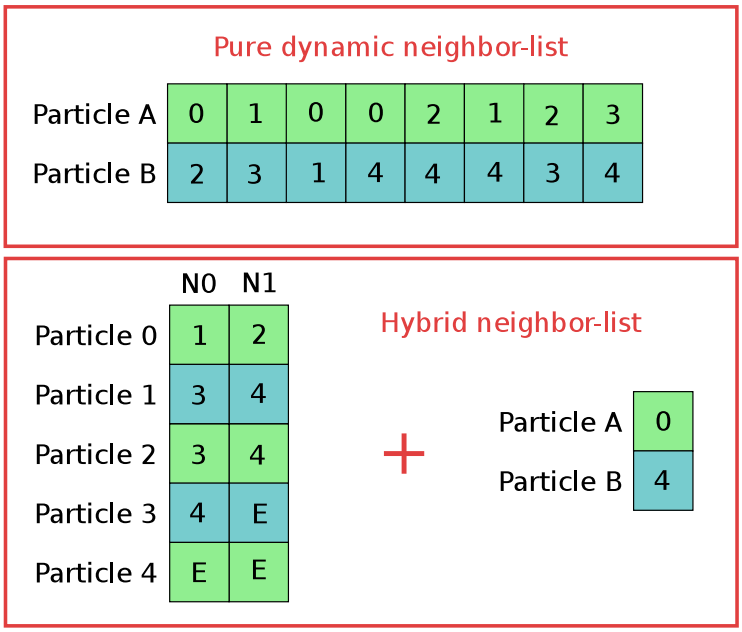

Figure 4.3: Structure of hybrid neighbor list with two static entries per particle. It contains the same information as the pure dynamic neighbor list given at the top.

4.4 Force History

As discussed in section 2.4.3, in order to make use of the force history, the contacts have to be tracked over time. For this, an additional value is stored per neighbor list entry. This can be done with the static and dynamic version in a similar way. How the in-place modification of the different neighbor list variants can be implemented, will be discussed in section 5.5.

Chapter 5

Implementation Details on Neighbor-Lists

The basic properties and differences between static and dynamic neighbor-lists were already analyzed in chapter 4. Still, various possibilities exist in order to process the lists, what will be discussed next. Additionally, a detailed description is given of how to modify neighbor-lists in-place in order to track contacts, e.g. used when force history is desired.

5.1 Static Neighbor List - Force Kernel

The two possibilities processing the static neighbor-list are illustrated in figure 5.1. The first variant is to launch one thread per particle, which then processes all the neighbor elements belonging to itself (one row in figure 5.1). The simplified lines of code look as follows, where *index* equals the particle index which is processed by the thread:

```
1  uint particleindex = index;
2  uint nbbegin = index * N_S;
3  uint nbend = index * N_S + N_S;
4
5  for(uint nbindex = nbbegin; nbindex < nbend; ++nbindex) {
6    uint neighborindex = nbids[nbindex];
7    if(neighborindex == INVALID_HASH)
8      break;
9    // ...load particle properties, calculate forces...
10 }
11
12 force [particleindex] += force;
13 torque[particleindex] += torque;
```

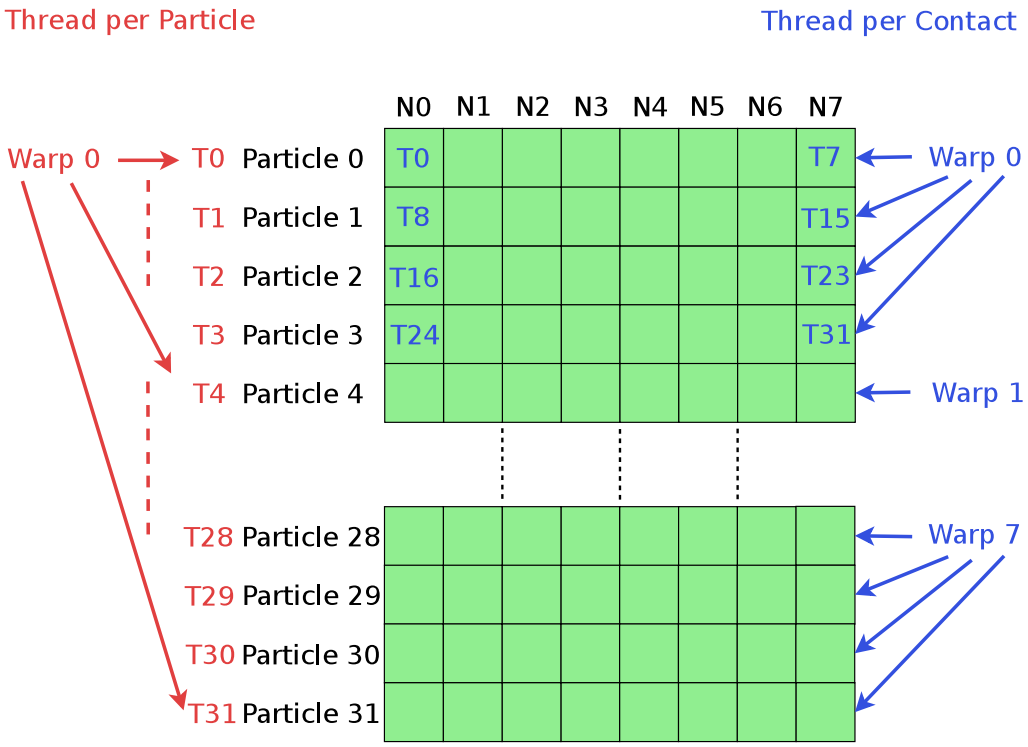

Figure 5.1: Static neighbor-list: force kernel configuration.

There is an even simpler-to-implement approach, when one thread is launched per element in the neighbor-list. This version is called thread-per-contact or thread-per-neighbor, and could be implemented in the following way, where *index* equals the static neighbor-list index which is being processed:

```
uint particleindex = index / N_S;
uint neighborindex = nbids[index];
// ...load particle properties, calculate forces...
if(!collision)
  return;

atomicAdd( &force[particleindex], force);
atomicAdd(&torque[particleindex], torque);
```

Here, the degree of parallelism is even higher. However, atomic operations have to be used to write the results, because multiple threads are simultaneously computing forces belonging to only one particle. Otherwise, conflicts would occur for sure - looking at figure 5.1, illustrating an example of 8 entries per particle: When one thread per contact is used, one warp (32 threads) will process 4 particles and its neighbors. As all these threads are executed concurrently, there are 8 conflicting operations per particle. To overcome performance limitations, reductions can be done using shared memory or warp-intrinsics,

and only one thread writes the resulting force in the end. To keep this simple enough, size restrictions for the number of entries are introduced: either 1, 2, 4, 8, 16, or 32 entries per particle are possible. In the following examples a size of 8 entries per particles is used.

To introduce the method, it is necessary to know about the so-called lane-id, which is a number between 0 and 31 and informs about the relative position within a warp. So, in the actual example, the threads with a lane-id of 0, 8, 16, or 24 are the first threads processing a particular particle's neighbors. The following helper functions are then needed:

```
__device__ inline
float4 __shfl_down(float4 var, unsigned int srcLane, int width=32) {
  float4 a = var;
  a.x = __shfl_down(a.x, srcLane, width);
  a.y = __shfl_down(a.y, srcLane, width);
  a.z = __shfl_down(a.z, srcLane, width);
  a.w = __shfl_down(a.w, srcLane, width);
  return a;
}
__device__ inline
float4 warpReduceSum(float4 val, int width) {
  for (int offset = width/2; offset > 0; offset /= 2)
    val += __shfl_down(val, offset, width);
  return val;
}
```

The *__shfl_down*-intrinsic returns values from the given source lane offset and is implemented only for the primitive types, so it is overriden to be useful for the *float4* types as well. The *warpReduceSum* function then does the tree-based reduction within a warp, which is illustrated in figure 5.2. The following lines of code then use this warp reduction method in the force kernel, where only the first thread processing this particular particle does the actual force addition:

```
uint particleindex = index / N_S;
uint neighborindex = pids[index];
// ...calculate forces...
// intra-warp reduce
force  = warpReduceSum(force,  N_S);
torque = warpReduceSum(torque, N_S);

// first thread processing this particleindex
if(lane_id() % N_S == 0) {
  force[particleindex] += force;
  torque[particleindex] += torque;
}
```

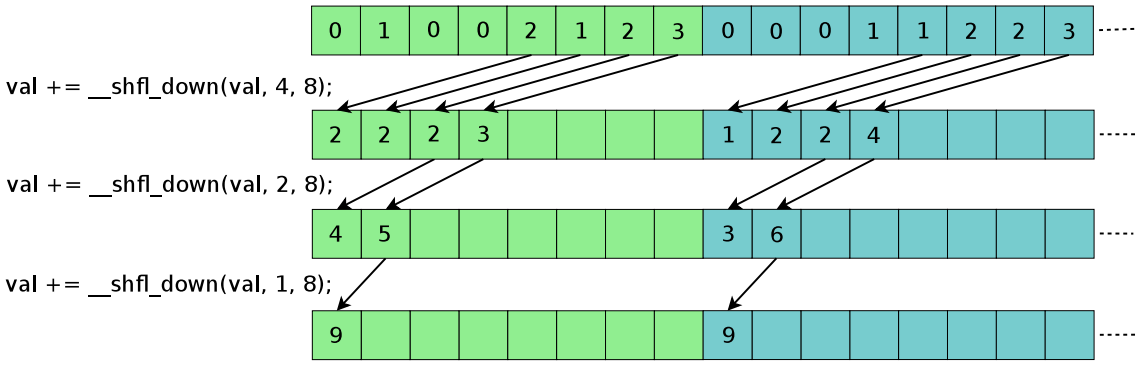

Figure 5.2: Warp reduction example: Blocks of 8 threads within a warp exchange values using shuffle-down operations. The intrinsic takes a offset lane-id (4, 2, 1) and works within a given width (8). Green threads illustrate lane-ids from 0 to 7, blue threads from 8 to 15.

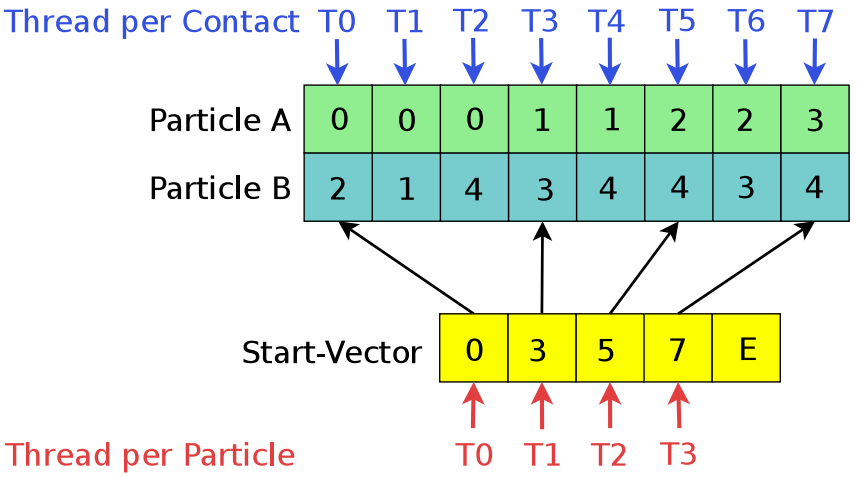

Figure 5.3: Dynamic neighbor-list: force kernel configuration possibilities.

