Georg Neubauer, BSc.

A Multi-GPU Implementation of the Discrete Element Method

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

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

In cooperation with
Research Center of Pharmaceutical Engineering GmbH

Graz, February 2018

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

Ursprünglich von Cundall im Jahre 1971 konzipiert ist die Diskrete-Elemente-Methode (DEM) heutzutage eine der Standardmethoden zur numerischen Bestimmung von Teilchenbewegungen in Simulationen. Grundlegend ist es für diese Methode zu erkennen ob und wann Partikel miteinander kollidieren und aus der dadurch resultierenden Kraftwirkung die neuen Bewegungsparameter (Richtung, Geschwindigkeit, Rotation) pro Teilchen zu errechnen.

Bei der Abbildung von realen Prozessen müssen oft große Partikelsysteme, mit relativ kleinen Partikeln (z.B.: Schüttgut, Puder, etc.) simuliert werden. Um das Simulationsergebnis so akkurat wie möglich zu machen benötigt die DEM jedoch sehr kleine Zeitschritte, wodurch die Berechnungsdauer oft jegliche vernünftige Zeitspanne übersteigt. Aufgrund der zugrundeliegenden Methodik der Diskretisierung wird jedes Partikel für sich separat betrachtet. Dies führt zu einem hohen Grad an Parallelisierbarkeit. Anfangs oft auf CPUs berechnet, werden heutzutage geeignetere Plattformen (z.B.: GPUs, CUDA®, Vektor-Rechner) verwendet um die benötigte Ausführungszeit zu senken.

Ziel dieser Arbeit ist es nun, die bereits im bestehenden Programm (XPS) vorhandene Algorithmik dahingehend zu erweitern, dass durch die Verwendung von mehreren GPUs eine höhere Stufe von Parallelität erreicht wird. Zudem soll im bestehenden Programm die Möglichkeit geschaffen werden die Simulation in Echtzeit zu überwachen ohne dadurch zu viel Prozesszeit zu verschwenden. Nach der Beschreibung der notwendigen Änderungen und der Implementierung wird anhand von Testfällen gezeigt, dass ein deutlicher Speedup erreicht wurde. Zudem wird auch noch darauf hingewiesen, dass durch die Verwendung von mehreren GPUs die abbildbare Problemgröße gesteigert werden kann.

Schlüsselwörter

Diskrete-Elemente-Methode, Partikelsimulation, CUDA®, Multi GPU, Shared memory

Abstract

Invented by Cundall in the year 1971, the Discrete-Element-Method (DEM) is a standard method for numerical determination particle motions in simulations. A fundamental aspect for this method is to determine if and when particles will collide with each other. By giving the resulting contact force the new motion parameter (like direction, velocity and rotation) can be calculated for each particle separably.

To illustrate real-world processes huge particle sets, consisting of relatively small particles (e.g. bulk solids, powders, etc.), have to be simulated. While getting the simulation outcome as accurate as possible the DEM needs rather small time-step sizes. This leads to unreasonable long total simulation times. The method itself is looking on the particles in a discrete way. With this it is possible to look at each discretization separately, leading to a high degree of parallelism. Modern vectorizing architectures (e.g. GPUs, CUDA) provide many advantages on simulating huge particle systems over the CPUs used formerly, resulting in much faster simulations.

The goal of this thesis is to extend an already existing simulation software (XPS) by introducing new algorithmic features. This is mostly enabling multi-GPU support to achieve a higher level of parallelism. Additionally a method should be invented to observe the simulation in real-time by not wasting to much of it's process's time. After describing the necessary changes and the implementation a result section will show some test cases. It will be shown there how the simulation data can now be shared and which significant speed-ups can be observed. Last but not least, it will be mentioned that the maximum illustratable problem size was increased.

Keywords

Discrete-Element-Method, particle simulation, CUDA®, multi GPU, shared memory

Acknowledgment

This master thesis was carried out during the years 2015 to 2018 on the Institute for Technical Informatics at Graz University of Technology.
First, I want to thank Mr. Christian Steger for the possibility to write my master thesis at the Institute for Technical Informatics.
I also want to express my special gratitude to Charles Radeke, who supported me with all of his possibilities over the whole time of this thesis.
A thank goes to Hermann Kureck, Eva Siegmann, Dalibor Jajcevic, Eyke Slama and all of my former colleagues from RCPE which supported me to finish the work.
Special thanks go to my wife Margit, who supported me with some graphical stuff, my daughter Eva and my son Tobias for their excitation.
Last but not least I want do thank my parents, my parents-in-law and all of my siblings and friends, who supported me throughout my student time and I am sure that they will support me in my future.

Graz, February 2018 Georg Neubauer, BSc.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Objectives 2
1.3 XPS - eXtended Particle System 3
1.4 Outline 5

2 Theory **6**
2.1 Discrete Element Method (DEM) 6
2.2 Compute Unified Device Architecture 7
2.2.1 Programming Model 8
2.2.2 Hardware Implementation 13
2.2.3 Multi-GPU 15
2.2.4 Thrust 16
2.2.5 CUDA Profiling Tools 18
2.3 Inter-process Communication (IPC) 18
2.3.1 Shared Memory 18
2.3.2 Resource Locking 19
2.3.3 Message-Passing Interface (MPI) 22
2.4 Program Libraries 22
2.4.1 Static Linking 23
2.4.2 Dynamic Linking 23
2.4.3 The BOOST C++ Libraries 24
2.5 XPS - eXtended Particle System 25
2.5.1 Overview 25
2.5.2 Current Implementation 26
2.5.3 Implemented DEM Algorithmic 27

3 Shared Simulation Data **34**
3.1 Design . . . 34
3.1.1 Pipes . . . 34
3.1.2 Streams . . . 34
3.1.3 Shared Memory . . . 35
3.1.4 Actual Design . . . 35
3.2 Implementation . . . 36
3.2.1 Define data to be shared . . . 36
3.2.2 The data sharing infrastructure . . . 37
3.2.3 View and update simulation data . . . 39
3.2.4 Deploy as program library . . . 40
3.2.5 Command line arguments . . . 40
3.3 Results . . . 40
3.3.1 Usage . . . 41
3.3.2 Example . . . 41

4 Multi-GPU DEM - Design **43**
4.1 Subdividing the simulation world . . . 43
4.1.1 Choosing a reasonable sub-division . . . 43
4.1.2 Subdivision overlap and halo distribution . . . 45
4.2 Hiding GPU memory transfers . . . 46
4.3 Load balancing . . . 47
4.4 Possible implementations . . . 50
4.4.1 Single-Threaded Multi-GPU (ST-MGPU) . . . 52
4.4.2 Multi-Threaded Multi-GPU (MT-MGPU) . . . 52
4.4.3 Multi-Process Multi-GPU (MP-MGPU) . . . 52

5 MultiGPU DEM - Implementation **53**
5.1 Preliminary work and re-factoring of the existing code . . . 53
5.2 Top-Down view . . . 54
5.2.1 Algorithm solvers . . . 54
5.3 Multi-Threaded implementation . . . 57
5.4 Optimizing execution speed . . . 58
5.5 Load balancing . . . 59

6 Multi-GPU DEM - Results **60**
6.1 The Test System . . . 60
6.2 The Test Case . . . 61
6.2.1 Evaluation Of Memory Usage . . . 61
6.2.2 Test Case Variations . . . 63
6.3 Test 1: Performance Gain . . . 65
6.3.1 Expectations . . . 66
6.3.2 Results . . . 67
6.3.3 Discussion . . . 71
6.4 Test 2: Reaching The Test System's Limits . . . 72

7 Conclusion and Outlook **75**
7.1 Future Work . . . 75

A Acronyms and Glossaries **76**

Bibliography **79**

List of Figures

1.1 Simulation of a pharmaceutical mixing device in labor/academic scale . . . 3
1.2 Simulation of a pharmaceutical coating device in labor/academic scale . . . 4

2.1 Grid of two-dimensional thread blocks organized in a two-dimensional grid . 10
2.2 CUDA memory hierarchy . . . 12
2.3 CUDA stream concurrency . . . 14
2.4 Dynamic thread-block assignment to available Streaming Multiprocessors . 15
2.5 Example views of the NVIDIA Visual Profiler . . . 17
2.6 Multiple processes map to same shared memory region . . . 19
2.7 Race condition, two threads using same memory resource . . . 20
2.8 Static vs. dynamic linking for the Linux operating system. . . . 24
2.9 The XPS Logo . . . 25
2.10 Particle shapes available in XPS . . . 27
2.11 Spherical objects sorted into a uniformed grid . . . 28
2.12 DEM Preparation phase: From particles in cellspace to *cellstart* vector . . . 29
2.13 DEM Collision phase - neighbor search via uniform grid . . . 31
2.14 The spring/damper model that is used for particle collisions . . . 32

3.1 Connect Live Simulation . . . 41
3.2 Left internal deprecated viewer, right external live viewer . . . 42

4.1 Splitting a 2D domain along an arbitrary path of cell boundaries . . . 44
4.2 Splitting a 2D domain along vertical and horizontal planes . . . 44
4.3 Splitting a 2D domain along vertical or horizontal planes . . . 45
4.4 The boundary between two sub-domains on grid cell layer, with the halo highlighted in red. . . . 46
4.5 A single particle traveling to another GPU domain . . . 47
4.6 Example of load balancing by particle number, of a setup with GPUs of different speed. . . . 48

4.7 Example of load balancing by execution time, of a setup with GPUs of different speed. . . . 49
4.8 Simplified flow-chart for a Discrete Element Method (DEM) implementation computed on n Graphics Processing Units (GPUs). . . . 51

5.1 Class diagram of the minimal *Simulator* interface. . . . 53
5.2 Collaboration diagram of the multi-GPU implementation of the DEM. . . . 55
5.3 Class diagram of the minimal *DEMSolver* interface. . . . 55
5.4 Dedicated footsteps for the multi-GPU based DEM. . . . 56
5.5 Detailed flow chart with marks for needed synchronization points between workers 58

6.1 A view on the test system as used to evaluate the Multi-GPU implementation 61
6.2 Particles falling into a box test case 62
6.3 Test 1: Sub-domain division for the 1 million particles example 66
6.4 Test 1: Evolution of processed simulation steps per seconds over the process time for 1 million particles. . . . 68
6.5 Test 1: Evolution of processed simulation steps per seconds over the process time for 10 million particles. . . . 68
6.6 Test 1: Evolution of processed simulation steps per seconds over the process time for 50 million particles. . . . 69
6.7 Test 1: Evolution of processed simulation steps per seconds over the process time for 100 million particles. . . . 69
6.8 Test 1: Evolution of processed simulation steps per seconds over the process time for 230 and 300 million particles. . . . 70
6.9 Test 1: Performance gain in percent compared from using 1 GPU to 2, 3 and 4 GPUs. . . . 72
6.10 Test 1: Kernel schedule analysis for the 100 million particles example running on three GPUs 72
6.11 Test 1: Kernel schedule analysis for the 100 million particles example running on four GPUs 73

List of Tables

2.1 Currently available CUDA®compute capabilities (as of 2016) 8

3.1 Description of data that needs to be shared between simulation and any observing process 36

6.1 Specifications of the test system as used to evaluate the Multi-GPU implementation. 60

6.2 Minimum GPU memory consumption per particle. 63

6.3 Test case variations. 64

6.4 Test 1: Mapping number of particles examples on number of GPUs. 65

6.5 Test 1: Average simulation steps per second. 70

6.6 Test 1: Wall clock times. 70

Chapter 1

Introduction

This chapter is intended to give a short overview of this master thesis, which was carried out in cooperation with the Research Center Pharmaceutical Engineering (RCPE) and the Institute for Technical Informatics of the Graz University of Technology.

This introduction starts with some words about the motivation and objectives of this thesis. Adjacent I will give a short explanation of the XPS Project from which this thesis arises from. At the end of this chapter you will find an outline of the structure of this master thesis.

1.1 Motivation

Many industrial fields are confronted with the optimization of their production processes. Especially companies operating in the fast growing market of pharmaceutical, cosmetic and food industries invest plenty of time and funds in process and prototype development. Common problems in these areas occur with the need for mostly specialized devices. Such devices might be used for mixing raw materials (e.g. powders), coating pills with active and inactive fluids and materials, tablet production (pressing), capsule filling and many other things.

Over the last centuries computer aided simulations were taking up an important role in the development stage of such industrial devices. Not only that simulations are able to vastly improve the development speed of new processes, but they will give also a deeper understanding on well known methods. A very potent simulation method is the DEM.

The DEM [Cun71], [CS79] is suitable to calculate the overall behavior of arbitrary particle flows providing really accurate results. Designed to work with discrete elements it basically acts on all forces emerging from any collisions a single element undergoes. As long as all major forces from different sources, like gravity, particle collisions or wall collision,

considered, this method is indeed capable of handling a large number of concrete elements. Elements can be chunks of any kind, like single well-shaped particles, clumps of particles but also more or less of abstract type, like used by Smoothed Particle Hydrodynamics (SPH). The algorithm itself does not care about chunk sizes. So the process will work for small scale (like in powders) as good as for large ones (like rocks).

Existing commercial solutions are acting mostly on Central Processing Units (CPUs). Due to the enormous computation effort needed for the simulation of high amounts of particles, such solutions will reach their reasonable limits in the sub-million range (even on mid-sized clusters). Other solutions follow a different approach. They use a technology introduced a few years ago by NVIDIA where massive parallel processors embedded into commercially available graphic cards can be used to perform miscellaneous computation tasks. This technology is called Compute Unified Device Architecture (CUDA®). You'll find an introduction about this technology in section 2.2.

However plain particle simulations are often not enough to take account of all industrial processes. For example, a common production task in pharmaceuticals is to stream fluids into the treated powders. Therefore, more complex simulations are needed. The good news is that the DEM can be extended by considering any forces targeting the particles. So, it is even possible that the introduced forces arose even from other simulations. Such a simulation using different input sources for their calculations is called *coupled*.

The RCPE invented a solver using CUDA® within their academic software eXtended Particle System (XPS). Although this programs main business is performing particle simulations on capable graphic cards, it also supports the coupling with a Computational Fluid Dynamics (CFD) simulation provided by AVL FIRE®. An introduction about XPS is given in section 1.3. Additionally, you will find a detailed explanation about XPS in section 2.5.

1.2 Objectives

The intention behind this thesis is to extend the existing XPS functionality. The first goal is to provide a procedure to share its simulation outcome with other programs in a quick and easy-to-use way. Once provided, the technique should enable the use of external programs providing an online view and analysis of the current internal simulation states.

As a second goal of this work a new DEM solver which utilizes more than one GPU by time was developed. This method is commonly known by the term *Multi-GPU*. On the one hand, this extends the maximal number of particles usable in one single simulation run. On the other hand, the total calculation time will be decreased for a certain number of

particles. As a drawback, massive synchronization issues must be solved and will produce timing overheads.

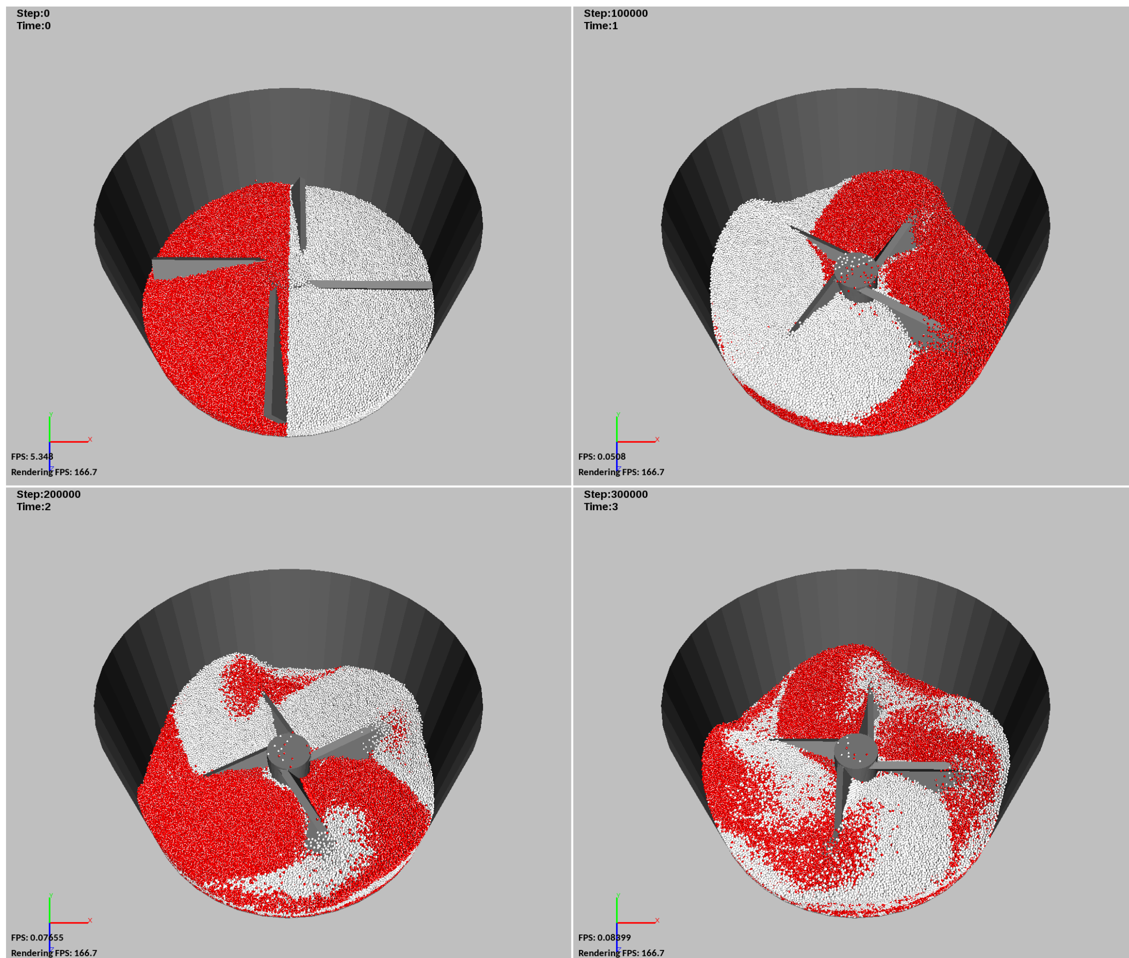

Figure 1.1: Simulation of a pharmaceutical mixing device in labor/academic scale. The blades are rotating with 40 revolutions per minute. Filled in particles are color tagged to allow a visible evaluation of the mixing quality. Upper-left is the initial state at $t = 0s$. From left to right and up to down are snapshots at $t = 1s$, $t = 2s$ and $t = 3s$, respectively. (Source: RCPE GmbH)

1.3 XPS - eXtended Particle System

In 2011 the RCPE started with the development of a software prototype named XPS within a joint research project together with some industrial partners. The program acts as a simulation framework embedding DEM based methodologies for usage with miscellaneous sized particles and varying characteristics. By utilizing CUDA® compatible

graphic cards of the newest generations, with a *compute compatibility* of 2.0 or higher, this software takes advantage of cutting edge technology. Real world geometries can be processed and may act as impenetrable walls. As shown in figure 1.1, allowing single parts of the geometries to move (e.g. translate or rotate) special devices can be simulated as well. Furthermore XPS supports coupling possibilities between a running simulation with the output of a fluid simulation, shown in figure 1.2. The fluid calculation is provided by AVL FIRE® which is a dedicated CFD software.

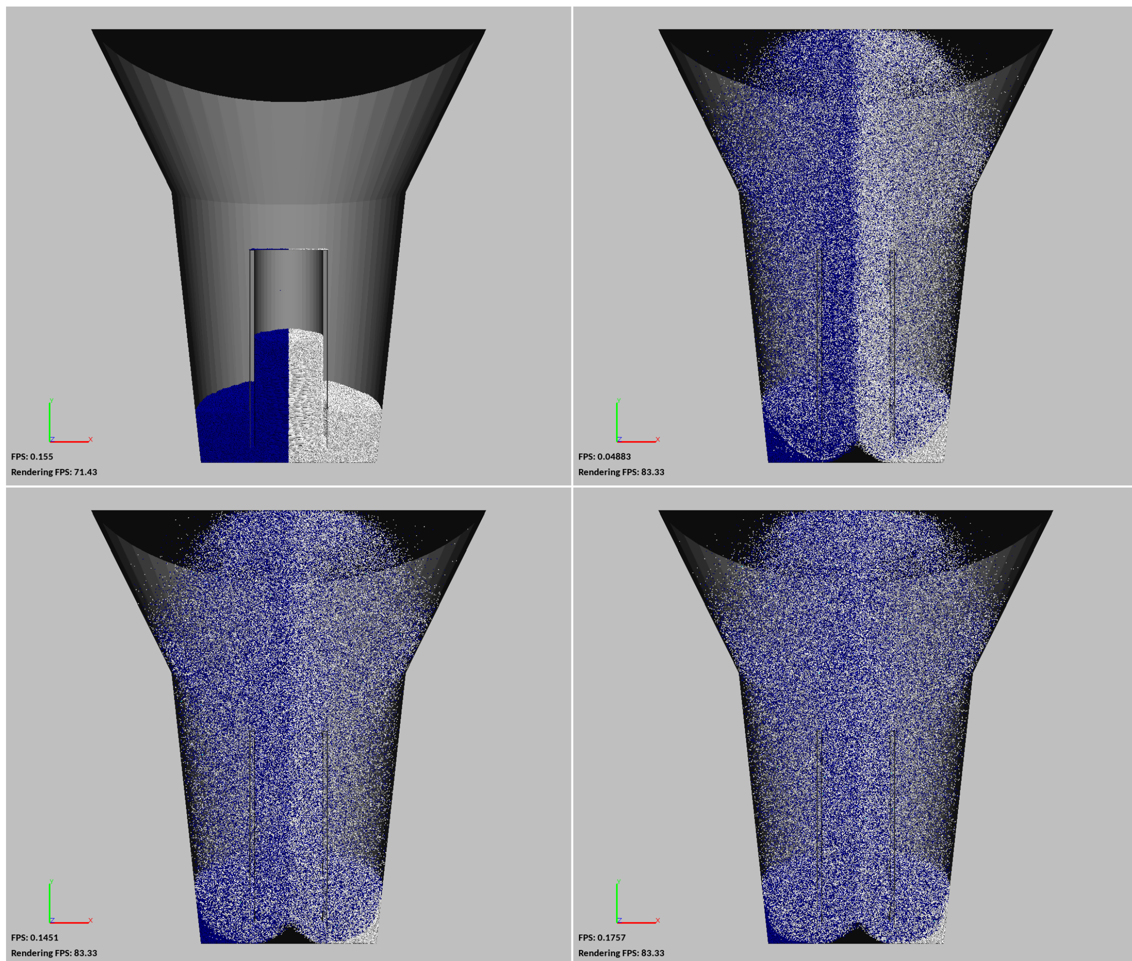

Figure 1.2: Simulation of a pharmaceutical coating device in labor/academic scale, a so-called Wurster-Coater. The particles get fluidized through the inner tube. Usually there sits a coating nozzle on the top of the tube - which is neglected here. Filled in particles are color tagged to allow a visible evaluation of the mixing quality. Upper-left is the initial state at $t = 0s$. From left to right and up to down are snapshots as time goes on. (Source: RCPE GmbH)

1.4 Outline

A short overview of the chapters and structure of this thesis is shown here. The first part is about theory in chapter 2. It shows current state of the art, explaining the DEM and CUDA®. Since the implementation part of this thesis heavily depends on them, some background about Inter-process Communication (IPC) and Program Libraries are given. A short overview about BOOST C++ libraries (BOOST) is given as well. The theory chapter ends with an introduction to the simulation framework XPS, showing the current implementation at start of this work. Especially the implemented DEM methodology is outlined.

