PARALLEL MULTI-BODY DYNAMICS ON GRAPHICS PROCESSING UNIT (GPU) CARDS

by

Hammad Mazhar

A thesis submitted in partial fulfillment of
the requirements for the degree of

Master of Science

(Mechanical Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2012
Approved:

_____________________

Professor Dan Negrut
Department of Mechanical Engineering
University of Wisconsin – Madison
ACKNOWLEDGMENTS

I would like to first and foremost thank my advisor Prof. Dan Negrut, for his support and invaluable words of advice. I also wanted to acknowledge the help I received from my peers in the Simulation Based Engineering Laboratory. Specifically the help of Toby Heyn, Dan Melanz, Arman Pazouki, Andrew Seidl and Justin Madsen. I would also like to thank Marco Quadrelli, Abhi Jain, and the DARTS Laboratory at the NASA Jet Propulsion Laboratory (JPL) for giving me an interesting problem to look at as part of my thesis. Finally I would like to thank my committee members, Professor Gregory F. Nellis, Professor Heidi-Lynn Ploeg, and Dr. David Lamb for their help.
# TABLE OF CONTENTS

Table of Contents ii  
List of Tables v  
List of Figures vi  
List of Algorithms viii  

1 Introduction 1  
1.1 Parallel Architectures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2  
1.2 Computing on the GPU . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2  
1.3 Thesis Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4  

2 Differential Variational Inequality Solver 6  
2.1 Problem Formulation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6  
2.2 Constraint Formulation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7  
2.2.1 Bilateral Constraints . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7  
2.2.2 Unilateral Constraints . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8  
2.2.3 Friction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8  
2.2.4 The Overall Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10  
2.3 Time-Stepping Scheme . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12  
2.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17  

3 Collision Detection Algorithm 18  
3.1 Broad-Phase Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23  
3.1.1 Axis Aligned Bounding Boxes . . . . . . . . . . . . . . . . . . . . . . . . . 23  
3.1.2 Spatial Subdivision Algorithm . . . . . . . . . . . . . . . . . . . . . . . . 24  
3.2 Narrow-Phase Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27  
3.2.1 Support Mappings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28  
3.2.2 Minkowski Difference . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28  
3.2.3 Minkowski Portal Refinement . . . . . . . . . . . . . . . . . . . . . . . . . 29  
3.2.4 Determining Contact Information . . . . . . . . . . . . . . . . . . . . . . 33  
3.3 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33  

4 Parallel Implementation of DVI Solver 34  
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34  
4.2 Solving the CCP in Parallel . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34  
4.3 Data Structures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36  
4.4 Complete Time Stepping . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39  
4.4.1 Stage 1: Pre-processing . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40  
4.4.2 Stage 2: CCP Loop . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.5 Performance Optimizations ........................................ 43

5 Parallel Implementation of Collision Detection 45
5.1 Introduction ......................................................... 45
5.2 Data Structures ..................................................... 45
5.3 Parallel Broad-Phase ............................................. 46
  5.3.1 Stage 1: AABB Generation .................................... 47
  5.3.2 Stage 2: Computing Bounds ................................ 49
  5.3.3 Stage 3: Offset AABB ......................................... 49
  5.3.4 Stage 4: Count AABB-to-Bin Intersections ................ 50
  5.3.5 Stage 5: Get Offsets and Total Intersections .............. 51
  5.3.6 Stage 6: Store AABB-to-Bin Intersections ................ 51
  5.3.7 Stage 7: Sort Intersections by Bin Number ................ 52
  5.3.8 Stage 8: Get Total Number of Active Bins .............. 52
  5.3.9 Stage 9: Inclusive Scan of AABBs per Bin to get Bin Offsets ... 53
  5.3.10 Stage 10: Per Bin Determine Number of AABB-AABB Contacts .... 53
  5.3.11 Stage 11: Inclusive Scan to get Offsets and Total Contacts .... 54
  5.3.12 Stage 12: Determine AABB AABB Contact Pairs ........... 55
  5.3.13 Stage 13: Determine Unique Contact Pairs ............... 55
5.4 Narrow-Phase .................................................... 56
5.5 Summary .......................................................... 58

6 Numerical Validation 59
6.1 Experimental Validation: Mass Flow Rate ....................... 59
  6.1.1 Experimental Model .......................................... 59
  6.1.2 Simulation Model ............................................. 60
  6.1.3 Results ....................................................... 62
6.2 Summary .......................................................... 64

7 Scaling Analysis and CPU vs GPU Comparison 66
7.1 CPU vs GPU Comparison .......................................... 66
7.2 Scaling Analysis .................................................. 67

8 Application 1: Anchoring in Granular Material 69
8.1 Introduction ........................................................ 69
8.2 Anchor Model ..................................................... 69
8.3 Numerical Experiments .......................................... 70
8.4 Results ............................................................ 70
8.5 Summary .......................................................... 72

9 Application 2: Tracked Vehicle on Granular Terrain 75
9.1 Introduction ........................................................ 75
9.2 Model Description ............................................... 75
9.3 Simulation Scenarios ............................................. 77
9.4 Results ................................................................. 79
9.5 Summary ............................................................ 79

10 Conclusions and Future Work 82

A Common Functions 84
A.1 Quaternion Math ................................................. 84

B Thrust 85
B.1 Introduction ..................................................... 85
B.2 Basic Algorithms ................................................ 85
  B.2.1 Reductions ...................................................... 85
  B.2.2 Prefix-Sums .................................................... 86
  B.2.3 Sorting ........................................................... 86
B.3 Miscellaneous ..................................................... 86
  B.3.1 Sequence ........................................................ 86
  B.3.2 Fill ............................................................... 86
  B.3.3 Iterators ........................................................ 87
  B.3.4 Unique .......................................................... 87
  B.3.5 Remove ........................................................ 87
  B.3.6 Transformations .............................................. 88

References 89
LIST OF TABLES

3.1 Support mappings, for a more complete list see [1] . . . . . . . . . . . . . . 29
5.1 Information required to define each type of shape . . . . . . . . . . . . . . . 46
# List of Figures

1.1 Million-body particle simulation .................................................. 5
2.1 2-Dimensional Representation of Friction Cone ................................. 10
2.2 Contact $i$ between two bodies $A, B \in \{1, 2, \ldots, n_b\}$ .................. 12
3.1 Double Track Simulation on Granular Terrain [2, 3] ............................ 19
3.2 Simulation of a ball floating on granular material in a wave tank .......... 19
3.3 Collision detection using overlap on all three principal axes ................. 21
3.4 Track Shoe Decomposed into spheres ............................................ 21
3.5 Example of 3D space divided into bins .......................................... 23
3.6 Example of AABB generation for 3D cylinder .................................. 24
3.7 Example of Computing Bounds ..................................................... 25
3.8 Example of a 2D space, containing 3 AABBs, divided into bins. B contains the list of bodies with associated bin intersections ......................... 26
3.9 Sort based on the Bin ID to so that the objects in each bin are known ..... 26
3.10 List of Potential Contacts ......................................................... 27
3.11 Example of a support point computed for a given vector $n$ ................. 28
3.12 Example of a Minkowski difference, $A \ominus B$, on two convex sets $A$ and $B$ ................................................................. 30
4.1 The concept which inspires the reduction algorithm. Sums are performed with a binary tree, to exploit the parallel nature of the stream processors. ...... 35
4.2 Different buffers used for contacts, constraints, and bodies .................. 37
4.3 Buffers needed to determine offset locations for each constraint ............. 41
4.4 Example with 4 constraints between 5 different bodies ....................... 42
4.5 $X$ is sorted by body number ..................................................... 42
4.6 $Y$ is sorted by update number .................................................... 42
4.7 Mapping of update onto offset buffer .......................................... 42
5.1 Data required for a collision shape .............................................. 45
5.2 Example of inclusive scan, note that the first element is not included in the output ................................................................. 51
6.1 Weight vs time for a gap size of 3 mm ........................................... 63
6.2 Weight vs time for a gap size of 2.5 mm ....................................... 63
6.3 Weight vs time for a gap size of 2 mm .......................................... 64
6.4 Weight vs time for a gap size of 1.5 mm ....................................... 65
7.1 Scaling of the CPU vs GPU for different amounts of bodies ................ 67
7.2 Speedup of CPU vs GPU .......................................................... 68
7.3 Scaling of the GPU for different amounts of bodies ............................ 68
8.1 Stages of anchoring, first anchor drops onto granular bed (a), then an applied torque anchors the anchor (b,c), after which it is pulled out (d).

8.2 Anchor with different applied torques and a constant pullout force of 300N.

8.3 Anchor with different pullout forces and a constant torque of 1000N.

8.4 Closeup of Fig. 8.3, Anchor with different pullout forces and a constant torque of 1000N.

9.1 Track model used for simulation.

9.2 Bed of granular material.

9.3 Tank on inclined surface, geometry of tank model is purely cosmetic.

9.4 Tank on a bed of granular material comprised of spheres of varying radii, geometry of tank model is purely cosmetic.

9.5 Magnitude of force experienced by one revolute joint.

9.6 Magnitude of force experienced by 5 revolute joints.

10.1 Simulation of tower collapsing with a granular flow.
## LIST OF ALGORITHMS

<table>
<thead>
<tr>
<th></th>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DVI: Inner Iteration Loop</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>DVI: Outer, Time-Stepping, Loop</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>MPR: Minkowski Portal Refinement, Phases</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>MPR: Minkowski Portal Refinement, Phase 2 Details</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>Complete Time Stepping, when GPU is Available</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>Constraint Solver, Pre-processing Stage</td>
<td>42</td>
</tr>
<tr>
<td>7</td>
<td>Constraint solver Iteration loop</td>
<td>43</td>
</tr>
<tr>
<td>8</td>
<td>Broad-Phase Stages</td>
<td>47</td>
</tr>
</tbody>
</table>
PARALLEL MULTI-BODY DYNAMICS ON GRAPHICS PROCESSING UNIT (GPU) CARDS

Hammad Mazhar

Under the supervision of Dr. Dan Negrut
At the University of Wisconsin-Madison

Engineers are increasingly relying on simulation to augment and, in some cases, replace experimental tasks. However, current simulation capabilities are often inadequate to capture phenomena of interest. In tracked vehicle analysis, for example, the track-terrain interaction has been difficult to characterize through simulation. This work addresses these limitations by developing a simulation framework that is capable of simulating up to one million bodies interacting through contact and friction by drawing on the parallel computing power of the graphics processing unit (GPU) card. With this simulation capability, it is possible to simulate tracked vehicles operating on terrain modeled as discrete granular particles rather than empirical formulas. This simulation capability is achieved through the implementation on the GPU of both the solution of the collision detection, and the solution of the multi-body dynamics problems. The collision detection problem is solved through the use of a spatial subdivision method that can quickly find collisions between convex geometries. The multi-body dynamics problem is posed as a set of differential variational inequality (DVI) equations, the solution of which leads to a cone complementarity problem (CCP) solved in parallel by an iterative method. These tools are combined to simulate a variety of applications such as anchoring in granular material and a tracked vehicle operating on granular terrain. The methods proposed have been implemented as a set of parallel algorithms leveraging NVIDIA’s Compute Unified Device Architecture (CUDA).
1 INTRODUCTION

Within traditional architectures (personal computers with a single CPU), multi-body systems with a very large number of components can lead to prohibitively long simulation times. This may happen, for instance, when simulating mechanisms with many frictional contacts such as in part feeders, conveyor belts, or size-segregation shakers. Larger problems involving millions or billions of parts such as granular flows or powder mechanics (relevant, for example, in pharmaceutical and mining industries) remain intractable due to the excessive computational overhead that their simulation can require [4].

In such cases of extreme complexity, see Fig. 1.1 for an example, parallel computing is the only viable alternative. However, the type of parallelism supported on high-end supercomputers has two major drawbacks: first, there is the price barrier, which makes these high-end supercomputers available almost exclusively to large organizations; second, programming these unique machines requires great expertise due to limited productivity tools such as profilers and debuggers. However, the increasing diffusion of intermediate high-performance-computing solutions based on small clusters of computers is making supercomputing more affordable, especially when exploiting the computational power of the multiprocessors that are used in recent graphical processing units (GPUs). GPU technology can boost the computational power of small computing centers, at a cost of a few graphic adapters [5]. Also, during the last years some GPU vendors introduced application programming interfaces, such as CUDA from NVIDIA [6], that ease the implementation of parallel algorithms to be executed on GPU boards [7].

The ability of chip manufacturers to produce processors with significantly higher clock frequencies has been drastically diminished during the last decade. Therefore future increases in computational power are expected to be the outcome of an increase in the number of parallel compute cores, as illustrated by GPU technology and its impact in scientific computing. This trend justifies the effort of implementing parallel algorithms
today, even if one does not necessarily aim at massive simulations, because the algorithms will scale well on tomorrow’s hardware architectures.

1.1 Parallel Architectures

Early attempts to implement parallel methods for multi-body problems were mostly concerned with the efficient simulation on custom parallel computers of simple mechanisms such as vehicles, especially where man-in-the-loop scenarios imposed real-time requirements [8]. In those implementations the formulation was recursive and one of the most critical issues was the partitioning of the constraint tree because, given the limited computational resources at the time, suboptimal partitioning of the few constraints would have had a major impact on performance. Recent solutions based on computer clusters allowed cheaper and more general-purpose implementations, for instance in [9] where the workload is distributed on a network of multiple PCs or on Symmetric Multi-Processing (SMP) architectures using the Message-Passing Interface (MPI) standard. Recently, Iglberger et al. used a BlueGene supercomputer (a cluster with 131,072 nodes) to perform a multi-body simulation with two billion particles [10]; the software performs a lattice domain decomposition and exploits MPI as a method for exchanging data between the domains. One of the first attempts to parallelize multi-body simulation on GPU boards is discussed in [11, 12]. Although a single GPU board cannot handle millions or billions of rigid bodies, as in a large supercomputer, this approach proves to be very cost-effective and enables Teraflop grade simulation on a workstation.

1.2 Computing on the GPU

Currently, high-end GPUs offer single precision floating-point parallel computing power close to one Teraflop, exceeding that of multi-core CPUs. This computational resource,
usually devoted to the execution of pixel shading fragments for the rendering of OpenGL or DirectX three dimensional visualization, can be also exploited for scientific computation.

Earlier experiments with scientific computing on the GPU required an intricate programming technique because GPU hardware and software was meant only for real-time graphical visualization. The developer had to use OpenGL calls to reformulate small scientific computation programs in the GLSL shading language native to the graphics board. These programs were executed with data organized in rectangular textures, with RGBA color representing some scientific data. In this way, the output was rendered in parallel, pixel by pixel, by the pixel-shading processors\(^1\) into a frame buffer which was never visualized on the screen; the RGBA colors of that frame buffer would in fact represent the output of the parallel scientific computation.

To alleviate the difficulty of this programming model, NVIDIA proposed in 2006 a development environment, called CUDA [13], which allows general-purpose programming on the GPU. Basically, the programmer can write algorithms using a subset of the C++ language, which can be compiled into machine code and executed on the GPU device. The GPU executes the same kernel on each parallel thread which in turn operates over data structures called streams, hence the name streaming processor. This computational architecture is called SIMT (Single Instruction Multiple Thread), and it can be considered an advanced form of SIMD (Single Instruction Multiple Data) architecture according to the Flynn taxonomy [14]. To efficiently execute hundreds of threads in parallel, GPU multiprocessors are hardware multithreaded: they can manage thousands of concurrent threads with almost zero scheduling overhead, so that hardware switching between threads is used effectively to hide the latency of memory access operations.

For the problem at hand, parts of the multi-body simulation have been ported to the

\(^1\)Earlier models of GPU implemented two kinds of parallel execution units, the pixel processors and the vertex processors; the former were more easily adapted to scientific computing. Modern GPUs, instead, implement a single type of execution units (called streaming processors or thread processors) which can be used for pixel shading, vertex processing, as well as for generic scientific computation.
The two main components, namely the collision detection and the solution of the Cone Complementarity Problem (CCP), have been ported to the GPU as the proposed algorithm fits well into the GPU multithreaded model because the computation can be split in multiple threads each acting on a single contact or kinematic constraint.