5.2 Dynamic Neighbor List - Force Kernel

The possibilities of processing the dynamic neighbor-list are illustrated in figure 5.3 and similar to the static version, it is possible to implement a thread-per-contact or thread-per-particle kernel. The simplest, and surprisingly efficient implementation then is structured as follows:

```
uint particleindex = idA[index];
uint neighborindex = idB[index];
// ...load particle properties, calculate forces...
if(!collision)
  return;
atomicAdd( &force[particleindex], force);
atomicAdd(&torque[particleindex], torque);
```

For a thread-per-particle implementation there will be a loop over all the neighbors of one particle. But, to make this feasible, preparation steps are necessary, as was already discussed in section 4.2.3, namely sorting the list and building an array containing the entry-points for each particle. Key-value sort is done efficiently by using the Thrust library, whereas the start-vector can be built by remembering positions where the key particle index changes:

```
if(i == 0 || idA[i] != idA[i-1])
  particlestart[idA[i]] = i;
```

That means for the force kernel that a loop is done over entries, until the key particle index changes or the end of the arrays is reached:

```
uint startindex = particlestart[index];
uint particleindex = idA[startindex];
do {
  uint neighborindex = idB[startindex];
  // ...load particle properties, calculate forces...
  ++startindex;
}
while(idA[startindex] == particleindex);
// adding forces
force [particleindex] += force;
torque[particleindex] += torque;
```

For the thread-per-contact implementation a warp-reduction is also possible by using warp voting functions and bit-masks to find all the particles in a warp which have the same index. With this information a similar reduction can be done, where only the first appearing thread processing a particular particle does the summation in the end. This method is discussed in [Wes15].

5.3 Symmetry Considerations

Whenever the neighbor-list is built symmetric (i.e. each collision pair contained only once), force contributions must be added to the neighbor particle as well:

```
1 // after force calculation: add neighbor forces
2 atomicAdd(&force[nbidx], other_force);
3 atomicAdd(&torque[nbidx], other_torque);
```

In this case, or when a thread-per-contact implementation is used, each force addition has to be an atomic operation, because not all force contributions are calculated by one single thread. After some experience with different implementations, the symmetric versions are preferred. Neighbor search is faster, force calculation seems to be faster in most cases as well, especially when it is computationally expensive, and the symmetric list needs only half of the memory since every contact is stored only once instead of twice. Additionally, if sorting of the dynamic list is required, this is faster as well for only half the length.

5.4 Particle/Geometry Contacts

Because a static list has many disadvantages for contacts with geometry, a pure dynamic neighbor list is used for all kind of walls. The main reason for this is that only a few particles may be in touch with a wall at all. In the current implementation it is necessary to have a compacted list and a thread-per-particle force kernel because knowledge about whether in touch with more than one triangle or not is required.

As usually only a few particles are in contact with walls, there is a sligthly different method used for creating the start-vector, to avoid starting a thread for particles currently not in wall contact. The vector containing entry points is appended the start-indices atomically and only *different_particle_count* threads are launched for force calculation:

```
1 if(i == 0 || idA[i] != idA[i-1]) {
2   uint old_count = atomicAdd(different_particle_count, 1);
3   particlestart[old_count] = i;
4 }
```

5.5 In-Place Modification of Neighbor-List

If keeping the neighbor list from the previous time step is desired, the need arises to modify the existing lists in-place. If the dynamic neighbor-list is used, an additional valid flag is introduced, which is initially marked as invalid for each contact. When neighbors are found, it has to be checked first if the contacts already exists, because re-use of some

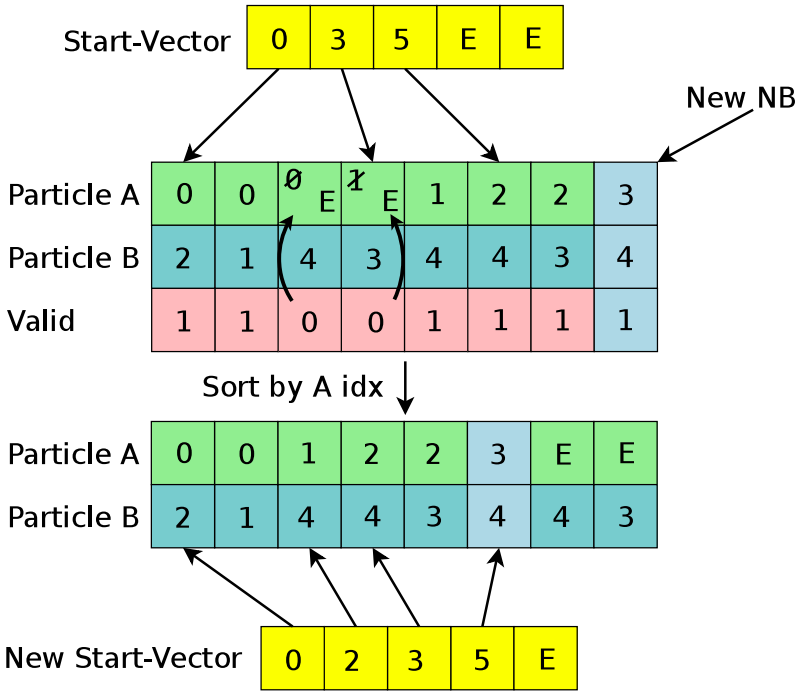

Figure 5.4: How to modify dynamic neighbor-list in-place when kept from previous step.

additionally stored values is required during the contact. The procedure is illustrated in figure 5.4. Following steps are done:

1. Before neighbor search, fill valid-flags with false.
2. If a potential collision partner is found during neighbor search, do:
   - If neighbor already contained, mark it as valid.
   - Append new contacts at the end of the list.
3. After neighbor search, do:
   - Overwrite index of invalid contacts with the empty-marker.
   - Key-value sort via particle A index.

For this to work, the dynamic neighbor-list has to be sorted and compacted to find existing contacts. The new start-vector which is created can then be used for the following force calculation kernel, before it is used in the next time step to find existing contacts.

If the hybrid neighbor-list is used, and therefore static entries are available, the best solution found was similar, with the difference that new contacts always are appended to the dynamic list, as there is no static neighbor counter available and hence, insertion into the static list is difficult. Additionally, for each contact found, this means looking up

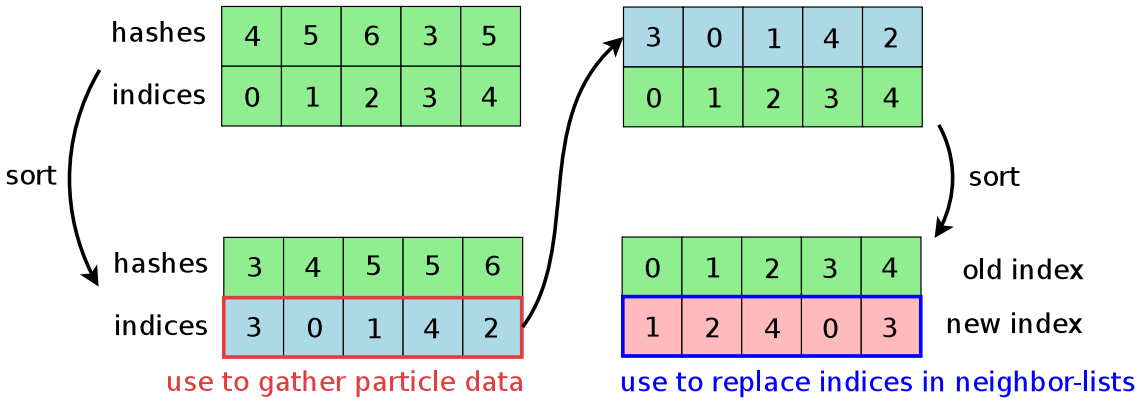

Figure 5.5: How to keep neighbor-list up-to-date when particle data is sorted. The new order is reverse-sorted to get a mapping vector from old to new particle indices.

existing contacts at two different places: the static entry point, which is fixed and known, and the dynamic entry point given by the start-vector. After the neighbor-search, static entries are compacted, and dynamic entries can be transferred to the static list if there is space left.

A pure static neighbor list is not considered at all - if it can store at least most of the contact pairs, the dynamic list will be short anyway.

5.5.1 Keeping List Up-To-Date when Particle Data is Sorted

In chapter 6 the effect of sorting particle data will be discussed. It is observed that sorting all the particle data (e.g. positions and velocities) from time to time is beneficial to data locality, with a huge performance degradation over time if particles are moving and sorting is omitted. But, as particle data is sorted, the indices effectively change. In figure 5.5 the necessary steps are illustrated to create an index-replacement array which then can be applied to all indices contained in the neighbor lists. To achieve this, the sorted indices array is inverse-sorted by using it as a key array to sort a sequence. The code to replace a index then looks as follows:

```
1 idxA[index] = new_indices[idxA[index]];
```

where *idxA* contains some particle indices, *new_indices* contains the index-mapping, with the *index*-th element of the array being processed. However, this step is only necessary for neighbor lists which are re-used.

5.6 Conclusion

Many different approaches exist for creation and processing of neighbor-lists, and lots of configurations (dynamic, hybrid list, re-using contacts) and combinations (symmetric or each contact pair contained twice) are possible, making it difficult to determine what to choose, which will also depend on the specific use-case. Various tests are done for evaluation in chapter 6.

Due to the random order of floating-point force summations the result will be a bit different in each simulation run when using symmetric lists or a thread per contact implementation. This is because floating point addition is commutative, but not necessarily associative or distributive. [PH08]

Chapter 6

Results

In this chapter test cases are introduced. They cover basic validation (i.e. physical correctness) and performance comparisons in terms of execution speed and memory consumption of the different algorithms.

6.1 Work Flow

The work flow for doing simulations consists of the following steps:

- Prepare shape information of used shapes.
- Define material properties of used materials.
- Write configuration file:
  - Give algorithmic details (e.g. BVH or uniform grid for particles/triangles, simple neighbor list, force history, time step size, simulation end time).
  - Select particle initialization details (method, domain, size distribution, material, shape).
  - Define simulation domain.
  - Optionally: Give geometry information (STL files, movement definition).
  - Optionally: Define CFD-DEM coupling information.
  - More options: Analysis (e.g. calculating cell-averaged velocities), spray nozzle definition, time interval for writing store files, etc.
- Optionally: give input file with particle and/or triangle data to do a re-start.