Starting with the first objective (shared simulation data), described in chapter 3, possible designs are shown and evaluated. Afterwards, the evidently best method is nominated and a possible implementation is shown. At the end of that chapter the results of this work is presented. This is done by explaining by the usage and showing some use cases.

The design of the second objective (Multi-GPU DEM) is presented in chapter 4, giving some background basics and explaining design decisions. Afterward, the implementation is shown in chapter 5. Test cases and their results are shown, evaluated and discussed in chapter 6. Finally, discussion about future development and a conclusion are given in chapter 7.

Chapter 2

Theory

2.1 Discrete Element Method (DEM)

The *DEM* is a numerical method for computing the stresses and motions of a large quantity of chunks in a limited space. It was originally designed by Cundall in 1971 [Cun71], [CS79] for two dimensional spaces only and was adopted to the third dimension later on. The algorithmic main idea is to handle materials consisting of a set of separate discrete elements. Those elements exist in a limited volume and get handled by the calculation method individually, meaning all contacts, impact forces and displacements are calculated for each particle separately. Mostly all chunks within the simulation space are simplified as regular spherical particles, but the calculation scheme can be extended to various kinds of other shaped elements [GWKE14]. Possible shapes as currently available within XPS are shown in figure 2.10.

The calculation steps are triggered on a time-step based approach successively. All forces the chunks are exposed to get calculated in each single step. The emerging position and velocity changes are integrated by a numerical integration scheme solving Newton's equations of motion. $\vec{F_i}$ donates the total Force and $\vec{M_i}$ the total torque affecting the particle i. In addition $\vec{p_i}$ and $\vec{L_i}$ names the translational and angular momentum of particle i.

$$\begin{aligned} \vec{F_i} &= \frac{\mathrm{d}\vec{p_i}}{\mathrm{d}t} \\ \vec{M_i} &= \frac{\mathrm{d}\vec{L_i}}{\mathrm{d}t} \end{aligned} \tag{2.1}$$

The time step size depends on the set of particles, used materials and overall properties. These are for instance the size, shape, stiffness, damping or maximum amount of velocity.

A value of around $T_{step} = 10^{-5}s$ might be useful for particle sizes in the millimeter range. For bigger sized particles a working time step value might be much bigger. In any case a trade-off between accuracy and calculation speed must be taken. Assume having a value of $T_{step} = 10^{-5}s$ and processing a single step takes $0.1s$ then calculating a reasonable process time of $10s$ will take more than one day ($10^5 s$).

On initialization time all particles must be positioned in the simulation domain giving an initial velocity and rotation information if desired. It is advisable that there are no overlaps between particles in the basic arrangement. Starting from this initial condition the time-step based method calculates all kinds of forces exposed to the particles. Except from collision impacts also other influences can be considered, like gravitation or a fluid surrounding the particles.

As the number of particles increases the complexity of neighbor search used for the collision detection increases. A trivial algorithm will loop over all particles trying to detect any existing overlaps by any two particles. Although such an approach might be fine for low particle numbers (e.g. <1000), it will suffer in terms of speed for larger simulations, as it shows a time complexity of $\Theta(n^2)$. Therefore using a feasible neighbor search algorithm gets important. The approach to increase this limit as followed by XPS is shown in section 2.5.

2.2 Compute Unified Device Architecture

Introduced by *NVIDIA* in 2006, the CUDA® is a general purpose massive parallel computing architecture. It enables access to the new computational features of any modern *GPU* developed by *NVIDIA*. Thus software developers can use implementations of this architecture to solve complex computational problems in a much more efficient way than on *CPUs*. CUDA® imposes new instruction sets and a new programming model to well known computer languages. The Application Programming Interface (API) extends well known programming languages by some special functionality and new keywords. In this section only the binding to the C/C++ programming language will be shown since the concrete practical part of this thesis uses this language. Anyway nearly all assumptions on the bindings can be transferred to all other programming languages supported (e.g. Fortran, JAVA, Matlab or .NET). Since GPUs are currently subject of heavy research and development efforts, the hardware architectures and CUDA® are evolving quickly. Therefore *NVIDIA* releases new computational capabilities on a regular basis, introducing more architectural features, like more sophisticated on-device scheduling and parallelism. Table 2.1 shows available architectures and their corresponding computational capabilities.

Name	Compute capability
Tesla	1.0 1.1 1.2 1.3
Fermi	2.0 2.1
Kepler	3.0 3.2 3.5 3.7
Maxwell	5.0 5.2 5.3
Pascal	6.0 6.1
Volta	7.0

Table 2.1: Currently available CUDA®compute capabilities (as of 2016) (Source: [NVI18])

2.2.1 Programming Model

The *CUDA C Programming Guide [NVI18]* defines CUDA® as a scalable programming model. Enabling development of application software that transparently scales its parallelism with the increasing number of computation cores. Furthermore CUDA® keeps the leaning curve for already experienced C and C++ developers low. Therefore the three key abstractions (thread group hierarchies, shared memory and barrier synchronization) are integrated into the C/C++ language. Developers simply have to use newly introduced keywords to invoke CUDA® related operations.

Kernels

For the C/C++ programming language CUDA® allows the software developer to define some specialized functions. Those are called *kernels*. A kernel is defined using the **__global__** declaration specifier. When a kernel is called it gets executed **N** times in parallel by **N** different CUDA® threads. In difference to a normal C function which would only be executed once. Beside the **__global__** also **__device__** and **__host__** specifiers exist. The **__device__** keyword determines solely device sided callable functions which can be called from a kernel only. Beside that, the **__host__** keyword determines functions callable from CPU threads only.

By using a special *execution configuration syntax* (<<<...>>>) the developer can specify the number of blocks, the thread count (**N**) per block, the size of shared memory and a CUDA® stream the kernel should be executed on. For convenience only the first two parameters are mandatory. If not given the shared memory size will default to zero and the stream to be used is the default stream.

Listing 2.1 illustrates a simple kernel call. Two vectors A and B of size N are added together by N different threads. Each thread is given a unique *thread ID* that is accessible within the kernel through the built-in **threadIdx** variable. This ID is used to determine the elements a thread should work with. The output is stored into the vector C.

```
1  // The kernel: each thread executes one pair-wise addition
2  __global__ void VecAdd(float* A, float* B, float* C) {
3    // Get current index
4    int i = threadIdx.x;
5    C[i] = A[i] + B[i];
6  }
7
8  int main() {
9    // Create data arrays on GPU and prepare data
10   ...
11   // Execute the kernel, N times using one single block
12   VecAdd<<<1, N>>>(A, B, C);
13   ...
14 }
```

Listing 2.1: Simple CUDA C kernel

Thread and Block Hierarchy

In the CUDA® programming model threads are defined as independent run-able instances utilizing exact one single core within a Streaming Multiprocessor (SM, SMX). This is analogous to the programming model on the host sided CPU core. A single thread is always bounded into a so called *thread block*. Blocks can be arranged in one-, two- or three-dimensional matters. Therefore the **threadIdx** is a 3-component vector identifying the explicit position of a thread within its block. A natural way of invoking computation across different shaped structures, such as a vectors, matrix or volume is achieved by doing so.

For a one-dimensional block the thread ID is given by the x component of the **threadIdx** vector. For a two-dimensional block of size $(D_x,\ D_y)$, the thread ID of a thread with index (x, y) is given by $(x + yD_x)$ and for three dimensions $(D_x,\ D_y,\ D_z)$ by $(x + yD_x + zD_yD_x)$ respectively.

Since all threads of a block reside on the same processor core sharing the limited processor resources, the maximum number of threads per block is bound by architecture details. Currently available GPUs support up to 1024 threads per block. Anyway, a kernel can be executed by multiple equally-shaped thread blocks. Therefore the total number of executable threads equals the number of blocks multiplied by the number of threads per block. Thread blocks are organized within one-, two- or three-dimensional grids as shown in figure 2.1

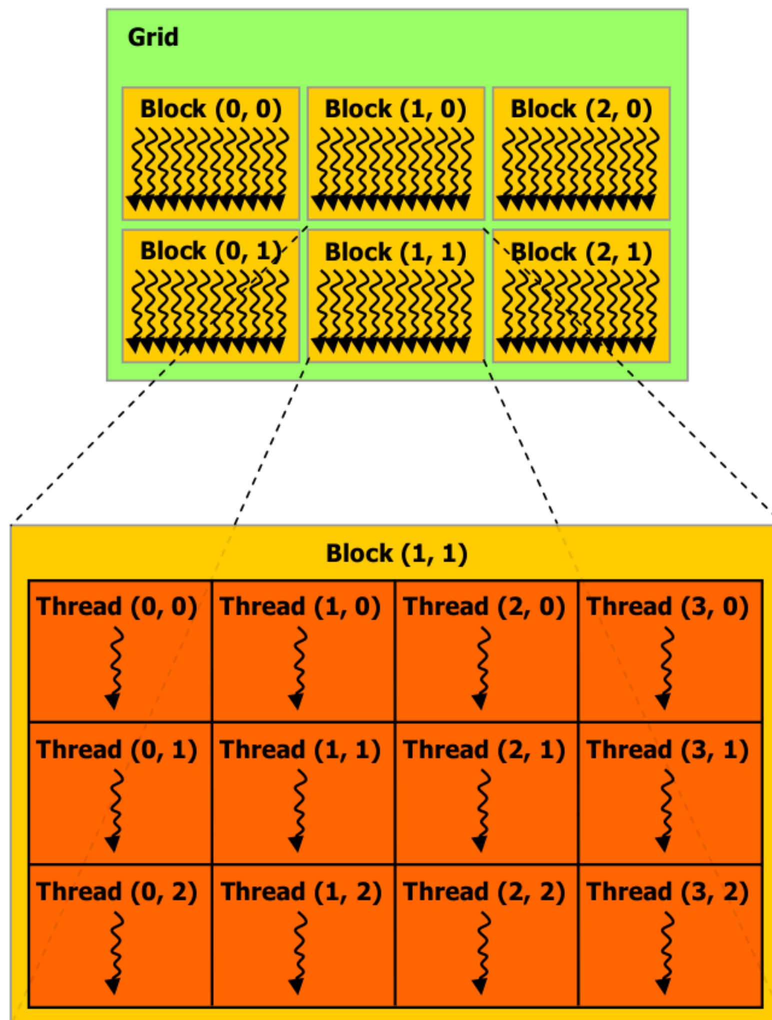

Figure 2.1: Grid of two-dimensional thread blocks organized in a two-dimensional grid (Source: [NVI18])

CUDA Thread Synchronization

Within a single thread block threads can cooperate with each other. This is done by sharing data through a special *shared memory* block. Therefore the programmer can specify explicit synchronization points within the kernel by calling the **__syncthreads()** intrinsic function. This acts as a barrier at which all threads in a block have to wait until all others have reached this point. The shared memory is expected to be of low-latency near each processor core.

Memory Hierarchy

In CUDA® different memory regions are defined which can be accessed by threads either in a read-write or a read-only manner, as shown in figure 2.2. Each thread has its own register set and some local memory with private access only. Every thread block shares a general purpose memory along all of his threads. This is the *shared memory* region. On the most recent architecture in 2015, the implementation year of this work, this region had a size of 96kB per block. This was compute capability 5.x (Maxwell).

The *global memory* region is the biggest memory block on the GPU. This region is globally accessible from every thread belonging to any block. Recent graphics cards, e.g. the Titan X, have global memory resources of 12GB or more.

Beside the read-write-able memory also two read-only memory regions exist. Those are the *constant* and *texture* memory spaces. As the name suggests, constant data should be placed into the constant space. Those might be pointers to huge data structures residing in the global space or various steady configuration variables. Through the *texture cache* read-only requests to the global memory can be accelerated. Additionally textures provide addressing calculations, e.g. transparent interpolation, are offered for arrays. Global, constant and texture regions are persistent across kernel launches by the same application.

Streams and Events

To enable more concurrency within the execution progress the CUDA® programming model introduces so called Streams. A Stream is defined as a sequence of operations that execute in the same order as they are issued, similar to a pipeline. Usually all operations issued to a CUDA® capable card are scheduled in a special Stream, known as the *NULL Stream* or *Default Stream*. The architecture allows to schedule a set of Streams to do multiple things in parallel, as long as the required resources are available. By default a single card can execute a device kernel and run up to two memory copy processes

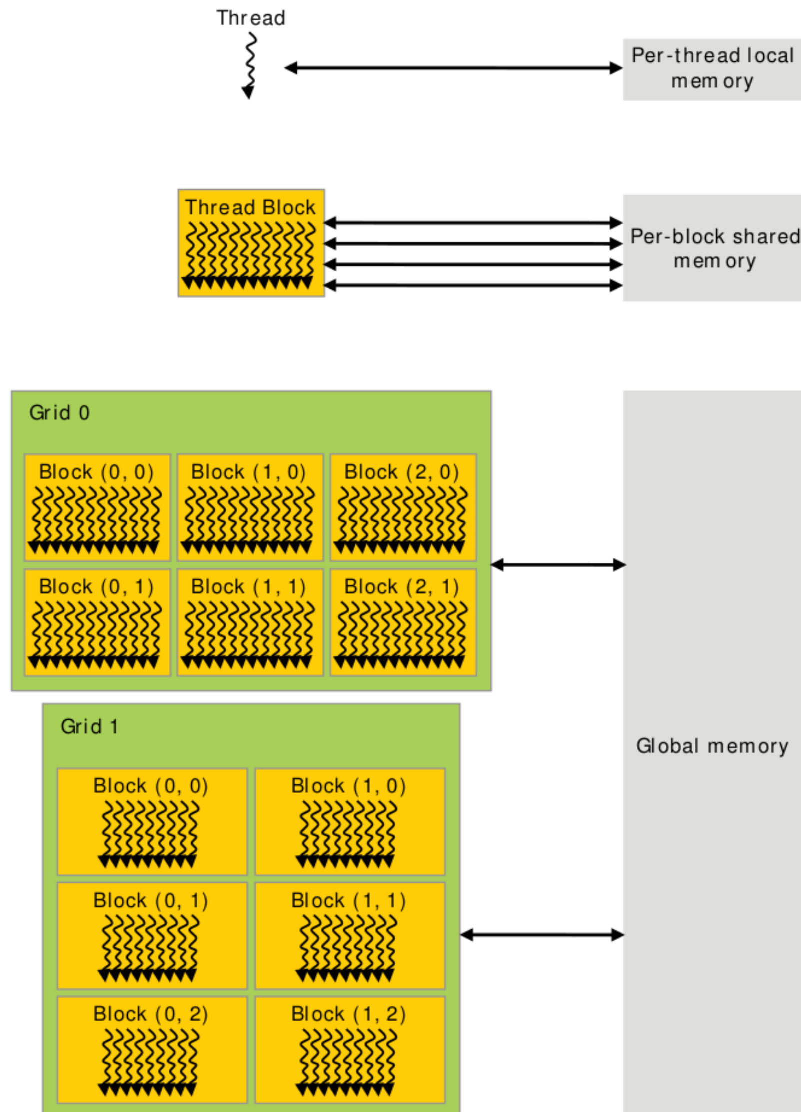

Figure 2.2: CUDA memory hierarchy (Source: [NVI18])

in parallel. One copying data from host to device and the second from device to host. However, since the introduction of *compute capability 2.0+ (FERMI)* streams may also be used to address concurrency between kernel executions, as long as there are unused resources. So, in theory it is possible to utilize a single card completely, nowadays. Listing 2.2 shows such an example. Figure 2.3 shows a possible execution order as well, as the situation would look like if stream concurrency is not used.

```
// Assume we have two host sided vectors *h_a* and *h_b* and two
// device sided once *d_a* and *d_b*. We now want to make a H2D copy from
// *h_a* to *d_a* and a concurrent D2H copy from *d_b* to *h_b*.
// Also, we will address concurrency for our two kernels.
// Therefore we use four cuda streams *s1*, *s2*, *s3* and *s4*.
// Assume there is no (!) data dependency.

// trigger H2D copy async (s1)
cudaMemcpyAsync(h_a, d_a, sizeof(h_a), cudaMemcpyHostToDevice, s1);

// trigger kernel executions async (s2)
kernel1 <<< grid, block, smem, s2 >>> ();
kernel2 <<< grid, block, smem, s3 >>> ();

// trigger H2D copy async (s4)
cudaMemcpyAsync(d_b, h_b, sizeof(d_b), cudaMemcpyDeviceToHost, s4);
```

Listing 2.2: Using Multiple CUDA Streams

As a special feature the default stream is always guaranteed to be synchronized with all other streams. Therefore, it is advisable to disclaim the use of the default stream if one wants to exploit stream concurrency. In contrast there is absolutely no implicit synchronization between all other streams. To circumvent this situation CUDA® defines some explicit synchronization methods. The first is **cudaDeviceSynchronize**. This function blocks host execution until all work in every used stream is finished, preventing the host program to issue more work to any stream. Another, not so restrictive, method is **cudaStreamSynchronize** which blocks host execution until all work in a specific stream is done. It can be said that both methods act as barriers for the host side only.

To use more convenient synchronization methods, CUDA® implements so called Events. Applications can record events to any stream, to keep track of progress within that certain stream. All events work by writing a shared memory location when all previous submitted work on the stream is done. By querying the event (**cudaEventQuery**) one can introduce specific synchronization points within streams. Furthermore, events can also be used to measure GPU execution time of work portions. Therefore, events use a high resolution in device's hardware to store a trigger time stamp. The elapsed time between the triggering of two events can be elected through by the use of **cudaEventElapsedTime**. [NVI18], [Neu13], [Wil13], [Ren11], [Har12]

2.2.2 Hardware Implementation

In general a CUDA® capable GPU is built around a certain number of SMs. A SM uses the little-endian representation. If a kernel is called for a specific grid size, all execution blocks

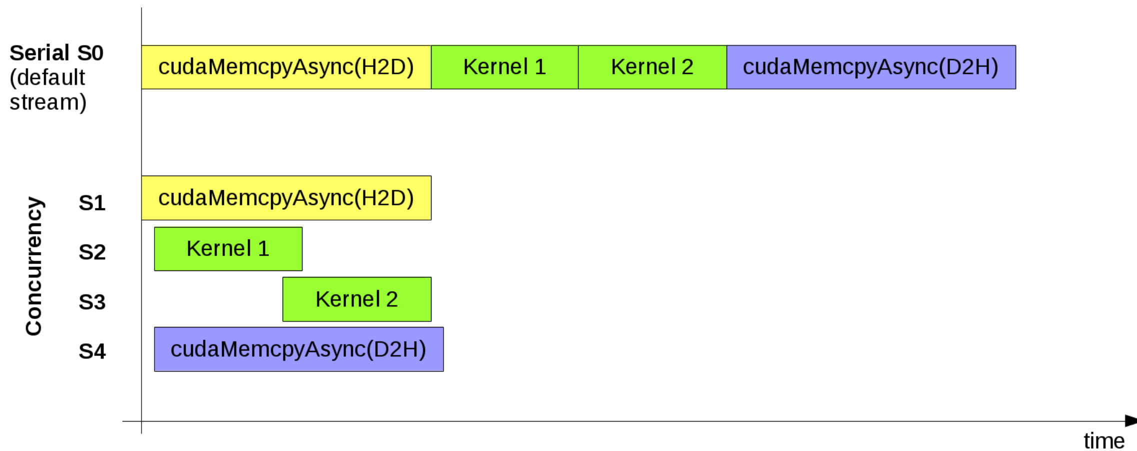

Figure 2.3: CUDA stream concurrency, possible device side execution sequence in case of sequenced execution (Serial) compared to concurrent execution (with multiple streams) (Source: RCPE GmbH)

are enumerated and distributed to the multiprocessors with free execution capacity. All threads of a thread block execute concurrently on the same multiprocessor. Also, multiple thread blocks can execute on the same multiprocessor in parallel. Once thread blocks finish work new ones will be launched on the abandoned processors. Therefore a GPU containing more SMs will often execute a program faster than one having less computing units. Figure 2.4 shows how a multithreaded program is split into a set of thread-blocks and distributed to all available SMs.

Single-Instruction Multiple-Thread (SIMT)

Since a multiprocessor is designed to run hundreds of threads in parallel it is built on a unique architecture called Single-Instruction Multiple-Thread. Every SM creates, manages, schedules and executes threads in groups of 32. This is called a thread warp, which is the smallest execution unit in CUDA®. All threads within a warp start together at the same program address and they are constrained to execute the same low level operation at the same time. But each thread has its own instruction address counter and register state. Therefore they are free to branch and execute independently. As a warp executes the same instruction for all of its threads, full efficiency is realized only if all of its 32 threads agree on their execution path. Therefore branching inside single thread warps will cause stalls which will decrease computation speed directly. [Kre11], [NVI18], [Wil13]

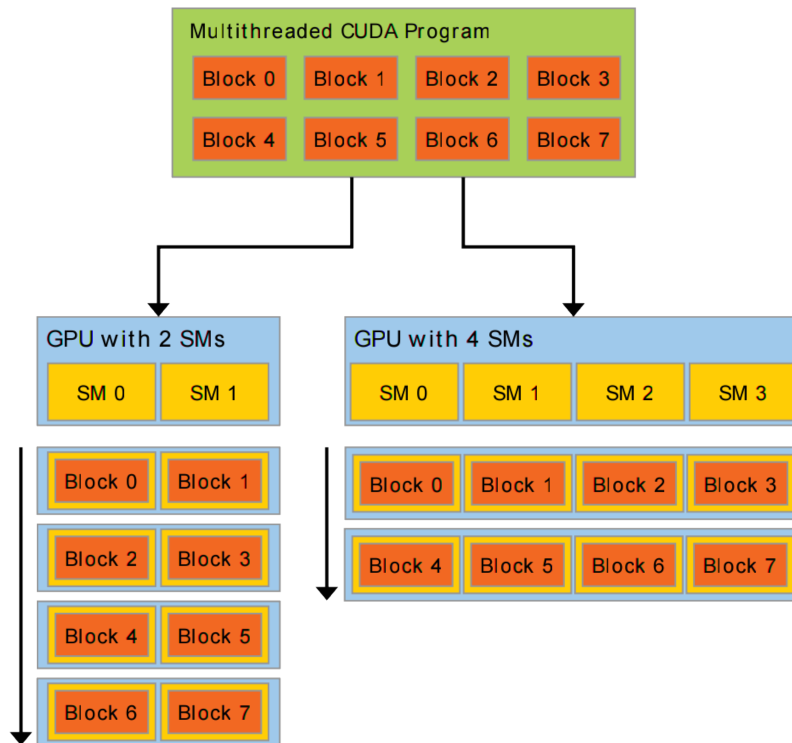

Figure 2.4: Dynamic thread-block assignment to available SMs (Source: [NVI18])

2.2.3 Multi-GPU

CUDA® has always supported multiple GPUs. Systems with several GPUs generally contain motherboards with two or more GPUs plugged in. There are also special GPUs, implementing a PCI Express bridge chip featuring multiple GPUs on the same board (e.g. GeForce GTX 690, GeForce Titan Z). Furthermore, a multi-GPU experience might also be exploited by having concrete computer nodes containing its own GPUs.