NVIDIA’s Tesla family of GPUs was selected as the target hardware for the proposed collision detection algorithm due to affordability and software support. Tesla C2070 has a set of 448 scalar processors, which are organized in groups of 32. Each group, along with associated shared memory (48 KB), a register file that can store 32,768 floats/integers, and a transcendental function unit, making up a stream multiprocessor (SM) [15]. There are 14 SMs in each GPU card, sharing 6 GB of high bandwidth (140 GB/s) memory, which is analogous to RAM in traditional CPU computing. The hardware is managed by the CUDA runtime Application Programming Interface (API) that provides a simple interface against which users can program the GPU. Data exchange between the CPU (the host) and GPU (the device) takes place over a PCIe X16 2.0 connection with an 8 GB/s bandwidth. The CUDA driver can simultaneously manage up to 21,504 computational threads running on the C2070 hardware. While some of these threads are actively executing instructions, others idle waiting for memory transactions or queue for access to the 448 scalar processors. This thread-level time slicing, or context switching, is extremely lightweight in CUDA and very effectively hides the latency associated with global, texture, and constant memory transactions [15].

1.3 Thesis Outline

In most multi-body dynamics simulations, the computational bottlenecks are the collision detection task [16], and the constraint solver [17]. An in-depth explanation about the constraint solver will be provided along with the specifics of the GPU implementation. The collision detection algorithm will also be explained and its implementation details
will be provided. It should be noted that the proposed collision detection algorithm described in chapters 3 and 5, along with the CCP solver described in chapters 2 and 4 was implemented in Chrono::Engine [18]. Validation of the methods described will be presented, both by looking at macro-scale behavior and also by looking at specific bilateral constraint types. Additionally, a comparison between CPU and GPU implementations of the collision detection and DVI solver, along with a GPU scaling analysis of both will be provided. Finally, two real-world applications of the methods described will be demonstrated, specifically an anchoring in granular material and tracked vehicle mobility over granular terrain.
2 DIFFERENTIAL VARIATIONAL INEQUALITY SOLVER

2.1 Problem Formulation

The formulation of the equations of motion, that is the equations that govern the time evolution of a multi-body system, is based on the so-called absolute, or Cartesian, representation of the attitude of each rigid body in the system.

The state of the system is denoted by the generalized positions \( \mathbf{q} = \begin{bmatrix} \mathbf{r}_1^T, \epsilon_1^T, \ldots, \mathbf{r}_{n_b}^T, \epsilon_{n_b}^T \end{bmatrix}^T \in \mathbb{R}^{7n_b} \) and their time derivatives \( \dot{\mathbf{q}} = \begin{bmatrix} \dot{\mathbf{r}}_1^T, \dot{\epsilon}_1^T, \ldots, \dot{\mathbf{r}}_{n_b}^T, \dot{\epsilon}_{n_b}^T \end{bmatrix}^T \in \mathbb{R}^{7n_b} \), where \( n_b \) is the number of bodies, \( \mathbf{r}_j \) is the absolute position of the center of mass of the \( j \)-th body and the quaternions \( \epsilon_j \) are used to represent rotation and to avoid singularities. However, instead of using quaternion derivatives in \( \dot{\mathbf{q}} \), it is more advantageous to work with angular velocities: the method described will use the vector of generalized velocities \( \mathbf{v} = \begin{bmatrix} \dot{\mathbf{r}}_1^T, \bar{\omega}_1^T, \ldots, \dot{\mathbf{r}}_{n_b}^T, \bar{\omega}_{n_b}^T \end{bmatrix}^T \in \mathbb{R}^{6n_b} \). Note that the generalized velocity can be easily obtained as \( \dot{\mathbf{q}} = \mathbf{L}(\mathbf{q})\mathbf{v} \), where \( \mathbf{L} \) is a linear mapping that transforms each \( \bar{\omega}_i \) into the corresponding quaternion derivative \( \dot{\epsilon}_i \) by means of the linear algebra formula \( \dot{\epsilon}_i = \frac{1}{2} \mathbf{G}^T(\mathbf{q})\bar{\omega}_i \), with 3x4 matrix \( \mathbf{G}(\mathbf{q}) \) as defined in [19].

Given the velocities considered, for a system of rigid bodies the generalized mass matrix \( \mathbf{M} \) remains constant and diagonal. Denoting the set of applied, or external, generalized forces by \( \mathbf{f}^A(t, \mathbf{q}, \mathbf{v}) \), the second order differential equations that govern the time evolution of the multi-body system expressed in matrix notation assume the form \( \mathbf{M} \ddot{\mathbf{v}} = \mathbf{f}^A(t, \mathbf{q}, \mathbf{v}) \). Note that this is true only when there are no constraints (either unilateral or bilateral) in the system.
2.2 Constraint Formulation

2.2.1 Bilateral Constraints

Bilateral constraints represent kinematic pairs, for example spherical, prismatic or revolute joints, and can be expressed as holonomic algebraic equations constraining the relative position of two bodies. Assuming a set $\mathcal{B}$ of constraints is present in the system, they lead to a collection of scalar equations:

$$\Psi_i(q,t) = 0, \quad i \in \mathcal{B}. \quad (2.1)$$

For instance, a ball joint requires three of these scalar equations. Assuming smoothness of constraint manifold, $\Psi_i(q,t)$ can be differentiated to obtain the Jacobian $\nabla_q \Psi_i = [\partial \Psi_i / \partial q]^T$.

All bilateral constraints must also be satisfied at the velocity level. This requirement stems from the full time-derivative of the $i$-th constraint equation:

$$\frac{d \Psi_i(q,t)}{dt} = 0 \Rightarrow \frac{\partial \Psi_i}{\partial q} \dot{q} + \frac{\partial \Psi_i}{\partial t} = \nabla_q \Psi_i^T \dot{q} + \frac{\partial \Psi_i}{\partial t} = \nabla_q \Psi_i^T L(q) \mathbf{v} + \frac{\partial \Psi_i}{\partial t} = 0$$

Defining

$$\nabla \Psi_i^T = \nabla_q \Psi_i^T L(q), \quad (2.2)$$

the constraints are consistent at velocity-level provided the following equation is satisfied:

$$\nabla \Psi_i^T \mathbf{v} + \frac{\partial \Psi_i}{\partial t} = 0 \quad (2.3)$$

Note that the $\frac{\partial \Psi_i}{\partial t}$ is nonzero only for rheonomic constraints (motors, actuators, imposed trajectories).
2.2.2 Unilateral Constraints

Given a large number of rigid bodies with different shapes, modern collision detection algorithms are able to efficiently find a set of contact points, that is points where a gap function $\Phi(q)$ can be defined for each pair of near-enough shape features. Where defined, such a gap function must satisfy the non-penetration condition $\Phi(q) \geq 0$ for all contact points.

Note that a signed distance function, differentiable at least up to some value of the interpenetration [20], can be easily defined if bodies are smooth and convex [21]. However, this is not always possible, for instance when dealing with concave or faceted shapes often used to represent parts of mechanical devices. In this case the gap function $\Phi(q)$ can be non-smooth or not well defined. Without loss of generality, for sufficiently small penetration, the following assumption can be made on the geometrical constraints: any contact is described by a gap function $\Phi(q)$ that is twice continuously differentiable. Most often, when one deals with convex geometries and small numerical integration step-sizes, this assumption is easily verified. The proposed implementation uses a GPU-based parallel collision detection algorithm, described in Chapter 3, to find the contact points between convex shapes.

2.2.3 Friction

Friction is introduced for each unilateral contact constraint present in the multi-body system. When a contact $i$ is active, that is $\Phi_i(q) = 0$, a normal force and possibly a tangential force act on each of the two bodies at the contact point. We use the classical Coulomb friction model to define these forces [22]. If the contact is not active,

---

1An efficient way to deal with convex geometries is to represent them as clusters of smaller convex shapes, performing a concave decomposition before the simulation starts. In this way, the convex algorithm can be used on the convex subparts for most geometries without significant impact on the robustness of the method.
that is $\Phi_i(q) > 0$, no friction forces and no normal forces exist. This implies that the mathematical description of the model leads to a complementarity problem [23]. Given two bodies $A$ and $B$ in contact, let $n_i$ be the unit normal at the contact pointing toward the exterior of the body of lower index, which by convention is considered to be body $A$. Let $u_i$ and $w_i$ be two vectors in the contact plane such that $n_i, u_i, w_i \in \mathbb{R}^3$ are mutually orthonormal vectors.

The frictional contact force is impressed on the system by means of multipliers $\gamma_{i,n} \geq 0$, $\gamma_{i,u}$, and $\gamma_{i,w}$, which lead to the normal component of the force $f_{i,N} = \hat{\gamma}_{i,n} n_i$ and the tangential component of the force $f_{i,T} = \hat{\gamma}_{i,u} u_i + \hat{\gamma}_{i,w} w_i$.

The Coulomb model imposes the following nonlinear constraints:

\[
\hat{\gamma}_{i,n} \geq 0, \quad \Phi_i(q) \geq 0, \quad \Phi_i(q) \hat{\gamma}_{i,n} = 0, \\
\mu_i \hat{\gamma}_{i,n} \geq \sqrt{\gamma_{i,u}^2 + \gamma_{i,w}^2}, \quad ||v_{i,T}|| \left(\mu_i \hat{\gamma}_{i,n} - \sqrt{\gamma_{i,u}^2 + \gamma_{i,w}^2}\right) = 0, \\
\langle f_{i,T}, v_{i,T} \rangle = -||f_{i,T}|| ||v_{i,T}||
\]

where $v_{i,T}$ is the relative tangential velocity at contact $i$. Defining $\langle \cdot, \cdot \rangle$ as the inner product of two vectors, the constraint $\langle f_{i,T}, v_{i,T} \rangle = -||f_{i,T}|| ||v_{i,T}||$ requires that the tangential force be opposite to the tangential velocity. Note that the friction force depends on the friction coefficient $\mu_i \in \mathbb{R}^+$. The original Coulomb model distinguishes between static $\mu_s$ and kinetic $\mu_k$ friction coefficients. For simplicity, an assumption is made that these coefficients are the same. If needed, it is possible to extend the method to make this distinction or also consider more complex constitutive equations such as the Stribeck friction model [24].

The first part of the constraint can be restated as

\[
f_i = f_{i,N} + f_{i,T} = \hat{\gamma}_{i,n} n_i + \hat{\gamma}_{i,u} u_i + \hat{\gamma}_{i,w} w_i \in \Upsilon,
\]
where $\Upsilon$ is a cone in three dimensions, whose slope is $\tan^{-1} \mu_i$, see Fig. 2.1. This results in the friction force being dissipative. An equivalent convenient way of expressing this constraint is by using the maximum dissipation principle:

$$
(\hat{\gamma}_{i,u}, \hat{\gamma}_{i,w}) = \arg\min \sqrt{\hat{\gamma}_{i,u}^2 + \hat{\gamma}_{i,w}^2} \leq \mu_i \hat{\gamma}_{i,n} 
$$

In fact, the the first-order necessary Karush-Kuhn-Tucker conditions [25] for the minimization problem in Eq. (2.5) correspond to the Coulomb model above [26, 27].

### 2.2.4 The Overall Model

We assume that at time $t$ several bodies are touching, interpenetrating or separated by a distance smaller than a threshold $\delta > 0$, so that a set $\mathcal{A}$ of relevant contact constraints can be assembled:

$$
\mathcal{A}(\mathbf{q}, \delta) = \{i \mid i \in \{1, 2, \ldots, p\}, \Phi_i(\mathbf{q}) \leq \delta\},
$$

Shapes which are separated by larger distances than the $\delta$ threshold are not considered for frictional contact analysis to minimize computational effort.
It is also assumed that there is a set $\mathcal{B}$ of active bilateral constraints, acting on the rigid bodies. Each constraint $i \in \mathcal{B}$ transmits reactions to the connected bodies by means of a multiplier $\hat{\gamma}_{i,b}$.

Considering the effects of both $\mathcal{A}(q, \delta)$ frictional contacts and $\mathcal{B}$ bilateral constraints, the time evolution of the dynamical system is governed by the following differential problem with set-valued functions and complementarity constraints, which is equivalent to a differential variational inequality [28]:

$$\begin{align*}
\dot{q} &= L(q)v \\
M\dot{v} &= f(t, q, v) + \sum_{i \in \mathcal{A}(q, \delta)} (\hat{\gamma}_{i,n} D_{i,n} + \hat{\gamma}_{i,u} D_{i,u} + \hat{\gamma}_{i,w} D_{i,w}) + \sum_{i \in \mathcal{B}} \hat{\gamma}_{i,b} \nabla \Psi_i \\
i \in \mathcal{B} : \Psi_i(q, t) &= 0 \\
i \in \mathcal{A}(q, \delta) : \hat{\gamma}_{i,n} \geq 0 \quad \perp \Phi_i(q) \geq 0,
\end{align*}$$

(2.6)

The tangent space generators $D_i = [D_{i,n}, D_{i,u}, D_{i,w}] \in \mathbb{R}^{6n_b \times 3}$ are defined as

$$D_i^T = \begin{bmatrix}
0 & \ldots & -A_{i,p}^T A_{i,p} A \tilde{s}_{i,A} & 0 & \ldots & 0 & A_{i,p}^T \tilde{s}_{i,B} & \ldots & 0
\end{bmatrix},$$

(2.7)

where we use $A_{i,p} = [n_i, u_i, w_i]$ as the $\mathbb{R}^{3 \times 3}$ matrix of the local coordinates of the $i$th contact, and introduce the vectors $\tilde{s}_{i,A}$ and $\tilde{s}_{i,B}$ for representing the contact point positions in body-relative coordinates, as illustrated in Fig. (2.2).

The Coulomb model used in this work is the predominant model used in the engineering literature to describe dry friction. Unfortunately, the model may be inconsistent: there exist configurations for which the resulting problem does not have a solution [29, 30]. This situation has led to the need to explore weaker formulations where the forces are measures and Newton’s law is satisfied in a measure differential inclusion sense [30]. It has been shown that solutions in that sense do exist and can be found by time-stepping
schemes [31].

2.3 Time-Stepping Scheme

We formulate the dynamical problem in terms of measure differential inclusions [30], whose numerical solution can be obtained using the following time-stepping scheme based on the solution of a complementarity problem at each time step.

Given a position $\mathbf{q}^{(l)}$ and velocity $\mathbf{v}^{(l)}$ at the time-step $t^{(l)}$, the numerical solution is found at the new time-step $t^{(l+1)} = t^{(l)} + h$ by solving the following optimization problem
with equilibrium constraints [32]:

\[
M(v^{(l+1)} - v^{(l)}) = hf(t^{(l)}, q^{(l)}, v^{(l)}) + \sum_{i \in A(q^{(l)}, \delta)} (\gamma_{i,n} D_{i,n} + \gamma_{i,u} D_{i,u} + \gamma_{i,w} D_{i,w}) + \sum_{i \in B} \gamma_{i,b} \nabla \Psi_i,
\]

(2.8)

\[
i \in B : \frac{1}{h} \Psi_i(q^{(l)}, t) + \nabla \Psi_i^T v^{(l+1)} + \frac{\partial \Psi_i}{\partial t} = 0
\]

(2.9)

\[
i \in A(q^{(l)}, \delta) : 0 \leq \frac{1}{h} \Phi_i(q^{(l)}) + D_{i,n}^T v^{(l+1)} \perp \gamma_{n}^i \geq 0,
\]

(2.10)

\[
(\gamma_{i,u}, \gamma_{i,w}) = \arg\min_{\mu, \gamma_{i,n} \geq \sqrt{\gamma_{i,u}^2 + \gamma_{i,w}^2}} v^T (\gamma_{i,u} D_{i,u} + \gamma_{i,w} D_{i,w}),
\]

(2.11)

\[
q^{(l+1)} = q^{(l)} + hL(q^{(l)})v^{(l+1)}.
\]

(2.12)

Here, \(\gamma_s\) represents the constraint impulse of a contact constraint, that is, \(\gamma_s = h\tilde{\gamma}_s\), for \(s = n, u, w\). The \(\frac{1}{h} \Phi_i(q^{(l)})\) term achieves constraint stabilization, and its effect is discussed in [33]. Similarly, the term \(\frac{1}{h} \Psi_i(q^{(l)})\) achieves stabilization for bilateral constraints. The scheme converges to the solution of a measure differential inclusion [34] when the step size \(h \to 0\).