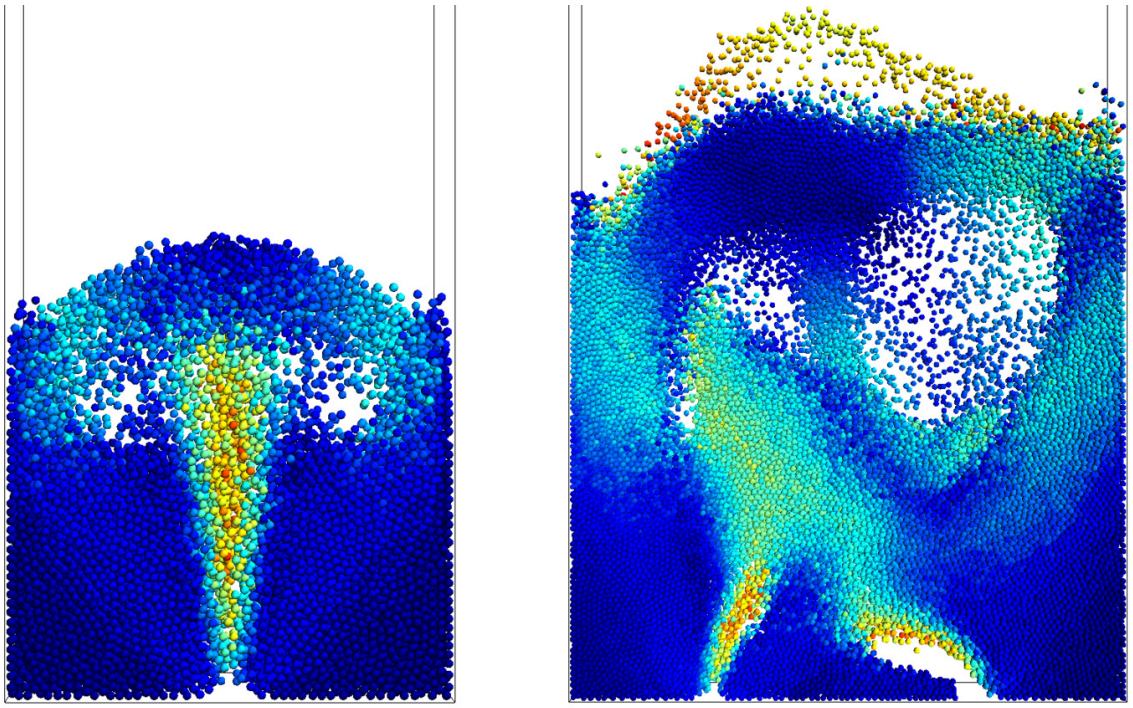

Figure 6.1: Single (left) and double (right) bed use-case: Glass particles fluidized at one and two air inlets, respectively. Used for validation regularly. Particles are colored by magnitude of velocity. (Source: RCPE)

6.2 Validation

6.2.1 Fluidized Bed Test Case

What can be simply compared with the old algorithms is using the BVH or the neighbor-list without force history. For validation of physical correctness there is one standard test case used by RCPE regularly to ensure that the basic particle-particle and particle-fluid interactions do not change. The basic test case is a single-spout bed where 12000 glass particles are fluidized. There is measurement data for validation of double- and triple-spout beds as well. For comparison the cell-based time-averaged velocity is written into VTK (Visualization ToolKit) files which then can be processed by external tools like ParaView. At specific bed heights the average particle velocities are then plotted. A description of the measurements and how simulation data is processed is given in [JSRK13]. Fluidized bed use-cases are illustrated in figure 6.1 and final results for the double bed are given in figure 6.2, using the material parameters and number of particles given in [JSRK13].

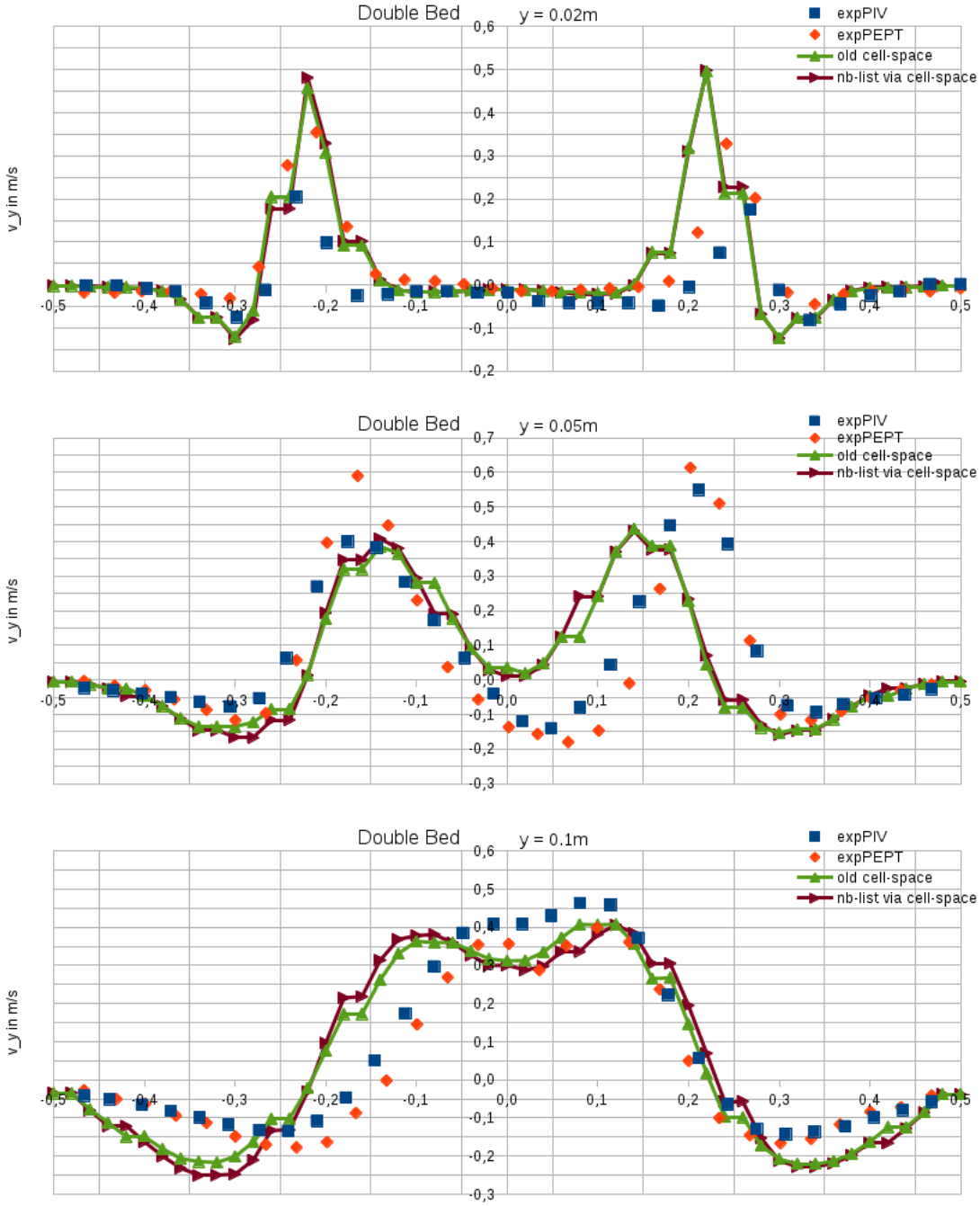

Figure 6.2: Double bed validation results. Average velocities plotted at different bed heights, using neighbor list, and comparison with old implementation and two different experiments.

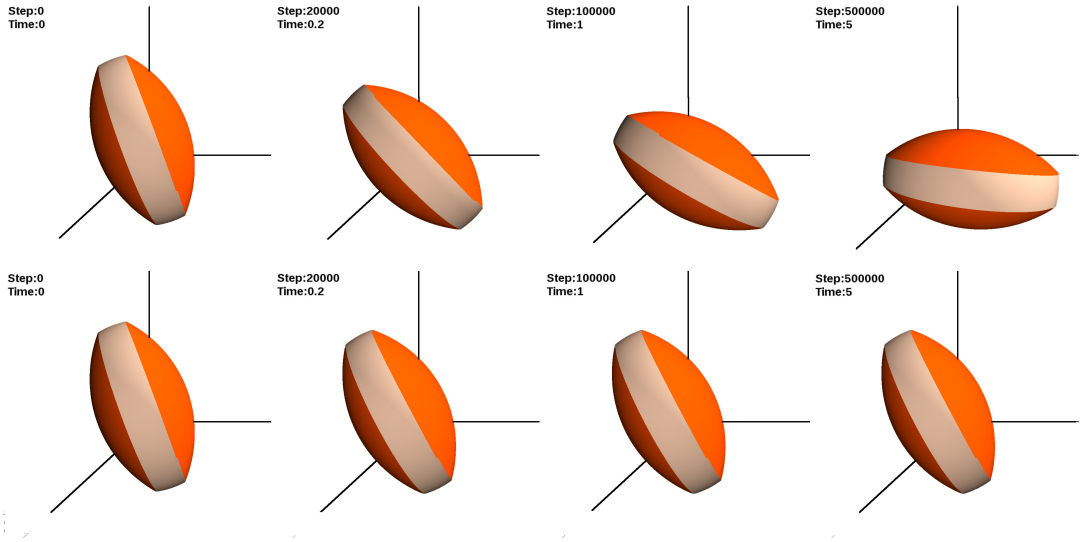

Figure 6.3: Picture series of a tablet leaning at a wall, at times 0 s, 0.2 s, 1 s and 5 s. The upper series shows that without force history the tablet slips away, whereas in the bottom simulation including force history the tablet stays in this position.

6.2.2 Force History Test Cases

The history-dependent tangential contact model is a completely new feature in XPS. Therefore the outcome of a simulation cannot be directly compared with an old implementation. For validation single-particle tests have been done to verify correctness of the tangential force evolution. A really simple test is a tablet leaning at a wall, which then stays in this position and does not slip away. Results are shown in figure 6.3. Another show case is a pyramid of 4 spheres, where 3 of them are touching the ground, and the fourth lies on them. Due to friction it is possible that these spheres stay in this position forever if there are no external influences, apart from gravity. Without having a working force history implementation this result would be impossible. Snapshots are shown in figure 6.4. For the sphere pyramid a radius of $r = 10$ millimeters was used, with initial positions as follows by tetrahedron edge points:

$$
\begin{aligned}
P0: &\quad (0 \quad 1 \quad 0) \cdot r \\
P1: &\quad (2 \quad 1 \quad 0) \cdot r \\
P2: &\quad (1 \quad 1 \quad \sqrt{3}) \cdot r \\
P3: &\quad (1 \quad 1 + \frac{2}{3}\sqrt{6} \quad \frac{\sqrt{3}}{3}) \cdot r
\end{aligned}
$$

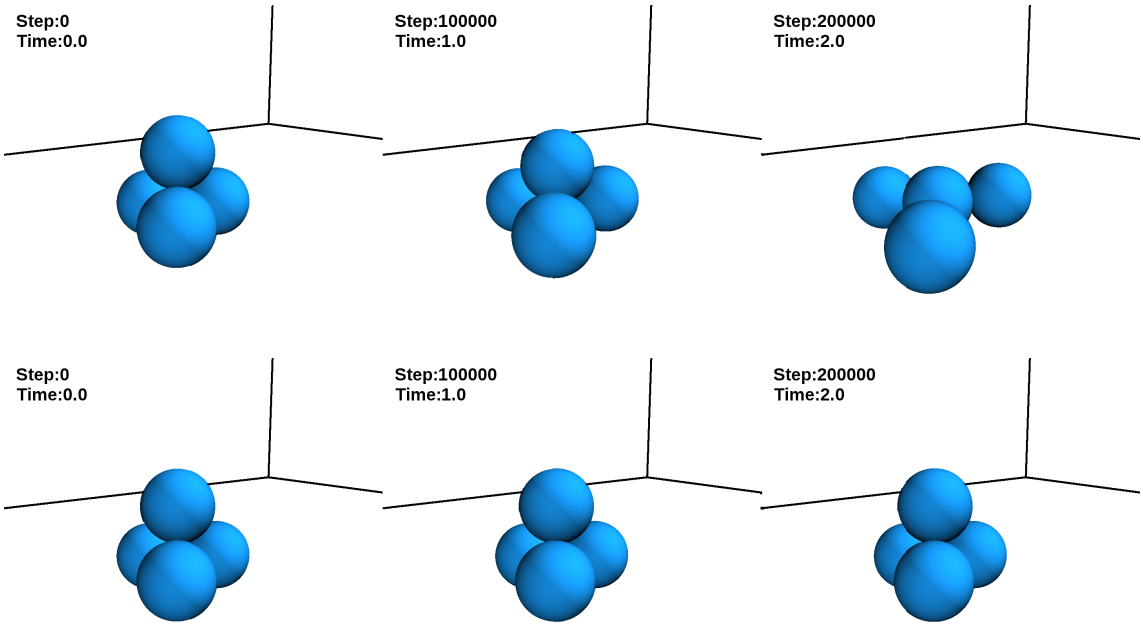

Figure 6.4: Picture series of a sphere pyramid with 4 spheres (bottom 3 are touching the ground), at times 0 s, 1 s, and 2 s. The upper series shows that without force history the spheres roll away, whereas in the bottom simulation including force history all spheres stay roughly in the same position because of effective friction.