Single-Threaded Multi-GPU (ST-MGPU)

Till the release of CUDA® 4.0 each GPU has to be controlled by a separate CPU thread. For workloads that require a lot of CPU power, this was never very onerous. However, for programs that do not require multiple CPU threads at all this fact adds additional, mostly unwanted, IPC duties. So, by using a modern CUDA® runtime, a single thread can drive multiple GPUs. Therefore, it has to keep track of the currently used GPU and needs to switch between the GPUs as needed. This is done by calling the **cudaSetDevice** function. The most important advantage of this method is that one does not need to keep an eye on pitfalls which can occur in a multi-threaded/process environment. This includes locking and race condition issues. As a drawback it has to be stated that managing multiple GPUs

in one single thread will cause the program code to get much more complicated.

Multi-Threaded Multi-GPU (MT-MGPU)

As already said, workloads that require a lot of CPU power can also utilize a set of GPUs in separate CPU threads. So, the full power of modern multi-core processors can be unlocked through multi-threading. As for the single threaded approach each thread has to be sure that its GPU is currently set as active device. That requires some inter-thread locking methodologies. Finally spoken, multi-threading just for driving multiple GPUs might add tremendous locking and inter-thread communication issues, but it might simplify the code base and reduce controlling overheads.

Multi-Process Multi-GPU (MP-MGPU)

As for the multithreaded approach it is as well possible to use multiple GPUs within a program splitting its work on several CPU processes. Like for normal multi-process programs one has to implement interprocess communication, such as Massage-Passing Interface (MPI). Beyond that, a multi-process-multi-GPU implementation will not differ that much from the multithreaded approach. At least every CPU process might utilize its assigned GPUs in a multithreaded way as well. So, the biggest advantage of porting multi GPU programs to many computing nodes is that it will give one a tremendous boost in speed and maximum problem size. [NVI18], [Wil13]

2.2.4 Thrust

Thrust, developed by NVIDIA and shipped within its parallel execution toolkit, is a remarkable and overall useful extension to the CUDA® functionality. It provides a C++ template library based on the Standard Template Library (STL), which is indeed fully interoperable with CUDA® C. By using this high level interface it only takes a minimum of effort to implement a maximum performance parallel application. Basically Thrust defines two template based container object types. The first one *(thrust::host_vector<T>)* is host side based and the other one resides on the device. Except from their GPU binding both of them are truly comparable to their STL relative *(std::vector<T>)*. The usage of this vectors hides any nested calls to low level functions like *cudaMalloc*, *cudaFree* and *cudaMemcpy* behind their object oriented API.

Additionally Thrust brings in generic and fast algorithms for data manipulation of parallel primitives, like scan, sort and reduce. Since all algorithms work for both provided vector types, they can be composed together and extended by own ones implementing

concise and readable source code. The data exchange procedure is simplified by the use of class operators. Listing 2.3 shows how Thrust simplifies CPU and GPU data utilization seamlessly.

In earlier CUDA® versions Thrust calls always used the default stream only. That leads to some performance issues if Thrust based algorithms were used in implementations driving multiple execution streams. In newer versions, this problem is solved. Now, there is an implementation using **cuda::par.on(stream)** do support multiple streams [NVI15].

```
// initialize random values on host
thrust::host_vector<int> h_vec (1000);
thrust::generate(h_vec.begin(), h_vec.end(), rand);

// copy values to device
thrust::device_vector<int> d_vec = h_vec;

// compute sum on host
int h_sum = thrust::reduce(h_vec.begin(), h_vec.end());

// find extremas on device
int d_min = thrust::min_element(d_vec.begin(), d_vec.end());
int d_max = thrust::max_element(d_vec.begin(), d_vec.end());
```

Listing 2.3: Simple Thrust Example

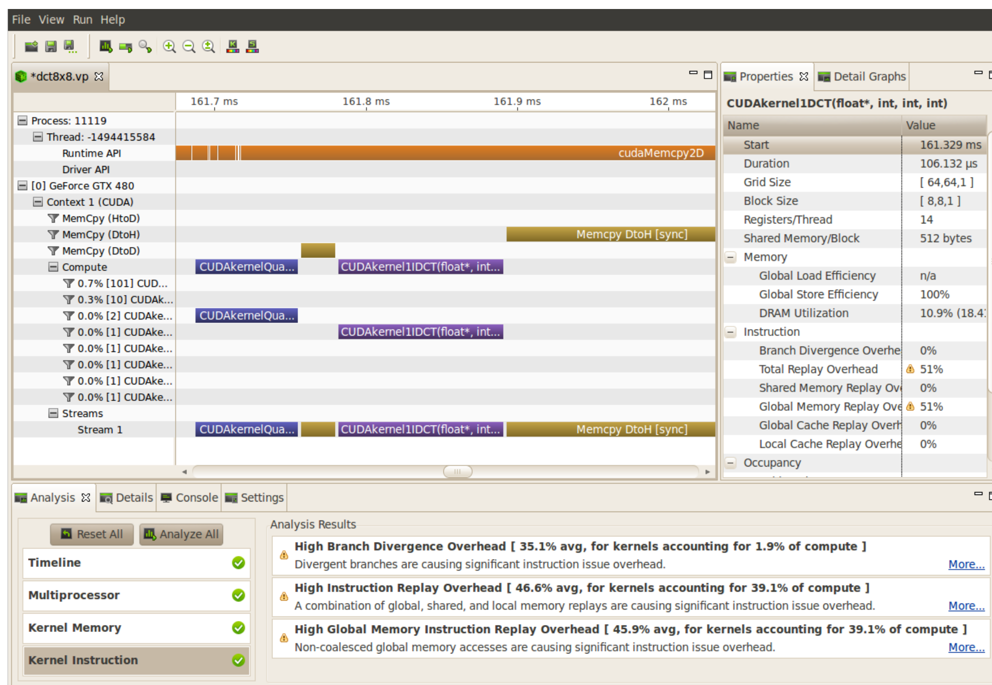

Figure 2.5: Example views of the NVIDIA Visual Profiler (Source: NVIDIA)

2.2.5 CUDA Profiling Tools

Profiling is of major interest when it comes to performance and runtime optimization. To ease the process of profiling CUDA® code the toolkit comes with a easy-to-use profiling tool, called *NVIDIA Visual Profiler (nvvp).* This tool is a cross-platform performance profiling tool. It delivers vital feedback for optimizing CUDA C/C++ applications to the developers and supports guided application analysis. A sample view of the tool is given in figure 2.5.

2.3 Inter-process Communication (IPC)

From time to time concurrent running processes need to communicate and exchange data with each other. A well known example is the communication of a web-server with a web-browser. In that case the data transfer is usually handled through sockets using dedicated routes within a network. Such connections are two-way based, because the client explicitly requests resources that are delivered by the server afterwards. Beside that also one-way communication schemes exist, like a pipe or FIFO (short for First-In-First-Out). However, every time two processes communicate with each other this is called IPC.

2.3.1 Shared Memory

A pretty simple IPC method is shared memory. In principal it allows two or more processes to access the same memory region, as shown in figure 2.6. Shared memory is a very fast way to exchange data. In a nutshell, the operating system will only map the same Random access memory (RAM) pages into each participating process. This makes access to it as fast as for a process' non-shared memory, and it does not require any additional system calls to the operating system kernel. Therefore, any modification done by any thread will be seen by all others immediately. Note, that this avoids unnecessary data copies which might be very resource intensive.

To use shared memory one process has to allocate a shared memory segment, using a dedicated system call. Once this is done all other processes desired to use the segment have to attach to it. This is done by executing an other system call. After usage of the shared memory, for instance all participating processes finished their work, one of them has to deallocate the segment.

Since shared memory resides in global RAM and all processes gain direct access to it there is no implicit synchronization logic. This means that multiple threads should not write to the same location at the same time. Doing so will lead to race conditions

causing unexpected and undetermined program behavior. To circumvent this situation the programmer has to take care of inter-process synchronization by himself.

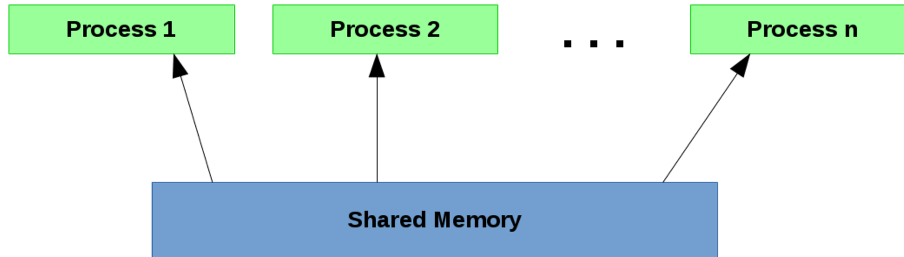

Figure 2.6: Multiple processes map to same shared memory region (Source: RCPE GmbH)

2.3.2 Resource Locking

Race Condition

Whenever multiple concurrent threads or processes use a shared resource, like dedicated data structures in a shared memory segment or shared devices, it is required to coordinate their accesses to it. Neglecting upcoming synchronization will lead to undefined or undetermined behavior. For example, suppose that two concurrent running threads will write to the same memory location, at the same time, and read from that location afterwards. Imagine the first thread writes 'A' and the second one writes 'B' to the memory. Now we have two cases. In the first one each of the two threads will not interrupt each other. So, the first thread will write and read 'A' and the second one does the same with 'B'. In the other case, one of the threads will interrupt the other one right after the write command. The interrupting thread will now override the data and both will read the last written value. Figure 2.7 illustrates the mentioned case.

Deadlock

A deadlock determines a situation where two or more actors wait for the other to finish. To show how simple a deadlock situation can occur, consider two concurrent processes. Both where developed independently, but they are designed to fulfill the same task, printing pressed keyboard letters directly to a printer. They need to take a lock on each of the two resources. Now imagine, that one tries to take the keyboard lock first and the other attempts to take the printer lock at first. This will work as long as not both processes held the locks at the same time. If, at any point in time, one process holds the lock for the

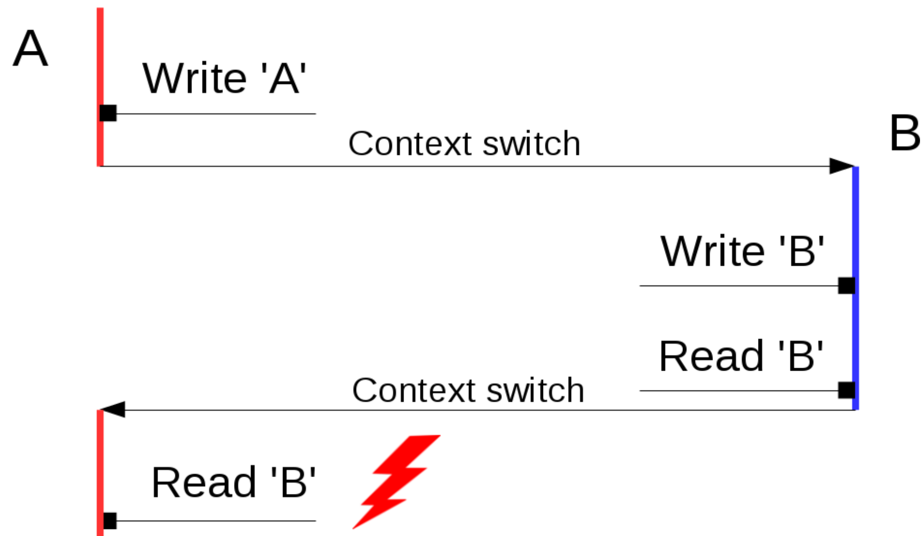

Figure 2.7: Race condition, two threads using same memory resource (Source: RCPE GmbH)

keyboard and the other one has already acquired the lock for the printer, both processes will stall on waiting for the other one to release his locked resource.

SpinLock

A very primitive synchronization method is the SpinLock. The thread, that likes to acquire the lock, simply waits in a loop ("spin") until the lock is free, as shown in listing 2.4. Since the thread is doing nothing useful while waiting for the resource to become available, this is a kind of "busy waiting".

```
1  // active wait until resource is free
2 while( ! spinLock.free() );
3
4 // acquire a lock to the resource
5 spinLock.acquire();
```

Listing 2.4: Thread try to acquire a SpinLock

Mutex

A Mutex is another method to resource locking. Although it is used like a SpinLock it does not introduce the problem of "busy waiting". In current implementations, regarding to hardware support, whenever a thread tries to acquire a locked Mutex it is sent to sleep. While a thread is sleeping it is ignored by the scheduler. Once the Mutex is released the

sleeping thread gets woken up by the scheduler again, and can try gain now exclusive access to the shared resource again.

Condition Variable

A Condition Variable is a synchronization primitive that enable threads or processes to wait until a certain condition occurs. Since a naive approach by just keep checking a value, will lead to race conditions a Condition Variable needs to be used together with a lock, protecting the shared resources. A well known example to this is the Producer-Consumer problem. Here the Producer creates data and puts it into a FIFO as long as the FIFO is not full and needs to be blocked while the FIFO is full. On the other hand, the Consumer takes elements from the FIFO as long as the FIFO is not empty and needs to be blocked while the FIFO stays empty. Listing 2.5 shows a basic solution for this problem, using a FIFO, two Condition Variables and a Mutex.

```
1  Fifo fifo;
2  Mutex lock;
3  ConditionVariable empty;
4  ConditionVariable full;
5
6  producer {
7    while(true) {
8      lock.acquire(); // acquire lock for initial check
9      // if fifo is full, wait until elements got removed
10     while ( fifo.full() )
11       wait(full, lock);
12     produce(fifo); // produce something
13     notifyAll(empty); // notify consumer
14   lock.release(); // release the lock
15 }
16
17 consumer {
18   while(true) {
19     lock.acquire(); // acquire lock for initial check
20     // if fifo is empty, wait until elements got inserted
21     while ( fifo.empty() )
22       wait(empty, lock);
23   consume(fifo); // consume
24     notifyAll(full); // notify producer
25     lock.release(); // release the lock
26 }
```

Listing 2.5: The producer-consumer problem solution with two ConditionVariables and one Mutex

Barrier

A Barrier is a synchronization primitive used in parallel computing. It is used when an algorithm, which is executed by multiple threads or process, needs to define a certain point of synchronization that needs to be reached by all actors before continuing. So, a Barrier in source code means that all threads or processes have to stop at this point and send to sleep by the scheduler. If all actors have successfully reached the synchronization point, they get waken up again and will continue. Therefore, a Barrier holds a counting variable, that is increased as long the number of waiting members is not reached. When the count reaches the given number of members to be wait for, all of them are released again and the count is set to zero again.

2.3.3 Message-Passing Interface (MPI)

MPI donates a portable and standardized programming model for IPC. Providing an easy to understand API it is used widely in computer science for parallel computing tasks. Having convenient C and Fortran bindings, it brings generalized process synchronization and data exchange methods to many vendor platforms, without the need of significant changes in the underlying communication and system software. [oT15] [MS01] [TB14]

2.4 Program Libraries

Within an enlarged meaning the term program library may involve different types of libraries, like source files, macros, object- or bytecode and tools. In terms of software development, a program library is the collection of non-volatile resources. This means subprograms and -routines in many cases. Those software parts often implement specialized functionality like algorithm and specific methods. Once provided within a library object it makes them reusable in widen field of different software solutions.

Depending on the operating system and the development solution a program library might be delivered in several ways. It could consist of either a single file or a directory containing multiple library files. An other method is do collect several libraries of different types inside on single file. This can be called as a library database. Last but not least the term program library does not need to explicitly name a library file. In that case some reusable components might be saved as single executable files providing standard routines on data manipulation. An example for a Linux shell processing some input data through a set of independent executables is shown in listing 2.6.

```
1 #!/bin/sh
2 # split string into tokens separated by ' ', take the second field and
3 # remove any exclamation marks
4 $> echo 'Hello !WORLD!!' | cut -d' ' -f2 | sed -e 's/\!*//g'
5 WORLD
```

Listing 2.6: Use executables as algorithm resources in a (linux) shell

For the 'in source' usage within a specific software tool a concrete connection to the providing library have to exist. The linkage and binding is resolved by specialized tools called linker or binder. Library resources can be jump addresses or other routine calls and get resolved into fixed or relocatable addresses. Conceptually two different approaches are possible as shown in figure 2.8.

2.4.1 Static Linking

The method of static linking is performed during the creation process of an object file, a library file or an executable. Sometimes this it also referred as *early binding*. Within static linking all of the modules needed by the compilation are copied into the created object. This is called a static build. As an advantage a statically linked executable might not need any other resources. Otherwise the resulting object size will be bigger than by using dynamic linking. Furthermore the linking has to be redone when any of the modules are recompiled.

2.4.2 Dynamic Linking

In contrast to static linking the process of dynamic linking does not copy the needed resources into the output file. Rather, only a reference to the containing library is added to the application file. This might include the full file path or just a simple file name, as shown in listing 2.7. On execution time this dependency is resolved by an dynamic loader trying to find and link the libraries resources. By using dynamic linking executable sizes will shrink and redoing the link process does not have to be done so often. Otherwise this method introduces some overhead while run time. [Jon08]

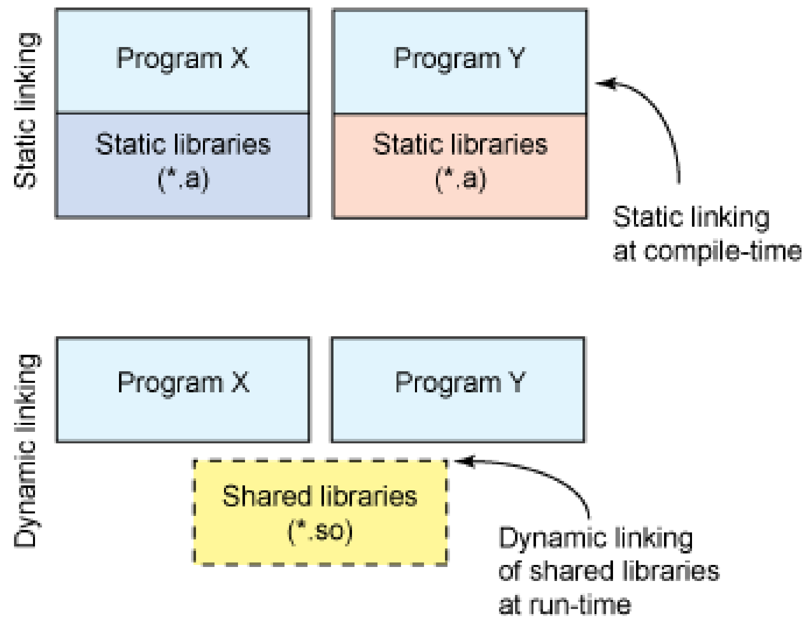

Figure 2.8: Static vs. dynamic linking for the Linux operating system. (Source: [Jon08])

```
1 #!/bin/sh
2 $> ldd ./a.out
3   linux-vdso.so.1 (0x00007ffd2b11b000)
4   libstdc++.so.6 => /lib64/libstdc++.so.6 (0x00007fcb73ff2000)
5   libm.so.6 => /lib64/libm.so.6 (0x00007fcb73cf0000)
6   libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007fcb73ad8000)
7   libc.so.6 => /lib64/libc.so.6 (0x00007fcb73717000)
8   /lib64/ld-linux-x86-64.so.2 (0x000055e16f7f6000)
```

Listing 2.7: Dynamic linkage of a simple "Hello World"-executeable (64bit linux system)

2.4.3 The BOOST C++ Libraries

BOOST provides a set of libraries for the C++ programming language. With first version released in 1999 BOOST currently offers more than eighty individual libraries. The main goal of all included sub-libraries is a gain in efficiency for the C++ development. Therefore it brings independence of the used operating system by building up an abstraction layer. Additionally it supplies well known and approved solutions for recurring tasks and introduces quiet a lot of useful data structures. Most of its libraries are header only based but a few of them provide independent library objects as well. Main parts consist of inline functions, templates and macros giving a huge impact on efficiency and flexibility.

Since BOOST is implemented on a high standard and its founders closeness to the C++ standards committee, several BOOST libraries already found their way into modern C++ standards. The following BOOST libraries are used heavily within this thesis execution.

Algorithm A collection of general purpose algorithms

Container Implements several well-known containers

Interprocess Simplifies the use of common interprocess communication and synchronization mechanisms

Smart Pointer A collection of general purpose smart pointers

Thread Enables the use of multiple threads of execution with shared data in portable C++ code

For a full list of libraries provided by BOOST see the BOOST homepage at `http://www.boost.org/`.

2.5 XPS - eXtended Particle System

2.5.1 Overview

XPS (figure 2.9) is a software developed at the RCPE together with some industrial partners, since 2011. It embeds DEM based methodologies for usage with miscellaneous sized particles and varying characteristics. The program utilizes CUDA® compatible graphic cards of the newest generation and is capable of performing large scale simulations with more then fifty million particles in reasonable time. E.g. a simulation with 50 million particles will proceed about 0.5 steps/s on a single modern GPU (e.g. a Titan X).

Figure 2.9: The XPS Logo (Source: RCPE GmbH)

Although having huge particle numbers is a nice attribute at all, the framework provides many more features. Real world geometries supplied via files encoded in the STereoLithography (STL) format can be processed and may act as impenetrable walls. Allowing single parts of the geometries to move (e.g. translate or rotate) special devices can be simulated as well. Figure 1.1 shows such an device within the simulation. As another remarkable feature XPS provides a method to process a so called coupled simulation. In this case the plain particle simulation can be coupled with the output of a fluid simulation. This calculation is provided by AVL FIRE® which is a dedicated CFD software. On the one hand XPS uses the provided fluid information to let additional forces act on the particles (e.g. drag- and pressure-forces). On the other hand some material and simulation characteristics of the solids acting as source terms in the fluid phase. Figure 1.2 shows an example application. Additionally a spraying module is available which enables the simulation of so called coating processes, which are often used in combination with one of the above methods.

2.5.2 Current Implementation

Acting as a framework the program is build up on a modular basis. The current implementation of XPS is split into a set of more or less independent parts, making it easily extendable. Core components are split into a small set of libraries, making their usage as independent as possible. A generalized configuration methodology, based on command-line parameters and input files, enables an easy and flexible way of parameterization. The common management kernel binds a specific, but exchangeable, simulation component to the core components. Currently, specialized simulator kernels have to be provided by dynamically linked libraries, but a solution using at run-time loadable simulator objects can be introduced with minor effort. This makes XPS working as a plug-in system.

By now, XPS provides a DEM as well as a SPH simulation kernel, which are under massive development. The DEM simulation kernel is capable to work on a set of different shaped particles, as shown in figure 2.10. Those particles can be modified and parametrized as well, by using shapes and materials from an editable databases. Via a certain plug-in a connection to a host-sided CFD simulation, provided by AVL FIRE® can be established.

For post-processing reasons all simulation output is provided via result files which can be written at dedicated time steps. An integrated live-viewer makes it possible to take a look at the current state of the simulation. Furthermore, a post viewing application is available which makes it possible to view and analyze existing simulation outcome.

The next section gives an introduction to the basic DEM methodology as implemented. Note, that special models, like SPH, might differ from them by a little.