Several approaches can be used to solve Eqs. (2.8)-(2.11). Some authors suggested to approximate friction cones as faceted pyramids, so that the system of equations above, originally a Nonlinear Complementarity Problem (NCP), turns into a Linear Complementarity Problems (LCP) [22]. The resulting LCPs are solved using algorithms based on the so-called pivoting methods or simplex methods. These numerical approaches that belong to the class of direct methods are computationally expensive, and their complexity class is in the worst case exponential [35].

Alternatively, the problem can be cast as a monotone optimization problem by introducing a relaxation over the complementarity constraints. Specifically, the time-
stepping scheme is modified by replacing Eq. (2.10) with

\[ i \in \mathcal{A}(q^{(l)}, \delta) : 0 \leq \frac{1}{h} \Phi_i(q^{(l)}) + D_{i,n}^T v^{(l+1)} - \mu_i \sqrt{v^T D_{i,u} v + (v^T D_{i,w} v)^2} \perp \gamma_i^n \geq 0 . \] (2.13)

Nonetheless, as \( h \to 0 \) the solution of the modified time-stepping scheme will approach the solution of the same measure differential inclusion as the original scheme [34].

It has been shown [36] that the modified scheme is a Cone Complementarity Problem (CCP), which can be solved efficiently by a family of iterative numerical methods that rely on projected contractive maps. One such algorithm is discussed below; it fits well into a parallel computing paradigm since it requires little data interdependency, similar to a projected-Jacobi fixed-point method. Omitting for brevity some of the details discussed in [36], the algorithm makes use of the following vectors:

\[ \tilde{k} \equiv \mathbf{M}v^{(l)} + hf(t^{(l)}, q^{(l)}, v^{(l)}) \] (2.14)
\[ b_i \equiv \left\{ \frac{1}{h} \Phi_i(q^{(l)}), 0, 0 \right\}^T i \in \mathcal{A}(q^{(l)}, \delta), \] (2.15)
\[ b_i \equiv \frac{1}{h} \Psi_i(q^{(l)}, t) + \frac{\partial \Psi_i}{\partial t}, \ i \in \mathcal{B} \] (2.16)

The solution, in terms of dual variables of the CCP (the multipliers), is obtained by iterating the following steps until convergence:

\[ \forall i \in \mathcal{A}(q^{(l)}, \delta) : \]
\[ \delta_i^{r+1} = \gamma_i^r - \omega_\eta_i \left[ D_i^T \mathbf{M}^{-1} \left( \sum_{z \in \mathcal{A}(q^{(l)}, \delta)} D_z \gamma_z^r + \sum_{z \in \mathcal{B}} \nabla \Psi_z \gamma_z^r + \tilde{k} \right) + b_i \right] \] (2.17)
\[ \gamma_i^{r+1} = \lambda \Pi_i \left( \delta_i^{r+1} \right) + (1 - \lambda) \gamma_i^r. \] (2.18)

\[ \forall i \in \mathcal{B} : \]
\[ \delta_i^{r+1} = \gamma_i^r - \omega_\eta_i \left[ \nabla \Psi_i^T \mathbf{M}^{-1} \left( \sum_{z \in \mathcal{A}(q^{(l)}, \delta)} D_z \gamma_z^r + \sum_{z \in \mathcal{B}} \nabla \Psi_z \gamma_z^r + \tilde{k} \right) + b_i \right] \] (2.19)
\[ \gamma_i^{r+1} = \lambda \delta_i^{r+1} + (1 - \lambda) \gamma_i^r. \] (2.20)
The iterative process uses the projection operator \( \Pi_{\Upsilon_i}(\cdot) \) \cite{32}, which is a non-expansive map \( \Pi_{\Upsilon_i} : \mathbb{R}^3 \to \mathbb{R}^3 \) acting on the triplet of multipliers associated with the \( i \)-th contact. Thus, if the multipliers fall into the friction cone, they are not modified; if they are in the polar cone, they are set to zero; in the remaining cases they are projected orthogonally onto the surface of the friction cone.

The overrelaxation factor \( \omega \) and the \( \lambda \) and \( \eta \) parameters are adjusted to control the convergence. Good default values for \( \eta \) are \( \eta_i = 3/\text{Trace}(D_i^T M^{-1} D_i) \) for \( i \in A(q^{(l)}, \delta) \), and \( \eta_i = 1/(\nabla \Phi_i^T M^{-1} \nabla \Phi_i) \) for \( i \in B \). When dealing exclusively with bilateral constraints these choices lead to the classical Jacobi fixed-point method. In regards to \( \omega \) and \( \lambda \), extensive numerical experiments suggest that choosing \( \omega = 0.3 \) and \( \lambda = 1 \) typically leads to good convergence speed. For a proof of the convergence of this method refer to \cite{36}.

Note that using Eqs.\((2.8)\) and \((2.14)\), one can rewrite the iteration in a more compact form:

\[
\forall i \in A(q^{(l)}, \delta) : \\
\gamma_{r+1}^i = \lambda \Pi_{\Upsilon_i} \left[ \gamma_r^i - \omega \eta_i \left( D_i^T \mathbf{v}^r + b_i \right) \right] + (1 - \lambda) \gamma_r^i
\]

\[
\forall i \in B : \\
\gamma_{r+1}^i = \lambda \left[ \gamma_r^i - \omega \eta_i \left( \nabla \Psi_i^T \mathbf{v}^r + b_i \right) \right] + (1 - \lambda) \gamma_r^i
\]

In this case, at each iteration, before repeating Eqs. \((2.21)\) and \((2.22)\), velocities \( \mathbf{v}^{(l+1)} \) are updated as

\[
\mathbf{v}^{r+1} = M^{-1} \left( \sum_{z \in A(q^{(l)}, \delta)} D_z \gamma_{z}^{r+1} + \sum_{z \in B} \nabla \Psi_z \gamma_{z}^{r+1} + \mathbf{k} \right)
\]

Note that the superscript \((l + 1)\) was omitted for brevity.

For small problems, good accuracy in the CCP solution is typically obtained after less than one hundred iterations. Note that iterating through Eqs. \((2.21)\), \((2.22)\) and
(2.23), also yields the primal variables (the velocities) at the end of the procedure with no additional effort.

The following pseudocode of Algorithm 1 shows how the iteration is implemented on a serial computing architecture:

Algorithm 1 DVI: Inner Iteration Loop

1. For $i \in A(q, \delta)$, evaluate $\eta_i = 3/\text{Trace}(D_i^T M^{-1} D_i)$.

2. For $i \in B$, evaluate $\eta_i = 1/(\nabla \Phi_i^T M^{-1} \nabla \Phi_i)$.

3. If some initial guess $\gamma^*$ is available for multipliers, then set $\gamma^0 = \gamma^*$ (warm start), otherwise $\gamma^0 = 0$.

4. Initialize velocities: $v^0 = \sum_{i \in A} M^{-1} D_i \gamma_i^0 + \sum_{i \in B} M^{-1} \nabla \Phi_i \gamma_i^0 + M^{-1} \tilde{k}$.

5. For $i \in A(q^{(l)}, \delta)$, compute changes in multipliers for contact constraints:
   \[
   \gamma_i^{r+1} = \lambda \Pi_{\gamma_i} \left( \gamma_i^r - \omega \eta_i \left( D_i^T v^r + b_i \right) \right) + (1 - \lambda) \gamma_i^r;
   \]
   \[
   \Delta \gamma_i^{r+1} = \gamma_i^{r+1} - \gamma_i^r;
   \]
   \[
   \Delta v_i = M^{-1} D_i \Delta \gamma_i^{r+1}.
   \]

6. For $i \in B$, compute changes in multipliers for bilateral constraints:
   \[
   \gamma_i^{r+1} = \lambda \left( \gamma_i^r - \omega \eta_i \left( \nabla \Psi_i^T v^r + b_i \right) \right) + (1 - \lambda) \gamma_i^r;
   \]
   \[
   \Delta \gamma_i^{r+1} = \gamma_i^{r+1} - \gamma_i^r;
   \]
   \[
   \Delta v_i = M^{-1} \nabla \Psi_i \Delta \gamma_i^{r+1}.
   \]

7. Apply updates to the velocity vector:
   \[
   v^{r+1} = v^r + \sum_{i \in A} \Delta v_i + \sum_{i \in B} \Delta v_i
   \]

8. $r := r + 1$. Repeat from 5 until convergence, or until $r > r_{\text{max}}$.

The stopping criterion is based on the value of the velocity update. The overall
algorithm that provides an approximation to the solution of Eqs. (2.8) through (2.12) relies on Algorithm 1 and requires the following steps:

**Algorithm 2** DVI: Outer, Time-Stepping, Loop

1. Set $t = 0$, step counter $l = 0$, provide initial values for $q^{(l)}$ and $v^{(l)}$.

2. Perform collision detection between bodies, obtaining $n_A$ possible contact points within a distance $\delta$. For each contact $i$, compute $D_{i,n}$, $D_{i,u}$, $D_{i,w}$; for each bilateral constraint compute the residual $\Phi_i(q)$, which also provides $b_i$.

3. For each body, compute forces $f(t^{(l)}, q^{(l)}, v^{(l)})$.

4. Use Algorithm 1 to solve the cone complementarity problem and obtain unknown impulse $\gamma$ and velocity $v^{(l+1)}$.

5. Update positions using $q^{(l+1)} = q^{(l)} + hL(q^{(l)})v^{(l+1)}$.

6. Increment $t := t + h$, $l := l + 1$, and repeat from step 2 until $t > t_{\text{end}}$

### 2.4 Summary

The inner/outer loop combination has been implemented on serial computing architectures and proved to be reliable and efficient. In Chapter 4, the time-consuming part of the methodology, that is, the CCP iteration of Algorithm 1, will be reformulated to take advantage of the parallel computing resources available on commodity GPUs.
3 COLLISION DETECTION ALGORITHM

Collision Detection (CD) is a ubiquitous task performed in computer simulations in a variety of fields such as computer games, nonlinear finite element analysis, smoothed particle hydrodynamics (SPH), multi-body dynamics analysis, and granular dynamics. An example application, which draws on the last two fields and aims at gauging the mobility of tracked vehicles on granular terrain [2], is shown in Fig. 3.1. The purpose of the collision detection task is to understand whether any pairs of geometries are in contact, and if so, to quantify the nature of each contact; i.e., depth of penetration, volume of penetration, center of mass, and normal vectors to the contact surface (if applicable).

For the dynamics analysis of the double tracked system in Fig. 3.1, a problem with $2 \cdot 10^5$ to $4 \cdot 10^5$ elements, the collision detection can become a significant computational bottleneck. For instance, when the time-stepping (CCP iteration, numerical integration) component of the simulation was solved using a GPU parallel implementation in [37], the sequential implementation of the collision detection stage prevented any significant speedup, a consequence of Amdahl’s law [38]. A similar scenario occurred when simulating the dynamics of granular material, as shown in Fig. 3.2 [39]. In order to fully leverage the potential of parallel computing in many-body dynamics problems it becomes apparent that parallelizing the collision detection task is mandatory. This chapter concentrates on the details of the the collision detection algorithm, while Chapter 5 provides specifics of its parallel implementation.

In the Multi-body Dynamics community, handling frictional contact dynamics and/or carrying out collision detection to that end have been topics of intense research, see for instance [40, 41, 42, 43] and references therein. These papers handle problems where the number of contacts at each time step is relatively small, that is, less than thousands of events. Large collision detection tasks, on the order of hundreds of millions of contacts, are encountered in granular dynamics, see for instance [44] and references therein. However,
this and similar many-body dynamics contributions either adopt sequential computing approaches, or are very focused in nature and fall short of presenting a solution that can address industrial applications of general interest such as the one in Fig. 3.2. A large body of literature addressing the collision detection topic has been compiled by the Computer Science community. One of the most widely used collision detection algorithms draws on a "Sort and Sweep" approach that leverages the properties of Axis Aligned Bounding Boxes (AABBs) in order to detect collisions between objects of varying size [45, 46]. An AABB is a special case of a bounding box that is always aligned to the global reference frame, simplifying collision detection as the bounding box (box that completely encloses the space around an object) cannot rotate. The basic premise of this algorithm is that if two objects are colliding, their projections/shadows will overlap on all three principal axes (see Fig. 3.3). To this end, bounding boxes are generated for all objects and projected onto the global X, Y, and Z-axes. Each axis is then sorted by position and iterated over in linear time. A stack-like data structure is used to keep track of collision pairs; a
collision will be recorded only if two bodies overlap on all three axes. The approach is simple to implement and ideally suited to execute sequentially on the Central Processing Unit (CPU). A maximum of three parallel threads are used to sort and traverse the axes sequentially and only one thread can be used to compare the three lists to detect collisions thus limiting this method’s effectiveness on parallel architectures. One salient feature of this algorithm is its ability to take advantage of temporal coherence. Since objects in a dynamics simulation rarely move large distances in one time step (the positions are temporally coherent), updates can be made to the sorted axes easily and with relatively little cost. This characteristic was exploited in I-Collide [47].

Parallel collision detection approaches became tractable with the introduction of easily programmable Graphics Processing Units (GPUs). Many algorithms that use GPUs rely on shaders to do computations. Shaders, which are instruction sets designed to perform calculations on polygons and textures for graphics processing, can also be used to perform basic scientific computations. CULLIDE [48], R-CULLIDE [49], and Q-CULLIDE [50] are three CD algorithms that take advantage of shaders. These algorithms could resolve contacts between tens of thousands of polygons at near real-time speed and were faster than their CPU counterparts. More recently, work on large dynamics problems has been reported in [51], where spheres of constant radius are used to decompose complex geometries in order to decrease simulation times. Spherical decomposition removed the overhead associated with the tasks of handling complex geometries represented as triangle meshes and the ensuing triangle-triangle collision detection. This in turn allowed for real-time collision detection between thousands of objects composed of spheres. Fig. 3.4 shows an example of the spherical decomposition of a track shoe from a tank tread.

The previously mentioned decomposition algorithm was revisited in [52]. Originally, when generating the three dimensional grid, see Fig. 3.5, textures were allocated to store the complete space, and empty bins (bins that contained no bodies at all) used the
same amount of memory as filled ones. A bin is a rectangular space in three dimensions, essentially one cell in the three dimensional grid. In the updated version of this algorithm, bins that were not in use were trimmed, forming a tighter dynamic grid that saved memory. While optimal for spherical decomposition, this method could not support any additional object types that were not decomposed into spheres, limiting its usefulness in simulations containing complex shapes made with boxes, planes, ellipsoids, etc.

With the introduction of NVIDIA’s Compute Unified Device Architecture (CUDA) [53], GPUs are now able to execute C code instead of relying on shader code. A parallel spatial subdivision algorithm that uses CUDA to compute interactions between spheres was introduced in [54]. This approach relies on the observation that the objects in a
given bin in the spatially subdivided space can only interact with the 26 surrounding bins in a 3x3x3 grid. Unlike the method introduced in [51], the CUDA based approach in [54] does not support spherical decomposition of objects. Moreover, it does not compute collisions; rather interaction forces between spheres are calculated using equations based on fluid dynamics principles. One unique feature, discussed at great length in [54], is the radix sort algorithm. Most collision detection algorithms require a fast sorting algorithm to arrange collision information into specific data structures. Commonly used sorting methods such as quick sort, while fast on CPUs, are difficult to parallelize. Radix sort is not hindered by this drawback, as it is able to sort key-value pairs in parallel extremely efficiently. Additional work in [55] was done to improve the performance and scalability of the sorting algorithm.