v_t/v_n	$-v_t'/v_n'$ without FH	$-v_t'/v_n'$ with FH	estimated slope of line
0	0	0	-0.368
1	0.147	-0.368	
2	0.322	-0.723	
3	0.531	-1.066	
3.5	-	-1.180	
4	0.781	-1.125	
5	1.087	-0.127	
6	1.476	0.925	
7	2.029	1.977	
8	3.029	3.029	
9	4.081	4.081	1.052

Table 6.1: Simulation - measurement of steel particle rebound from a wall

A more sophisticated example found in literature is measurement of the reflexion angles of an initially non-rotating steel puck floating on an air film, see figure 6.5 where test set-up, experimental results and exact solution are given, including the comparison with simulation results done with a steel sphere in XPS, with and without force history. In the literature (see [Wal94]) the solutions for the rolling and sliding case (both straight lines) were given as:

$$\begin{aligned} -\left[\frac{v_t'}{v_n'}\right]_{\text{rolling}} &= -\frac{\beta_0}{e}\cdot\frac{v_t}{v_n} \\ -\left[\frac{v_t'}{v_n'}\right]_{\text{sliding}} &= -\frac{7}{2}\mu(1+\frac{1}{e})+\frac{1}{e}\cdot\frac{v_t}{v_n} \end{aligned} \tag{6.1}$$

where v_t and v_n are the tangential and normal velocity in the contact point before wall collision and v_t' and v_n' the values after the collision, respectively. Here, μ is the static friction coefficient, and β_0 and e are the coefficients of restitution in the tangential and normal direction, respectively. These restitution coefficients are directly related to the particle velocity change due to collision. In table 6.1 the simulation outcome is given as well as an estimation of the slope. Following material parameters were used:

$$\beta_0 = 0.35 \qquad e = 0.95 \qquad \mu = 0.75$$

The exact slopes from equation 6.1 for the rolling and sliding solution then equal $-\frac{\beta_0}{e} = -0.3684$ and $\frac{1}{e} = 1.0526$, respectively, and the zero point which lies on the sliding solution line occurs at $\frac{v_t}{v_n} = \frac{7}{2}\mu\cdot e\cdot(1+\frac{1}{e}) = 5.1188$, which is a good match to the simulation outcome.

6.2.3 Discussion

The results keep statistically the same when using the new algorithms. But, as already discussed in section 5.6, due to the random order of single-precision force summations the result will be a bit different in each simulation run when using the neighbor-list. Additionally, using the force history leads to stable simulation outcomes and correct results in terms of physics - and the validation cases show where the history really is needed, namely low tangential-to-normal velocity ratio. To understand this, a look at the bottom figure of 6.5 shows that results without force history are sufficient for large values of v_t/v_n. This is the case because the tangential force is usually limited by μF_N (the so-called Coulomb condition) which is reached quickly for large v_t/v_n and in this case the increments in tangential displacement are thrown away and have no effect anymore.

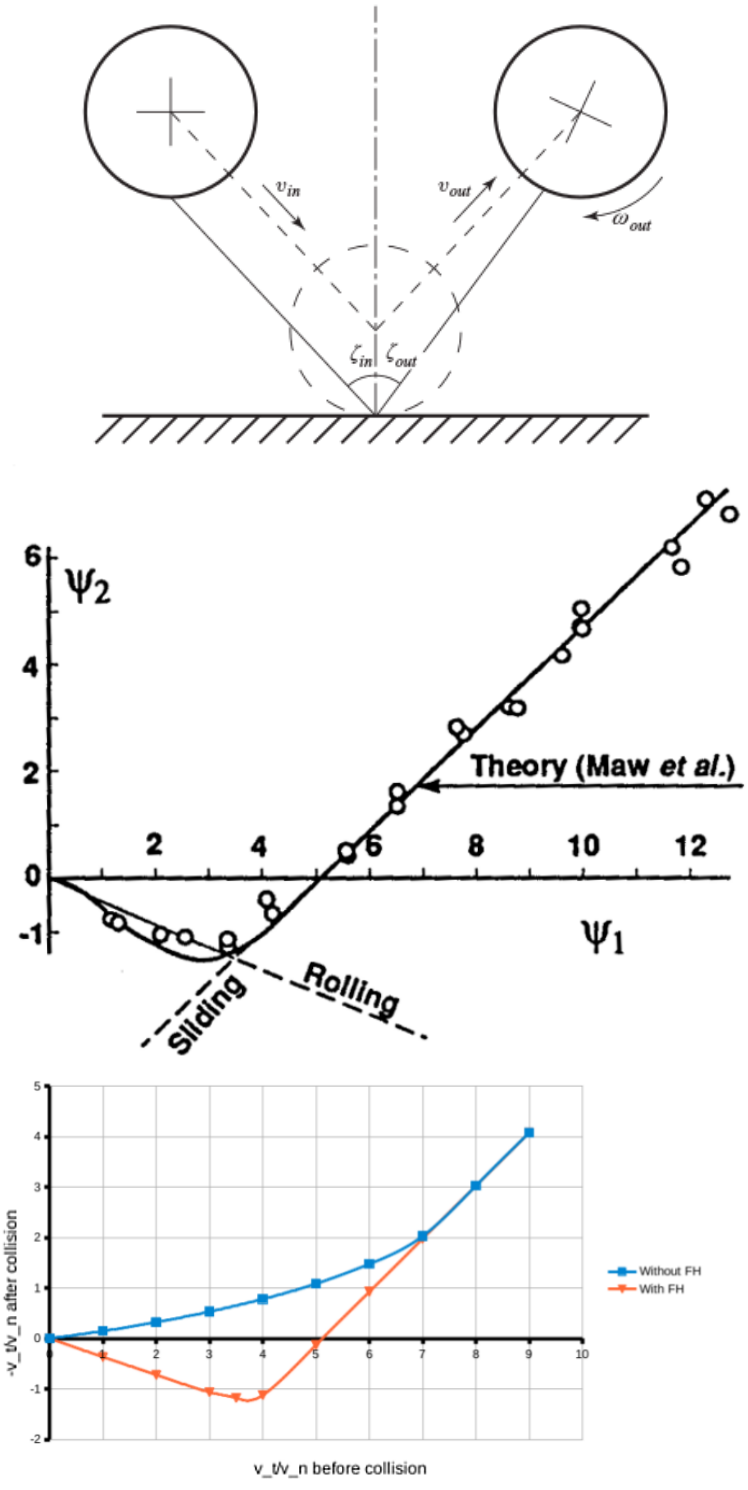

Figure 6.5: Top: Test set-up showing trajectories of the sphere's center of mass and the contact point. Middle: Non-dimensional angles of reflexion plotted against non-dimensional angles of incidence for a steel disk. Bottom: Simulation results for a steel sphere at different incident angles. (Source: Top: [JH04]. Middle: [Wal94, page 902])

Figure 6.6: Test system used for validation and performance tests

6.3 Performance Comparison

Use cases for testing performance of the algorithms are difficult to design as performance strongly depends on the problem, e.g. particle configuration (i.e. initialized on a grid or already mixed up during the process), particle shape, number of contacts, geometry, and how many triangles are used to represent the geometry. Following cases are analyzed for a sufficient large number of spheres and bi-convex tablet-shapes:

- Tuning size of the static neighbor list for particle-particle contacts.
- Particles freshly initialized in a grid, without gravity acting on particles:
  - Not mixed, no collisions at all
- Particles moving in a rotating drum coater, measure starts after one full revolution, particles loaded from store file:
  - Well mixed particles
- Difference BVH and uniform grid (for particles):
  - Mono-disperse spheres
  - Bi-disperse spheres with a big size ratio

- Slow-down when re-using neighbor list

Influence of sorting particle data is analyzed as well. In all cases where a comparison between different number of particles / shape types is made, the particle sizes were scaled so that approximately the same filling level of the box or of the drum coating device is reached. The performance numbers measured include:

- Overall average steps calculated per computation second.
- Measured execution time for collision kernels.

Measuring kernel execution times was done using the NVIDIA visual profiler tool. Figure 6.6 shows the machine used for testing purposes. It is a GPU barebone designed to accomodate 8 graphics cards, which was filled with 4 NVIDIA GeForce®GTX Titan X, already powered by the newest micro-architecture available, namely Maxwell™, driven by 2 Intel®Xeon®CPUs (E5-2650 v3 @ 2.30GHz) and 128 GB of DDR4 memory. The NVIDIA driver used for testing was 352.63, and CUDA® toolkit version 7.5.18 was installed on the CentOS 7 system (Linux Kernel 3.10).

6.3.1 Test Case 1 - Tuning Size of Static Neighbor-List

This test case determines appropriate sizes for the static neighbor-list. Possible numbers for static neighbor entries per particle are only powers of two up to 32. Included are tests with tablets and spheres, as they behave differently in terms of code convergence or actual collisions with entries in the neighbor-list. The test-cases are illustrated in figure 6.7. In table 6.2 measured execution times for neighbor search, processing static and dynamic neighbor-list are given for one million spheres settled in a box. Table 6.3 proves that also for 100 million spheres with many collisions the same static size of the neighbor-list should be chosen.

For tablets the results are different. Execution times shown in tables 6.4 and 6.5 for one and 10 million tablets, respectively. It can be seen that for non-spherical particles there is nearly no advantage using the static neighbor-list. The reason might be that for spherical particles processing the neighbor-list has high convergence (each entry is a real collision partner) and only needs low computational resources, so that memory loads and especially the atomic writes become more important, which is better handled by the static neighbor-list due to automatically being in a sorted state and using warp reduction for the thread-per-contact implementation. For tablets, not each entry is actual a real collision, leading to less conflicts at the atomic force summation.