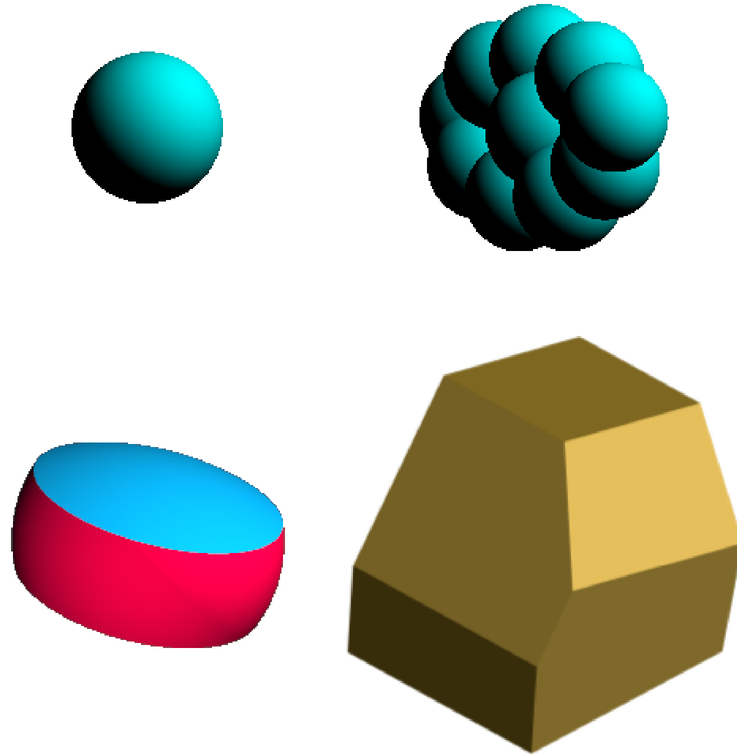

Figure 2.10: Particle shapes available in XPS. From left to right and top to down single-sphere, multi-sphere, true-shape tablets and polyhedrons are shown. (Source: RCPE GmbH)

2.5.3 Implemented DEM Algorithmic

In section 2.1 the DEM was already outlined. When it comes to huge particle numbers it has been shown that a trivial solution suffers in terms of speed because of algorithmic complexity. Therefore special attention has to be given especially on the neighbor search part. In XPS algorithmic the DEM workload is subdivided into successive phases. A preparation phase is done to determine regional relationships between all particles. Afterwards the collision phase calculates all forces exposed to each single chunk. Finally the integration phase sums up all forces and calculates the local displacement and new particle velocities and rotation.

Preparation Phase

Although, the DEM itself is a grid-less-method logical-grids (mostly uniformed) are used for detecting regional relationships between the chunks. A necessary precondition for these

logical grids is that their cell edge sizes exceed the maximal elongation of all particles inside the simulation domain in each direction. It has to be said that in an optimal case particle sizes should not diverge intensely. Figure 2.11 shows such a uniformed grid containing some spherical particles for a two dimensional world. The particles use the grid to gain information of their direct neighbors. In this figure the blue balloon detects those other balloons marked with the green check as direct neighbors. On the other hand, the one having marked with the red cross is not detected as a neighbor.

Figure 2.11: Spherical objects sorted into a uniformed grid (Source: [Kar12])

As a precondition a certain set of vectors is needed for the preparation step:

- **hashes** - holding a scalar value determining cell affiliation; size is equal to the number of particles in grid domain
- **indices** - holding particle indices and linking them to their belonging hash values; size is equal to the number of particles in the grid domain
- **cellstart** - determining the starting indexes of sorted hashes and particle indices in the above vectors; size is equal to the number of cells of the grid

The grid cells of the uniformed logical grid are subsequently indexed in each dimension. For a two dimensional grid this would be {x, y} and {x, y, z} for a three dimensional world, respectively. Based on that indexing scheme and giving the maximum number of

cells in each direction a hash value can be calculated via

$$
\begin{aligned}
\text{cPos}.xyz &= \left\lfloor \frac{\text{pos}.xyz - \text{worldOrigin}.xyz}{\text{cellSize}.xyz} \right\rfloor \\
\text{hash}_{3\text{d}} &= \text{cPos}.z \cdot \text{gridSize}.y \cdot \text{gridSize}.x + \text{cPos}.y \cdot \text{gridSize}.x + \text{cPos}.x
\end{aligned}
\tag{2.2}
$$

Note that the hash value is unique for each grid cell and leads to a linear indexing scheme of all cells. We will call this the "cellid". For each particle the corresponding *cellid* is calculated and written into the *hash* vector at array index particle id. Furthermore the *indices* array is filled up by an linear sequence over the particle ids. The *cellstart* vector needs to be initialized with a value "InvalidHash" which is bigger than the maximum value for *cellid*. Afterwards a key-value sort is done over the *hash* and *indices* vectors. The values in the *hash* are the keys and the indices as values. Now having the hash values in an ascending order the start indices for the cells can be determined easily. Note the found indices in *cellstart* correspond to the changes of values in the *hashes* array. Figure 2.12 shows the described actions. Now having the cellstart filled up each particle can easily determine all particles in their own and their neighbor cells. For two dimensional domains this will give a total of nine cells or twenty-seven for three dimensions, respectively. Although the described method of logical grids is commonly used there are other ways to retrieve regional relationships. For instance the usage of a method called Bounding Volume Hierarchy (BVH) makes the algorithm independent from particle size divergences. Possible implementations are described by Kureck ([Kur16]) or Karras [Kar12].

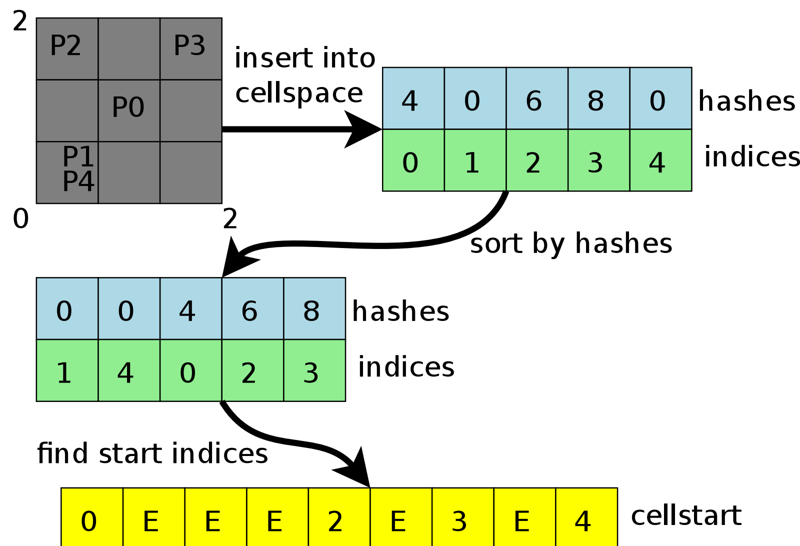

Figure 2.12: DEM Preparation phase: From particles in cellspace to *cellstart* vector (Source: RCPE GmbH)

At last it has to be said that the preparation step is not fundamentally needed for the force calculation itself, rather it is done in order to minimize the overhead for neighbor detection of all particles. Basically it minimizes the run-time complexity for the neighbor search from $\Theta(n^2)$ to $\Theta(k \cdot n)$, with $k \ll n$.

Collision Phase

Within the collision phase each particle computes the total amount of force and torque affecting it separately. Based on information of the preparation phase the collision kernels are called for all possible collisions, like particle-particle and particle-walls collisions. From an algorithmic view, a single thread is running for each particle on the utilized GPU. As outlined in figure 2.13 for the particle-particle collision, each thread first determines its associated clump and evaluates its cell position, as given from the preparation phase. In the three-dimensional simulation world each particle now checks its own and the 26 neighbor cells. Given the current cell identification as a three component index x, y, z, the 27 cells to check can be determined as shown in equation 2.3.

$$\begin{aligned} x + x_i,\ x_i &\in [\text{-}1, 1] \\ y + y_i,\ y_i &\in [\text{-}1, 1] \\ z + z_i,\ z_i &\in [\text{-}1, 1] \end{aligned} \tag{2.3}$$

Next, the cell under process has to be checked if it is outside the world. If so it will be not processed and the algorithm proceeds with the next one. Since all clump are given through a sorted vector, based on the cell indices, the first particle in the cell is indexed by information from the *cellstart* vector. If this index is invalid ("InvalidHash") the cell does not contain any particles and the process continues with the next cell. A valid index will now point to the first particles in the examined cell. Now the algorithm iterates through all sequenced particles until the hash value, which determines the corresponding cell, changes or it reaches the end of the vector, which is of length *numParticles*. Because the cell containing the current particle, given by $x_i = y_i = z_i = 0$, is checked as well, this case has to also be proven. This prevents the force calculation of a particle with itself, as this will be certainly wrong to do. For all remaining possible collision partners, the thread now calculates the upcoming collision forces using a linear spring/damper model, as shown in figure 2.14. All resulting forces are summed up in a single three component vector, per particle. If the simulation accounts for rotation information as well, the upcoming torque has to be calculated and considered too. Therefore, a second force vector is needed.

For collisions with any given geometry the collision calculation has to be done as well.

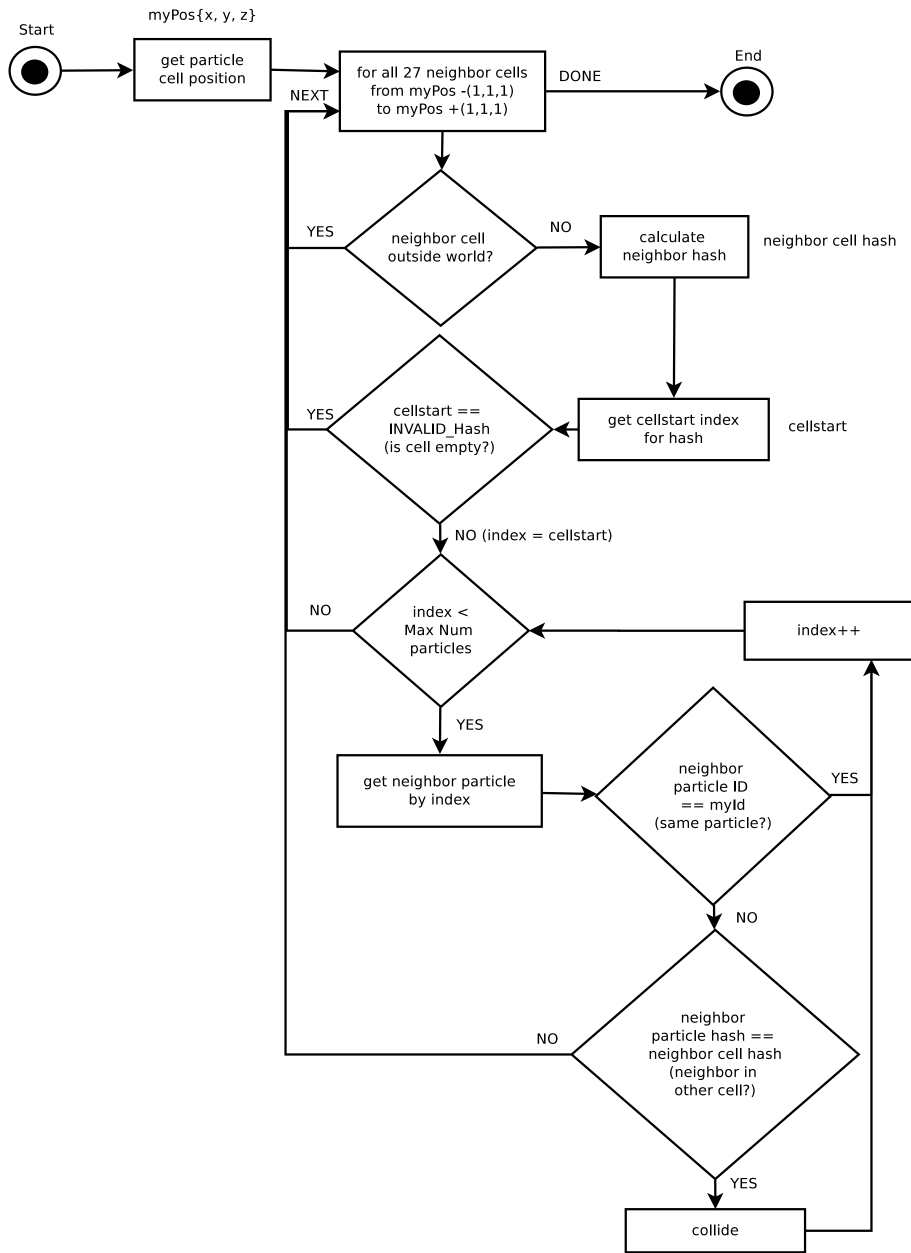

Figure 2.13: DEM Collision phase - neighbor search via uniform grid (Source: RCPE GmbH)

Because each geometry is given as a set of triangles, all triangles have to undergo the preparation phase. This associates each of them with their linked cells. Now, a unique kernel is called in which all particles check and calculate all upcoming collisions with the triangle set, similar to the particle-particle collision process.

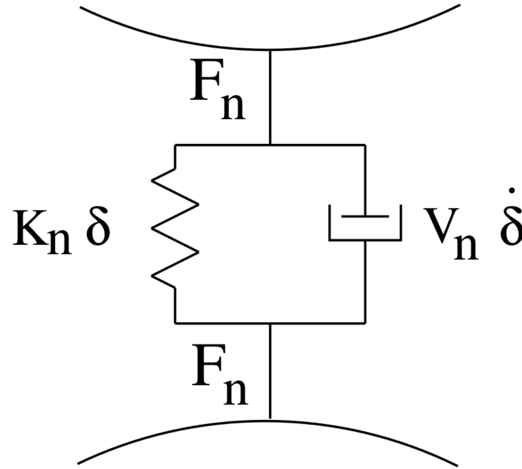

Figure 2.14: The spring/damper model that is used for particle collisions (Source: [Rad06])

Integration Phase

The last phase in XPS' DEM algorithmic is the integration phase. Given all acting forces and torques, for all particles, a single integration kernel is started for all particles. As shown in equation 2.4, by knowing the current time-step size ($\mathrm{d}t$), the current velocity ($\vec{v}$) and the angular velocity (ω), the position ($\vec{p}$) and the rotation angle ($\vec{a}$) vectors are updated at first.

$$\begin{aligned} \vec{p}_{i+1} &= \vec{p}_i + \vec{v}_i \mathrm{d}t \\ \vec{a}_{i+1} &= \vec{a}_i + \vec{\omega}_i \mathrm{d}t \end{aligned} \tag{2.4}$$

By transforming the equations given in 2.1 into terms shown in 2.5 with knowing the particle mass (m) as well as the time-step size, each thread now calculates the delta velocities for his corresponding particle. Note, that especially the equations for the rotation component are valid for spheres solely.

$$\begin{aligned} \mathrm{d}\vec{v}_i &= \frac{\vec{F}_i}{m}\mathrm{d}t \\ \mathrm{d}\vec{w}_i &= \frac{\vec{M}_i}{m}\mathrm{d}t \end{aligned} \tag{2.5}$$

If set, the influence resulting from the acting gravitational force ($\vec{G}$), as shown in equation 2.6, is calculated as well. Note, that the gravitational component can be chosen freely and

might be zero or directed to any direction.

$$\mathrm{d}\vec{g}_i = \vec{G}\mathrm{d}t \tag{2.6}$$

At the end of the integration phase the new translational and rotational velocities are summed up as shown in equation 2.7.

$$\begin{aligned} \vec{v}_{i+1} &= \vec{v}_i + \mathrm{d}\vec{v}_i + \mathrm{d}\vec{g}_i \\ \vec{\omega}_{i+1} &= \vec{\omega}_i + \mathrm{d}\vec{\omega}_i \end{aligned} \tag{2.7}$$

[Neu13], [JSRK13], [RGK10], [Rad06]

Chapter 3

Shared Simulation Data

3.1 Design

The main goal of this task is to extend the XPS framework to share all data of a running simulation with other processes in real time. Therefore, some kind of IPC needs to be implemented. This section will give an overview over design thoughts related to this task. Possible methods are outlined and evaluated for their practical application.

3.1.1 Pipes

To share the data among concurrent programs a link has to be created between their processes. An easy-to-use and very common method on doing this to create a pipe in between. By chaining the processes's standard streams, connecting the ones output stream to the others input stream, pipelining provides one-way communication only. Since it is required that the simulation processes can receive control commands and does not only provides its data, two-way communication is needed. Therefore, the pipeline method is not applicable here.

3.1.2 Streams

By using streams and sockets a two-way communication can be achieved. This technique is used when a server communicates with a client, for example. Although the use of this method might solves the problem of the two-way communication there is another fact to be considered in this design process. Every time when data needs to be send through a stream it needs to be copied on the one hand and also do not have to be altered while it is not completely send. Since the amount of data that needs to be shared might get very huge, this circumstance will block the simulation process while the data is passed through

the stream. Additionally, fast data updates might happen and will make it necessary to resend the whole data set again.

3.1.3 Shared Memory

As illustrated in section 2.3.1 shared memory is a method that can be used to share ones process data among a certain number of other programs easily. It uses dedicated memory segments containing arbitrary data structures, which are mapped into the processes virtual memory. Therefore unwanted and time-consuming memory copies can be reduced by this technique. Since all data shared via such a segment shows up in all attending processes, which might be far more than just two, at the same time, shared memory can be used for inter-process synchronization also. This makes the shared memory method most appropriate for the given task.

3.1.4 Actual Design

The principal duty in case of any shared memory implementation is on operating systems side. Therefore, any process, wanting to use shared memory, has to register his needs by the operating system. The main idea is to build some infrastructure around the low-level shared memory management and pack it into a single library to make is reusable by the different players. Ideally, the whole shared memory communication can be integrated seamlessly into the available code base. By this means a single developer should not need to know anything about shared memory at all.

At this point BOOST's library on inter-processing needs to be mentioned here. It already contains an implementation for shared memory which is almost operating system independent. Basically the usage of this library requires that a set of steps are needed to be taken. First the shared memory segment has to be created or opened, if it already exists. Then the segment needs to mapped into the process's address space. Afterwards data can be placed into the segment or read from it. A basic example is given in listing 3.1

```
int main(int argc, char *argv[])
{
  //Create a shared memory object.
  shared_memory_object shm (create_only, "MySharedMemory", read_write);
  //Set size
  shm.truncate(1000);

  //Map the whole shared memory in this process
  mapped_region region(shm, read_write);

  //Write all the memory to 1
  std::memset(region.get_address(), 1, region.get_size());
  // Do some work ...
  // cleanup if needed
  shared_memory_object::remove("MySharedMemory");
  return 0;
}
```

Listing 3.1: Basic usage of BOOST shared memory implementation

3.2 Implementation

Thoughts on the design of this task and possible methods for realization where already discussed in section 3.1. Based on that, this section will describe what the implementation actually looks like.

3.2.1 Define data to be shared

At first all data that needs to be shared has to be defined. Therefore a data structure has to be created holding all simulation data and extend it with necessary metadata as shown in table 3.2.1.

Type	Description
Particle data (dynamic)	position, velocity and rotation information
Particle data (static)	identification, shape and material
Geometry data (dynamic)	per obstacle STL triangle set
Geometry data (static)	simulation world, boundaries and obstacle transformation description
Time step data	time step timings, like step size, start and end times
Configuration	general simulation settings, like gravity

Table 3.1: Description of data that needs to be shared between simulation and any observing process

From a single particle view the current state information like position, velocity and rotation is needed on the one hand. On the other hand static information, as that is the identification number, shape and material characteristics is needed additionally. For definition of the simulation world, characteristics information about the bounding box is needed. Furthermore, the wrapping triangle set described by a STL geometry and information about geometry movement is needed for every solid obstacle within the simulation, as well. Last but not least simulation configuration and dynamic data as time step information and settings are shared too.

In the current implementation all of this data is collected within a single object instance called *XPSConfig*. As a single point of information it implements the *Singleton* design pattern. It is used within the whole program and therefore it is already implemented in a thread-safe way.

3.2.2 The data sharing infrastructure

Making the data sharing method highly integrable is one if the most important goals within this task. Therefore, an infrastructure is created that gives access to the data regions in a seamless way. By providing a self-contained program library the implementation effort is lowered on developers side. An easy-to-use access pattern, through program options or command line arguments, allows any user to enable the sharing option on program start.

Wrapping the general implementation

As outlined in section 3.1.4 the BOOST library on inter-processing is used. Therefore, a new infrastructure, encapsulating this library is created, giving access to the shared memory regions. It is build of up by two parts.

The first part of the introduced infrastructure is a managing object, called *SHM_Holder*. It is implementing the *Singleton* design pattern to act as a central point of access. This object provides two ways to access the wrapped general shared memory implementation. By the use of the first one a new shared memory region can be created. The other one is to open a previously created region, or load it respectively.

Introducing a static allocator

Beside the managing object a way for dynamically allocating memory within the shared region needs to be defined. Therefore a allocator class, called *MutableAlloc*, is introduced. This class implements a easy to use interface for allocating memory either in the mapped shared memory region or the normal process heap. For this it implements static, template

based, access methods for the allocation and deallocation of memory. Where the memory resides is defined by an initial step. How this new infrastructure is initialized is shown in listing 3.2.

```
std::string shm = "MySharedMemory";

if (create_shm_region)
{
    // create a shared memory region if requested
    // set 2GB as default
    std::size_t shm_size = 2.1475e+09;
    SHM_Holder::Init(shm_size, shm);
}
else if (load_shm_region)
{
    SHM_Holder::Open(shm);
}
else
{} // nothing to do

// in any case we need to initialise MutableAlloc instances here,
// since they all use the same SegmentManager we can use any templated
// version of MutableAlloc here
// Note: if SHM_Holder was not initialized explicitly
// std::allocator (Heap) will be used
MutableAlloc<char>::Init(SHM_Holder::SegmentManager());

// Note: the SegmentManager can be altered later on, to support
    multiple
// shared memory regions or to switch to Heap memory
```

Listing 3.2: Initialization and usage process of the *SHM_Holder* and *MutableAlloc* class

Placing data in the shared memory region

As mentioned before all necessary data is already held by the *XPSConfig* class. The singleton instance of this class now needs to be placed into the shared memory region, or loaded from it respectively. Therefore, the current instantiation method is extended. The original *GetInstance* method now accepts an optional parameter for the segment manager to use. To allocated memory dynamically into the shared memory region the previous defined allocator class needs to be utilized. For initialize object creation it is needed to forward the outcome of the allocation, which is the memory address, to a *placement new* call. Note, that there is the possibility of a named and a anonymous creation. The named one is used if the object should be directly locatable by any other process. For example,

this is used to locate the main *XPSConfig* object. The process of placing the context object into shared space and allocating memory accordingly is shown in listing 3.3.

```
1   // assume the SHM_Holder and MutableAlloc classes
2   // are initialized properly
3
4   if (load_shm_region)
5   {
6       // if we want to load from SHM we need to search
7       // for an instance created previously
8       XPSConfig::FindInstance(SHM_Holder::SegmentManager());
9   }
10  else
11  {
12      // creates XPSConfig in SHM if SHM_Holder is initialized,
13      // otherwise Heap is used
14      XPSConfig::GetInstance(SHM_Holder::SegmentManager());
15  }
16
17  // ...
18  // get direct access to the allocator in use
19  void* alloc = SHM_Holder::SegmentManager();
20
21  // create a named string in shared memory
22  std::string* str1 = reinterpret_cast<boost::interprocess::
        managed_shared_memory::segment_manager*>(alloc)->construct<std::
        string>("MyString") ("My String Content");
23
24  // create an anonymous object in SHM
25  std::string* str2 = new(MutableAlloc<std::string>().create_anonymous())
         std::string("My String Content");
26
27  // ...
28  // clean up objects, it's the same for named and anonymous objects
29  MutableAlloc<std::string>().remove_anonymous(str1);
30  MutableAlloc<std::string>().remove_anonymous(str2);
31  // here we need to destroy the singleton
32  XPSConfig::Destroy();
```

Listing 3.3: Place the *XPSConfig* object in shared memory and create new objects there

3.2.3 View and update simulation data

To observe a simulation, the process now needs to initialize its data within the shared memory region. Contrariwise, the observing process has to load the simulation context from the same space. The code needed for that is shown above and an example is given in the results, section 3.3. To provide continuous updates of the simulation data, a simple

command-response protocol was implemented. With that the observing process can either change a predefined refresh rate or directly request an update of the data at any time. Note, that updates of the simulation data can only be done directly after a simulation time-step was finished and the next one has not started yet. Since the simulation is running in concurrent threads (see chapter 5) additional synchronization work between the data management code and the simulation algorithmic is required.