The collision detection approach proposed herein is geared at solving many-body dynamics problems. Examples include the dynamics of sand flowing inside an hourglass, a rover running over sandy terrain, an excavator/frontloader digging/loading granular material, etc. In this context, the collision detection task is performed on a rather small collection of rigid and/or deformable bodies of complex geometry (hourglass wall, wheel, track shoe, excavator blade, dipper), and a very large number of bodies (millions to billions) that make up the granular material. On this scale, the collision detection task, particularly when dealing with the granular material, fits perfectly the Single Instruction Multiple Data (SIMD) computation paradigm. Specifically, the same sequence of instructions needs to be applied to every individual body and/or contact in the granular material. An overview of the algorithm is provided in this chapter, with specifics presented in Chapter 5.
Figure 3.5: Example of 3D space divided into bins

3.1 Broad-Phase Algorithm

The Broad-Phase algorithm is used to compute whether two bodies might be in contact at a given time. The purpose of the broad-phase algorithm is not to find actual contact information, but rather to determine if a contact could potentially occur based on the AABBs of the bodies involved.

3.1.1 Axis Aligned Bounding Boxes

An Axis Aligned Bounding Box (AABB) is a special case of a bounding box that is always aligned to the global reference frame, simplifying collision detection as the bounding box cannot rotate. Because of this, the volume enclosed by the bounding box will always be equal to or greater than the volume of the shape it encloses. AABB generation is simple and can be easily parallellized on a per object basis.

An AABB is computed by determining the maximum and minimum point of a shape. For example, an AABB for a sphere can be computed using its center and radius, with the minimum and maximum points being the radius subtracted and added to the center respectively. Computing the AABB for other shapes is more complicated. For an ellipsoid, for instance, the orientation of the object needs to be taken into account, and rather than
one radius there are three, one for each principle axis. See Fig. 3.6 for an example of AABB computation for a cylinder in 3D space.

3.1.2 Spatial Subdivision Algorithm

A high-level overview of the GPU-based collision detection is as follows. The collision detection process starts by identifying the intersections between AABBs and bins, see Fig. 3.5 for a visual representation of a bin. The AABB-bin pairs are subsequently sorted by bin id. Next, each bin’s starting index is determined so that the bins’ AABBs can be traversed sequentially. All AABBs touching a bin are subsequently checked against each other for collisions. This high level process is implemented in a sequence of 5 stages, each of which is discussed next. Fig. 3.8 shows what a typical set of data used for collision detection looks like and will be used in what follows to explain the proposed approach.

Step 1: Find Bounds of Space

The collision detection process begins by identifying the absolute minimum and maximum points for all of the AABBs in the global coordinate system. Starting with a global AABB $A$ of zero size, the AABB for each collision shape is added, growing $A$ as necessary. This process returns $A$ which contains the min/max point information, see Fig. 3.7. Using this information the dimensions of the collision space are computed in 3 dimensions. For
each dimension the space is divided into $s_x$, $s_y$, and $s_z$ bins respectively. The vector $s$ is updated in Stage 5, starting from an initial value. Once the bounds of the space and the bin size are computed the AABB-bin intersections can be computed.

**Stage 2: Determine AABB-to-Bin Intersections**

In this stage all AABB-to-bin intersections are identified. An AABB can ‘touch’ more than one bin; there is no limit to how many intersections take place. The minimum and maximum points are stored in the AABB, therefore the AABB-bin intersections can be quickly determined, see Fig. 3.8. For each AABB the bin ID is determined for the maximum and minimum points. Because the AABB is not oriented all bins between these two indices will contain the AABB. For each AABB-bin intersection a unique bin identifier is computed based on that bin’s 3-dimensional index. This unique identifier is computed using a hashing algorithm that takes a [x,y,z] triplet and returns a unique number, this number along with the AABB identifier is stored in array $B$, see Fig. 3.8.

**Stage 3: Reverse Mapping**

The AABB-bin pairs, stored in $B$ are currently sorted by the AABB index. This needs to be reversed so that for each bin, the indices of the associated AABBs are known. To this end a key-value based sort is performed on $B$ using the bin number as the key and the
Figure 3.8: Example of a 2D space, containing 3 AABBs, divided into bins. B contains the list of bodies with associated bin intersections.

Figure 3.9: Sort based on the Bin ID to so that the objects in each bin are known AABB index as the value. The result of this operation is an array of key-value pairs E, see Fig. 3.9.

**Stage 4: Determine AABB Contacts**

Using E the AABB’s in each bin are traversed. For each AABB in the bin, an exhaustive $N^2$ search is performed for AABB-AABB intersection, where N is relatively small compared to the total number of AABBs and is usually between 2 and 100. If each condition in equation 3.1 is met for two AABBs A and B, an intersection has occurred.
The collision pair associated with each intersection is stored in $G$, see Fig. 3.10, and represents the list of potential contacts for all collision shapes. This list is the input for the narrow-phase algorithm which will determine if an actual contact occurred.

**Stage 5: Optimize Broad-Phase**

One of the bottlenecks for this algorithm is the $N^2$ AABB-AABB intersection test that is performed within each bin. To do better load balancing so that some bins do not have hundreds more AABBs compared to other bins, the bin size can be modified based on the maximum number of AABBs for all bins. If there are too many AABBs in a bin, the bin size is reduced. Likewise, if the number is too low the bin size is increased. These changes are applied when collision detection is performed during the following time step.

### 3.2 Narrow-Phase Algorithm

Once potential contacts have been determined from the broad-phase collision detection stage, the Narrow-Phase algorithm needs to process each possible contact and determine if it actually occurs. To this end an algorithm capable of determining contacts between convex geometries was implemented on the GPU. This algorithm, called “XenoCollide” [56], is based upon Minkowski Portal Refinement (MPR) [1]. Given a simple mathematical
function called a *support mapping* the method can detect a contact between two convex geometries.

### 3.2.1 Support Mappings

A *support mapping* is a simple mathematical function which, given a direction $\mathbf{n}$, will return the furthest point on the object, in its local reference frame, in this direction. For example, Fig. 3.11 shows an example of a support point $\mathbf{p}$ given a vector $\mathbf{n}$. If multiple support points are found, any one of them is a valid candidate to be the support point for that normal. See Table 3.1 for some example support mappings. Note that support mappings can be combined to form more complex shapes. For example, a cylinder’s support function is simply the support functions of a disk and a line segment added together.

### 3.2.2 Minkowski Difference

The concept of *Minkowski difference* is important when trying to compute the collision between convex objects. A convex shape can be described as a convex set of points in the global coordinate system. Given two convex sets $A$ and $B$, the Minkowski difference is defined as $A \ominus B = \{a - b : a \in A, b \in B\}$, see Fig. 3.12. This process essentially
Table 3.1: Support mappings, for a more complete list see [1]

<table>
<thead>
<tr>
<th>Description</th>
<th>Support Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_{\text{point}}(n)$</td>
<td>$p$</td>
</tr>
<tr>
<td>$S_{\text{segment}}(n)$</td>
<td>$[r_x \text{sgn}(n_x), 0, 0]$</td>
</tr>
<tr>
<td>$S_{\text{disk}}(n)$</td>
<td>$r [n_x, n_y, 0] / |n_x, n_y, 0|_2$</td>
</tr>
<tr>
<td>$S_{\text{box}}(n)$</td>
<td>$[r_x \text{sgn}(n_x), r_y \text{sgn}(n_y), r_z \text{sgn}(n_z)]$</td>
</tr>
<tr>
<td>$S_{\text{sphere}}(n)$</td>
<td>$rn$</td>
</tr>
<tr>
<td>$S_{\text{ellipsoid}}(n)$</td>
<td>$[r_x^2 n_x, r_y^2 n_y, r_z^2 n_z] / |n_x, n_y, n_z|_2$</td>
</tr>
<tr>
<td>$S_{\text{cylinder}}(n)$</td>
<td>$S_{\text{segment}}(n) + S_{\text{disk}}(n)$</td>
</tr>
</tbody>
</table>

amounts to taking the difference in three dimensions of every point in $A$ by every point in $B$ in the global coordinate system. $A \ominus B$ has the interesting property that if the sets $A$ or $B$ have any common points, the origin lies within $A \ominus B$ and the two shapes are intersecting. Therefore, if the Minkowski difference $A \ominus B$ can be computed, the state of contact between the two shapes can be determined.

The computation of the Minkowski difference can be expensive depending on the type of shape, as subtracting every point in one shape from the other would be time consuming. If, however, the support mapping of sets $A$ and $B$ are provided, the support mapping of the Minkowski difference can be readily computed using Equation 3.2 [1]. This removes the need to compute the Minkowski difference; which, depending on the complexity of the two convex shapes, could require high computational effort.

$$S_{A \ominus B}(n) = S_B(n) - S_A(n) \quad (3.2)$$

### 3.2.3 Minkowski Portal Refinement

The algorithm used for collision detection is based upon the Minkowski Portal Refinement (MPR) technique [1] and is called “XenoCollide” [56]. The MPR algorithm consists of
two steps or phases which are listed below.

**Algorithm 3** MPR: Minkowski Portal Refinement, Phases

1. **Phase 1:** Using four points, generate a tetrahedron so that it has one point, \( p_0 \) that is in the interior and three points, \( p_1 \ p_2 \ p_3 \), that are on the boundary of the Minkowski difference \( A \ominus B \). The triangle made with the three points, \( p_1 \ p_2 \ p_3 \) is referred to as the "portal". Generate an origin ray, i.e. a vector from the center of \( A \ominus B \) to the origin. If the ray does not pass through this portal, find a new portal candidate.

2. **Phase 2:** Portal refinement. Find a new portal such that it still intersects the origin ray. This new portal should be closer to the surface of \( A \ominus B \) than the previous. Iterate until either (a) the origin cannot be reached and it is now touching or within the tetrahedron formed by \( p_0 \ p_1 \ p_2 \ p_3 \), or (b) the origin is inside the tetrahedron.

**Phase 1: Finding a Portal**

The first step to finding the portal is to determine the origin ray. To do this a point known to be on the interior of \( A \ominus B \) is necessary. One possible point is the center of
mass for $A \ominus B$ which can be computed by taking the center of $B$ and subtracting it from the center of $A$. The vector from this point $p_0$ to the origin is the origin ray. If this vector intersects the surface of $A \ominus B$ before reaching the origin, the origin is on the outside of $A \ominus B$ and therefore by the property of the Minkowski difference, no contact occurs. Using the support mapping $S_{B-A}(n)$, where $n$ is the vector from the center of $A$ to the center of $B$, three points, $p_1 \ p_2 \ p_3$, which are not co-linear are determined. These three points are used to create a triangular portal; if the origin ray does not pass through this portal, a new portal must be determined.

Using $p_0$, the center of $A \ominus B$, which was found earlier, and the three non co-linear points, $p_1 \ p_2 \ p_3$, a tetrahedron is formed. If the triangular portal is not intersected by the the origin ray, the face of the tetrahedron closest to the origin is determined. The normal of this face is used to determine a new support vertex $p_{new}$ using the support mapping, see See Table 3.1. The vertex on the tetrahedron, $p_0 \ p_1 \ p_2 \ p_3$, forming the plane closest to the origin is replaced with $p_{new}$.

This phase is complete once a portal that intersects the origin ray is found.

**Phase 2: Refining the Portal**

In this phase we loop constantly until either a hit or a miss has occurred. An overview is provided in Algorithm 4, details are beyond the scope of this document, for an in depth discussion see [1] and [57].
Algorithm 4 MPR: Minkowski Portal Refinement, Phase 2 Details

1. \( A \ominus B \) is convex, the tetrahedron formed by, \( p_0 \ p_1 \ p_2 \ p_3 \) is therefore guaranteed to lie inside. If the origin lies within the portal the origin is also inside the tetrahedron, therefore, if the origin lies within the tetrahedron it lies within \( A \ominus B \) and a contact has occurred.

2. If no contact occurred the origin lies outside the portal. It is unknown, however, if the origin is still within \( A \ominus B \) and outside the portal or if the origin is completely outside \( A \ominus B \). The outward facing normal of the portal is used as the new support vector because we want more information of what lies on the outside of the portal. Using this vector a new support point and plane are computed.

3. If the origin is found to still be outside the portal, a miss has occurred. Because the previous portal was known to contain the origin ray and the origin is still on the outside of the support plane found in the direction of the origin, the origin cannot be inside \( A \ominus B \).

4. If the support plane for the new point lies too close to the portal it means that the algorithm has stalled and convergence will not be achieved. In this case either a hit or a miss can be reported. It is better to report a hit so that contacts are not missed. This can be changed, however, depending on the application.

5. Using the support plane previously found a new portal is generated. To determine which vertex of the portal to replace: \( p_1, p_2, \text{ or } p_3 \), a check between the three planes of the tetrahedron and the origin point in performed. The origin will lie on two of the three planes, the face that touches two of the planes will make up the new portal.

6. Return to the first step and continue.
3.2.4 Determining Contact Information

Once we have determined that a contact has occurred, collision information can be determined. Using MPR, the origin ray is continued to be projected to the surface of the Minkowski difference $A \ominus B$. Once the ray reaches the surface it provides a good estimate of the contact normal.

Contact points can be determined by using the final portal determined and computing the location of the origin in barycentric coordinates relative to this triangular portal. This point, along with the contact normal are used to compute one final support point for $A$ and $B$. These support points are the points of contact on each object required by the constraint solver. The penetration distance for the contact is the distance between the two contact points.

3.3 Summary

The MPR algorithm is a useful approach to determining contact between convex shapes. Unlike other methods, such as GJK [58], which cannot determine collision information directly and relies on a secondary algorithm, MPR provides contact information directly. This makes the approach more efficient and better suited to parallel applications [57].
4 PARALLEL IMPLEMENTATION OF DVI SOLVER

4.1 Introduction

This chapter will focus on the parallel implementation of the algorithm described in Chapter 2. The implementation uses the CUDA API [13] and was designed to run on any of NVIDIA’s CUDA capable GPUs. Additionally the Thrust library [59] was used for many basic operations, such as parallel prefix sum, parallel radix sort, and parallel reductions. A listing of the functionality used is provided in Appendix B.

4.2 Solving the CCP in Parallel

A parallel version of an algorithm must respect the Lamport consistency model, that is the parallel execution must produce the same results as the sequential program, regardless of the number of threads [60].

Data dependency poses a constraint on the possibility of a straightforward parallelizations of algorithms. In fact, denoting $I_i$ and $O_i$ the sets of input and output variables of the $i$-th program fragment, Bernstein’s conditions state that two fragments $i, j$ can be executed in parallel only if the following conditions are satisfied: $I_i \cap O_j = \emptyset$, $O_i \cap I_j = \emptyset$. If all these conditions are satisfied, the program requires no synchronization of memory and it belongs to the so called embarassingly-parallel class, the type of parallel execution most suitable for GPU computing.

One can see that a parallelization of this class can be easily implemented for computations in steps 5 and 6 of Algorithm 1, by simply assigning one contact per thread (and, similarly, one constraint per thread). In fact the results of these computations would not overlap in memory, and it will never happen that two parallel threads need to write in the same memory location at the same time. These are the two most numerically-intensive
Figure 4.1: The concept which inspires the reduction algorithm. Sums are performed with a binary tree, to exploit the parallel nature of the stream processors.

steps of the CCP solver, called the CCP contact iteration kernel and the CCP constraint iteration kernel.

However, the sums in step 7 of Algorithm 1 cannot be performed with embarrassingly-parallel implementations: for example, it may happen that two or more contacts need to add their velocity updates to the same rigid body. A possible approach to overcome this problem is presented in [51] for a similar problem. An alternative method was used for this application. It uses an offset buffer to store the locations of each update as it would appear if the list of updates was sorted by body number. A parallel reduction can then be done during each iteration of the CCP solver.

Summation of array values into a single memory destination, called data reduction, is a problem which can be performed in parallel only at the cost of some fine-grained data synchronization [61]. Recent research on GPU parallel architectures proposed hierarchical algorithms as a way to perform data reduction [62, 63]. The basic idea is depicted in Fig. (4.1): the summation is performed as a sequence of in-place parallel binary sums with exponentially-increasing strides. This way, at least for large data, a large number of
threads are kept busy.