The difference between using the thread-per-contact or thread-per-particle algorithm is small. Thread-per-contact leads to slightly better performance but is more sensitive to the

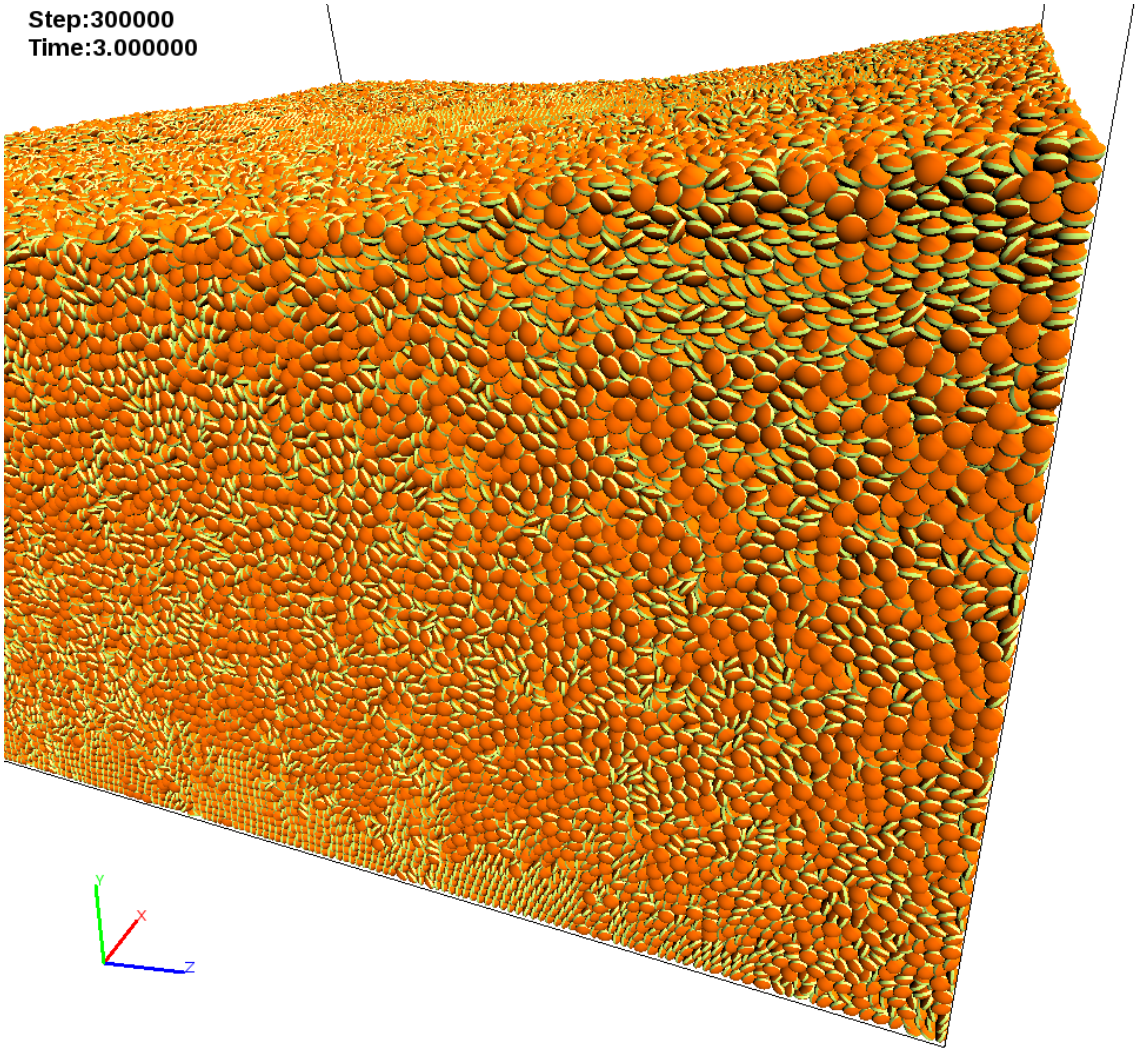

Figure 6.7: Neighbor-list tuning use-case: particles settled in a box. Exemplarily shown is the one million tablet case.

Static Entries	Process time neighbor search	Process time static list Thread per Contact/Particle	Process time dynamic list	Overall Steps/s Thread per Contact/Particle
0	1.98	0 / 0	2.50	136 / 136
1	1.97	0.46 / 0.47	2.01	134 / 133
2	1.92	0.74 / 0.75	1.51	137 / 136
4	1.82	1.36 / 1.28	0.70	142 / 146
8	1.77	**1.79 / 1.75**	0.09	**149 / 150**
16	1.78	3.46 / 1.83	0.00	126 / 148
32	1.80	5.66 / 1.93	0.00	92 / 139

Table 6.2: Hybrid neighbor-list tuning for one million spheres. Times given in milliseconds. Average number of neighbors: 9.31

Static Entries	Process time neighbor search	Process time static list Thread per Contact/Particle	Process time dynamic list	Overall Steps/s Thread per Contact/Particle
0	194	0 / 0	153	2.04 / 2.04
1	193	45 / 46	153	1.85 / 1.84
2	190	75 / 75	137	1.82 / 1.81
4	182	97 / 121	69	2.08 / 1.91
8	176	**122** / 150	2	**2.19** / 2.05

Table 6.3: Hybrid neighbor-list tuning for 100 million spheres. Times given in milliseconds. Average number of neighbors: 7.56

number of static entries. This is because the thread-per-particle force kernel stops when there are no more valid neighbor indices, whereas the thread-per-contact kernel is started for each entry in the neighbor-list. For 32 static entries per particle this means that there is one full warp per particle, but more than half of the warp is inactive because of invalid neighbor indices. But, performance also decreases for large static list sizes in thread-per-particle mode due to worse memory alignment - 16 consecutive integers read from one array, per thread, is already the maximum to comply computing guide recommendations for maximum memory throughput.

What also can be seen in these results is that the neighbor search performances increases slightly with the number of static entries. This is because for the static list a local counter can be used as there is one thread per particle created for neighbor search, whereas for adding an element to the dynamic list an atomic increment operation on a global counter variable is necessary. Additionally, indices have to be stored for both collision partners, opposed to only one index when adding a neighbor to the static list.

Static Entries	Process time neighbor search	Process time static list Thread per Contact/Particle	Process time dynamic list	Overall Steps/s Thread per Contact/Particle
0	2.88	0 / 0	8.6	68 / 68
1	2.87	1.36 / 1.4	7.28	67 / 67
2	2.84	2.55 / 2.65	5.9	68 / 67
4	2.72	**5.05** / 5.25	3.22	**69** / 68
8	2.5	9.5 / 10.25	0.67	61 / 59
16	2.48	17.7 / 12	0	41 / 55

Table 6.4: Hybrid neighbor-list tuning for one million tablets. Times given in milliseconds. Average number of neighbors: 11.95

Static Entries	Process time neighbor search	Process time static list Thread per Contact/Particle	Process time dynamic list	Overall Steps/s Thread per Contact/Particle
0	29	0 / 0	88	**7.37 / 7.37**
1	29.5	13.5 / 13.7	76	7.22 / 7.21
2	29	25 / 26	62	7.32 / 7.31
4	28	49 / 51	47	6.91 / 6.84
8	26	92 / 100	8	6.7 / 6.43
16	26	175 / 118	0	4.41 / 6

Table 6.5: Hybrid neighbor-list tuning for 10 million tablets. Times given in milliseconds. Average number of neighbors: 11.9

6.3.2 Test Case 2 - No Collisions

The no-collision test cases are special test cases for performance comparison when having nearly no collisions. Shown in tables 6.6 and 6.7 significant speed-ups are observed. The standard configuration of the static neighbor list size (8 entries per particle) here is not the optimal choice because it has to be processed even without valid neighbors in the list.

Algorithm	Collect NBs	Forces	Overall Steps/s
old		61	12.9
new: pure dynamic list	12.07	0	**36.8**
new: with static list	12.07	5.71	30

Table 6.6: Execution times for 10 million spheres in a grid without any collisions. Times given in milliseconds.

Algorithm	Collect NBs	Forces	Overall Steps/s
old		334	2.8
new	18.85	8.35	**21.6**

Table 6.7: Execution times for 10 million tablets in a grid without any collisions. Times given in milliseconds.

6.3.3 Test Case 3 - Particles in a Drum-Coater

For this use case a industrial drum coater geometry is used. Initially 10 millions of particles were placed with the so-called fill placer of XPS which can be used to fill geometries with particles as an initial condition. Afterwards the simulation was run up to one full revolution of the coater rotating with 8 rpm, that is $t = 7.5$ seconds. From there a restart is done using different algorithms, including the old one which was implemented at the start of this master thesis. The use-cases are illustrated in figure 6.8. Results are shown in tables 6.8 and 6.9. Additionally, integration of particle positions and velocities takes about 8 ms for both cases, and the sorting into the uniform spatial grid takes between 11 ms and 13 ms. In this real-world case the fastest versions were:

- Using thread-per-contact processing for static and unsorted dynamic neighbor list
- Particle neighbor search via uniform grid
- Triangle neighbor search via BVH
- Static neighbor list size of 8 for spheres, and 0 for tablets (i.e. a pure dynamic list)
- Symmetric Lists

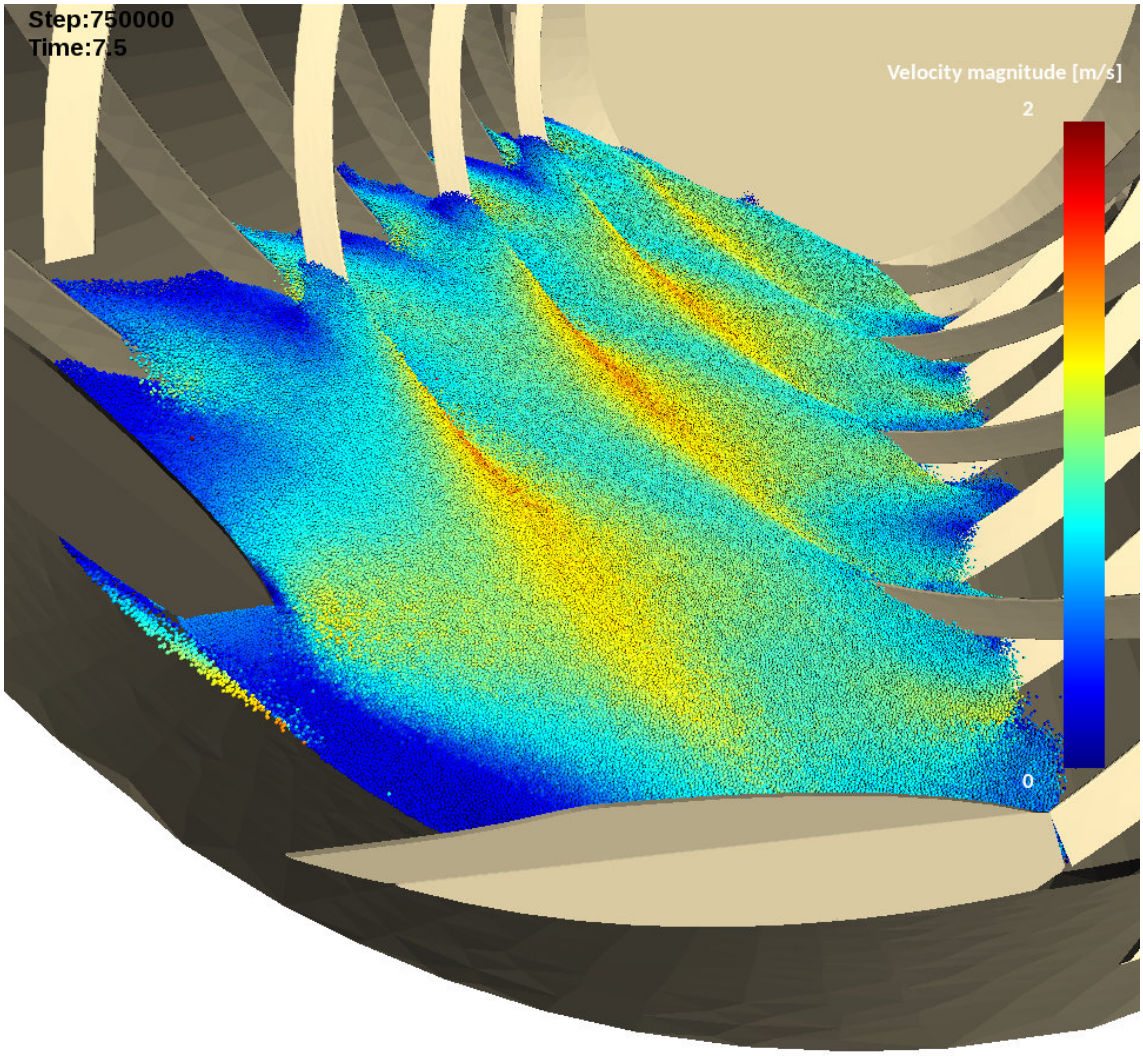

Figure 6.8: Drum coating device use-case. Exemplarily shown for 10 million spheres, colored by magnitude of velocity.