3.2.4 Deploy as program library

To lower the developers effort on implementing an consumer application, like for viewing, data forwarding (e.g. over TCP/IP), online analysis or similar, the implementation is packed into a program library. When used, either linked dynamically or statically, access is given to the shared memory by just initializing the *SHM_Holder* class and search for an existing *XPSConfig* object in shared memory. Similar as shown in the listing above. Afterwards, all shared simulation can be accessed in the consuming program, by querying the *XPSConfig* object.

For the case that it is needed to place other objects into the shared memory region, the allocator implementation is provided as well. Since it consists mostly of template based code it is shared as a source only library.

3.2.5 Command line arguments

As stated before, some new command line arguments and program options where introduced, allowing an user to enable the data sharing routines. The new options are shown below.

- **shm** - *no argument* - use shared memory, default name is "XPS_{PID}", where *PID* is the process id given by the operating system
- **name-shm** - *name* - use/create the shared memory block identified by *name*, implies the *smh* option
- **load-shm** - *name* - load the shared memory block identified by *name*, implies the *smh* option

3.3 Results

To show the basic functionality of the suggested implementation an arbitrary simulation can be started. The only thing that has to be done is to actually use the given functionality.

How this is obtained will be shown below. Additionally, a side by side view of the old internal simulation viewer and a connected one will be presented as well.

3.3.1 Usage

At first the simulation process has to be instructed using the data sharing functionality. This is done by passing one of the newly introduces command line arguments shown in section 3.2.5. The execution line is shown below in listing 3.4. After this call the simulation starts to run with calculation settings given in the *xps.config* file. Furthermore, the process will register a shared memory segment named *XPS_Test*.

```
1 $> xps --config xps.config --name-shm XPS_Test
```

Listing 3.4: Call to XPS for using shared memory

On the other hand, the remote viewer application needs to be adjusted to connect to the shared memory segment. This is done within the graphical user interface settings as shown in figure 3.1. Afterwards, the current simulation outcome, from the running XPS process, will be shown within the remote viewer.

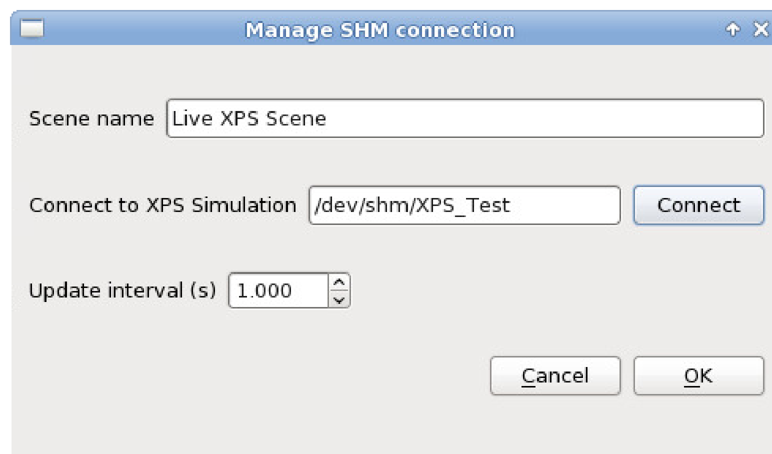

Figure 3.1: Connect Live Simulation (Source: RCPE GmbH)

3.3.2 Example

At the bottom of the settings window, as shown in figure 3.1, the refresh rate for updating the current data in shared memory can be adjusted. In this example it is set to one second. Note, that this means simulation time and not real time. So, for example, by using a time-step size of $10^{-5}s$ by the simulation, an update will be triggered after every

ten thousand simulation steps. This setting might be altered later on.

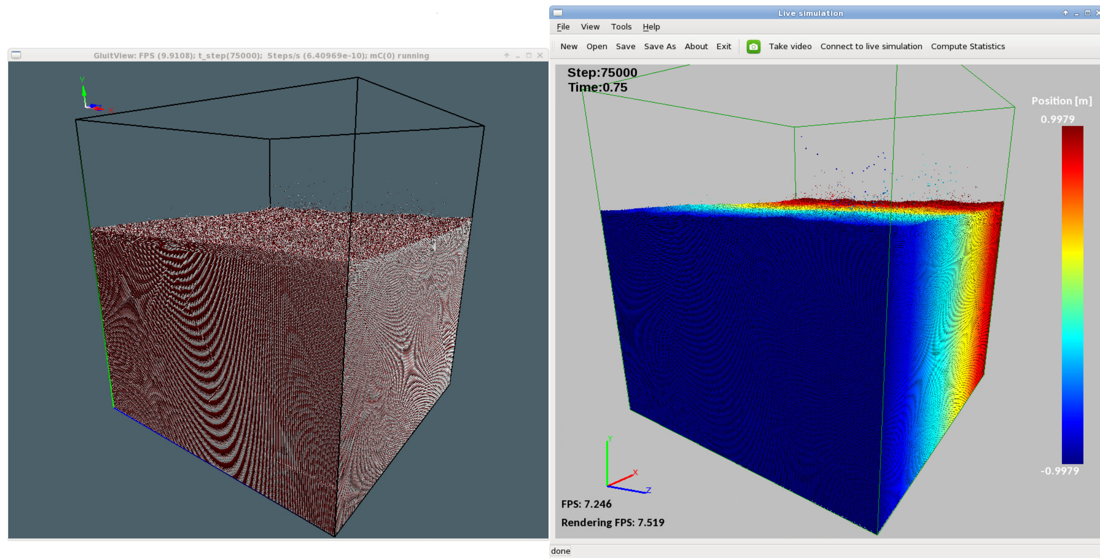

Figure 3.2: Left internal deprecated viewer, right external live viewer (Source: RCPE GmbH)

Figure 3.2 shows a running simulation within the built-in, deprecated viewer on the left side and the newer remote capable user interface on the right side. As it can be seen, the view is the same on both sides. The simulation running in the background was at $0.75s$ simulation time.

Chapter 4

Multi-GPU DEM - Design

The use of multiple GPUs for exploiting an additional level of parallelism, requires structural and logical changes in the original implementation of the XPS algorithmic outlined in section 2.5. Currently, the algorithm works on a single particle set bounded to exactly one single GPU utilized by one CPU thread. This procedure was absolutely fine until now, but is inappropriate for further multi-GPU implementations. This chapter is mentioned to give an overview through design thoughts for this particular task.

4.1 Subdividing the simulation world

The main idea for utilizing multiple GPUs within the existing DEM implementation is to split the upcoming computational burden across all utilizable devices. The DEM algorithm used in XPS is strictly sequential in its time step based iterations and all particles have a spatial locality. While using the grid based particle localization approach, as presented in 2.5.3, the split might be done along any cell boundary. We will call such a boundary *a halo*. For the DEM, dealing with concrete particles only, each halo needs to have a thickness of one grid cell on every side of the boundary.

4.1.1 Choosing a reasonable sub-division

Although, any closed path might be used for domain separation, not all of them will make sense. For example, a split, along an arbitrary path of cell boundaries, as shown in figure 4.1 for the 2D domain, is not wise to choose. In the shown situation, a single GPU has to deal with multiple halos in multiple dimensions, all of them rather small in size. This gives the need to calculate and dispatch all information for each halo separately, giving a tremendous additional workload to each GPU.

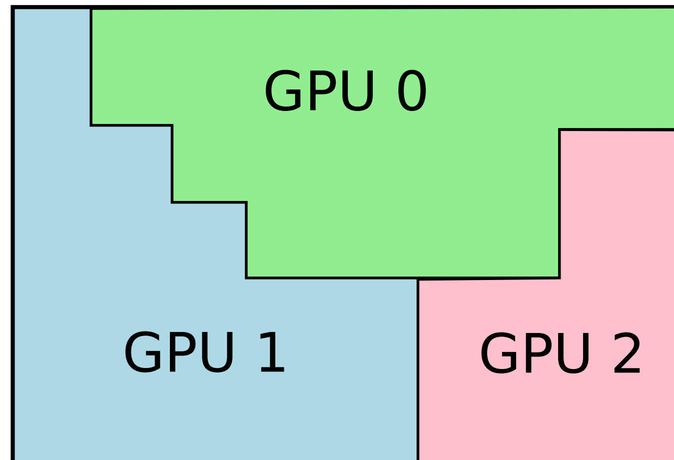

Figure 4.1: Splitting a 2D domain along an arbitrary path of cell boundaries (Source: RCPE GmbH)

Therefore, it is wise to split the domain along planes consisting of cell boundaries. When having multiple splits, which means using three or more GPUs working on the same domain, splits as shown in 4.2 might be suitable. Here the simulation domain is divided along planes in the vertical and horizontal directions. As a drawback it has to be mentioned that now each sub-domain may has a boundary plane in every dimension (up to three for a three dimensional simulation world). As stated before, this will also increase the algorithmic and computational effort needed for handling the halos.

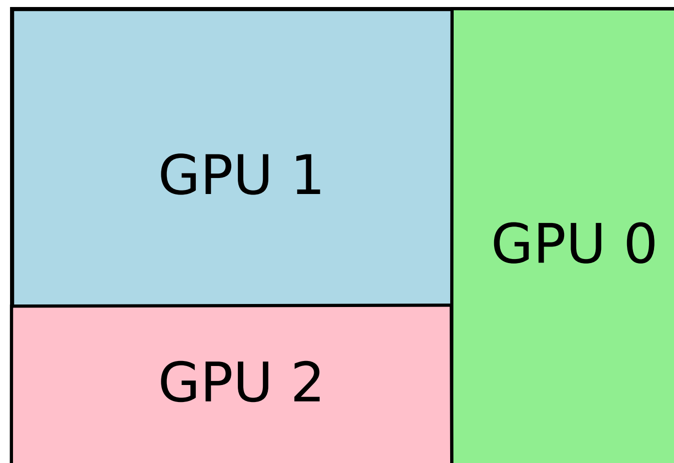

Figure 4.2: Splitting a 2D domain along vertical and horizontal planes (Source: RCPE GmbH)

So the perfect split will be along one simple dimension for all sub-divisions, as shown in figure 4.3. In this configuration each GPU has to handle only a maximum of two halos for

its assigned sub-domain. Both aligned with the same axis, making the algorithmic effort as low as possible. Assuming all cells are enumerated with one most significant axis and the split plane chosen along this axis, such a configuration will give additionally memory accesses enhancement on a single GPU. By occupying contiguous memory areas for the halo planes, the number of needed memory transfers is reduced, as shown by [RBH+14].

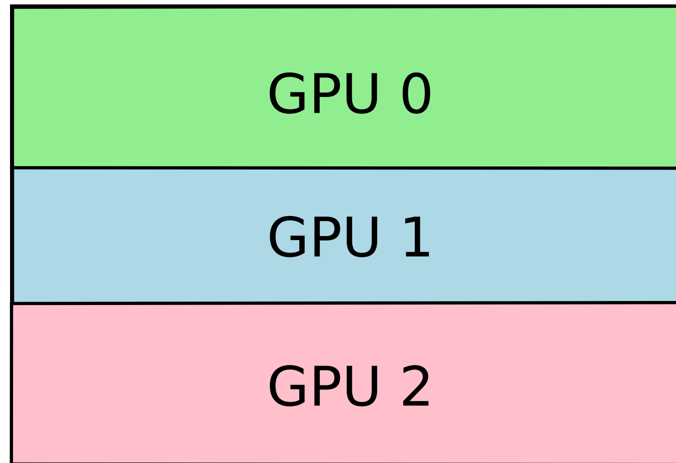

Figure 4.3: Splitting a 2D domain along vertical or horizontal planes (Source: RCPE GmbH)

4.1.2 Subdivision overlap and halo distribution

As shown in the sections 2.1 and 2.5.3 each particle needs to know all of its direct neighbors at each time step. This means, the GPU, on which the particle currently resides on, must be able to access all the needed information. If the particle resides in the inner part of the assigned sub-domain, this is no problem at all, since all of the information required resides in local GPU memory. On the other hand, if the particle currently stays within a domain halo, the GPU needs to access the halo information shared by the neighbor GPU as well. As mentioned before the halo thickness needs to be two grid cells wide, having a thickness of one grid cell on each associated sub-domain on both sides. The overlap is pictured in figure 4.4. [DCVB+13]

The halo information has to be shared in every time step. There are two communication steps needed. At the beginning of each time-step, the preparation takes place. Within this step all particles positions are evaluated and they are sorted into the grid. Afterwards all particles residing in a halo have to be collected and distributed to the neighbor GPU. At the end of a time step calculation all particles leaving the sub-domain need to be detected and transferred into their new domain. If having all sub-division boundaries bound to

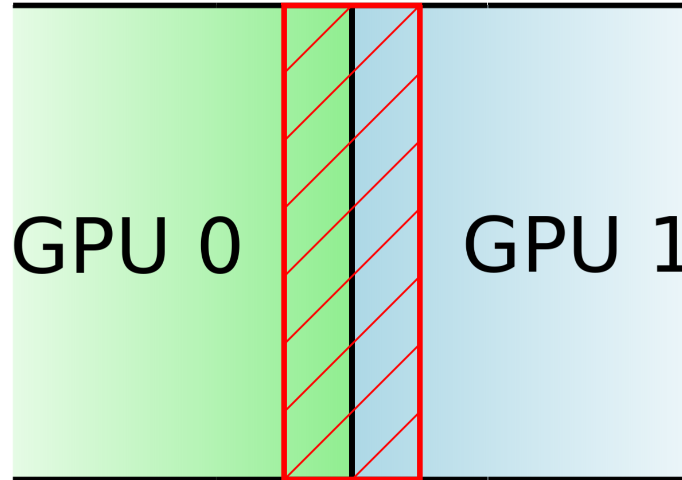

Figure 4.4: The boundary between two sub-domains on grid cell layer, with the halo highlighted in red. (Source: RCPE GmbH)

planes parallel to an major axis, as described above, the detection of a leaving particle can be fulfilled by just checking the crucial position component. The described process is pictured in figure 4.5, for a two dimensional simulation world split in the x-direction. [RBH+14]

4.2 Hiding GPU memory transfers

To provide all neighboring sub-domains with the updated positions of halo particles two data transfers have to be done in every time step. Since direct GPU-GPU memory communication is not an option for the available cards, while doing this thesis, all memory transfers needed to be queued through the hosts RAM. This will introduce a significant overhead slowing down the overall computational speed.

To avoid this problem, one can exploit the hardware capability to perform concurrent computations and data transfers. By using asynchronous work queues, provided by the *asynchronous API* of the CUDA® platform, transfers can be started as soon as the needed data is ready. As described in the theory section about the CUDA® programming model 2.2.1, we can use events and streams for doing so. Additionally the asynchronous approach can be used to parallelize smaller and independent computational workloads also. This will give an extra performance gain in some situations. What can exactly be done will be shown in the thesis part about the implementation in chapter 5. [Yua13], [RBH+14]

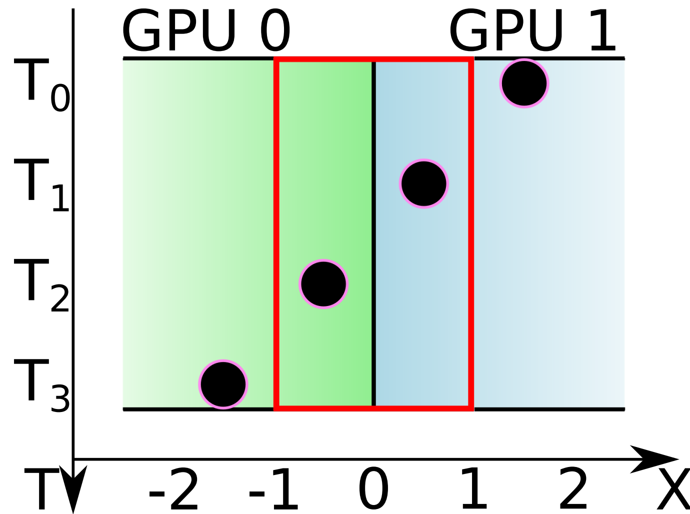

Figure 4.5: A single particle traveling to another GPU domain (Source: RCPE GmbH)

T_0: $P_x = 1.5$ Particle is known to the right sided domain solely

T_1: $P_x = 0.5$ Particle entered the halo recently and is shared with the left domain

T_2: $P_x = -0.5$ Particle left the right sided domain and was transfered to the right one. Since it is located in the halo region, it is still shared with the right domain

T_3: $P_x = -1.5$ Particle is known to the left sided domain solely

4.3 Load balancing

In the ideal case, the whole workload is distributed to all available GPUs equably. Assuming the halo distribution, in detail the data exchange, slows done all participating GPU by the same factor, then the speed-up of the multi-GPU is linear to the total number of GPUs used.

Analysis show that most of the time is spent during force calculation. The per particle workload which is needed for this step depends heavily on how many neighbors the particle has. By processing a certain amount of particles on a single GPU, the total time the device will need to perform a single time step is directly proportional to the mean number of neighbors per particle. Due to the Lagrangian nature of the DEM, particles move through space during the simulation is running. Therefore, ideal load balancing, giving a linear speed-up is rather hard to achieve.

For utilizing all available GPUs the whole simulation world needs to be split into a set of sub-domains. One for each device. At simulation start all sub-domains should

contain the same amount of particles, if all of the used computing devices obtain the same computational power. Otherwise the number of particles for a single sub-domain should be proportional to its computing capacity. Attention, has to be given to the total available memory of the used device, because in any case the particles simulation data has to fit into device memory.

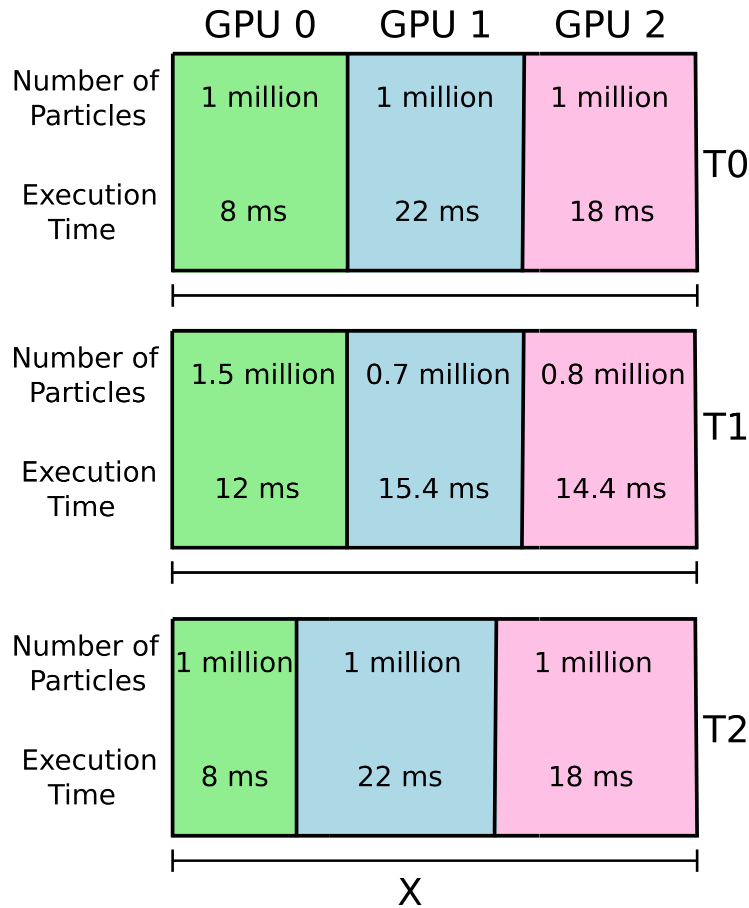

Figure 4.6: Example of load balancing by particle number, of a setup with GPUs of different speed. (Source: RCPE GmbH)

Two different load balancing algorithms can be used now. The first one assures that always the same amount of particles is assigned to all sub-domains. This algorithm is suitable if all used GPUs have the same computing power. The example in figure 4.6 shows how the algorithm works. Note, that GPUs with different speeds where choosen do show the infeasibilities of this simple method for optimizing the execution time. First, all three sub-domains contain the same amount of particles. After a certain number of time steps the particle count diverges. Now a balancing action is executed. The task is to move sub-domain boundaries to equalize the particle count over all domains. It can be

seen, that the situation at time T_1 is more ideal in respect to simulation time, than the equalized situation at time T_2.

The other possible balancing algorithm optimizes the total per time step execution time over all devices. A weighted average of computing time per simulation step over a suitable number of steps is measured per device. The average should be used, because a single time step might present high fluctuations on execution time. As stated above this is due to the mean number of particle-particle interactions within a single domain. After a defined number of steps, these values are evaluated and further load balancing actions are triggered. It has to be noted, that this algorithm will not unify the computation time, over all sub-domains, by a single execution. But it will equalize the workload over the total simulation time, increasing the overall performance. It can be easily seen that this algorithm binds more process time then the first, simpler one, but it is more adaptive and therefore best suitable for devices with different computational capacities.

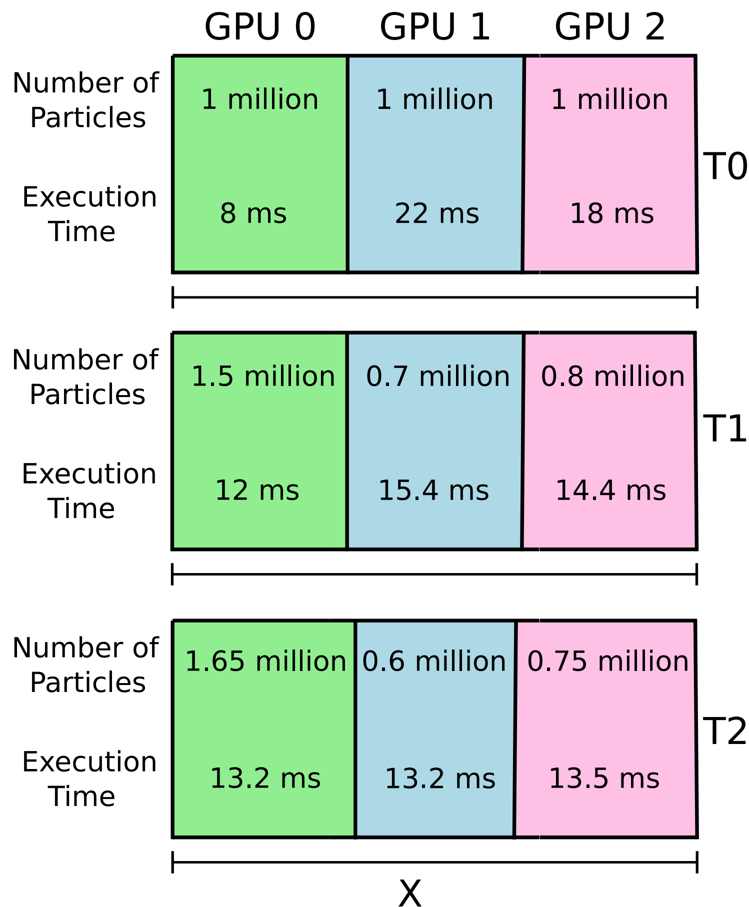

Figure 4.7: Example of load balancing by execution time, of a setup with GPUs of different speed. (Source: RCPE GmbH)

Figure 4.7 shows an example on how this method will balance the workload for two computing devices. The number of steps used for averaging is five. Used weights are the same for all samples $w = \frac{1}{5}$. The balance action is executed every five time steps. Similar to the example before, the workload diverges over time. After the balancing step, the execution times of the next step are adjusted. [DCVB+13] [RBH+14]

4.4 Possible implementations

Based on the given DEM implementation shown in section 2.5.2, the algorithmic should be extended to run on multiple GPUs in parallel. As stated in section 2.2.3 there exist three different methods on how multiple GPUs can be utilized from a host program. Figure 4.8 shows a simplified flow-chart of all steps needed in the multi-GPU Discrete Element Methods simulation. Computational tasks that might be running in parallel are already drawn side by side. All needed synchronization steps are shown explicitly. Execution tasks that needs to be synchronized on two or more GPUs are shaded light green. Synchronization that has to be done for a single device only is shaded light blue.