4.3 Data Structures

In the proposed approach, the data structures on the GPU are implemented as large arrays (buffers) to match the execution model associated with NVIDIA’s CUDA. Specifically, threads are grouped in rectangular thread blocks, and thread blocks are arranged in rectangular grids. Three main buffers are used: the contacts buffer, the constraints buffer, and the bodies buffer.

When designing the data structures of these buffers, special care should be paid in order to minimize the memory overhead caused by repeated transfers of large data structures. Moreover, data structures should be organized to exploit fast GPU coalesced memory access to fetch data for all parallel threads in a warp, which is a set of 32 threads all running simultaneously in parallel. Provided that bytes are contiguous and that the $k^{th}$ thread accesses the $k^{th}$ element in the data structure, up to 512 bytes can be fetched in one global memory read operation by a warp of threads. Failing to perform coalesced memory access may slow the kernel significantly.

Numerical experiments show that for high memory throughput, it is better to pad the data into a four-float width structure even at the cost of wasting memory as several entries end up not being used. Also, the variables in the data structures are organized in a way that minimizes the number of fetch and store operations. This approach maximizes the arithmetic intensity of the kernel code, as recommended by the CUDA development guidelines [13].

In the actual implementation of the method, the data structure for the contacts were not mapped onto a padded four-float structure, see Fig. (4.2). A trade off was made between a small drop in compute performance which provided a reduction in the total memory used. Memory is important when solving large problems with millions of contacts,
so a reduction in the amount of memory required to store 1 contact will result in large memory savings.

There is no need to store the entire $D_i$ matrix for the $i^{th}$ contact because it has zero entries for most of its part, except for the two 12x3 blocks corresponding to the coordinates of the two bodies in contact. In fact, once the velocities of the two bodies $\dot{r}_{A_i}, \omega_{A_i}, \dot{r}_{B_i}$ and $\omega_{B_i}$ have been fetched, the product $D_i^T \mathbf{v}^r$ in step 5 of Algorithm 1 can be performed as

$$D_i^T \mathbf{v}^r = D_{i,v,A}^T \dot{r}_{A_i} + D_{i,\omega,A}^T \omega_{A_i} + D_{i,v,B}^T \dot{r}_{B_i} + D_{i,\omega,B}^T \omega_{B_i}$$

Figure 4.2: Different buffers used for contacts, constraints, and bodies

with the adoption of the following 3x3 matrices

$$D_{i,v,A}^T = -A_{i,p}^T$$

$$D_{i,\omega,A}^T = A_{i,p}^T A_{A} \mathbf{S}_{i,A}$$

$$D_{i,v,B}^T = A_{i,p}^T$$

$$D_{i,\omega,B}^T = -A_{i,p}^T A_{B} \mathbf{S}_{i,B}$$
Since $\mathbf{D}^T_{i,v_A} = -\mathbf{D}^T_{i,v_B}$, there is no need to store both matrices, so in each contact data structure only a matrix $\mathbf{D}^T_{i,v_{AB}}$ is stored, which is then used with opposite signs for each of the two bodies.

Also the velocity update vector $\Delta \mathbf{v}_i$, needed for the sum in step 7 of Algorithm 1, is sparse: it can be decomposed in small subvectors. Specifically, given the masses and the inertia tensors of the two bodies $m_A$, $m_B$, $\mathbf{J}_A$, and $\mathbf{J}_B$, the term $\Delta \mathbf{v}_i$ can be computed and stored in four parts as follows:

\begin{align}
\Delta \dot{r}_A &= m_A^{-1} \mathbf{D}_{i,v_A} \Delta \gamma_{r_{i+1}} \quad (4.3a) \\
\Delta \omega_A &= \mathbf{J}_A^{-1} \mathbf{D}_{i,\omega_A} \Delta \gamma_{\omega_{i+1}} \quad (4.3b) \\
\Delta \dot{r}_B &= m_B^{-1} \mathbf{D}_{i,v_B} \Delta \gamma_{r_{i+1}} \quad (4.3c) \\
\Delta \omega_B &= \mathbf{J}_B^{-1} \mathbf{D}_{i,\omega_B} \Delta \gamma_{\omega_{i+1}} \quad (4.3d)
\end{align}

Note that those four parts of the $\Delta \mathbf{v}_i$ terms are not stored in the $i$-th contact data structure or in the data structure of the two referenced bodies (because multiple contacts may refer the same body, hence they would overwrite the same memory position). These velocity updates are instead stored in a reduction buffer, which will be used to efficiently perform the summation in step 7 of Algorithm 1. This will be discussed shortly.

The constraints buffer, shown in Fig. (4.2), is based on a similar concept. Jacobians $\nabla \Psi_i$ of all scalar constraints are stored in a sparse format, each corresponding to four rows $\nabla \Psi_{i,v_A}$, $\nabla \Psi_{i,\omega_A}$, $\nabla \Psi_{i,v_B}$, $\nabla \Psi_{i,\omega_B}$. Therefore the product $\nabla \Psi_i^T \mathbf{v}^r$ in step 6 of Algorithm 1 can be performed as the scalar value

\begin{equation}
\nabla \Psi_i^T \mathbf{v}^r = \nabla \Psi_{i,v_A}^T \dot{\mathbf{r}}_A + \nabla \Psi_{i,\omega_A}^T \omega_A + \nabla \Psi_{i,v_B}^T \dot{\mathbf{r}}_B + \nabla \Psi_{i,\omega_B}^T \omega_B 
\end{equation}
Also, the four parts of the sparse vector $\Delta \mathbf{v}_i$ can be computed and stored as

$$
\Delta \dot{r}_{Ai} = m_{Ai}^{-1} \nabla \psi_{i,vA} \Delta \gamma_i^{r+1} \tag{4.5a}
$$

$$
\Delta \omega_{Ai} = J_{Ai}^{-1} \nabla \psi_{i,\omega A} \Delta \gamma_i^{r+1} \tag{4.5b}
$$

$$
\Delta \dot{r}_{Bi} = m_{Bi}^{-1} \nabla \psi_{i,vB} \Delta \gamma_i^{r+1} \tag{4.5c}
$$

$$
\Delta \omega_{Bi} = J_{Bi}^{-1} \nabla \psi_{i,\omega B} \Delta \gamma_i^{r+1} \tag{4.5d}
$$

About the bodies buffer, Fig. (4.2) shows that each body is represented by a data structure containing the state (velocity and position), the mass moments of inertia and mass values, and the external applied force $F_j$ and torque $C_j$. Forces and torques, if any, are used to compute the third step of Algorithm 1. Note that to speed the iteration, it is advantageous to store the inverse of the mass and inertias rather than their original values, because the operation $M^{-1} D_i \Delta \gamma_i^{r+1}$ must be performed multiple times.

### 4.4 Complete Time Stepping

The process used to advance the simulation by one time step is outlined in Algorithm 5. In step 3 (Stage 1 of the solver), the pre-processing stage is carried out where the gyroscopic torque is computed and any external torques and forces are applied as impulses before the constraint iteration stage. In step 4 (Stage 2 of the solver), the velocity corrections for each constraint, both unilateral and bilateral, are computed. Once the termination criteria are met, the solver has converged to an acceptable solution. There is a tradeoff of speed vs. accuracy at this point: more iterations result in a more stable and accurate solution while fewer iterations means faster computation at the expense of accuracy.
Algorithm 5 Complete Time Stepping, when GPU is Available.

1. (GPU, Geometry-Parallel) Perform collision detection between bodies, obtaining $n_A$ possible contact points within a distance $\delta$, as contact positions $s_{i,A}, s_{i,B}$ on the two touching surfaces, and normals $n_i$. If warm start is used, then fetch last reactions in contact point $\gamma_i^*$ (obtained in previous frame, if the contact is persistent) and set $\gamma_i = \gamma_i^*$; otherwise set $\gamma_i = 0$.

2. (Host, serial) Copy bilateral constraint data (residuals $b_i$ and Jacobians) into the constraint buffer.

3. **Stage 1**: (GPU, Algorithm 6) Pre-processing. At each time step the pre-processing stage described in Algorithm 6 is performed.

4. **Stage 2**: (GPU, Algorithm 7) CCP Iteration. The solver iterates on the constraints until it converges as described in Algorithm 7.

5. (GPU, body-parallel) Time integration kernel. For each $j$ body, perform time integration as $q_j^{(l+1)} = q_j^{(l)} + hL(q_j^{(l)})v_j^{(l+1)}$

4.4.1 Stage 1: Pre-processing

Algorithm 6 outlines the computations carried out in this stage. First the gyroscopic torque is computed on the GPU (Algorithm 6 item 1), this computation is body-parallel so body $i$'s gyroscopic torque is computed by thread $t$ on the GPU. The same process is performed for the Force computation see (Algorithm 6 item 2).

In the CCP loop, Stage 2, each contact updates the velocity information for two bodies; due to the random nature of the contacts that need to be processed, all velocity updates need to be computed before they can be reduced and applied to the respective bodies. Rather than perform a reduction for every CCP iteration, a mapping can be computed that produces a sorted list of updates (by body number) given a random list of contacts. As contacts do not change within a time step, these offsets can be precomputed. Several buffers are necessary for this step, where $M$ represents the total number of constraints (both unilateral and bilateral) in the system. See Fig. 4.3 for a visual representation of how the buffers are setup. Data initialization for the sequences, Buffer $Y$ and Buffer $Z$,
was performed using the Thrust sequence command which generates a sequence of values on the GPU, see Appendix B.3.1

1. Buffer $X$, of size $2M$, contains the two body number $i$ and $j$ associated with each constraint.

2. Buffer $Y$, of size $2M$, where the elements are initialized to $\{1, 2, 3 \ldots 2M\}$. This represents the velocity update number.

3. Buffer $Z$, of size $2M$, where the elements are initialized to $\{1, 2, 3 \ldots 2M\}$. This represents the update offset or the mapping for a velocity update given the contact number.

The process begins by running a constraint-parallel kernel on the GPU. This kernel for each constraint $w$ stores its $i$th and $j$th body index in $X$, where $X[w] = i$ and $X[w + M] = j$. Fig. 4.4 presents an example with four constraints between five different bodies. A Thrust Sort By Key is performed using $X$ as the key and $Y$ as the value, see Appendix B.2.3. The result is that $X$ is now sorted by body number, see Fig. 4.5. $Y$ is then used as the key with $Z$ as the value (see Fig. 4.6). This effectively takes $Z$, which was initialized to a sequential list of numbers and sorts it so that it now maps a velocity update from a constraint to a new location where the updates for each body are contiguous. $Z[w]$ and $Z[w + M]$ are effectively the update positions in the reduction buffer, see Fig. 4.7. In order to find the final velocity update for each body a Thrust Reduce By Key, see Appendix B.2.1, operation needs to be performed every CCP iteration.
Algorithm 6 Constraint Solver, Pre-processing Stage

1. (GPU, body-parallel) **Computing Gyroscopic Torque.** For each body, compute the gyroscopic torque using the moment of inertia $I$ and rotational velocity $\omega$ where $\tau = \omega \times (I\omega)$.

2. (GPU, body-parallel) **Force kernel.** For each body, using applied forces $f(t^{(l)}, q^{(l)}, v^{(l)})$ stored in $F_j$ (forces) and $C_j$ (torques), apply impulses.

3. (GPU, constraint-parallel) **Determine Constraint Offset.** For each constraint $[w..M]$, given the body indices $i$ and $j$ for that constraint, store $i$ and $j$ in a new buffer $X$.

4. (GPU, Thrust) **Determine Update Mapping.** Using the functionality of Thrust, offsets for the constraint updates are computed.
4.4.2 Stage 2: CCP Loop

Algorithm 7 outlines the iteration process for the CCP solver. In this stage $\Delta \gamma^{r+1}$ is iteratively computed until convergence is attained. The convergence tolerance is based upon the change in $\Delta \gamma^{r+1}$ between iterations: once the value falls below the tolerance the iteration loop is terminated and time integration is performed at the position level.

**Algorithm 7** Constraint solver Iteration loop.

1. (GPU, contact-parallel) CCP contact iteration kernel. For each contact $i$, given contact normal and position, compute in-place the matrices $D_{i,vA}^T$, $D_{i,\omega A}^T$ and $D_{i,\omega B}^T$, then compute $\eta_i^r$ and the contact residual $b_i^r = \{\frac{1}{\mu_i}(q), 0, 0\}^T$. Compute: $\gamma_i^{r+1} = \lambda \Pi_{r} \left(\gamma_i^r - \omega \eta_i^r \left(D_{i,v}^T v^r + b_i^r\right)\right) + (1 - \lambda) \gamma_i^r$. Note that $D_{i,v}^T v^r$ is evaluated with sparse data, using Eq. (4.1). Store $\Delta \gamma_i^{r+1} = \gamma_i^{r+1} - \gamma_i^r$ in contact buffer. Compute sparse updates to the velocities of the two connected bodies $A$ and $B$, that is the four vectors of Eq. (4.3), and store them at the locations specified in $Z$.

2. (GPU, constraint-parallel) CCP bilateral iteration kernel. For each constraint $i$, do $\gamma_i^{r+1} = \lambda \left(\gamma_i^r - \omega \eta_i \left(\nabla \Psi_i^T v^r + b_i\right)\right) + (1 - \lambda) \gamma_i^r$. Note that $\nabla \Psi_i^T v^r$ is evaluated with sparse data, using Eq. (4.4). Store $\Delta \gamma_i^{r+1} = \gamma_i^{r+1} - \gamma_i^r$ in contact buffer. Compute sparse updates to the velocities of the two connected bodies $A$ and $B$, that is the four vectors of Eq. (4.5), and store them in the locations specified in $Z$.

3. (GPU, body-parallel) Reduction phase. Using the Thrust Reduce By Key function, Appendix B.2.1, the list of updates is reduced on a per body basis. After reduction is complete, the update is then applied to each body.

4. Repeat from step 1 until convergence or until number of CCP steps reached $r > r_{max}$.

4.5 Performance Optimizations

Because contacts do not change within the CCP iteration loop, the matrices $D_{i,vA}^T$, $D_{i,\omega A}^T$ and $D_{i,\omega B}^T$, along with $\eta_i$ and the contact residual $b_i = \{\frac{1}{\mu_i}(q), 0, 0\}^T$ do not change. As a result it is possible to compute these before the iteration loop, during the pre-processing phase. However, this requires that at every iteration the matrices $D_{i,vA}^T$, $D_{i,\omega A}^T$ and $D_{i,\omega B}^T$
would need to be fetched from the GPU global memory. Due to the highly random nature of the contacts and the amount of data required from global memory, the cost of computing this data every iteration becomes less than the cost of the copying the data. Finally, it should be pointed out that $b_{i,v}$ and $b_{i,w}$ are always zero.
5 PARALLEL IMPLEMENTATION OF COLLISION DETECTION

5.1 Introduction

This chapter will present the implementation details for the collision detection algorithm. Code listings are provided to better explain the implementation details for this algorithm, though some have been simplified for brevity.

5.2 Data Structures

The data structures used for the collision detection algorithm can be divided into several categories, collision geometry, body information, and contact information.

1. Collision geometry contains all info specifying the size, position and rotation of the geometry with respect to its local reference frame. This allows for compound collision geometries to be created, allowing for convex or spherical decomposition to be performed for collision detection purposes. Each collision geometry stores the local position and rotation offset and, depending on the type of object, its size information, see Table 5.1 and Fig. 5.1. A sphere would store a radius, an ellipsoid

<table>
<thead>
<tr>
<th>Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local position</td>
</tr>
<tr>
<td>Local rotation</td>
</tr>
<tr>
<td>Body ID</td>
</tr>
</tbody>
</table>

Figure 5.1: Data required for a collision shape
Table 5.1: Information required to define each type of shape

<table>
<thead>
<tr>
<th>Description</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sphere</td>
<td>radius</td>
</tr>
<tr>
<td>Box</td>
<td>halfwidth along each axis</td>
</tr>
<tr>
<td>Ellipsoid</td>
<td>radius along each axis</td>
</tr>
<tr>
<td>Cone</td>
<td>radius and height</td>
</tr>
<tr>
<td>Cylinder</td>
<td>radius and height</td>
</tr>
<tr>
<td>Plane</td>
<td>halfwidth along 2 axis</td>
</tr>
<tr>
<td>Disk</td>
<td>radius</td>
</tr>
</tbody>
</table>

would store three radii, one for each principle direction, and a cylinder would store a radius and a height.