Algo.	Prep. Triangles	Collect Tr. NBs	Collect Part. NBs	Forces Triangles	Forces Particles	Overall Steps/s	Memory
old	760			104.5	135	0.96	2071 MiB
new	2.5	11.75	16.1	1	19.6	**13.4**	2011 MiB

Table 6.8: Execution times for 10 million spheres in a drum coater. Times given in milliseconds.

Algo.	Prep. Triangles	Collect Tr. NBs	Collect Part. NBs	Forces Triangles	Forces Particles	Overall Steps/s	Memory
old	470			511	1051	0.48	1737 MiB
new	2	10.8	24	4.3	63	**7.9**	1773 MiB

Table 6.9: Execution times for 10 million tablets in a drum coater. Times given in milliseconds.

These settings seem to be the optimum and are now used as standard in XPS, giving much better performance in general for simulations done at RCPE compared to the old implementation. For comparison, the average entries in the collision list in this use-case are 7.85 neighbors per tablet particle and 4.3 neighbors per spherical particle, which means much less contacts than in the tuning examples for the static neighbor list size where particles fill up a box. Still, 8 static neighbor entries is fine for spheres, whereas tablets tend to perform better with dynamic list use only.

What can be seen in the results is the huge amount of time needed for triangle inserting into the uniform grid in the old algorithm. This one was already known to be very inefficient and sensitive to actual cell sizes. The automatically chosen grid size according to particle sizes was 410|655|410 for spheres and 348|556|348 for tablets which is responsible for the different execution times of 470 and 760 milliseconds for triangle cell sorting. The huge speed-up of $7.9/0.48 = 16.5$ for tablets and $13.4/0.96 = 14$ for spheres therefore may not apply to each real-world example. But, also when neglecting geometries, the reached speed-up is significant with only moderately higher memory consumption. For example the whole speed-up neglecting geometries, but including numerical integration and uniform grid sorting, is calculated approximately as following:

- Spheres (see table 6.8, where 19 ms are needed for preparation and integration): $(19 + 135)/(19 + 16.1 + 19.6) = 2.82$
- Tablets (see table 6.9, where 21 ms are needed for preparation and integration): $(21 + 1130)/(21 + 24 + 63) = 10.7$

These results show that contact lists are necessary to get good performance out of DEM simulations, especially for non-trivial shapes.

The additional memory consumption for the particle-particle neighbor lists is approximately:

- Spheres: 8 static entries, 4 byte each: $8 \cdot 4\,\text{byte} \cdot 10^7 = 320\,\text{MiB}$
- Tablets: Approx. 4 dynamic entries, 8 byte each: $4 \cdot 8\,\text{byte} \cdot 10^7 = 320\,\text{MiB}$

Additionally there is some more memory needed for particle-triangle collision pairs. When compared with the overall memory consumption given in tables 6.8 and 6.9, in the spheres use-case there is less memory needed overall. In the first place this is because for the spheres a smaller cell-size can be used, and therefore more cells are used. Also, the BVH structure needs less memory than the uniform grid, for this particular geometry.

6.3.4 Test Case 4 - Effect of Sorting in a Real-World Drum-Coater

During implementation of the new algorithms the matter of sorting particle data became more important. As shown in figures 6.9 and 6.10, illustrating the execution speed in steps per second over time, there is a big impact on performance whether particle data sorting is performed or not. Taken as use-cases are the previous examples of particles in a drum coating device, which are moving due to the rotation of the whole geometry. Therefore the particles are continuously being mixed up in the process. Due to worse memory alignment and less locality, leading to more global memory loads per warp, the performance reached without sorting is not optimal. Speed-ups due to sorting all the particle data are in the region of 4-5 in these examples and can further be increased when sorting is only done every now and then. Especially when simulating spheres the overhead of sorting is significant (see bottom figure in 6.9), leading to approximately 20% better performance when only sorting every 0.0001 to 0.001 seconds, which seems to be a good choice for tablets as well. But, the best value depends on the actual simulation problem and the degree and speed of mixing happening. The speed losses which occur at simulation time approximately $t = 7.505\,s$ caused attention and could be linked to the driver throttling GPU clock cause of thermal problems due to low fan speeds at the simulation start.

For a detailed view the execution time for the kernels at sorted vs. unsorted particle data is shown in tables 6.8, 6.9, 6.10, and 6.11. In general all the kernels regarding neighbor search or force calculation, either particles or particle-triangle contacts, benefit from sorting because of better data locality. This effect is much stronger when the new, more performant, kernels are used for separating broad- and narrow-phase collision detection, as also can be observed in figure 6.11, showing the execution speed over time at different sorting intervals for the original algorithms.

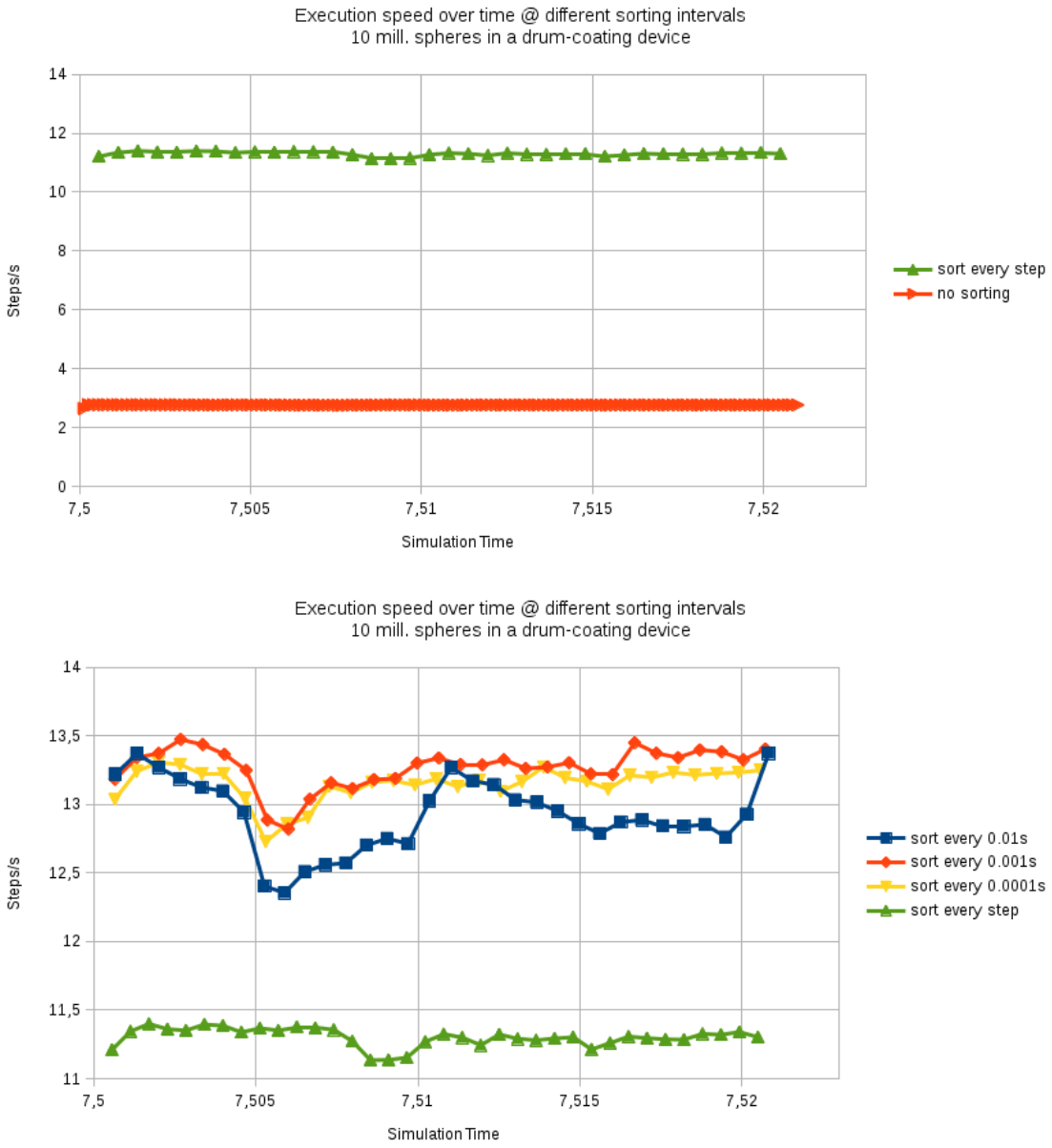

Figure 6.9: Execution speed over time at different sorting intervals for spheres. Top: Difference in execution speed when sort is enabled or not. Bottom: Only sort after specific time interval.

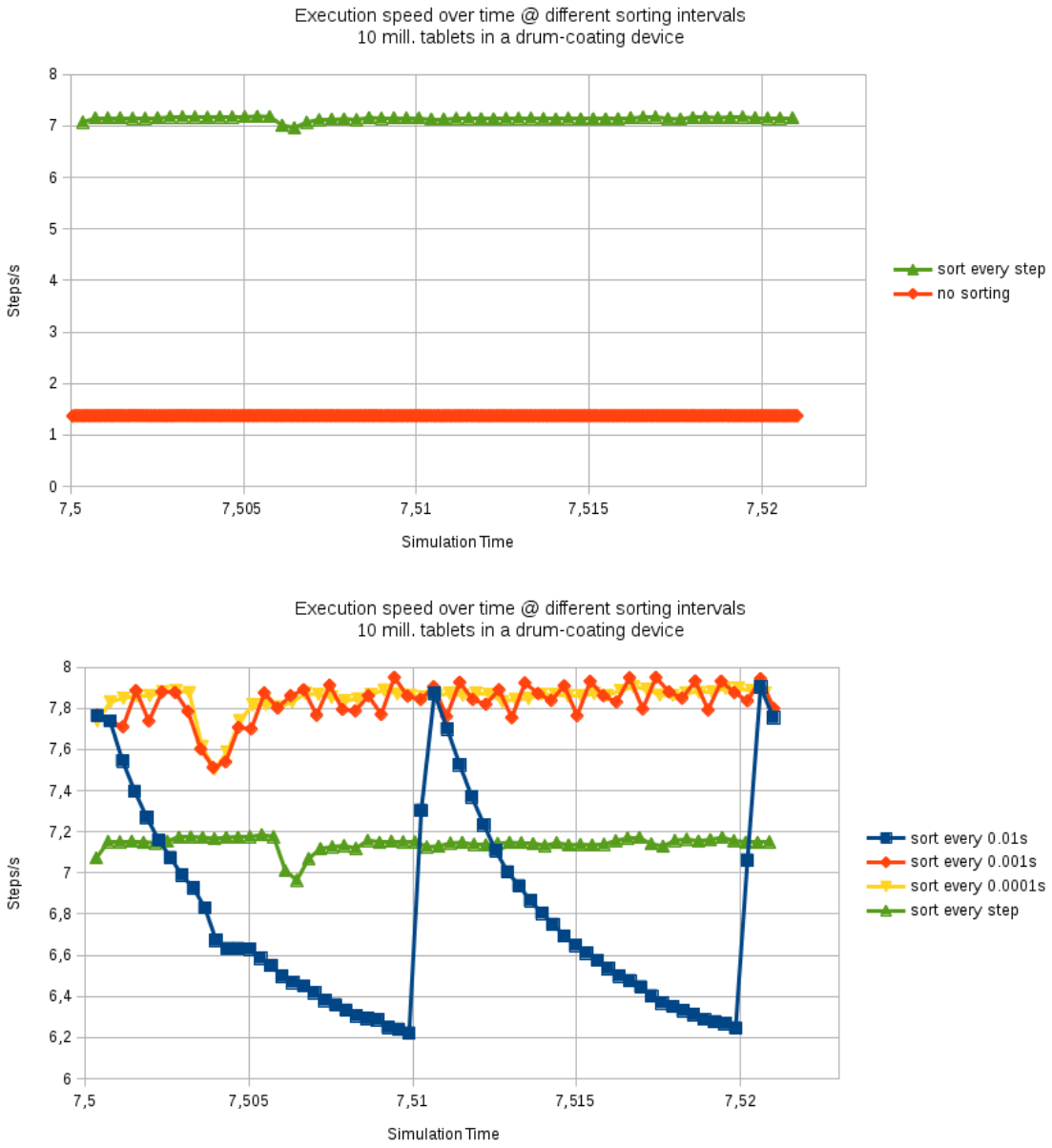

Figure 6.10: Execution speed over time at different sorting intervals for tablets. Top: Difference in execution speed when sort is enabled or not. Bottom: Only sort after specific time interval.