In the first step, temporary time step information needs to be cleared. This is acting forces, grid mapping, halo information and so forth. No synchronization needs to be done afterwards since the next step, which is preparation, needs to be done for each sub-domain solely. After synchronization all particle positions will be known. Now, the simulation can be split into two execution paths. The one is colliding all particles within the domain, including all halo particles. Since all calculation outcome is hold internally, this step does not need inner synchronization.

The other path has to exchange the halo information with the neighboring GPU at first. This needs two internal synchronization steps. The first has to synchronize directly after downloading from the GPU memory for exchanging data through host memory. The second synchronization has to take place after the actual transfer. Consider this two synchronization steps are needed per single halo. So a sub-domain having two halos will need four synchronization points. After successful data exchange each halo needs to calculate all collisions happening halo-internally.

Before the integration step, all calculations regarding particle collisions within the domain must be completed. Therefore, every computing device has to be synchronized internally. After, the integration is fulfilled the second halo-exchange step is performed. All particles leaving their domains have to be detected. By using a halo division as shown in figures 4.3 and 4.4, domain split parallel to an single axis, this step is computational inexpensive. The only thing that needs to be done, do detect a leave, is to check if

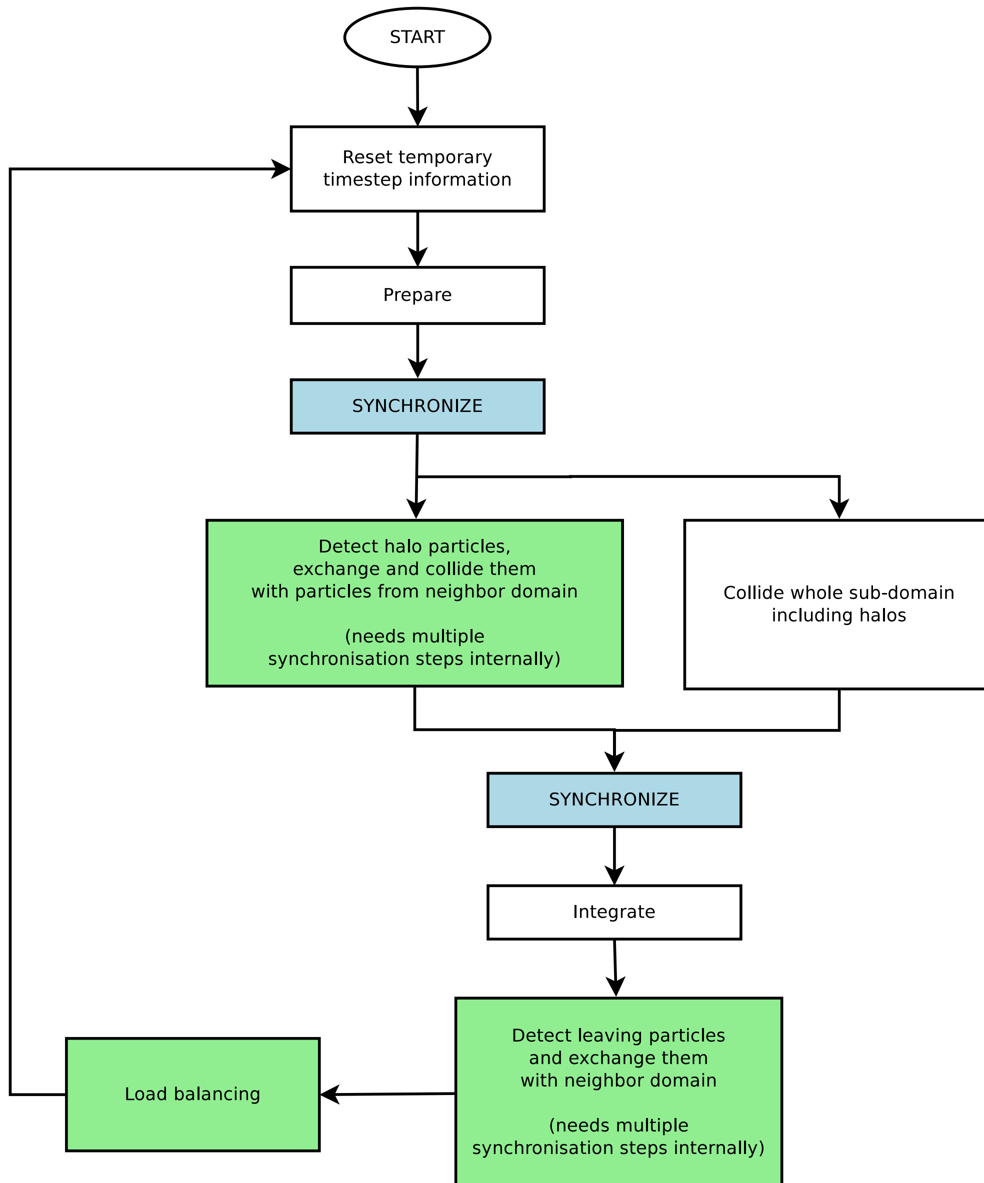

Figure 4.8: Simplified flow-chart for a DEM implementation computed on n GPUs. (Source: RCPE GmbH)

the particle is outside of its domain in the split direction. Since a particle should not move further than about ten percent of the grid edge size, only particles located inside a halo have to be checked here. Detected displaced particles now have to be transferred to the neighbor GPU, where they will enter the sub-domain. This requires the same synchronization points, as the previous halo exchange. The last step that needs to be taken is load balancing. This requires additional synchronization, due to the implemented algorithm as specified in section 4.3.

4.4.1 Single-Threaded Multi-GPU (ST-MGPU)

As stated in the theory section about multi-GPU, it is possible to drive multiple GPUs from a single CPU thread. To do so, the controlling thread needs to know all information about the current simulation and the state of all GPUs under it's control. Having such a single authority prevents the need of IPC among multiple players, as this has to be taken in focus for the other two methods.

One the other hand, it can be easily seen that the GPU synchronization overhead prevalences really quick as the number of used GPUs increases. This is because all computing devices have to be checked for their current state on an periodic base. It is needed to react on all changes specially but in respect to the state of all other devices. This makes the code base even more complex and introduces algorithmic overhead at the expense of process time.

4.4.2 Multi-Threaded Multi-GPU (MT-MGPU)

In contrast to the single threaded approach, the multi-threaded multi-GPU method removes nearly all inter GPU synchronization issues. Additionally, one might gain further CPU sided performance by doing host side tasks in parallel, like host sided memory copies. Since one GPU is driven by a single CPU thread the handling of computing devices is much simpler here. The algorithmic just needs to query the simulation steps one by one into the execution pipe. So the prior needed GPU synchronization will be transformed into host sided synchronization tasks, for which a form of IPC is needed.

4.4.3 Multi-Process Multi-GPU (MP-MGPU)

If multi-threading is not suitable or the used GPUs are distributed to different physical computing nodes, connected by an network, multi-process multi-GPU can be implemented. If so all IPC has to be done on process level. An implementation using MPI is described by [Yua13].

Chapter 5

MultiGPU DEM - Implementation

The basic properties for a multi-GPUs implementation of the DEM method where already given in the design chapter 4, where the basic ideas behind the implementation had been described. Based on this ideas, this chapter is supposed to show what the explicit implementation of the DEM algorithm on a multi-GPU system could look like.

5.1 Preliminary work and re-factoring of the existing code

After deep analyses of the existing implementation, as presented in section 2.5, the existing code base was undergoing a huge re-factoring step. The monolithic calculation part got extracted and made more configurable and reusable. Thus, a base class of an abstract concrete simulator object, simply called *Simulator*, was implemented. This class defines a specialized interface to address time-step based algorithms, as shown in figure 5.1.

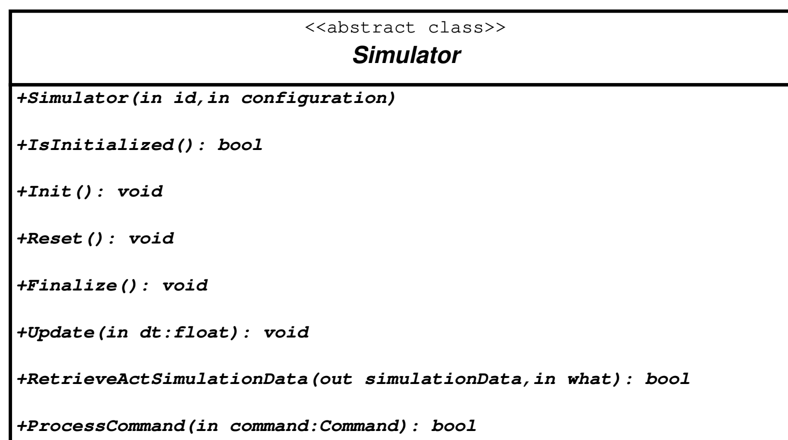

Figure 5.1: Class diagram of the minimal *Simulator* interface. (Source: RCPE GmbH)

After creating a simulator object it has to initialized with an configuration object through the *Init()* function. In almost the same manner, *Reset()* will set all internal data back to there default state and *Finalize()* will finish an ongoing simulation. After successful initialization the *Update(dt)* routine is mentioned to be triggered at each time-step to calculate the next algorithm step, with a step size of *dt*. Through the *ProcessCommand(command)* interface the simulation can be altered while processing. Therefore, a command has to be submitted between two algorithmic steps. Finally, the *RetrievActSimulationData(simulationData, what)* method is used to retrieve the current simulation data, like particle positions, velocity and others.

The existing single-GPU DEM-solver was re-implemented along with the existing existing CFD-coupling. With a new algorithmic design of the managing core-parts of the XPS software it is now possible to run multiple simulators in parallel. This characteristic is now used by the CFD-DEM co-simulaton two.

5.2 Top-Down view

As shown in the collaboration diagram in figure 5.2, a multi-GPUs solver is implemented based on the new simulator interface. The multi-GPU simulator object instruments a certain amount of concrete workers. One for each used GPU. Note, in a more developed version of the a multi-GPU simulator the worker to GPU arrangement and the number of used graphical units might not be fixed and may change over time. Due the complexity of such an implementation and the inconsiderable implications on this thesis such a feature is not implemented.

Stated previously in chapter 4, about the design of this task, the three dimensional simulation domain is split into clusters of cubic forms. The size of this blocks is affected directly by the algorithm and outcome of load balancing. A single block, and all particles within it, is always hard bounded to a single GPU. Particle data is organized in a special data structure. For that reason, all objects located in a given block form a particle set. Subsequently, a worker obtains ownership of the particle set assigned to its correlated GPU and block, respectively. The worker now executes the DEM algorithm on this particles. To do so the worker class instruments a list of highly specialized solvers, which are explained below.

5.2.1 Algorithm solvers

To be able to proceed the DEM algorithmic in a step wise manner so called solvers where introduced. A solver is an object that implements the *DEMSolver* interface, given in figure

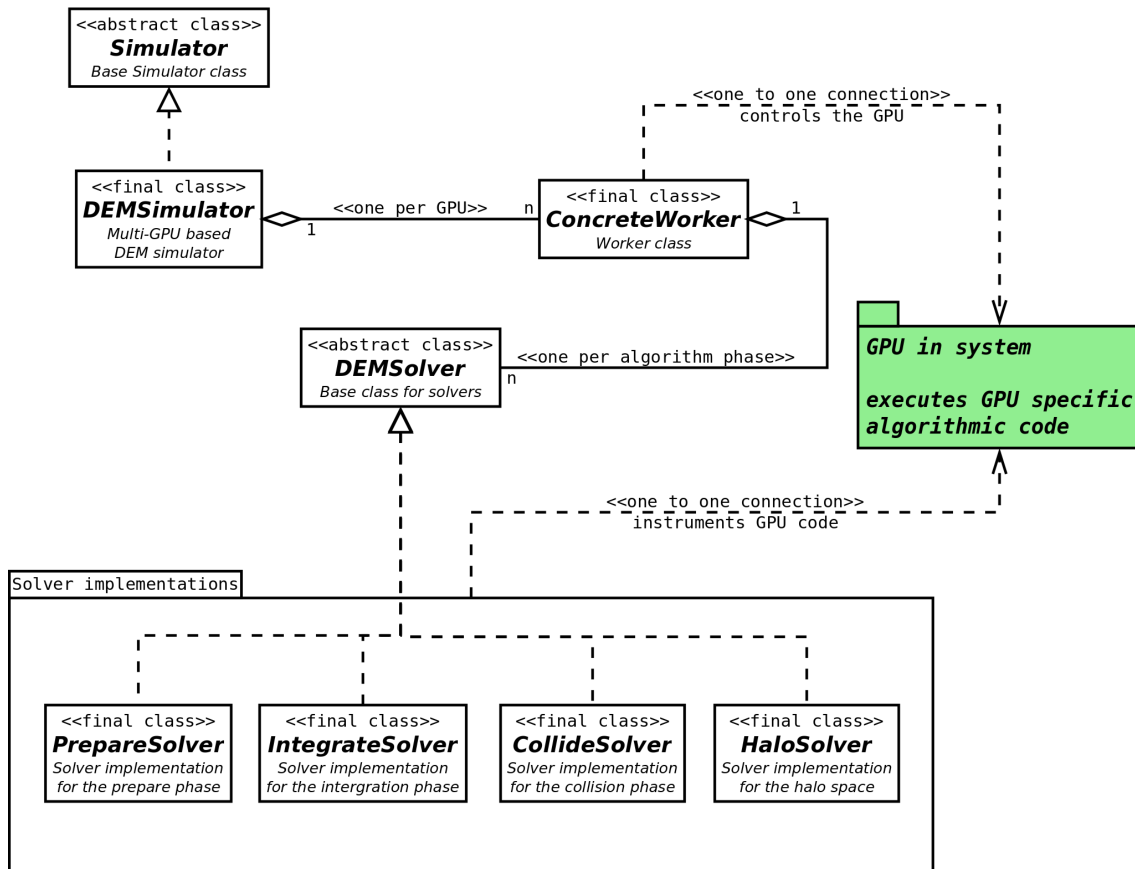

Figure 5.2: Collaboration diagram of the multi-GPU implementation of the DEM. (Source: RCPE GmbH)

5.3. This abstract class looks similar to the *Simulator* interface shown before. Therefore it is suitable for being used within the time-step based algorithm of the DEM, too.

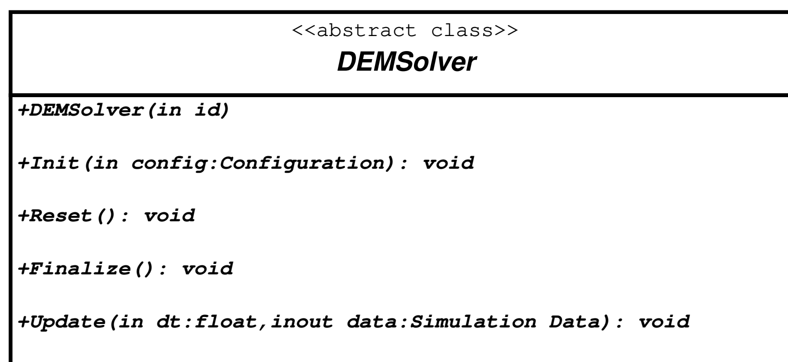

Figure 5.3: Class diagram of the minimal *DEMSolver* interface. (Source: RCPE GmbH)

As indicated by figure 4.8, dedicated footsteps, presented in figure 5.4, can be extracted for the multi-GPU based DEM. By having a specialized solver for each dedicated footstep, the whole calculation is split into small parts, that can be run subsequently and, if possible, in parallel. A solver comprises the algorithm to fulfill one or more of the presented footsteps. So, a particular list of specific solvers will solve a complete DEM time-step if executed in the right sequence. The solver execution is triggered by calling the *update()* method. Furthermore, this design allows easily replacement of used algorithmic details, as exploiting cell based or BVH based [Kur16] methods within the preparation phase. The solvers used in this implementation, as shown in figure 5.2 are explained below.

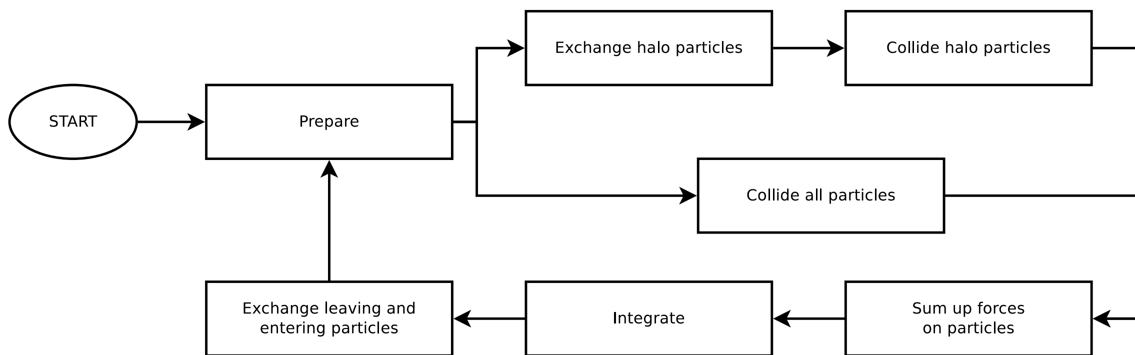

Figure 5.4: Dedicated footsteps for the multi-GPU based DEM. (Source: RCPE GmbH)

Prepare solver

Described in the sections 2.1 and 2.5.3 (preparation phase), the DEM itself is a gridless-method. But for the faster detection of local relationship of the particles, some kind of helper method might be used. This can be local grids, the BVH method, describe by [Kur16], or any other suitable technique. The class of preparation solvers are used to provide and apply the chosen procedure onto the particle set. The delivered data structure for the regional correspondence vary with the method used. Therefore, all subsequent solvers, using the outcome, have to have support for the technique chosen.

Collide solver

A collision solver needs to work on the information provided by a preparation solver, as shown before. To detect all collisions happening, the solver needs to check all neighbors for all particles. This can be done in parallel for all particles. Once a collision is detected, interacting forces are calculated and added to the resulting force and momentum vectors

for the particle observed. This procedure is described in section 2.5.3 (collision phase) in detail. It may happen, that other solvers, like the halo solvers, work in parallel on the same particles at the same time. Especially the mentioned class of halo solvers will calculate interacting forces too. Therefore, all additions to the resulting vectors need to be done in a thread-safe way. A method to do so is using the atomic functions provided by CUDA® as described by [NVI18].

Halo solver

By splitting the simulation domain into separate blocks halo spaces arise, like illustrated in section 4.1.2. The class of halo solvers are responsible to solve the special tasks bound to this spaces. A halo solver will first collect all particles in his observed halo space and exchange this information with the related halo solver of the neighboring simulation domain and vice versa. After receiving the neighbor information, the halo solver needs to do a collide step for all particles within his control and all of the particles in the adjacent halo. This is done similar like the collision solver does this, because localization information is exchanged as well. After the integration step of the DEM, it is also in the hand of a halo solver to collect all particles leaving the simulation domain block and hand them over to the domain they are entering.

Integration solver

After having all forces, for all single particles in the observed domain, summed up an integration solver is used to calculate the new particle positions, velocities and rotation information. This is done by starting a calculation per particle, solving the equations and doing the steps as described in section 2.5.3 (integration phase).

5.3 Multi-Threaded implementation

Due to the design decision by having each GPU used controlled by only one single worker object it makes sense to use a multi-threaded approach of the host-sided code base. So, each worker runs as an own thread, with its own context, controlled by the simulator object. As mentioned in section 4.4, IPC techniques, like locking of data-structures and synchronization, with methods described in section 2.3, are needed.

As shown in figure 5.5 synchronization is needed between the workers and the controlling simulator object. All workers have to communicate with their neighbors when doing data exchanges. This is needed for solving the halo specific tasks, like exchanging halo

domain particles information, especially. Therefore, the communication has to be synchronized after collecting and sending this information, like marked in light green within the figure. Additionally, global synchronization between the managing simulator object and all workers need to be done at certain points, marked in light blue.

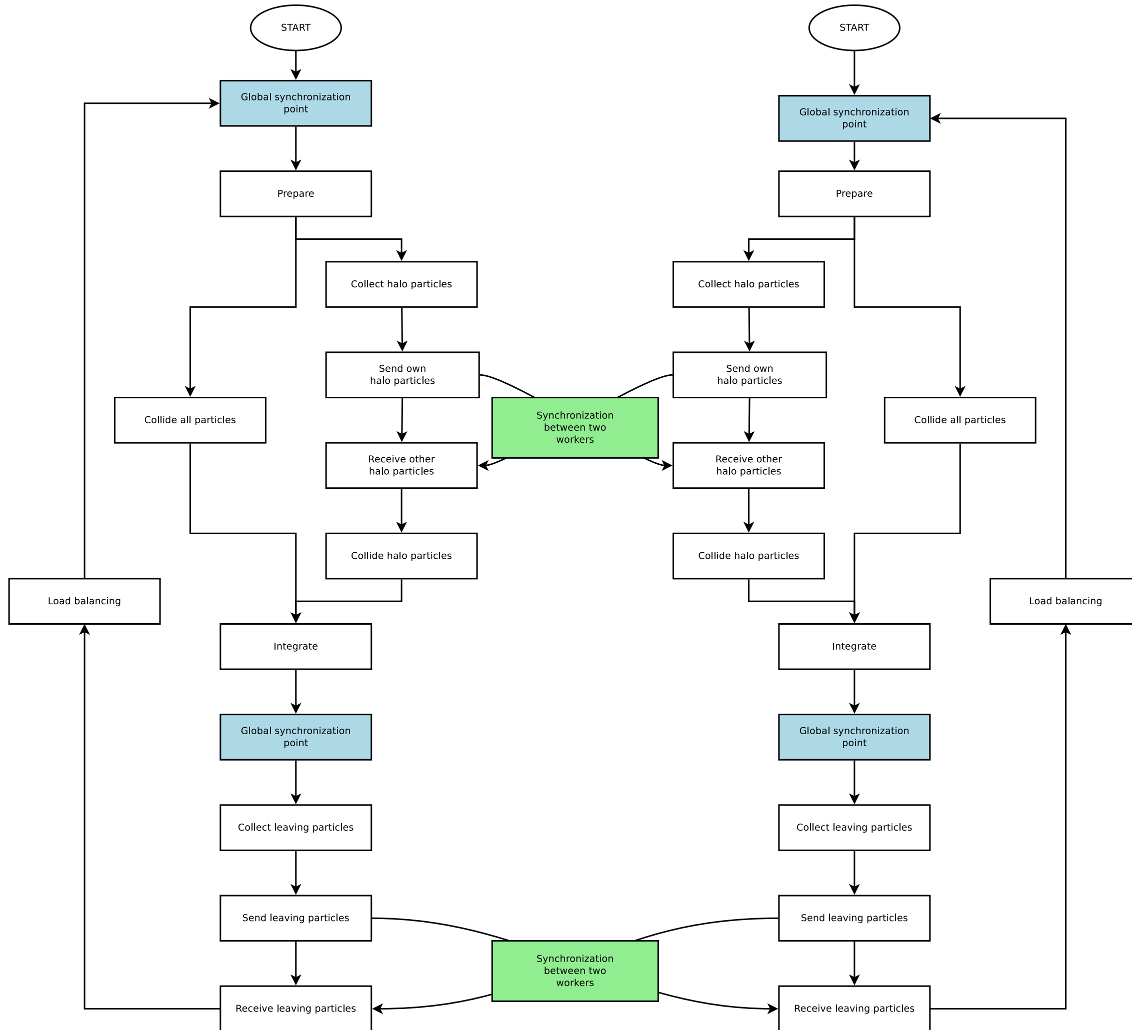

Figure 5.5: Detailed flow chart with marks for needed synchronization points between workers. (Source: RCPE GmbH)

5.4 Optimizing execution speed

When it comes to speed optimization of GPU code execution, there are two things to needed to be counted for. On the one hand, transferring data from a GPU to the CPU or the other way around, is expensive in terms of bandwidth and time. On the other

hand, small kernels, executed just for a few threads, may not fully utilize a GPU core. If so, a CUDA® core will have space for executing other kernels in parallel. Both of this issues can be addressed by using CUDA® streams for parallel execution and events for synchronization, like described in section 2.2.1.