2. **Body information** refers to the global position and rotation of an object. This information is updated by the solver and is used to determine the global coordinates of each body necessary to generate their AABB and determine collision data during the narrow-phase portion of the algorithm. The body buffers used in the DVI solver, see Fig. 4.2, are the same for the collision detection.

3. **Contact Information** contains the penetration distance, contact normal and point of contact on each object. This information is computed during the Narrow-Phase portion of the algorithm and is sent directly to the solver to determine the necessary velocity corrections. Similar to the body information, the contact information is identical to that needed by the solver, see Fig. 4.2.

### 5.3 Parallel Broad-Phase

The parallel broad-phase algorithm, outlined in section 3.1.2 is performed with the following stages outlined below. These stages are similar to the ones described earlier,
however due to the parallel nature of this implementation, extra stages are needed to allocate memory and prepare data.

The pseudocode below outlines the steps performed by the algorithm on the GPU. Calls marked with (Thrust) are performed using the Thrust library, see Appendix B.

Algorithm 8 Broad-Phase Stages

1. Compute AABB for each object
2. (Thrust) Determine bounding points for entire space
3. Offset AABB to positive coordinate system
4. Count AABB-Bin Intersections
5. (Thrust) Inclusive Scan to get offsets and total intersections
6. Store AABB-Bin Intersections
7. (Thrust) Sort intersections by bin number
8. (Thrust) Reduce to get total number of bins active
9. (Thrust) Inclusive scan of AABBs per bin to get bin offsets
10. Per bin determine number of AABB-AABB contacts
11. (Thrust) Inclusive Scan to get offsets and total contacts
12. Per bin determine AABB-AABB contact Pairs
13. Find unique pairs
14. Send pair data to Narrow-Phase

5.3.1 Stage 1: AABB Generation

Based on the type of geometry being processed, different AABB generation functions need to be used. Listing 5.1 shows the function used to compute an AABB for a sphere. AABBs must be transformed from the local to global reference frame, this process involves adding the global position to the AABB and rotating it by the associated body’s quaternion.
Note that this process needs to be performed at every time step as the body position and rotation are continually changing, while collision geometry is assumed to be static in relation to the body.

Unlike a sphere, the AABB for a box or ellipsoid is highly dependent on its orientation. Therefore, to compute the AABB, each point on the unrotated AABB is determined and then rotated into the global reference frame. Each of the three principle axes are iterated over and for each dimension, in x, y and z the largest and smallest values are computed. In this manner the AABB for the geometry is grown until the maximum is computed see Listing 5.2. The AABB for a cylinder or a cone is computed similarly.

```
void AABB_Sphere( radius, lposition, position, rotation, minp, maxp) {
    position += quatRotate(lposition, rotation); // Get Global Position
    minp = position - [radius, radius, radius]; // Minimum AABB point
    maxp = position + [radius, radius, radius]; // Maximum AABB point
}
```

Listing 5.1: AABB min/max point computation for Sphere

```
void AABB_Box(dim, lposition, position, rotation, A, minp, maxp) {
    float3 pos = quatRotate(lposition, rotation) + position; // New position
    // For all three axes
    for (int i = 0; i < 3; i++) {
        minp[i] = maxp[i] = pos[i]; // Start by adding in translation
        for (int j = 0; j < 3; j++) { // Form extent by summing smaller, larger terms
            float e = A[i][j] * dim[j];
            float f = A[i][j] * -dim[j];
            if (e < f) {
                minp[i] += e;
                maxp[i] += f;
            } else {
                minp[i] += f;
                maxp[i] += e;
            }
        }
    }
}
```

Listing 5.2: AABB min/max point computation for Box, Ellipsoid, Cylinder
5.3.2 Stage 2: Computing Bounds

This process begins by identifying the absolute minimum and maximum points for all of the AABBs in the global coordinate system. Starting with a global AABB \( \mathbf{A} \) of zero size, the AABB for each collision shape is added, growing \( \mathbf{A} \) as necessary. For this stage the Thrust transform_reduce function is utilized, see Appendix B.3.6. This transform reduce operation performs a reduction on a list of AABBs after a given transformation function is applied. This transformation function takes two AABBs and returns one AABB that contains both. At the end of the reduction process the final AABB will be the global AABB.

```cpp
typedef thrust::pair<float3, float3> bbox;
// reduce a pair of bounding boxes (a, b) to a bounding box containing a and b
struct bbox_reduction: public thrust::binary_function<bbox, bbox, bbox> {
    __device__ bbox operator()(bbox a, bbox b) {
        float3 ll; // lower left corner
        ll.x = fminf(a.first.x, b.first.x);
        ll.y = fminf(a.first.y, b.first.y);
        ll.z = fminf(a.first.z, b.first.z);

        float3 ur; // upper right corner
        ur.x = fmaxf(a.second.x, b.second.x);
        ur.y = fmaxf(a.second.y, b.second.y);
        ur.z = fmaxf(a.second.z, b.second.z);

        return bbox(ll, ur); // convert a point to a bbox containing that point,
    }
};
```

```cpp
bbox init = bbox(AABB[0], AABB[0]); // initialize bounding box to the first AABB
bbox_transformation unary_op; // bbox_transformation unary_op;
bbox result = thrust::transform_reduce(AABB.begin(), AABB.end(), unary_op, init, bbox_transformation); // bbox_transformation
float3 min_bounding_point = result.first; // minimum point
float3 max_bounding_point = result.second; // maximum point
```

Listing 5.3: Reduce a list of AABBs to a global AABB

5.3.3 Stage 3: Offset AABB

The spatial subdivision method used for the broad-phase requires that no AABBs exist in the negative coordinate system. As the maximal AABB \( \mathbf{A} \) was computed earlier, the
position of each AABB is offset by the absolute value of the minimum point of $A$. The following code listing provides the simple function that performs this operation.

```c
// min_bounding_point was computed earlier.
void Offset_AABBs(float3* AABB) {
    AABB[index] += abs(min_bounding_point);
    AABB[index + number_of_AABB] += abs(min_bounding_point);
}
```

Listing 5.4: Offset all AABBs to positive coordinate system

### 5.3.4 Stage 4: Count AABB-to-Bin Intersections

In this step all AABB-to-bin intersections are identified. An AABB can ‘touch’ more than one bin: there is no limit to how many intersections take place. The minimum and maximum points are stored in the AABB, therefore the AABB-to-bin intersections can be quickly determined. For each AABB, the 3-dimensional bin index, see Listing 5.5, is determined for the maximum and minimum points. Because the AABB is not oriented, all bins between these two indices will contain the AABB. The total number of bins intersected is stored during this stage, see Listing 5.6. The actual AABB-to-bin intersection will be stored in Stage 6.

```c
template<class T> inline unsigned int3 Hash(const T & A) {
    unsigned int3 B;
    B.x= A.x * bin_size.x;
    B.y= A.y * bin_size.y;
    B.z= A.z * bin_size.z;
    return B;
}
```

Listing 5.5: Function to compute bin index

```c
__global__ void AABB_Bins_Count(float3* device_aabb_data, uint* Bins_Intersected) {
    uint index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index >= number_of_models_const) { return; }
    //Get 3D bin index for minimum and maximum point of AABB
    uint3 gmin = Hash(device_aabb_data[index]);
    uint3 gmax = Hash(device_aabb_data[index + number_of_models_const]);
    //Total number of bins intersected, no need to count, can be computed analytically
    Bins_Intersected[index] = (gmax.x - gmin.x + 1) * (gmax.y - gmin.y+ 1) * (gmax.z - gmin.z + 1);
}
```

Listing 5.6: Counting the Number of AABB bin intersections
Figure 5.2: Example of inclusive scan, note that the first element is not included in the output.

### 5.3.5 Stage 5: Get Offsets and Total Intersections

In order to allocate the appropriate amount of memory on the GPU to store the body-bin intersection information in stage 6, an inclusive scan is performed on the list containing the number of AABB-bin intersections per AABB. After the inclusive scan operation, the last item in this array is the total intersection count. Fig. 5.2 provides an example of the input and output to an inclusive scan operation.

```plaintext
thrust::inclusive_scan(Bins_Intersected.begin(), Bins_Intersected.end());

number_of_bin_intersections = Bins_Intersected.back();
```

Listing 5.7: Inclusive Scan Code

### 5.3.6 Stage 6: Store AABB-to-Bin Intersections

For each AABB-bin intersection a unique bin identifier is computed based on that bins 3-dimensional index. This unique identifier is computed using a hashing algorithm that takes a [x,y,z] triplet and returns a unique number, see Listing 5.8. This number along with the associated AABB identifier is stored in an array, see Listing 5.9. The location used by each thread to store the intersection is determined by the inclusive process from the previous step. This array now contains the starting and ending indices for the AABB-bin intersections for each AABB, see line 10 of Listing 5.9 for usage in code.

```plaintext
template<class T>
inline unsigned int Hash_Index(const T &A){
    return (A.x * 73856093) ^ (A.y * 19349663) ^ (A.z * 83492791);
}
```

Listing 5.8: Hashing function
__global__ void AABB_Bins(float3* device_aabb_data, uint* Bins_Intersected, uint* bin_number, uint* body_number) {
    uint index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index >= number_of_models_const) {
        return;
    }
    uint count = 0, i, j, k;
    uint3 gmin = Hash(device_aabb_data[index]);
    uint3 gmax = Hash(device_aabb_data[index + number_of_models_const]);
    uint start_index = (index == 0) ? 0 : Bins_Intersected[index - 1];
    for (i = gmin.x; i <= gmax.x; i++) {
        for (j = gmin.y; j <= gmax.y; j++) {
            for (k = gmin.z; k <= gmax.z; k++) {
                bin_number[start_index + count] = Hash_Index(U3(i, j, k));
                body_number[start_index + count] = index;
                count++;
            }
        }
    }
}

Listing 5.9: Hashing function

5.3.7 Stage 7: Sort Intersections by Bin Number

The AABB-bin pairs, stored in B are currently sorted by the AABB index. This needs to be reversed so that for each bin, the indices of the associated AABB is known. To this end a key-value based sort is performed on B using the bin number and the key and the AABB index as the value.

```
thrust::sort_by_key(B_key.begin(), B_key.end(), B_value.begin())
```

Listing 5.10: Sorting by the bin number

5.3.8 Stage 8: Get Total Number of Active Bins

After the sorting process is complete, the number of objects in each bin needs to be determined. To do this a Thrust Reduce By Key operation is performed (see Appendix B.2.1). The inputs to this function are the list of sorted bin numbers which are going to be reduced (key), a Thrust Constant Iterator (value), see Appendix B.3.3, and the output of the reduced keys along with the number of objects in each bin. The Thrust
Constant Iterator with a value of 1 is used as the value for the reduction because each entry in the sorted B array represents one AABB-to-bin intersection.

```
last_active_bin= thrust::reduce_by_key(B_key.begin(),B_key.end(),thrust::
constant_iterator<uint>(1),B_key.begin(),F.begin()).first-B_key.begin()
```

Listing 5.11: Reduction to get the number of AABBs in each bin

### 5.3.9 Stage 9: Inclusive Scan of AABBs per Bin to get Bin Offsets

Using the output from the reduction, F, the start and end indices of each bin need to be computed, therefore an inclusive scan operation is performed on this list.

```
//in-place inclusive scan
thrust::inclusive_scan(F.begin(), F.end(), F.begin())
```

Listing 5.12: Inclusive Scan Code

### 5.3.10 Stage 10: Per Bin Determine Number of AABB-AABB Contacts

The AABB’s in each bin are traversed using F to determine the starting and ending indices of the AABBs touching that bin. For each AABB in the bin, an exhaustive $N^2$ search is performed for AABB-AABB intersection. If each condition in equation 3.1 is met for two AABBs A and B, an intersection has occurred. Only the number of AABB-AABB contacts is stored in this stage. The total number of contacts is used to allocate memory so that the contact information can be stored in Stage 12.

```
__global__ void AABB_AABB_Count(float3* device_aabb_data, int2* device_fam_data,
uint* bin_number, uint* body_number, uint* bin_start_index, uint*
Num_ContactD, int3* device_typ_data) {
    uint index = blockIdx.x*blockDim.x + threadIdx.x; // threadIdx.x + blockDim.x
    *threadIdx.y + (blockIdx.x*blockDim.x + blockDim.y) + (blockIdx.y *
blockDim.x + blockDim.y);
    if (index >= last_active_bin_const) {
        // Code...
    }
```

...
Listing 5.13: Counting AABB-AABB contacts

5.3.11 Stage 11: Inclusive Scan to get Offsets and Total Contacts

An inclusive scan operation is performed on the list containing the number of AABB-AABB contacts per bin. This operation results in the total number of AABB-AABB contacts as well as the starting and ending offsets necessary for storing the contact pairs in the proper locations.

Listing 5.14: Inclusive Scan
5.3.12 Stage 12: Determine AABB AABB Contact Pairs

The collision pair associated with each intersection is stored and represents the list of potential contacts for all collision shapes. This list is the input for the narrow-phase algorithm which will determine if an actual contact occurred. Collision pairs, represented by two 32 bit values, are stored in a single 64 bit value by using bit-shifts. The benefit of this is that the contact pairs can be checked for uniqueness in stage 13.

```c
__global__ void AABB_AABB(float3* device_aabb_data, int3* device_typ_data, int2* device_fam_data, uint* bin_number, uint* body_number, uint* bin_start_index, uint* Num_ContactD, long long* pair) {
    uint index = blockIdx.x * blockDim.x + threadIdx.x; // threadIdx.x + blockDim.x * threadIdx.y + (blockIdx.x * blockDim.x * blockDim.y) + (blockIdx.y * blockDim.x * blockDim.y);
    if (index >= last_active_bin_const) {
        return;
    }
    uint end = bin_start_index[index], count = 0, i = (!index) ? 0 : bin_start_index[index - 1], Bin = bin_number[index];
    uint offset = (!index) ? 0 : Num_ContactD[index - 1];
    if (end - i == 1) { return; }
    AABB A, B;
    int3 A_type, B_type;
    for (;;) {
        A.min = device_aabb_data[body_number[i]];
        A.max = device_aabb_data[body_number[i] + number_of_models_const];
        A_type = device_typ_data[body_number[i]];
        int2 FA = device_fam_data[body_number[i]];
        for (int k = i + 1; k < end; k++) {
            B.min = device_aabb_data[body_number[k]];
            B.max = device_aabb_data[body_number[k] + number_of_models_const];
            B_type = device_typ_data[body_number[k]];
            int2 FB = device_fam_data[body_number[k]];
            bool inContact = (A.min.x <= B.max.x && B.min.x <= A.max.x) && (A.min.y <= B.max.y && B.min.y <= A.max.y) && (A.min.z <= B.max.z && B.min.z <= A.max.z);
            if ((FA.x == FB.y || FB.x == FA.y) == false && inContact == true) {
                pair[offset + count] = ((long long) A_type.y << 32 | (long long) B_type.y);
                // the two indicies of the objects that make up the contact
                count++;
            }
        }
    }
}
```

Listing 5.15: Storing AABB-AABB contacts

5.3.13 Stage 13: Determine Unique Contact Pairs

The list of potential contact pairs is sorted such that contacts that were determined twice can be removed. Multiple contacts can occur if the two AABBS intersecting overlap
several bins. All bins touching both AABBs will find the contact. A Thrust Unique (see Appendix B.3.4) is then used to take the sorted list and only keep those contact pairs which are unique. This unique list is sent to the narrow-phase for further processing.