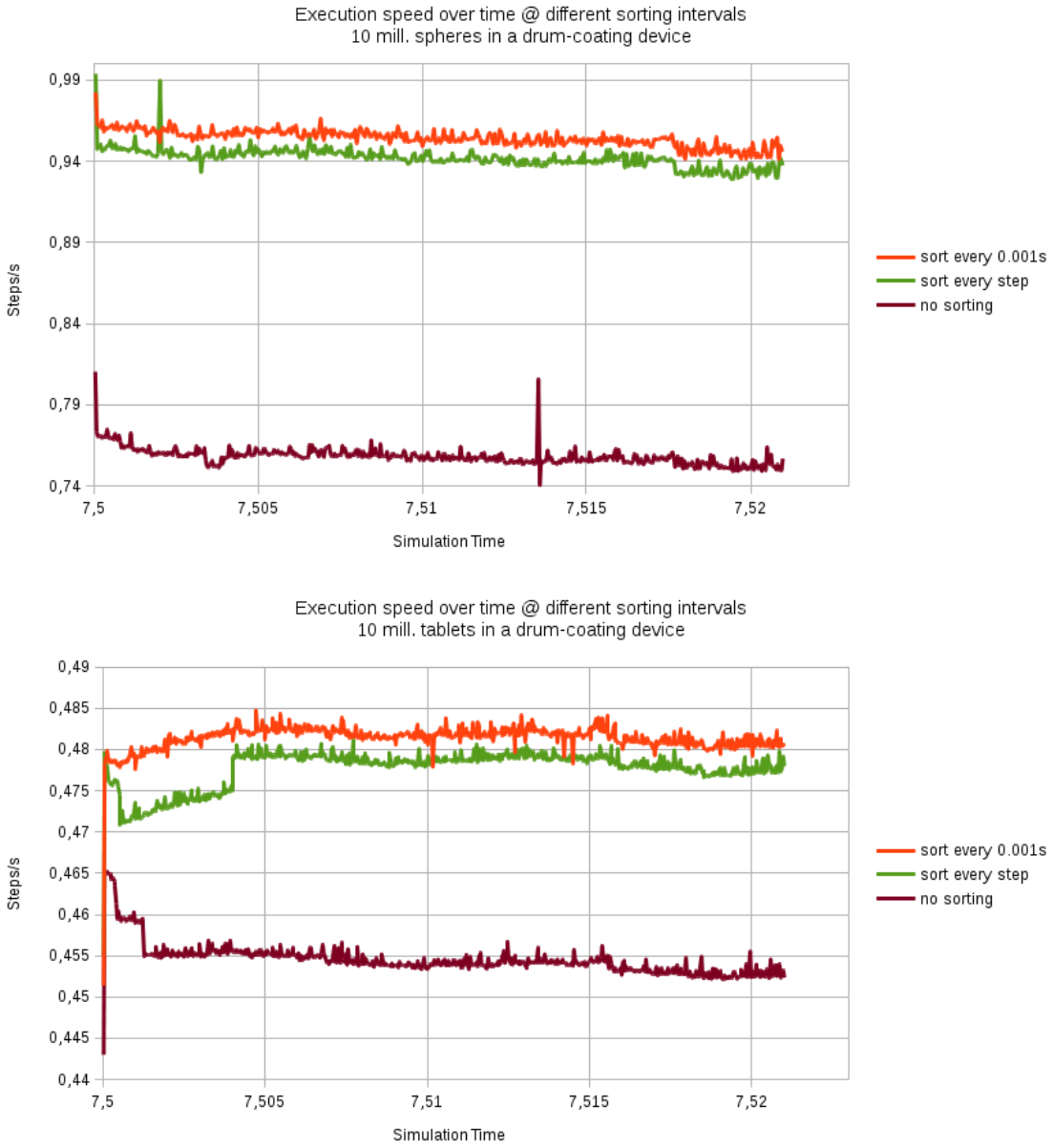

Figure 6.11: Execution speed over time at different sorting intervals for spheres (top) and tablets (bottom), using the original implementation.

Algo.	Preparation Triangles	Collect Triangle NBs	Collect Particle NBs	Forces Triangles	Forces Particles	Overall Steps/s
old	760			110	400	0.76
new	2.5	15	216	1.2	103	2.8

Table 6.10: Execution times for 10 million spheres in a drum coater. Particle data is not sorted. Times given in milliseconds.

Algo.	Preparation Triangles	Collect Triangle NBs	Collect Particle NBs	Forces Triangles	Forces Particles	Overall Steps/s
old	470			516	1130	0.45
new	2	14	248	5.1	428	1.4

Table 6.11: Execution times for 10 million tablets in a drum coater. Particle data is not sorted. Times given in milliseconds.

6.3.5 Test Case 5 - Particle Neighbor Search via BVH

To test the performance of the final BVH implementation also for particles, it is sufficient to use the particles-in-a-box use-case as this is not a matter of geometries used. In table 6.12 the results are given. Not surprising, traversing the tree takes much longer than the uniform-grid search, but also creating the tree is more effort. Nonetheless the overall slowdown compared to uniform-grid is approximately 3 in this case. Also, there is significantly more memory used, which was already explained in section 3.3.

NB-search algo.	Build structure	Collect NBs	Forces	Steps/s	Memory
Uniform-grid based	12	18	31.5	**13.96**	1700 MiB
BVH based	27	153	31	4.46	2187 MiB

Table 6.12: Execution times for 10 million monodisperse spheres in a box, comparison between uniform-grid and BVH. Times given in milliseconds.

One of the initial ideas of where using the BVH could be useful is simulating polydisperse particles with a big size-ratio, when many small particles are contained. The next use-case is shown in figure 6.12 and the simulation results are given in table 6.13, where one million small spheres (radius: one millimeter) were mixed with 276 large spheres (radius: 10 millimeter). As can be observed, the BVH neighbor search performs much better because its insensitivity to particle sizes, opposed to the uniform-grid algorithm which now has to loop over up to approximately 10^3 of the small particles in one cell, instead of just one if there were no large particles in the simulation which would allow to keep the cells small.

NB-search algo.	Collect NBs	Forces	Steps/s	Memory
Uniform-grid based	328	2.3	3	326 MiB
BVH based	14	2.3	**45**	379 MiB

Table 6.13: Execution times for one million 1:10 bi-disperse spheres in a box, comparison between uniform-grid and BVH. Times given in milliseconds.

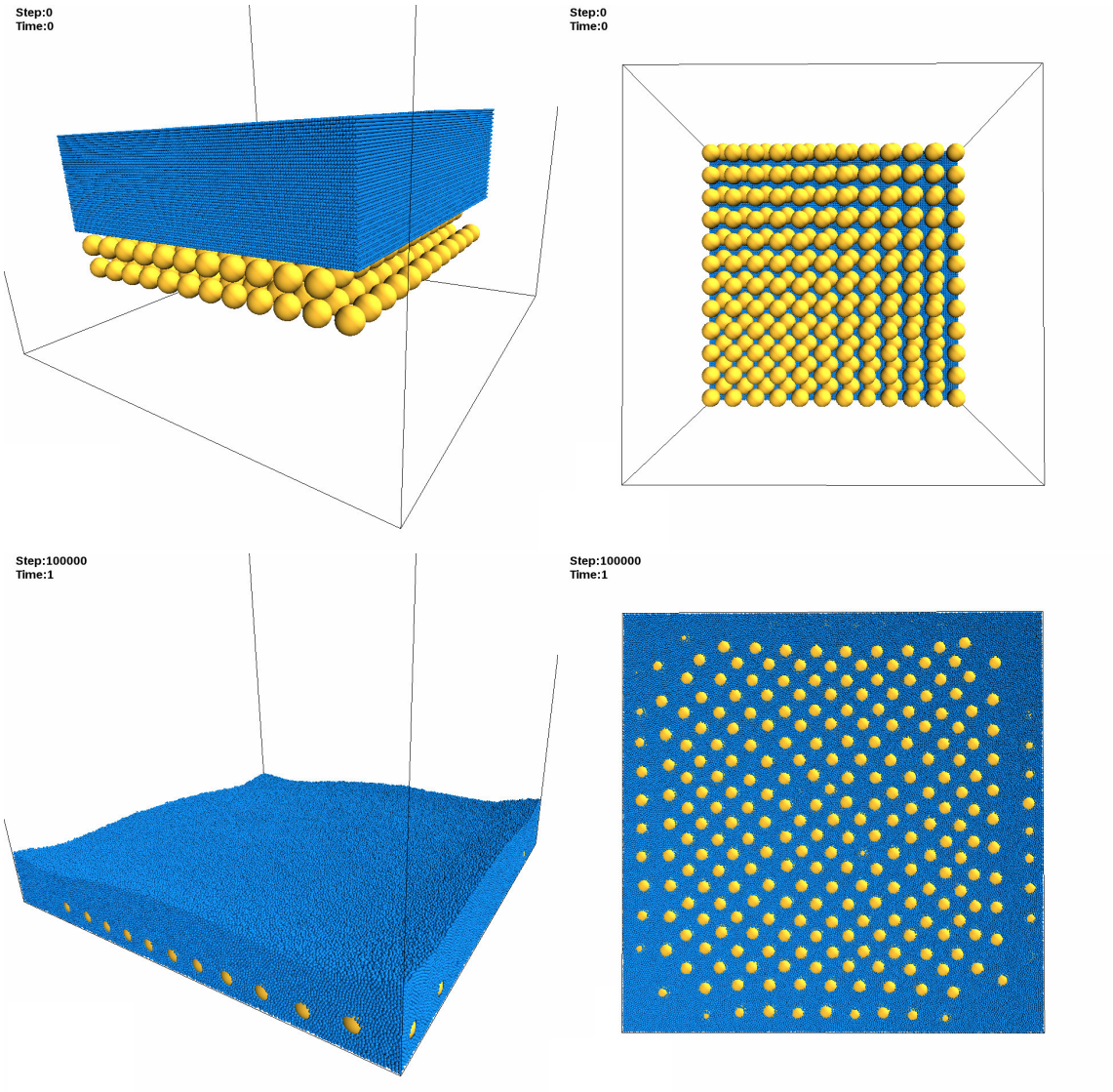

Figure 6.12: Bi-disperse particles with a 1:10 size-ratio use-case: initial positions of particles (top) vs. settled positions after one second (bottom). Left pictures were made at side view, right pictures are bottom view.