5.5 Load balancing

As stated in section 4.3 load balancing is needed to equalize the execution time over all GPUs used. Therefore, the execution time of each time-step is recorded for each worker separately. After a certain number of time-stamp loops, an averaged time value is calculated for each worker. Now a factor is evaluated between all workers and as a consequence the sub-domain blocks, assigned to the used GPUs and workers, are shifted and resized. To prevent that shifting gaps are getting too large, the load balancing is done every 32 time-steps. Additionally to prevent particles falling out of any simulation sub-domain while load-balancing is done, the resize difference must not bigger than half of the diameter of the smallest particle used. By considering this, particles that might leave the domain while balancing, will be still considered in the next time-step loop and leave the sub-domain accordingly.

Chapter 6

Multi-GPU DEM - Results

This chapter shows the results of this thesis implementation of the Multi-GPU objective. As a start, the used test system is described. Afterwards, results for two improvement characteristics introduced by this work are shown. The first one targets on pure performance gains that where achieved. The other shows how the Multi-GPU approach can increase the maximum number of particles per simulation.

6.1 The Test System

For testing the suggested implementation a potent test system is needed. Since it was available during this thesis execution time, a Multi-GPU blade, built by TYAN, was used. Although, this system offers space for up to eight GPUs driven by two CPUs only four GPUs where available during the test phase. Table 6.1 shows specifications of the test system. A picture of the system is given in figure 6.1.

Model name	Tyan GPU Barebone B7079F77CV10HR
Operating System	CentOS 7, Kernel 3.10
NVIDIA Driver	352.63 stable
CUDA version	7.5.18
CPU	2 x Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz
Memory	128 GB DDR4
GPUs	4 x NVIDIA GeForce Titan X (Maxwell)

Table 6.1: Specifications of the test system as used to evaluate the Multi-GPU implementation.

Figure 6.1: A view on the test system as used to evaluate the Multi-GPU implementation, featuring four GeForce Titan X GPUs driven by two Xeon(R) CPUs. (Source: RCPE GmbH)

6.2 The Test Case

To get detailed insights on the suggested implementation a basic test case was chosen. As shown in figure 6.2, the test example consists of a regular box and a pack of spheres falling into it. This is like pouring marbles into a transport box. On the one hand, this is a very realistic and practical example. On the other hand it is very computation intensive too.

6.2.1 Evaluation Of Memory Usage

To get knowledge about how many particles can be simulated on a dedicated GPU, the memory consumption of a single particle has to be evaluated first. For the simple test case, that was used for the evaluation of performance optimization, as shown in section 6.3, each particle needs a set of variables. Those variables are shown and explained in table 6.2.

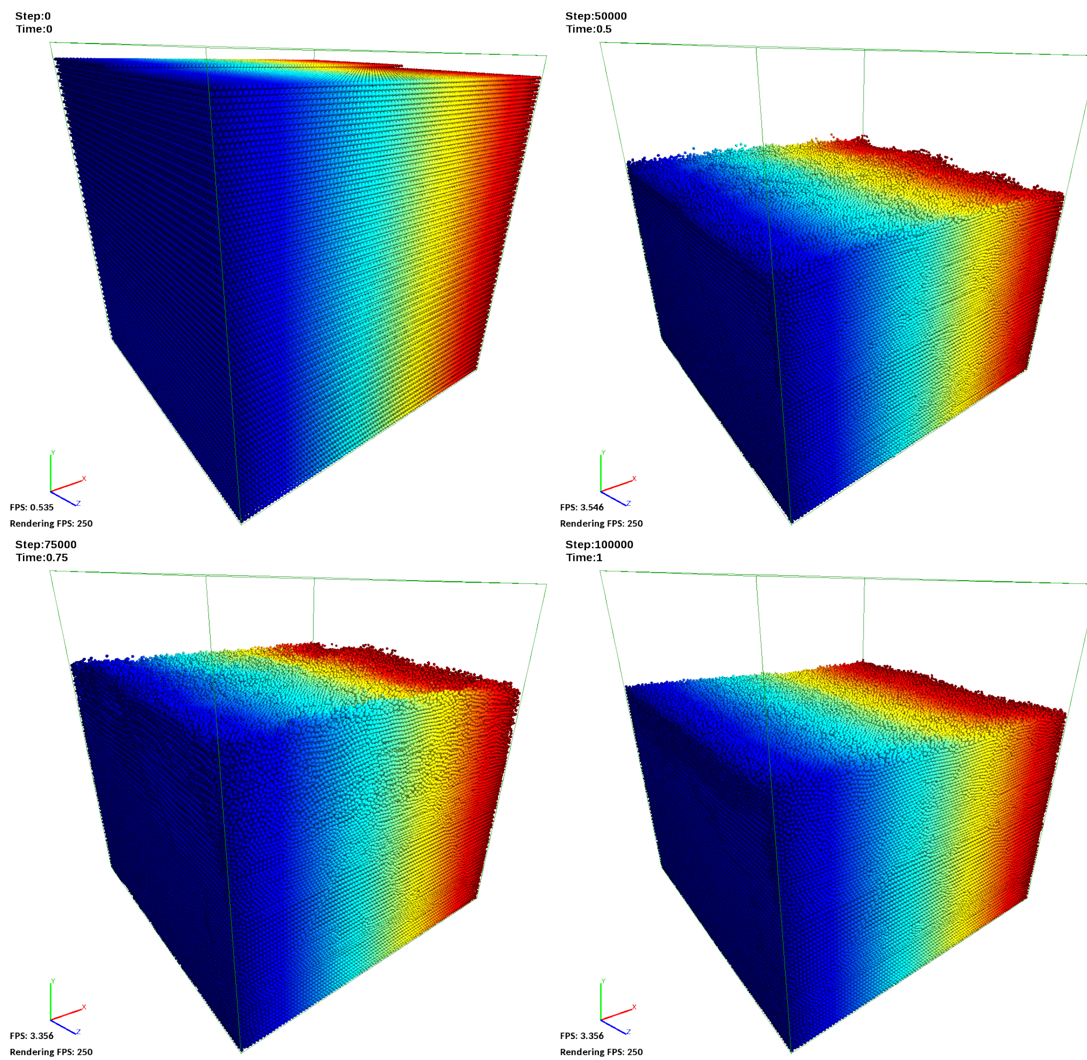

Figure 6.2: Particles falling into a box test case, with one million particles at different process times. (Source: RCPE GmbH)

Name	Type	Bytes	Explanation
id	uint64	8	unique particle id
species	uint32	4	a value masking the shape type and material of the particle
position	float4	16	current position of the particle in the simulation world (x, y, z) and the shape scale/size (w)
velocity	float4	16	current velocity of the particle (x, y, z)
force	float4	16	current force acting on the particle (x, y, z)
Total		60 Bytes	

Table 6.2: Minimum GPU memory consumption per particle.

A *float4* determines a special vector type introduced by CUDA®, as shown in listing 6.1. It consists of four float values *(x, y, z, w)*. To optimize memory throughput it has to be known that a warp coalesces all accesses into one or more memory transactions. Those depend heavily on word sizes, which should be aligned as 1, 2, 4, 8 or 16 bytes. Since the float4 consists of 16 bytes in total, this data type is optimized for usage on GPUs. Therefore this type should be preferred, even if other structures will match the data-layout better. [NVI18]

```
struct __device_builtin__ __builtin_align__(16) float4
{ float x, y, z, w; };
```

Listing 6.1: The float4 type as defined by the CUDA toolkit.

As outlined in section 2.5.3 additionally storage for cell information is needed. Per particle in the simulation, the hash is a four byte value and the index is of eight bytes length. Considering this and the values given in table 6.2, for a basic simulation without rotation and other properties, each particles will need a total of 72 bytes. Moreover, the **cellstart** vector needs another four bytes per cell.

6.2.2 Test Case Variations

The test case considers a simulation world size of $2m \cdot 2m \cdot 2m$, that is a total volume of $8m^3$. To get this box filled up by a desired particle quantity a maximum particle size has to be approximated. In case of spherical particles this is the diameter (d). Equation 6.1 shows a simple approximation. Note, the spherical particles are assumed as cubes. The number of particles to be used is given as N and the simulation volume is denoted by V.

$$P_{size} = \sqrt[3]{\frac{V}{N}} \tag{6.1}$$

Since XPS has support of some kind of filling algorithms, the so called *Grid-Placer* was used to fill the simulation volume. The upper left spot of figure 6.2 shows a regularly filled start condition. As the name suggests, it works on a regular basis and considers some offset and minimal spacing between the particles it places. Therefore, it is need able that the chosen particle size is somewhat smaller than approximated.

Assuming the simulation contains a quantity of N particles with a diameter of d the cell size of an optimal uniformed grid has to be adjusted. As stated in section 2.5.3, the ideal cell size will be likewise the particle size. Equation 6.2 determines how the number of cells in each direction (gridSize.xyz) can be calculated for the ideal case. The maximum particle elongation is given as d and worldSize.xyz donates the world sizes.

$$\text{gridSize}.xyz = \left\lfloor \frac{\text{worldSize}.xyz}{\text{d}} \right\rfloor \tag{6.2}$$

Equation 6.3 shows the calculation of the total amount of memory M needed on the GPU for a given quantity of particles N and a certain number of grid cells C. For the test case 72 bytes are needed per particle and additional 4 bytes per cell.

$$\begin{aligned} M &= N \cdot M_P + C \cdot M_C \\ &= \text{N} \cdot 72B + C \cdot 4B \end{aligned} \tag{6.3}$$

To profile the implementation, in terms of speed up and maximum reachable particle count, the described test case was executed in different variations as shown in table 6.3.

Number of particles [million]	Particle size [mm]	Cells		Memory usage		
		per dimension	count [million]	particles [GiB]	cells [GiB]	total [GiB]
1	20	100	1	0.067	0.004	0.071
10	9.2	217	10.22	0.671	0.038	0.71
50	5.4	370	50.65	3.353	0.188	3.54
100	4.2	476	107.85	6.706	0.402	7.1
230*	3.26	613	230.35	15.423	0.858	16.28
300*	2.96	675	307.55	20.12	1.15	21.27
600**	2.36	847	607.65	40.23	2.26	42.52

* only run-able on multiple GPUs

** only run-able on four GPUs with massive changes on optimization code parts

Table 6.3: Test case variations.

As shown in the variations table 6.3, the total amount of memory needed on the GPU for the 100 million example is around 7.1GiB. Given these values, about another 3.2GiB have to be added for algorithmic overhead and performance optimization. This is for sorting the particle indices every time-step, in the preparation phase, and sorting the whole particle set every N time-steps, to have better memory alignment. A GeForce Titan X, as it is available in the test system, provides a total memory amount of 12GiB. So, the 100 million particle test case will fit quite good on this kind of GPUs.

When it comes to the Multi-GPU implementation further memory consumption on the GPUs has to be considered. This arises from the need to collect particles in the halos and sorting particles leaving the domain, as shown in chapter 5. Therefore the maximal number of particles per simulation is not scalable by the GPU count. This leads to the fact, that a maximum of about 75 million particles per GeForce Titan X GPU was able to be utilized.

6.3 Test 1: Performance Gain

To test the plain speed-up gain from a single GPU up to the four GPUs available in the test system, the test case was executed in the 1, 10, 50 and 100 million setups. Furthermore, as shown in table 6.4, the 230 and the 300 million setups where executed on three and four GPUs, respectively. Since preliminary tests do not show a major difference in run-times between the two shown implementations, that one which is using halo threads and that which do not, only the version without halo threads was used for test case executions.

Number of particles [million]	GPUs			
	1	2	3	4
1	X	X	X	X
10	X	X	X	X
50	X	X	X	X
100	X	X	X	X
230			X	X
300				X

Table 6.4: Test 1: Mapping number of particles examples on number of GPUs.

As a precondition the simulation world is split into N different domains, where N is the number of GPUs used. The split is done automatically and since all GPUs in the test system are of the same type, all domains have the same amount of particles in the start condition. For this test no automatic load balancing was used. However, due to the

test case species having load balancing won't cause any big difference on the results. The initial domain division for the 1 million particles run is given in figure 6.3, for instance.

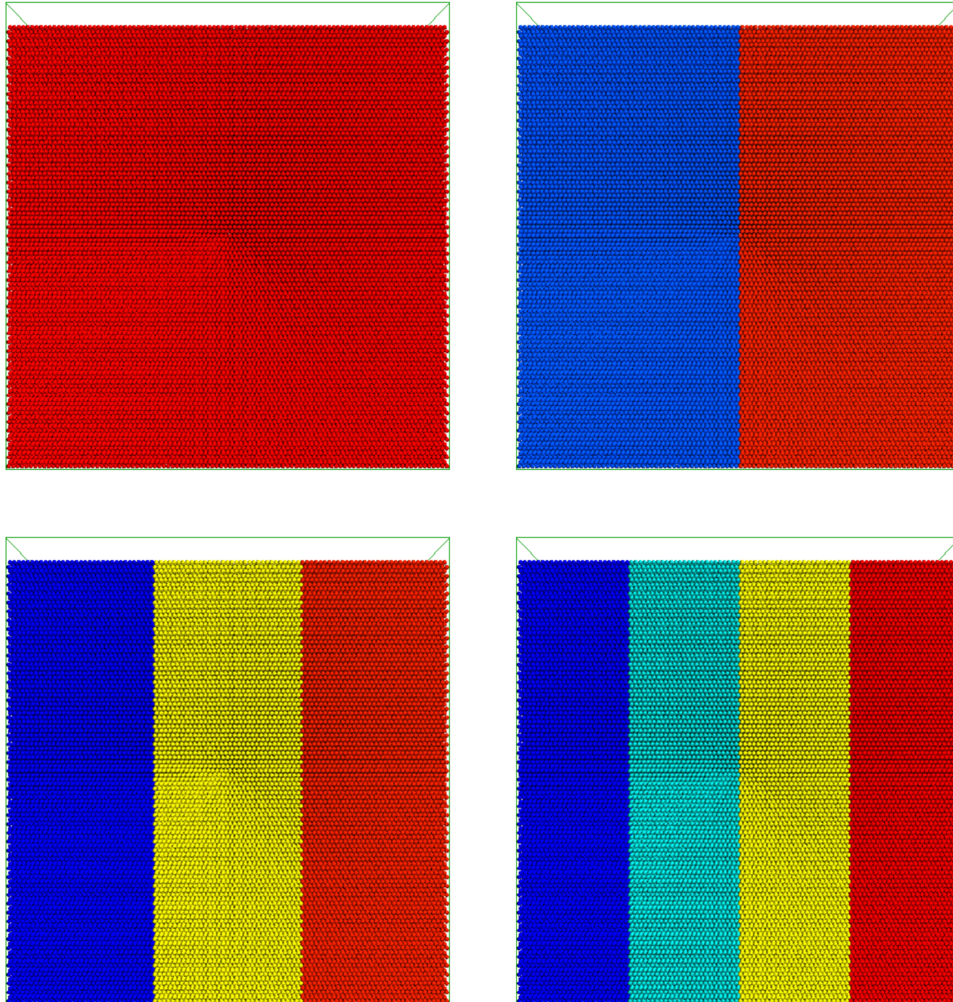

Figure 6.3: Test 1: Sub-domain division for the 1 million particles example. From one GPU, on the upper-left, to four GPUs on the lower right. For each configuration, all sub-domains are equal in the amount of particles they contain. (Source: RCPE GmbH)

6.3.1 Expectations

From a extremely naive sight, one might expect that the achieved performance gain scales directly with the number of used GPUs. In practice, communication and synchronization overhead has to be considered. As shown in the implementation chapter 5 there are several synchronization points while processing a single time step. Moreover, particles inside the

halos and those leaving their domain have to be send to the neighbor domains. As a matter of fact, it has to be known that all halos will always be full of particles in the supposed test case. Therefore the practical speed-up will always be lower than the number of used GPUs.

6.3.2 Results

At first it has to be noted, that there exists a natural relation between the simulation performance and the number of particles in the simulation. This is as long true, as the GPUs utilization stays over a certain level. As shown by the results for the average performance in table 6.5 and the simulation wall clock times in table 6.6 this behavior arises here as well.

The simulation starts at time zero and is carried out for one second of process time. At the beginning, all particles are loosely packed, thus there are no interactions between the spheres at all. This makes the example not really computational intense at this point. Since gravity is acting on the particles, they are falling down until the lowest ones rebound at the boxes bottom side. From this time on, particles start to interact with each other. This increases the computational effort dramatically, lowering the simulation speed. Due to the rebound forces, particles start to diverge again for a short time. At the process's end time, the particles are packed together very closely. There is not much motion left, but the number of particle interactions reaches a maximum. Consequently, the simulation reached a steady state, which is extremely computational intensive. The described behavior can be seen in figure 6.2 visualizing the test case. Additionally, the calculation costs at a specific point in process time can be derived from the figures showing the evolution of processed simulation steps per second, which are:

- Figure 6.4 for the 1 million particles example
- Figure 6.5 for the 10 million particles example
- Figure 6.6 for the 50 million particles example
- Figure 6.7 for the 100 million particles example
- Figure 6.8 for the 230 and the 300 million particles example

Table 6.5 shows the average of the processed simulation steps per second for the performance test cases, while table 6.6 shows the total wall clock times of the test runs. Timings are measured from program start until the end of the simulation process. Therefore, set-up and cleanup time consumptions are also included.

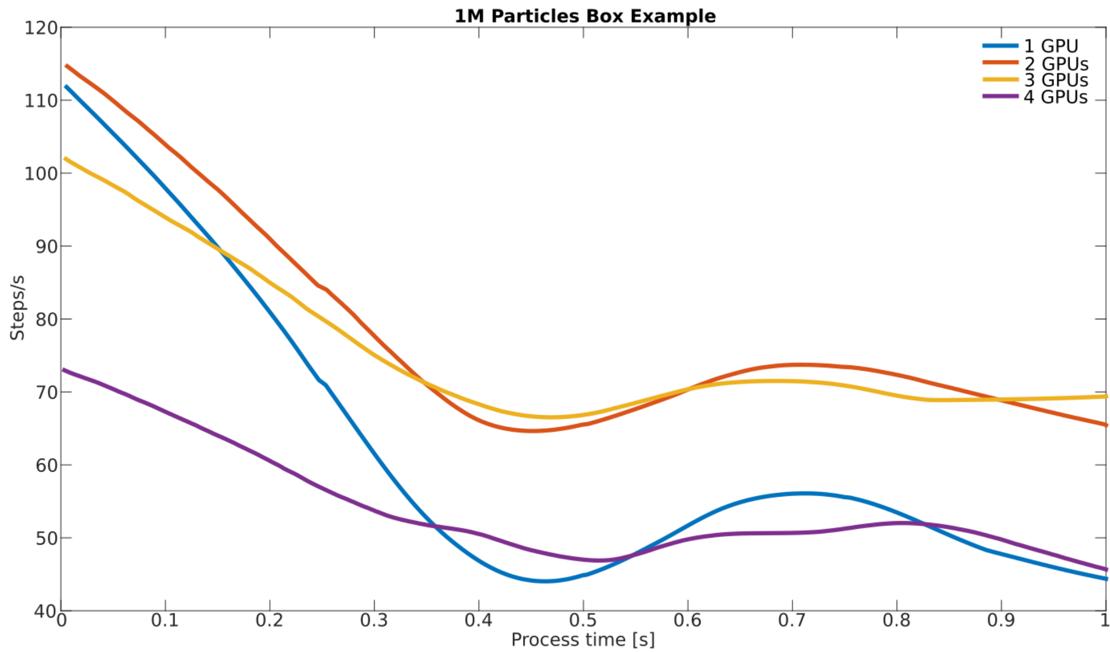

Figure 6.4: Test 1: Evolution of processed simulation steps per seconds over the process time for 1 million particles. (Source: RCPE GmbH)

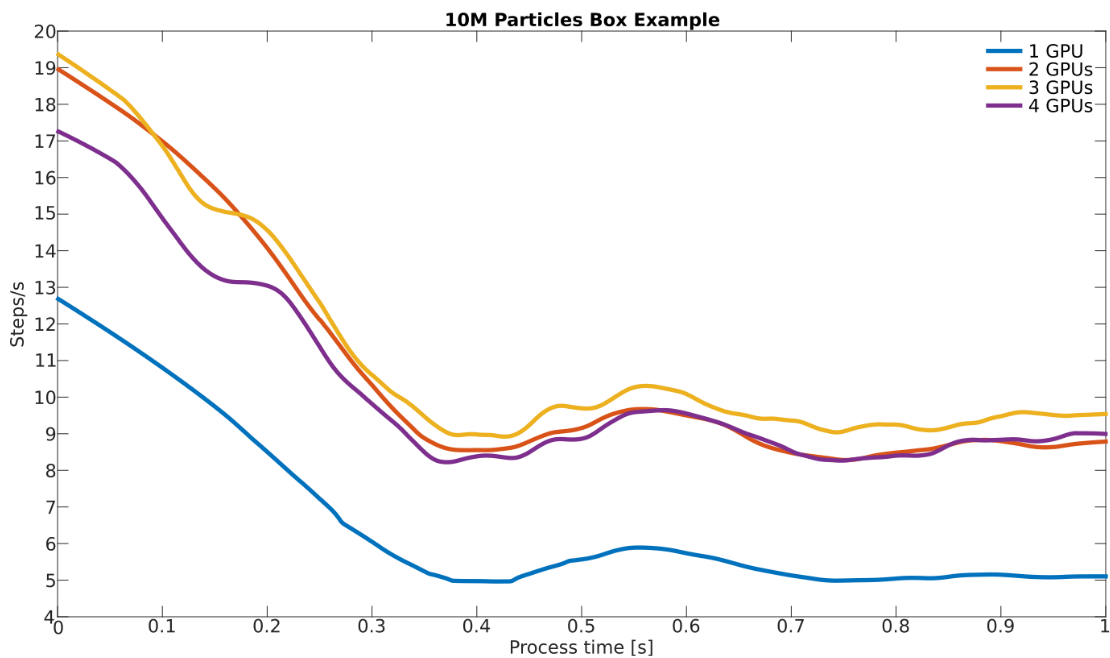

Figure 6.5: Test 1: Evolution of processed simulation steps per seconds over the process time for 10 million particles. (Source: RCPE GmbH)