5.4 Narrow-Phase

The algorithm presented in Chapter 3 was implemented in one function on the GPU. The code in Listing 5.16 is not ideally suited to the GPU due to its complexity, however for large problems the code was never a bottleneck. Specific details are omitted for brevity, but a general overview of the source code is provided. Future work will look at improving performance and tailoring the algorithm so that it is better suited to GPU computing.

```cpp
CollideAndFindPoint() {
    float3 v01, v02, v0, n, v11, v12, v1, v21, v22, v2;
    // v0 = center of the Minkowski difference
    v01 = GetCenter(typeA, A_X, A_Y, A_Z);
    v02 = GetCenter(typeB, B_X, B_Y, B_Z);
    v0 = v02 − v01;
    // For the case where centers overlap, pick a direction
    if (IsZero3(v0)) {v0 = make_float3(1, 0, 0);}
    // v1 = support is in the direction of the origin
    n = normalize(−v0);
    v11 = GetSupportVert(typeA, A_X, A_Y, A_Z, A_R, −n);
    v12 = GetSupportVert(typeB, B_X, B_Y, B_Z, B_R, n);
    v1 = v12 − v11;
    if (dot(v1, n) <= 0) {return false;}
    // v2 = support perpendicular to v1, v0
    n = cross(v1, v0);
    // Return a hit if the origin is inside the tetrahedron
    if (IsZero(n)) {
        ...
        return true;
    }
    v21 = GetSupportVert(typeA, A_X, A_Y, A_Z, A_R, −n);
    v22 = GetSupportVert(typeB, B_X, B_Y, B_Z, B_R, n);
    v2 = v22 − v21;
    if (dot(v2, n) <= 0) {return false;}
    // Determine whether origin is on outside or inside side of plane
    n = normalize(cross((v1 − v0), (v2 − v0)));
    // If the origin is on the inside side of the plane, reverse direction of plane
    if (dot(n, v0) > 0) {
        Swap(v1, v2); Swap(v11, v21); Swap(v12, v22);
        n = −n;
    }
    // Phase 1: Portal identification
    float3 v31, v32, v3;
    while (true) {
        // Obtain the support point in a direction perpendicular to the existing
```
plane, this point is guaranteed to lie off the plane
v31 = GetSupportVert(typeA, A_X, A_Y, A_Z, A_R, -n);
v32 = GetSupportVert(typeB, B_X, B_Y, B_Z, B_R, n);
v3 = v32 - v31;
if (dot(v3, n) <= 0) { return false;
else if (dot(cross(v1, v3), v0) < 0) {
    // Origin is outside (v1,v0,v3), eliminate v2 and loop ...
    continue;
} else if (dot(cross(v3, v2), v0) < 0) {
    Origin is outside (v3,v0,v2), eliminate v1 and loop ...
    continue;
}
break;
}
bool hit = false;
// Phase 2: Portal refinement
int iteration_counter = 0;
while (true) {
    iteration_counter++;
    // Compute normal of the face
    n = cross((v2 - v1), (v3 - v1));
    n = normalize(n);
    // Compute distance from origin to the face.
    // If the origin is inside, a hit has occurred
    if (dot(n, v1) >= 0. && !hit) {
        hit = true; // a contact has occurred
    }
    // Find the support point in the direction of the wedge face
    float3 v41 = GetSupportVert(typeA, A_X, A_Y, A_Z, A_R, -n);
    float3 v42 = GetSupportVert(typeB, B_X, B_Y, B_Z, B_R, n);
    float3 v4 = v42 - v41;
    float delta = dot((v4 - v3), n);
    depth = -dot(v4, n);
    // If the boundary is thin enough, based on tolerance, or the origin is
    // outside the support plane for the new vertex, terminate
    if (delta <= kCollideEpsilon || depth >= 0. || iteration_counter > 100) {
        if (hit) {
            // Compute the barycentric coordinates (b0,b1,b2,b3) of the origin in the
            // coordinate system of the portal
            ...
            float inv = 1.0f / sum;
            point = (b0 * v01 + b1 * v11 + b2 * v21 + b3 * v31) + (b0 * v02 + b1 *
                v12 + b2 * v22 + b3 * v32);
            point *= inv * .5;
            returnNormal = normalize(n);
        } return hit;
    }
if (dot(cross(v4, v1), v0) < 0.) { // Compute the tetrahedron face (v4,v0,v1)
    if (dot(cross(v4, v2), v0) < 0) { // Compute the tetrahedron face (v4,v0,v2)
        // Inside face1 & inside face2, eliminate v1 ...
    } else { // Inside face1 & outside face2, eliminate v3 ...
    }
} else { // Compute the tetrahedron face (v4,v0,v3)
    if (dot(cross(v4, v3), v0) < 0.) { // Outside face1 & inside face3 =>
        eliminate v2 ...
    } else { // Outside face1 & outside face3 => eliminate v1 ...
}
5.5 Summary

This chapter proposed a spatial subdivision based approach for parallel collision detection on the GPU. The algorithm combines two paradigms, CPU computing and GPU computing, to enable in an open source code effective and scalable parallel collision detection for scenarios with billions of collision events. The proposed algorithm significantly enlarges the solvable problem size: from 6 million collision events with the Bullet Physics Engine, to either billions of events in a multi-GPU configuration, or to approximately 23 million collision events when using only one GPU card [64]. Moreover, when compared to Bullet, the proposed algorithm was two orders of magnitude faster and eliminated collision detection as a computational bottleneck in the physics based simulation of granular material dynamics. Ongoing work aims at streamlining the implementation to leverage the Message Passage Interface (MPI) standard for cluster simulation setups, which represents a more challenging undertaking due to load balancing issues. An ongoing effort will aim at large scale collision detection of geometries defined through Delaunay triangulations. The collision detection framework outlined herein should accommodate these geometries with minimal implementation changes.
6 NUMERICAL VALIDATION

6.1 Experimental Validation: Mass Flow Rate

This section focuses on comparing experimental measurements against the theoretical results of granular flow through a channel obtained in the Chrono::Engine simulation environment [18]. To this end, an aluminum rig was designed and fabricated to measure the flow rate of a given amount of granular material flowing through a slit due to gravity. Additionally, a model of the system was created in Chrono::Engine and the results were matched to experimental runs by changing the friction coefficient between particles. After the friction coefficient of the particles was calibrated, several experimental runs with differing gap sizes were run. These flow rates were compared to the weight versus time data that Chrono::Engine output for the corresponding slit size. Along with comparing mass flow rates, the pile angle from the experimental runs was compared to the simulation results.

6.1.1 Experimental Model

Due to the limitations of the experimental hardware at the time, this experiment was performed with 50000 to 60000 particles (weighing about a tenth of a Newton) could be simulated. Because of this small amount of granular mass, it was necessary to be able to open the slit quickly so that the transient nature of the startup did not impact the measure of mass flow rate. Along with this, the opening mechanism had to have enough damping to reduce any dynamic response that might occur. To deal with these issues, a preexisting translational stage controlled by an actuator was used. The cavity was created using two sub-parts; an angled slot and the static slot. The angled slot was attached to the translational stage whereas the static slot was fixed to the base. The static slot component was a single part with the slot machined into the piece. This design aspect
was chosen for portability and ease of installation. The dimensions of the parts were such that the use of the aluminum stock that was made available by the University of Wisconsin-Madison Student Shop was optimal [65].

Due to the ease of which the actuator could alter the gap size of the flow meter, several experimental runs were taken for varying gap sizes. The desired gap size was measured using a digital caliper. A fixed weight of 0.0624 N of granular material was used for each test with an uncertainty of 0.0004 N due to measurement error. Based on the results of these experiments, an allowable "bandwidth" could be created with which the simulation results could be compared to.

6.1.2 Simulation Model

In order to validate the experimental results described above with those produced by the simulation engine, it is necessary to build a model that represents the actual setup with as much detail as possible. The experimental setup consists of four major parts: the trough, the spheres, the translational stage and the load cell.

The glass spheres are described as perfect spheres. Each sphere has the same mass and coefficient of friction. The trough, with its three walls and slope, is represented as rectangular boxes with finite dimensions. In the experiment the trough is opened by a translational stage. This stage moves the slope out of the holder. In the simulation model this behavior is described as a motion of the box representing the slope. The motion is captured from the data sheet of the translational stage. Different gap sizes can be adjusted for each simulation and the opening time varied. The load cell measures the outflow through the gap. Because this measurement is based on the weight per time it can be implemented as a count of spheres under a certain height for each time step and be outputted into a data file for later post processing. The number of spheres multiplied by the mass and gravity yields the weight which can be compared with experimental results.
The spheres leaving the trough are falling on a plane described as the collector in the experimental setup.

In order to save computational time, the simulation is split into two parts: one containing the filling process of the trough and the other the opening and measure process. If the same design of the trough is observed multiple times, for example at different gap sizes, the filling process is always identical. Therefore it is unnecessary to compute this process for each observation and only needs to be run once. The second part can then be used for observation of the behavior. In the filling process the same amount of spheres will be created as used in the real experiment and randomly distributed in the trough. After the spheres have settled, the x-, y-, and z-position of each sphere is saved for the following simulation. Settling was complete when the kinetic energy of the system was below 0.001 Joules and had reached a relatively constant value. At the beginning of the outflow simulation the position data set of the spheres is loaded into the model and the spheres will be created at the same positions they appeared in the filling process. The material properties of the spheres can be set as desired. The positions of each sphere can be saved at each time step for further post processing observation and the outflow is also saved at each time step.

The simulations setup consisted of 39000 rigid body spheres with a radius of 2.5e-4 m and a mass of 1.631e-7 kg. The slope of the inclined plane was 45° and the friction value of the wall-glass sphere interface was set to the friction coefficient was .6 which is roughly the friction coefficient of glass against metal. The friction coefficient was calibrated by running several tests and determining which friction value resulted in the most accurate simulation. This friction value was determined to be 0.15, using a gap size of 1.5 mm, and was used for every gap size. The following parameters were set for this simulation. A time step of 1e-4 [s] with 500 CCP iterations, a contact omega of 0.8 and a tolerance of 1e-7 for the maximum velocity correction. Simulations were generally run for 8 seconds.
SI units were used for all parameters.

**Procedure Used to Select the Value of \( \mu \)**

The friction coefficient of a certain material is not a constant value. It can depend on various environmental influences such as humidity, surface quality, temperature etc. The friction coefficient of glass was an unknown in the validation process and needed to be determined before further observations could be done. To achieve this, one experiment at a gap size of 1.5 mm was performed and multiple simulations with the same setup and different friction coefficients were computed. The resulting mass outflows distributed in a certain manner and the experimental result lay somewhere in between.

### 6.1.3 Results

Several experimental runs were performed for varying gap sizes. A fixed weight of 0.0624 N of granular material was used for each test with an uncertainty of 0.0004 N due to measurement error. Based on the results of these experiments, a range of values was obtained that was later used in validation.

**Results of 3mm Gap**

Shown in Fig. 6.1 are the results with a gap size of 3mm and 5 experimental runs. The average slope of the experimental runs was 1.41E-2 [N/s] and the slope of the simulation was 1.40E-2 [N/s].

**Results of 2.5mm Gap**

Shown in Fig. 6.2 are the results with a gap size of 2.5mm and 5 experimental runs. The average slope of the experimental runs was 2.59E-2 [N/s] and the slope of the simulation was 2.62E-2 [N/s].
Figure 6.1: Weight vs time for a gap size of 3 mm

Figure 6.2: Weight vs time for a gap size of 2.5 mm
Results of 2.0mm Gap

Shown in Fig. 6.3 are the results with a gap size of 2mm and 5 experimental runs. The average slope of the experimental runs was 4.00E-2 [N/s] and the slope of the simulation was 4.05E-2 [N/s].

Results of 1.5mm Gap

Shown in Fig. 6.4 are the results with a gap size of 1.5mm. The average slope of the experimental runs was 4.44E-2 [N/s] and the slope of the simulation was 4.48E-2 [N/s].

6.2 Summary

The experimental flow rate is based off of the average of several experimental runs and the uncertainty of the rates is determined by a Student’s T distribution with 95% confidence.
Based on an error of less than 2%, the simulated results match the experimental results well. Despite this, there are several factors that could be improved upon. Namely, the design of the rig did not exactly match the simulated trough. The rig was machined from aluminum, whereas the trough in the simulation had the same coefficient of friction as the granular material. Likewise, the simulation was conducted in essentially vacuum, ignoring any aerodynamic forces, while the experiment was performed at ambient conditions. Lastly, the effects of humidity were not taken into account by the experiment.
In this chapter the performance of the GPU implementation will be analyzed. The GPU parallel algorithms, implemented in the Chrono::Engine physics engine [18] will be compared to sequential CPU based methods (provided by Chrono::Engine). Specifically, CPU collision detection functionality is provided by the bullet physics library [66] and the constraint solver is a sequential CPU implementation of the DVI method presented in chapter 2. The GPU used for the following tests is an NVIDIA GTX 480 [67] and the CPU used is an Intel Xeon E5520 quad-core processor [68]. It should be noted that the algorithms running on the CPU were performing double precision arithmetic while the GPU algorithms were performing computations in single precision.

7.1 CPU vs GPU Comparison

For the CPU vs GPU comparison, simulations with increasing number of bodies were run. In each scenario, granular material was dropped into a box enclosed on all sides by rigidly fixed walls. Three seconds of this simulation were simulated, the total time taken for each simulation is presented in Fig. 7.1. The number of bodies simulated ranged from about 2300 for the smallest case to 16000 for the largest. It should be noted that for simulations smaller than 2000 bodies, the overhead associated with transferring data between the GPU and CPU is very large compared to the computational time. Also when solving such a small problem on the GPU, the large number of processing cores (480 in the case of the GTX 480) means that many cores will be idle and the full compute power of the GPU will not be utilized. Therefore for small problems it is recommended that the CPU algorithms be used.

Fig. 7.2 shows the speedup when comparing the total time taken to simulate on the CPU vs the GPU. The speedup increases as the problem size increases, for this scenario a
speedup of $11\times$ was achieved at the largest simulation size.

## 7.2 Scaling Analysis

For an identical simulation setup to that presented in section 7.1, larger amounts of bodies were dropped into the box. Note that for the previous comparison between the CPU and GPU the maximum number of bodies simulated was approximately 16000, for this analysis the maximum number of bodies simulated was an order of magnitude higher at around 250000. Fig. 7.3 shows that the GPU algorithms scale linearly with respect to the number of bodies and subsequently, the number of contacts as more bodies will result in more contacts. For small numbers of bodies, from 2000 to 10000, the total simulation time for the GPU is relatively flat. This is because the overhead associated with transferring memory from the CPU to the GPU is higher than the time taken for computations. Once the number of bodies gets higher than 20000 the total simulation time begins to increase.
Figure 7.2: Speedup of CPU vs GPU

Figure 7.3: Scaling of the GPU for different amounts of bodies
8 APPLICATION 1: ANCHORING IN GRANULAR MATERIAL

8.1 Introduction

NASA is interested in designing a spacecraft capable of visiting a Near Earth Object (NEO), performing experiments, and then returning safely. Certain periods of this mission will require the spacecraft to remain stationary relative to the NEO. Due to the low gravity, such situations require an anchoring mechanism that is compact, easy to deploy and upon mission completion, easily removed. Using Chrono::Engine [18, 69], a simulation package capable of utilizing massively parallel GPU hardware, extensive validation experiments will first be performed. A set of parametric studies will concentrate on the simulation of the anchoring system. The outcome of this effort will be a systematic study that considers several different anchor designs, along with a recommendation on which anchor is better suited to the task of anchoring. The anchors will be tested against a range of parameters relating to soil, environment and anchor penetration angles/velocities on a NEO to better understand its performance characteristics.