6.3.6 Test Case 6 - Slow-Down when Contact List is Reused

The drum-coating use-case was also simulated for one million spheres, which is used now to show the difference in execution times when using the force history or not. A restart at time $t = 7.5\,s$ was done with and without force history, using dynamic lists only, leading to the following results:

- Execution time for collecting particle neighbors increased from 1.55 ms to 3.42 ms
- Execution time for collecting triangle neighbors increased from 1.94 ms to 6.10 ms
- Execution time for calculating particle neighbors forces increased from 2.21 ms to 2.32 ms
- Execution time for calculating triangle neighbors forces increased from 0.40 ms to 0.46 ms
- Overall execution speed dropped from 83 to 36 steps per second
- Additional memory consumption of one *float4* per contact (4 floats = 16 bytes for the tangential displacement)

These results show that there is a significant additional cost when reusing the neighbor-list to make features like force history possible. The main reason for this are the additional un-coalesced memory loads due to check if the neighbor is already in the list and deleting vanished contacts. Additionally, if dynamic neighbor lists are used, there is a huge overhead for sorting the neighbor-list, which is necessary for looping through already existing neighbors and to move contacts which are over to the end of the list.

6.3.7 Discussion

To summarize the achievements following speed-ups were reached for tablet-shaped particles:

- One million particles in a box (see table 6.4): $69/8.63 = 8.00$
- 10 million particles in a box (see table 6.5): $7.37/0.91 = 8.10$
- 10 million particles in a box without collisions (see table 6.7): $21.6/2.8 = 7.71$
- 10 million particles in a drum-coating device (see table 6.9): $7.8/0.45 = 17.33$

Here it can be seen that for complex-shaped particles it is really beneficial to use neighbor-lists.

The following speed-ups were reached for spherical particles:

- One million particles in a box (see table 6.2): $149/60 = 2.48$
- 100 million particles in a box (see table 6.3): $2.19/0.64 = 3.42$
- 10 million particles in a box without collisions (see table 6.6): $36.8/12.9 = 2.85$
- 10 million particles in a drum-coating device (see table 6.8): $13.4/1 = 13.4$

Still, a good speed-up results from using neighbor lists. In any case the new BVH based algorithm for spatial subdivision of the geometrie's triangles should be used to avoid having potential performance issues with the old cell-space based algorithm, being too sensitive to the cell size used in the simulation.

When the BVH is used for particles there is a significant impact on performance when having poly-disperse particles in the simulation with a big size ratio and many small particles included. For the bi-disperse 1:10 case given in table 6.13 a speed-up of $45/3 = 15$ was reached comparing cell-space and BVH based neighbor search algorithms. Also, the BVH would make simulations with large empty areas feasible.

An issue when doing DEM simulations is that particles potentially are in continuous motion, and it is therefore reasonable to sort all the particle data at least every now and then to assure a better data locality among collision partners.

Still to be analyzed is the role of static neighbor lists and what is the fastest way of tracking and re-using contacts.

Chapter 7

Conclusion and Outlook

The goal of this thesis was to improve the current implementation of the CUDA®-based DEM implementation in the XPS software package. Alternative and more sophisticated approaches were discussed and good speed-ups were reached.

The most significant improvements yielded the LBVH implementation for non-uniformly sized particles with a big size-ratio. Recalling the introduction, this was the reason to implement bounding volume trees in the first place.

Decoupling of neighbor search and force calculation demonstrated its strength especially for non-spherical particles, where huge speed-ups were achieved. Also, tracking contacts is now possible which opens lots of new applications in the future.

At this point, the fastest known DEM implementation for spheres (BLAZE-DEM, see [GWKE14]) is performing at approximately one hundred steps per second for one million spheres, which now is reached by XPS easily as well.

7.1 Future Work

As already mentioned in the introduction, there is a simple coating algorithm implemented in XPS, a uniform grid cell-based ray-tracing method. This implementation is rather slow because to work correctly it has to loop over many cells to finally find the particle which is hit. For this purpose the implemented BVH would probably be a better acceleration structure.

Instead of the LBVH there are also other approaches which could improve tree quality for a faster traversal, but probably slower tree construction. One approach is presented in [LGS+09], discussing a hybrid tree using surface area heuristic in combination with an LBVH to get a good trade-off between fast construction and good-quality trees. Also, the memory consumption of the BVH is quite high compared with the uniform grid approach. Both memory consumption and traversal time could be reduced with similar approaches,

for example an Octree can be used, where each node has 8 instead of 2 children.

Last, but not least, the neighbor-list implementation can still be improved. The designed algorithms are also used in a Smoothed Particle Hydrodynamics (SPH) implementation currently under development, where each particle has much more neighbors and additional elements are stored per contact, therefore it is likely to be modified to meet those special requirements.

Appendix A

Acronyms and Glossaries

Acronyms

AABB Axis Aligned Bounding Box 18

AoS Array of Structures 33

BVH Bounding Volume Hierarchy 4

CFD Computational Fluid Dynamics 1

CPU Central Processing Unit 7

CUDA® Compute Unified Device Architecture 2

DEM Discrete Element Method 1

GPGPU General Purpose Computation on Graphics Processing Units 7

GPU Graphics Processing Unit 7

ITI Institute for Technical Informatics of the TU Graz 1

LBVH Linear Bounding Volume Hierarchy 18

RCPE Research Center Pharmaceutical Engineering 1

SIMT Unique architecture employed by a SM. It creates, manages and schedules threads of a warp. 8

SM Streaming Multiprocessor (SM, SMX) 8

SoA Structure of Arrays 33

SPH Smoothed Particle Hydrodynamics 75

STL STereoLithography 16

Visualization ToolKit VTK 51

XPS eXtended Particle System VII

Glossary

AVL FIRE® A powerful multi-purpose thermo-fluid software representing the latest generation of 3D CFD. It is being developed and continuously improved to solve the most demanding problems in respect of geometrical complexity, physics and chemistry. 1

Central Processing Unit A integrated electronic circuit (processor) performing logical, control and I/O operations. 7

Discrete Element Method A numerical simulation method for computing stresses and motion of arbitrary particle systems. 1

eXtended Particle System eXtended Particle System (XPS) is the name of a particle simulation software using a CUDA® implementation of the DEM developed at the RCPE. ..... VII, 1

Graphics Processing Unit A specialized electronic circuit (processor) mostly used to compute graphics and images. 7

Kepler™ Kepler™ is the codename for a GPU microarchitecture developed by Nvidia as the successor to the Fermi microarchitecture, introduced in March 2012. 10

Maxwell™ Maxwell™ is the codename for a GPU microarchitecture developed by NVIDIA as the successor to the Kepler microarchitecture, introduced in September 2014. 59

NVIDIA The NVIDIA Corporation 2

ParaView ParaView is an open-source, multi-platform data analysis and visualization application. 51

Smoothed Particle Hydrodynamics A numerical simulation method for computing fluid flows, based on particles. 75

Thrust Thrust is a C++ template library for CUDA, based on the Standard Template Library (STL). 33

Bibliography

[ASA+09] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Real-time Parallel Hashing on the GPU. In *ACM SIGGRAPH Asia 2009 Papers*, SIGGRAPH Asia '09, pages 154:1–154:9, New York, NY, USA, 2009. ACM.

[CS79] P. A. Cundall and O. D. L. Strack. A discrete numerical model for granular assemblies. *Géotechnique*, 29(1):47–65, 1979.

[CS13] J. Conway and N.J.A. Sloane. *Sphere Packings, Lattices and Groups*. Grundlehren der mathematischen Wissenschaften. Springer New York, 2013.

[Cun71] P.A. Cundall. A Computer Model for Simulating Progressive Large Scale Movements in Blocky Rock Systems. In *Proc. Int. Symp. Rock Fracture, ISRM*, pages 2–8, Nancy (F), 1971.

[GWKE14] Nicolin Govender, Daniel N. Wilke, Schalk Kok, and Rosanne Els. Development of a convex polyhedral discrete element simulation framework for NVIDIA Kepler based GPUs. *Journal of Computational and Applied Mathematics*, 270:386 – 400, 2014. Fourth International Conference on Finite Element Methods in Engineering and Sciences (FEMTEC 2013).

[JH04] G. G. JOSEPH and M. L. HUNT. Oblique particle-wall collisions in a liquid. *Journal of Fluid Mechanics*, 510:71–93, 7 2004.

[JSRK13] Dalibor Jajcevic, Eva Siegmann, Charles Radeke, and Johannes G. Khinast. Large-scale CFD-DEM simulations of fluidized granular systems. *Chemical Engineering Science*, 98:298 – 310, 2013.

[Kar12a] Tero Karras. Maximizing parallelism in the construction of BVHs, Octrees, and K-d Trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, EGGH-HPG'12, pages 33–37, Aire-la-Ville, Switzerland, Switzerland, 2012. Eurographics Association.

[Kar12b] Tero Karras. Thinking Parallel, Part I: Collision Detection on the GPU. `https://devblogs.nvidia.com/parallelforall/thinking-parallel-part-i-collision-detection-gpu/`, November 2012. Accessed: 2016-02-14.

[KEWS08] H. Kruggel-Emden, S. Wirtz, and V. Scherer. A study on tangential force laws applicable to the discrete element method (DEM) for materials with viscoelastic or plastic behavior. *Chemical Engineering Science*, 63(6):1523 – 1541, 2008.

[Kre11] Yossi Kreinin. SIMD <SIMT <SMT: parallelism in NVIDIA GPUs. `http://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html`, November 2011. Accessed: 2016-02-12.

[LGS+09] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH Construction on GPUs. In *IN PROC. EUROGRAPHICS 09*, 2009.

[NVI15] NVIDIA Corporation, 2701 San Tomas Expressway; Santa Clara, CA 95050. *NVIDIA CUDA C Programming Guide*, version 7.5 edition, 2015.

[OL12] Vitaliy Ogarko and Stefan Luding. A fast multilevel algorithm for contact detection of arbitrarily polydisperse objects. *Computer Physics Communications*, 183(4):931–936, 2012.

[PH08] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design, Fourth Edition: The Hardware/Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2008.

[Rad06] Charles Radeke. *Statistische und mechanische Analyse der Kräfte und Bruchfestigkeit von dicht gepackten granularen Medien unter mechanischer Belastung*. PhD thesis, Fakultät für Mathematik und Informatik der TU Bergakademie Freiberg, 2006.

[RGK10] Charles A. Radeke, Benjamin J. Glasser, and Johannes G. Khinast. Large-scale powder mixer simulations using massively parallel gpuarchitectures. *Chemical Engineering Science*, 65(24):6435 – 6442, 2010.

[Wal94] O.R. Walton. Numerical simulation of inelastic frictional particle-particle interaction. In M.C. Roco, editor, *Particulate Two-Phase Flow*, chapter 25, pages 884–907. Butterworth-Heinemann, Boston, 1994.

[Wan10] Peng Wang. Short-Range Molecular Dynamics on GPU. `http://on-demand.gputechconf.com/gtc/2010/presentations/S12006-Short-Range-Molecular-Dynamics-GPU.pdf`, September 2010. Accessed: 2016-04-15.

[Wes15] Elmar Westphal. Voting And Shuffling for Fewer Atomic Operations. `http://http://on-demand.gputechconf.com/gtc/2015/presentation/S5151-Elmar-Westphal.pdf`, April 2015. Accessed: 2016-05-10.

[Wil13] N. Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley, 2013.