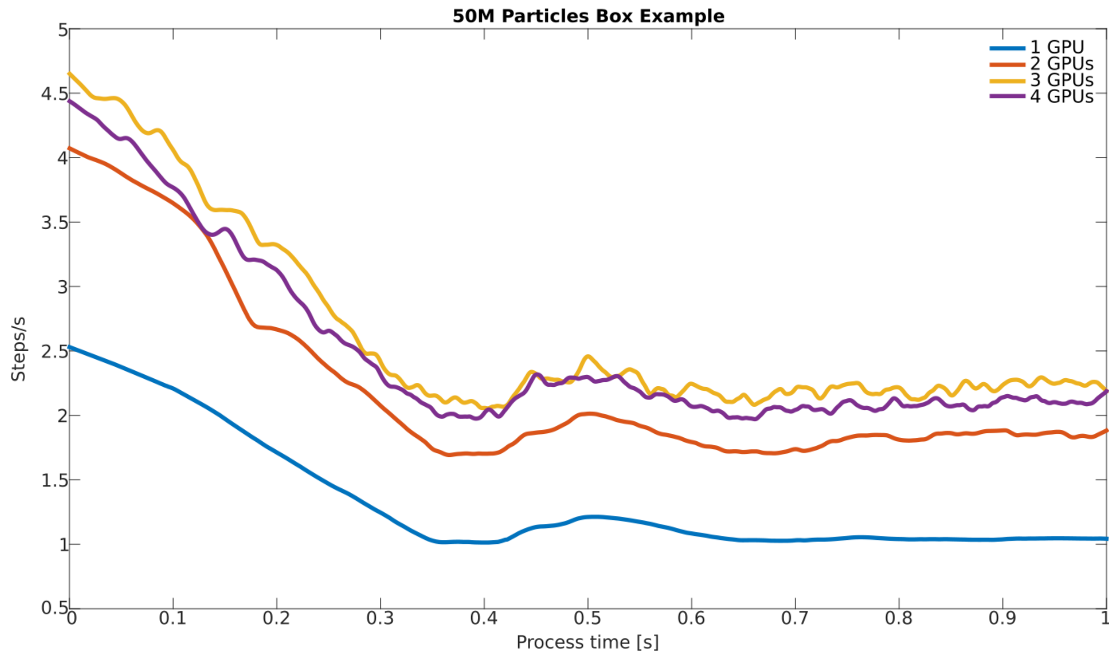

Figure 6.6: Test 1: Evolution of processed simulation steps per seconds over the process time for 50 million particles. (Source: RCPE GmbH)

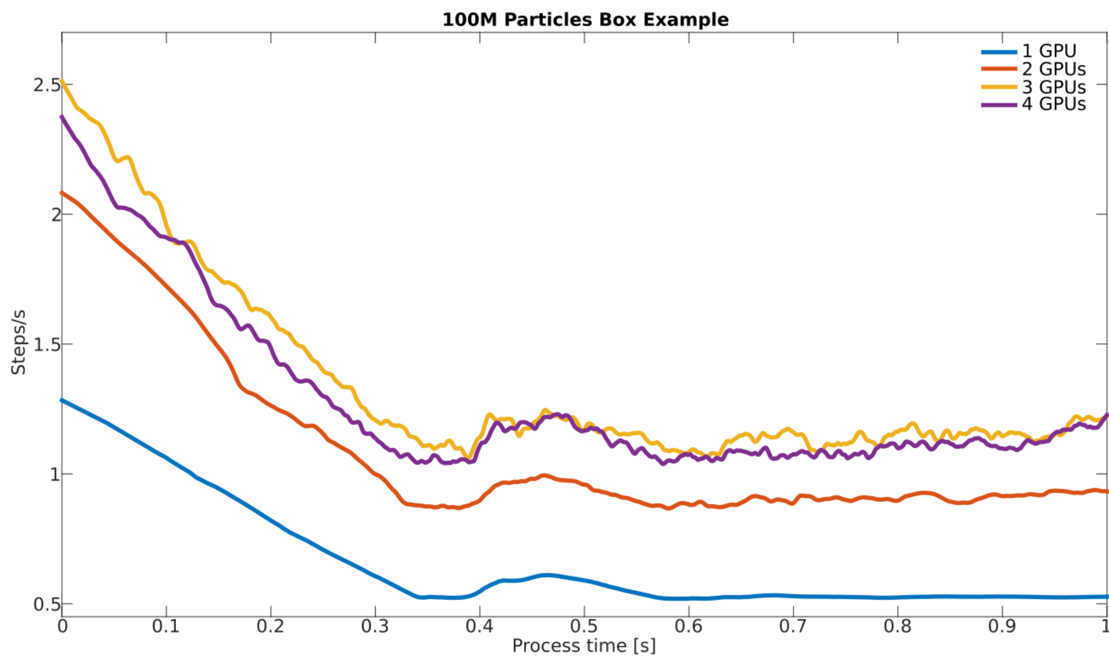

Figure 6.7: Test 1: Evolution of processed simulation steps per seconds over the process time for 100 million particles. (Source: RCPE GmbH)

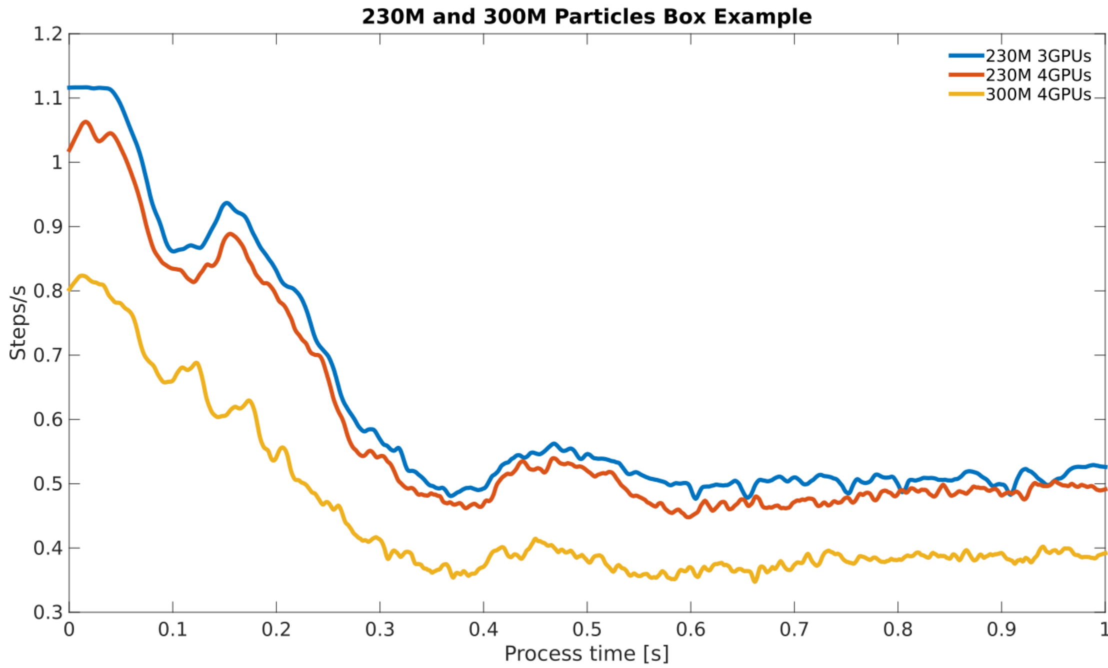

Figure 6.8: Test 1: Evolution of processed simulation steps per seconds over the process time for 230 and 300 million particles. (Source: RCPE GmbH)

Number of particles [million]	Average steps/s			
	1 GPU	2 GPUs	3 GPUs	4 GPUs
1	56.62	75.29	73.37	53.12
10	6.03	10.05	10.65	9.82
50	1.24	2.07	2.48	2.37
100	0.62	1.03	1.27	1.23
230			0.56	0.55
300				0.43

Table 6.5: Test 1: Average simulation steps per second.

Number of particles [million]	Runtime in minutes			
	1 GPU	2 GPUs	3 GPUs	4 GPUs
1	29.5	22.25	22.75	31.5
10	277	166.5	157	175.25
50	1362.5	810	675.25	707.75
100	2725.25	1632.5	1318.5	1363.25
230			2999.5	3029.5
300				3912.25

Table 6.6: Test 1: Wall clock times.

6.3.3 Discussion

To achieve a high speed up it is not only necessary to have many GPUs working on the problem, moreover it has to be assured that all of them are utilized to a level as high as possible. This is all the more true when there is CPU work that needs to be hidden. In case of Multi-GPU this is mostly communication overhead. For the used test case it has to be said, that all of the halos always have nearly the same amount of particles in it, regardless of how many GPUs are used. Additionally, this value stays approximately constant over simulation time. Therefore, the communication overhead can be seen of somewhat constant for the used test case. This is a lucky circumstance, because the results will show performance loss if the communication gets prevalence.

As it can be seen in figure 6.9 the 1 million particles example does not benefit from using more than two Titan X GPUs. Even worse the speed up gets negative in this example when the problem is split to four GPUs. By using two GPUs, the 10, 50 and 100 million particles examples show pretty good behavior in terms of speed-up. All of them increased their performance as expected by about 67%. This means the communication overhead is about 30%. If we use more GPUs the communication overhead gets more and more important. This is, because we have now less particles in the domains on the one hand, but have more inter-domain halos, on the other hand, with the same amount of particles as before.

By using three GPUs the 10 million particles example does not gain a high speed-up any longer, here only 10%. Otherwise the 50 and 100 million particles examples gain another 33% to 40%. This is absolutely fine and within the expectations. By adding a fourth GPU working on the problems, we will see a drop in performance in all of the examples pictured here. For the 1 million particles case it even gets negative. Figure 6.10 shows the situation for three GPUs. It can be seen that the longest running kernel, which is colliding all particles inside the domain and painted here in turquoise, is called on a regular basis. So the same kernel is running on all GPUs within all worker threads at the same time. When running this example with four GPUs a problem seems to raise up. About every three to six time-steps, one or two workers are scheduled lately as marked by the red boxes in figure 6.11. This is causing a massive performance drop because the other two have to wait for them, at the next synchronization point. This bad scheduling behavior reveals a big problem of the supposed implementation, which occurs on the used test system, at least.

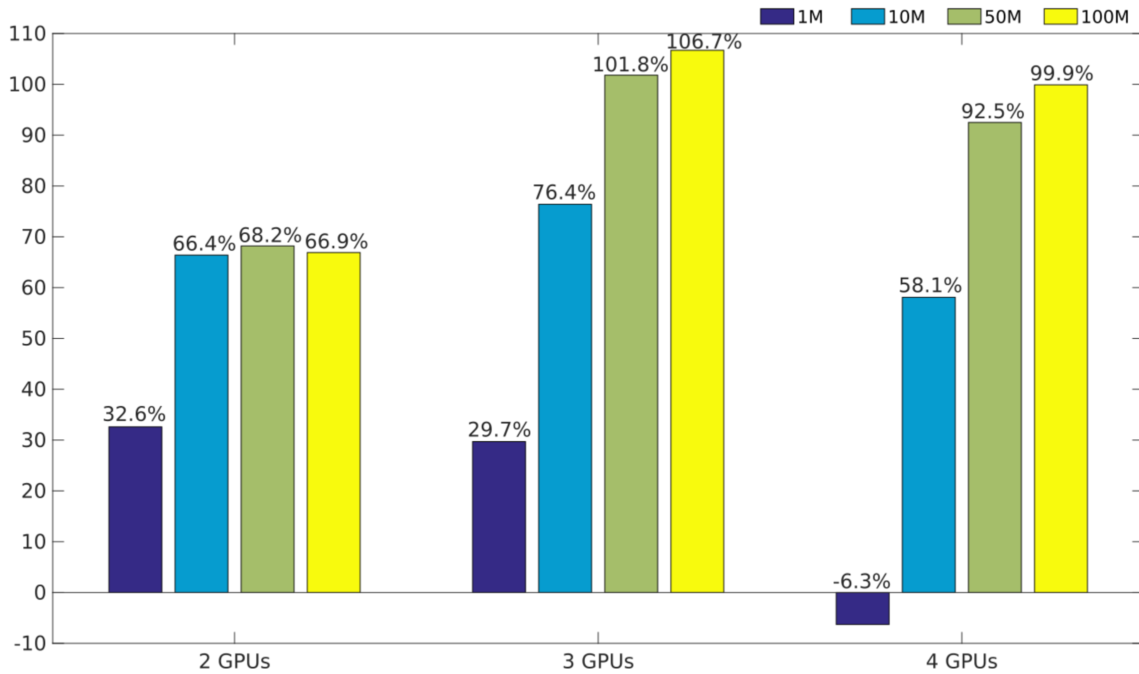

Figure 6.9: Test 1: Performance gain in percent compared from using 1 GPU to 2, 3 and 4 GPUs. (Source: RCPE GmbH)

Figure 6.10: Test 1: Kernel schedule analysis for the 100 million particles example running on three GPUs. All workers executed the same kernel at the same time. Therefore, there are no huge synchronization problems expected. (Source: RCPE GmbH)

6.4 Test 2: Reaching The Test System's Limits

Although, gaining a speed up in performance is fine enough, the supposed implementation also offers the possibility to calculate simulation containing many more particles. By having our four Titan X cards in the test system, providing a total memory amount of 48 GiB, a simulation with 600 million will theoretically fit on them. As shown in table

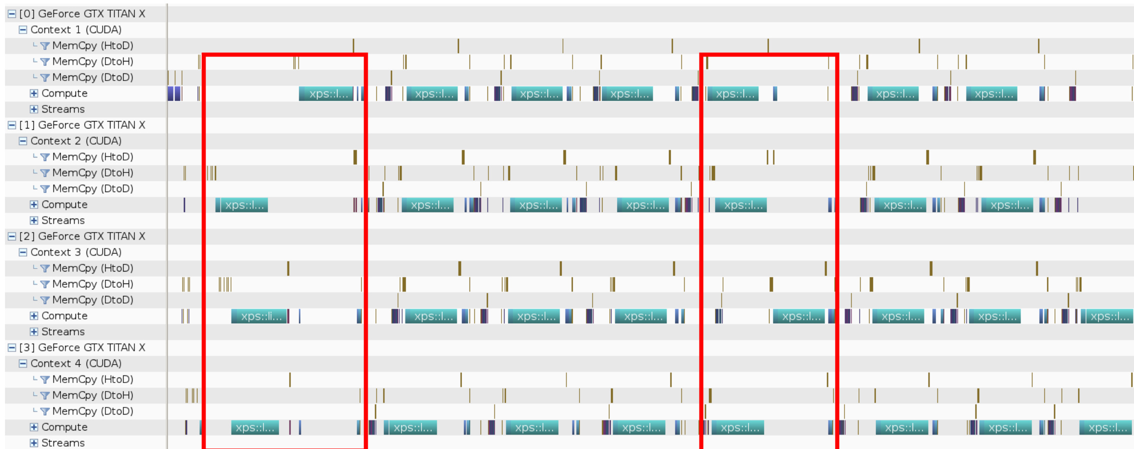

Figure 6.11: Test 1: Kernel schedule analysis for the 100 million particles example running on four GPUs. Compared to figure 6.10 scheduling problems do occur here. Some of the kernels are deferred, causing other workers to wait for them. (Source: RCPE GmbH)

6.3 that will require about 42 GiB of memory in total. This is for the particles and cells only. As mentioned before, there are some performance boosting procedures used which are consuming additional memory. By having them in the code the total memory consumption will be around 60 GiB in total, so the example will be not run-able on the test system. But by eliminating those procedures and doing some other memory saving tricks, like increasing the grid size dramatically, it is possible to get the 600 million particle simulation running. Listing 6.2 shows the memory consumption on the four GPUs while running the 600 million example.

```
+-----------------------------------------------------------------+
| Processes:                                           GPU Memory |
|  GPU       PID   Type   Process name                 Usage      |
|=================================================================|
|    0     25211      C   xps                            10067MiB |
|    1     25211      C   xps                            10196MiB |
|    2     25211      C   xps                            10196MiB |
|    3     25211      C   xps                            10069MiB |
+-----------------------------------------------------------------+
```

Listing 6.2: Test 2: Part of nvidia-smi output, showing total memory usage of the 600 million particles test case.

Although this example was executable it runs terrible slow under the given conditions, this is because there where no performance optimization done. Given an average of less than 0.1 simulation steps per second, on 100% utilization of all four GPUs, the example

was only executed for a short time. This is because, other, more important simulations, needed to be preferred. By extrapolating the values, from the short run, a single second of process time would take about two weeks to be calculated.

Chapter 7

Conclusion and Outlook

The goals of this thesis were to increase the usability of the XPS software package and speed up the included CUDA® based DEM implementation. On the one hand, the implementation for sharing the simulation results, implemented in the first task of this thesis, reveals more possibilities for the program usability. A simple viewer, running completely independent of the simulation process, was implemented showing the simulation state in real-time. Additionally, any consumer might be implemented giving even more possibilities to work on current data, like doing analysis steps, transferring data over networks or creating live pictures.

The implementation of the multi-GPU version of the underlying DEM solver gives a very good speed-up for ordinary simulation examples. Additionally the total number of particles that can be used in one single run, was increased massively as shown in the results by up to 600 million by using four TitanX, for example.

7.1 Future Work

Kureck [Kur16] showed how the implementation of his *(Linear-) Bounding Volume Hierarchy* work gained a speed-up as well, especially for non-uniformed particles. So based on his work and this thesis a combined Multi-GPU solver can be implemented which benefits from both. Assuming such an implementation even more realistic real-world problems might be simulated in future.

By exploiting the shared data task, a network transfer stack for real-time simulation results and many more analysis tools are easily to implement. Making the whole XPS software package even more user friendly and adoptable to new fields.

Appendix A

Acronyms and Glossaries

Acronyms

API	Application Programming Interface	7
BOOST	The BOOST C++ libraries	5
BVH	Bounding Volume Hierarchy	29
CFD	Computational Fluid Dynamics	2
CPU	Central Processing Unit	2
CUDA®	Compute Unified Device Architecture	2
DEM	Discrete Element Method	IX
GPU	Graphics Processing Unit	IX
IPC	Inter-process Communication	5
MPI	Massage-Passing Interface	16
RAM	Random access memory	18
RCPE	Research Center Pharmaceutical Engineering	1
SIMT	Unique architecture employed by a SM. It creates, manages and schedules threads of a warp.	14
SM	Streaming Multiprocessor (SM, SMX)	9
SPH	Smoothed Particle Hydrodynamics	2
STL	STereoLithography	26
XPS	eXtended Particle System	2

Glossary

Term	Definition
Application Programming Interface	An Application Programming Interface is a set of routines, protocols and tools for building software and applications. 7
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. 2
Central Processing Unit	A integrated electronic circuit (processor) performing logical, control and I/O operations. 2
Discrete Element Method	A numerical simulation method for computing stresses and motion of arbitrary particle systems. IX
Event	In CUDA® a event determines a synchronization point within a Stream when recorded. 13
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. 2
Graphics Processing Unit	A specialized electronic circuit (processor) mostly used to compute graphics and images. IX
hash value	A hash value is the result of a certain hashing algorithm that can be performed on an arbitrary data source. A hashing function generates a byte array of a specific length to store a "thumbprint" of the source data. Well known hash functions are defined by the MD family (e.g. MD5) or the SHA family (e.g. SHA-1, SHA-256). 29

Massage-Passing Interface	Massage-Passing Interface is a portable and standardized system for IPC. 16
NVIDIA	The NVIDIA Corporation 2
Smoothed Particle Hydrodynamics	A numerical simulation method for computing fluid flows, based on particles. 2
Stream	In CUDA® a Stream determines a sequence of operations that execute in the same order as they are issued on the GPU. 11
Thrust	Thrust is a C++ template library for CUDA, based on the Standard Template Library (STL). 16

Bibliography

[CS79] P. A. Cundall and O. D. L. Strack. A Discrete Numerical Model For Granular Assemblies. *Géotechnique*, 29(1):47–65, 1979.

[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.

[DCVB+13] J.M. Domnguez, A.J.C. Crespo, D. Valdez-Balderas, B.D. Rogers, and M. Gmez-Gesteira. New multi-GPU Implementation for Smoothed Particle Hydrodynamics on Heterogeneous Clusters. *Computer Physics Communications*, 184(8):1848 – 1860, 2013.

[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).

[Har12] Mark Harris. How to Implement Performance Metrics in CUDA C/C++. `https://devblogs.nvidia.com/parallelforall/how-implement-performance-metrics-cuda-cc/`, 2012. Accessed: 2016-02-14.

[Jon08] M. Tim Jones. Anatomy of Linux dynamic libraries. `http://www.ibm.com/developerworks/library/l-dynamic-libraries/`, 8 2008. Accessed: 2016-02-14.

[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.

[Kar12] 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.

[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.

[Kur16] Hermann Kureck. Design and Implementation of Massively Parallel Algorithms on GPUs for Particle Simulation. Master's thesis, Graz University of Technology, 2016.

[MS01] Mark Mitchell and Alex Samuel. *Advanced Linux Programming*. New Riders Publishing, Thousand Oaks, CA, USA, 2001.

[Neu13] Georg Neubauer. XPS - A GPU based framework for coupled particlefluid simulation methods. B.S. Thesis, Graz University of Technology, 2013.

[NVI15] NVIDIA Corporation, 2701 San Tomas Expressway; Santa Clara, CA 95050. *Thrust Quick Start Guide*, version 7.5 edition, 2015.

[NVI18] NVIDIA Corporation, 2701 San Tomas Expressway; Santa Clara, CA 95050. *NVIDIA CUDA C Programming Guide*, version 9.1 edition, 2018. Accessed: 2018-02-05.

[oT15] University of Tennessee. *MPI: A Message-Passing Interface Standard*. Forum, Message Passing, Knoxville, TN, USA, 2015. http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[Rad06] Charles Radeke. *Statistische und mechanische Analyse der Kr"afte und Bruchfestigkeit von dicht gepackten granularen Medien unter mechanischer Belastung*. PhD thesis, Fakult"at f"ur Mathematik und Informatik der TU Bergakademie Freiberg, 2006.

[RBH+14] Eugenio Rustico, Giuseppe Bilotta, Alexis Herault, Ciro Del Negro, and Giovanni Gallo. Advances in Multi-GPU Smoothed Particle Hydrodynamics Simulations. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):43–52, 2014.

[Ren11] Steve Rennich. CUDA C/C++ Streams and Concurrency. `http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf`, 2011. Accessed: 2016-02-14.

[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.

[TB14] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems.* Prentice Hall Press, Upper Saddle River, NJ, USA, 4th edition, 2014.

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

[Yua13] Yuan Tian, Ji Qi, Junjie Lai, Qingguo Zhou, Lei Yang. A Heterogeneous CPU-GPU Implementation for Discrete Elements Simulation with Multiple GPUs. 2013.