8.2 Anchor Model

The anchor was modeled using three types of regular primitives. The tip of the anchor was modeled using a sphere, the shaft was modeled using a cylinder, and the helix was modeled using 67 thin boxes swept along a helical spline. This configuration is optimal compared to using a triangulated mesh for the anchor. The triangular mesh requires several thousand triangles, decreasing the performance of the collision detection and increasing the memory requirements for the simulation. Using primitives has an added benefit, modifying the geometry of the anchor becomes straightforward, the pitch of the anchor along with its diameter and thickness can be varied easily allowing parametric
studies to be completed.

The anchor had several constraints and forces which were used to control its motion. First there was a bilateral constraint restricting the planar motion of the anchor, forcing it to move straight up or down, this constraint also prevented the anchor from rotating in all axes except the vertical axis. A pressing force was used to press the anchor into the material. This simulated an actuator which may be attached to the other end of the anchor, forcing it to penetrate. A torque was applied to the anchor to cause it to rotate and screw into the material, and a vertical force was used to pull the anchor out of the material at the end of the simulation, see Fig. 8.1.

### 8.3 Numerical Experiments

The numerical experiment consisted of a granular bed made up of spheres of randomly varying radii that was pre-settled and loaded at the start of each simulation. For each simulation only the parameters associated with the anchor were varied. The anchor’s mass was 10 kg with a radius of 0.5m. The granular material had a mass of 0.005 kg and a radius randomly varying between 0.025 m and 0.036 m. The granular material had a friction coefficient of 0.4, and gravity was set to \(-9.806 \, \text{m/s}^2\). The time step was 5e-4 with 1000 CCP iterations performed per time step.

### 8.4 Results

Several sets of parametric tests were simulated using the anchor model: the torque applied to the anchor was varied, and the pullout force applied after anchoring was varied. Fig. 8.2 shows an anchor with the same mass that has 4 different torques applied to it. For each test at the end the pullout force remained constant at 300N. The plot shows that at 2 seconds, when the anchoring torque was applied, the anchor with the highest torque went
Figure 8.1: Stages of anchoring, first anchor drops onto granular bed (a), then an applied torque anchors the anchor (b,c), after which it is pulled out (d).

in the deepest, fastest, which is as expected. With the constant pulling force that was applied at 7 seconds only the anchor with the lowest anchoring torque, 600 N-m, was pulled upwards, the mass of the granular material above the 3 other anchors was too high for the force to have any effect.

Fig. 8.3, with a close up in Fig. 8.4 show a different set of simulations. Here the anchoring torque was kept constant but the pullout force was changed. The purpose of the test was to gauge the magnitude of the pullout force required for a given applied torque. The plot shows that only a force of 2000N was able to pullout easily, gaining
Figure 8.2: Anchor with different applied torques and a constant pullout force of 300N velocity as it moved upwards. The pullout force of 1600N was able to start pulling out slowly and at a constant velocity.

8.5 Summary

Capturing the physics of granular material accurately is important when studying the interaction between an anchor and the material that it anchors into. One important factor in such simulations not captured currently are the cohesive forces. Simulating cohesion accurately is an important step in capturing rigid body dynamics, especially in a NEO (low gravity) environment. A short range force model needs to be added to the simulation capability. Once this is complete, simulations of the anchor will be performed in a low gravity environment, where the effects that cohesion has on anchoring will be investigated.
Figure 8.3: Anchor with different pullout forces and a constant torque of 1000N

The goal of this effort is to simulate several different anchor designs, and at the end provide a recommendation on which type of anchor performs the best when tested against a range of parameters relating to soil, environment and anchor penetration angles/velocities. Before the simulations of the anchor can be trusted, experimental validation needs to be completed. When comparing actual simulation data to that presented in literature several problems are still being worked upon, such as contact stiffness, and rate of convergence for the CCP solver.
Figure 8.4: Closeup of Fig. 8.3, Anchor with different pullout forces and a constant torque of 1000N
9 APPLICATION 2: TRACKED VEHICLE ON GRANULAR TERRAIN

9.1 Introduction

This application focuses on simulating a complex system comprised of many bilateral and unilateral constraints. Using a combination of revolute joints and linear actuators a track model was created and then simulated navigating over terrain made up of granular material. The vehicle is modeled to be a small lightweight tracked vehicle much like an autonomous robot that could be sent to another planet or one used to navigate dangerous terrain.

9.2 Model Description

The track model consists of two tracks, each with 61 track shoes (see Fig. 9.1). Each track shoe is made up of two cylinders and 3 rectangular plates. Each shoe is connected to its neighbors using one pin joint on each side, allowing the tracks to rotate relative to each other only along one axis. Within each track there are 5 rollers, one idler and one sprocket. The purpose of the rollers is to keep the tracks separated and support the weight of the vehicle as it moves forward. The idler is necessary as it keeps the track tensioned. It is usually modeled with a linear spring/actuator but for the purposes of demonstration it was fixed using a revolute joint to the vehicle chassis. The sprocket is used to drive the vehicle and is attached using a revolute joint to the chassis. Torque is applied to drive the track, with each track driven independent of the other. When the sprocket rotates it comes into contact with the cylinders on the track shoe and turns the track with a gear like motion.

The track for the vehicle was created by first generating a ring of connected track
shoes. This ring was dropped onto a sprocket, 5 rollers, and an idler which was connected to the chassis using a linear spring. The idler was pushed with 2000 N of force until the track was tensioned and the idler had stopped moving. This pre-tensioned track was then saved to a data file and loaded for the simulation of the complete vehicle.
9.3 Simulation Scenarios

The first simulation scenario was of the tracked vehicle dropping onto a flat plate. This test was designed to show that the pre-tensioning step was successful and that the model was stable and working properly. Using this scenario the simulation parameters, such as the time step and correction factors for the solver, were tuned. Once the proper values were determined, a second scenario was simulated. In this scenario the tank was again dropped onto the flat surface, but this time a torque was applied to the sprocket to move it forward. The motion of the tank was simulated for 10 seconds and observations were made to determine if any irregular behavior occurred in the track as it rotated around the idler, rollers and sprocket. Due to the tension of the track and the forces that it experienced, the track could fail when the constraints holding the shoes together were violated. Sufficient solver iterations and a small time step ensured that this did not occur. For the purposes of this application configuration with a time step of 0.005 s and 1000 CCP iterations was found to perform well.

The third scenario simulated involved the tank dropped on an inclined surface, see Fig. 9.3. The purpose of this scenario was to see whether the track was stable when resting on non-spherical geometry. This simulation was a stepping stone for when the tracked vehicle is simulated on a bed of granular material with similar characteristics, see Fig. 9.2

The final scenario involved the tracked vehicle navigating a surface comprised of 100,000 randomly varying spheres of different radii, see Figure 9.4. This was the first test where the tracked vehicle was driven over a granular surface, rather than just resting on it. The simulation provided important qualitative information about how well the vehicle performed and if problems occurred as it rolled over the terrain. More work needs to be done with a larger bed (more than 100,000 bodies) so that the track-terrain interaction can be studied further.
Figure 9.3: Tank on inclined surface, geometry of tank model is purely cosmetic.

Figure 9.4: Tank on a bed of granular material comprised of spheres of varying radii, geometry of tank model is purely cosmetic.
9.4 Results

For the second simulation scenario, where the tracked vehicle was dropped onto a flat surface and a torque was applied to the sprocket to drive it forwards, the forces on several revolute joints connecting the track shoes were analyzed as they traveled around the sprocket. Fig. 9.5 shows the forces in one revolute joint after the track has dropped onto the flat surface. Transient behavior is observed when the torque is applied to the sprocket at 1 second and the track shoe connected to this joint comes into contact with the sprocket at 5 s. The oscillatory behavior of the joint forces can be attributed to several factors, first the tension in the track was very high, there was no spring/linear actuator attached to the idler so high tension forces could not be dampened. Secondly, the combination of a high pre-tensioning force (2000 N) and lack of a linear actuator on the idler resulted in high revolute joint forces. Finally, the masses and moments of bodies making up the track model were not accurate as the masses of the track shoes especially were too low.

Figure 9.6 shows the joint forces for several revolute joints as their associated track shoes go around the sprocket. This plot shows that the forces in the joint are highest when the track shoe first comes into contact with the sprocket. As the track shoe moves around the sprocket, the force decreases as subsequent track shoes and their revolute joints help to distribute the load. It should be noted that the gearing motion between the track shoes and the sprocket was not ideal as it was not very smooth. In a more realistic model forces between track shoes would be overlapping so that the movement of the tracks is more smooth and the forces experienced by the revolute joints is smaller.

9.5 Summary

The simulation capability has demonstrated that it can simulate a tracked vehicle navigating a bed of granular material. Work remains to make the tracked vehicle model more
Figure 9.5: Magnitude of force experienced by one revolute joint

![Force vs Time](image)

Figure 9.6: Magnitude of force experienced by 5 revolute joints

![Force vs Time](image)
accurate; the gearing motion between the sprocket and the track shoes is not smooth causing large forces in the track shoes as they go around the sprocket. The mass and inertial properties of the tracked vehicle need to be adjusted so that they are more accurate. For the purposes of this experiment, and to demonstrate the simulation capability, these values were not important.

Work is currently being done on simulating the tracked vehicle navigating a rough granular bed, see Fig. 9.2. This bed is made up of cylinders, rectangular boxes, spheres and ellipsoids. This granular surface was generated by dropping a random distribution of these shapes and allowing them to settle.
About 50% of all traded products worldwide are in granular form. Grain, rice, sugar, coffee, cereal, salt, sand, gravel, drug pills, and fertilizers are examples of granular material. Granular material and its storing/packing/motion come up in the design of a combine, the mobility of a Mars rover, avalanche dynamics, earthquakes, and the formation of asteroids and planets.

In this context, this work concentrated on the two main components of granular dynamics simulation, the constraint solver and the collision detection algorithm. Parallel implementations of both were provided along with numerical results relating to validation and scaling. The methods were then used to analyze two different applications, namely anchoring in granular material and the simulation of a tracked vehicle operating over granular terrain.

Specific contributions of the author include the following:

1. Implementation and optimization of CCP solver on the GPU.
2. Development and implementation of GPU parallel broad-phase for large collision detection problems.
3. Implementation and optimization of MPR based narrow-phase convex-convex collision detection on the GPU.
4. Optimization of GPU based Chrono::Engine to decrease CPU-GPU transfers and improve performance.
5. Development of anchor model to perform a parametric analysis of how anchor geometry affect anchoring behavior in different types of granular materials.
6. Visualization setup for rendering and visual debugging purposes using Pov-Ray [70]. (See Fig. 10.1 for an example)

Future work in the Simulation Based Engineering Laboratory will seek to improve the performance and robustness of the DVI solver and collision detection implementations. Multi-body dynamics simulations on multiple GPUs is also being actively pursued. This
Figure 10.1: Simulation of tower collapsing with a granular flow

effort will leverage the lab’s cluster of GPU enabled machines allowing for simulations larger by an order of magnitude or more in terms of the number of objects simulated and contacts processed.
A COMMON FUNCTIONS

A.1 Quaternion Math

```c
__device__ __host__ inline float4 inv(const float4& a) {
    return (1.0f / (dot(a, a))) * F4(a.x, -a.y, -a.z, -a.w);
}
```

Listing A.1: Function to compute inverse of a quaternion

```c
__device__ __host__ inline float4 mult(const float4 &a, const float4 &b) {
    float w0 = a.x;
    float w1 = b.x;
    float3 v0 = F3(a.y, a.z, a.w);
    float3 v1 = F3(b.y, b.z, b.w);
    float4 quat = F4(w0 * w1 - dot(v0, v1), w0 * v1 + w1 * v0 + cross(v0, v1));
    return quat;
}
```

Listing A.2: Function to multiply two quaternions

```c
__device__ __host__ inline float3 quatRotate(const float3 &v, const float4 &q) {
    float4 r = mult(mult(q, F4(0, v.x, v.y, v.z)), inv(q));
    return F3(r.y, r.z, r.w);
}
```

Listing A.3: Function to rotate a vector by a quaternion
B THRUST

B.1 Introduction

Thrust is a C++ template-based library implemented using the CUDA API and is based on the C++ Standard Template Library (STL). Using Thrust, parallel applications can be developed on the GPU with minimal effort, which result in high performance through a high-level API.

The Thrust Library provides a collection of data parallel primitive algorithms such as scan, sort and reduce, among others. These simple operations can be combined together to implement complex algorithms with minimal source code. Because of its high-level API, Thrust can freely select the most optimal parameters for the GPU being used so that maximum performance is achieved. Algorithms are designed to be robust and are extremely useful for prototyping purposes. Thrust also implements the STL vector on the GPU, increasing productivity by abstracting low-level CUDA functions.

For more information see: [59]

B.2 Basic Algorithms

The examples shown below are taken from the Thrust documentation and are meant to illustrate the basic functionality of Thrust. For a more complete overview of functionality refer to [59].

B.2.1 Reductions

Reduce By Key

```
#include <thrust/reduce.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1}; // input keys
int B[N] = {9, 8, 7, 6, 5, 4, 3}; // input values
```
int C[N]; // output keys
int D[N]; // output values

thrust::pair<int*, int*> new_end;
new_end = thrust::reduce_by_key(A, A + N, B, C, D);
// The first four keys in C are now {1, 3, 2, 1} and new_end.first - C is 4.
// The first four values in D are now {9, 21, 9, 3} and new_end.second - D is 4.

Listing B.1: Function to rotate a vector by a quaternion

B.2.2 Prefix-Sums

#include <thrust/scan.h>

int data[6] = {1, 0, 2, 2, 1, 3};

// in-place scan

// data is now {1, 1, 3, 5, 6, 9}

Listing B.2: Function to rotate a vector by a quaternion

B.2.3 Sorting

Sort By Key

#include <thrust/sort.h>

... const int N = 6;
int keys[N] = {1, 4, 2, 8, 5, 7};
char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};

thrust::sort_by_key(keys, keys + N, values);
// keys is now {1, 2, 4, 5, 7, 8}
// values is now {'a', 'c', 'b', 'e', 'f', 'd'}

Listing B.3: Function to rotate a vector by a quaternion

B.3 Miscellaneous

B.3.1 Sequence

#include <thrust/sequence.h>

... const int N = 10;
int A[N];

thrust::sequence(A, A + 10, 1);
// A is now {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

Listing B.4: Function to rotate a vector by a quaternion

B.3.2 Fill
#include <thrust/fill.h>
#include <thrust/device_vector.h>

... 

thrust::device_vector<int> v(4);
thrust::fill(v.begin(), v.end(), 137);


Listing B.5: Function to rotate a vector by a quaternion

B.3.3 Iterators

#include <thrust/iterator/constant_iterator.h>

thrust::constant_iterator<int> iter(10);

*iter; // returns 10
iter[0]; // returns 10
iter[1]; // returns 10
iter[13]; // returns 10

// and so on...

Listing B.6: Function to rotate a vector by a quaternion

B.3.4 Unique

#include <thrust/unique.h>

... 

const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1};
int *new_end = thrust::unique(A, A + N);
// The first four values of A are now {1, 3, 2, 1}
// Values beyond new_end are unspecified.

Listing B.7: Function to rotate a vector by a quaternion

B.3.5 Remove

#include <thrust/remove.h>

... 

struct is_even
{
  __host__ __device__
  bool operator()(const int x)
  {
    return (x % 2) == 0;
  }
};

... 

const int N = 6;
int A[N] = {1, 4, 2, 8, 5, 7};
int *new_end = thrust::remove_if(A, A + N, is_even());
// The first three values of A are now {1, 5, 7}
// Values beyond new_end are unspecified.

Listing B.8: Function to rotate a vector by a quaternion
B.3.6 Transformations

```cpp
#include <thrust/transform_reduce.h>
#include <thrust/functional.h>

template<typename T>
struct absolute_value : public unary_function<T,T>
{
    __host__ __device__ T operator() (const T &x) const
    {
        return x < T(0) ? -x : x;
    }
};

int data[6] = {-1, 0, -2, -2, 1, -3};
int result = thrust::transform_reduce(data, data + 6, absolute_value<int>(), 0, thrust::maximum<int>());
// result == 3
```

Listing B.9: Function to rotate a vector by a quaternion
REFERENCES


