

École polytechnique de Louvain

Multi Bodies Simulation using Non Smooth Contact Dynamics on GPU

Author: **Alexandre DEWILDE**
Supervisors: **Vincent LEGAT, Jonathan LAMBRECHTS**
Readers: **Tom BARBETTE, Simon YANS**
Academic year 2024–2025
Master [120] in Computer Science

Abstract

The growing computational power of GPUs over the past three decades, combined with the stagnation of single-core CPU performance, has led mechanical engineering to leverage GPUs for complex simulations. This shift has significantly expanded the range of problems that can be solved within a reasonable timeframe.

Non-Smooth Contact Dynamics (NSCD) is a numerical method for simulation of numerous rigid bodies with contacts, such as granular materials. This thesis presents an implementation of a multi-circular simulation approach using a Jacobi solver, enabling the simulation of complex shapes with NSCD on GPUs. Several implementations are presented and compared in terms of precision, convergence, and performance.

Acknowledgements

First, I would like to thank my supervisors, Jonathan Lambrechts and Vincent Legat, for their guidance throughout this master's thesis year.

I would also like to thank Simon Yans (alias *Fat Cat*, he will understand) for his assistance with the implementation and for his patience in answering my questions.

I am grateful to Simon Yavari for answering my questions about smooth methods and for providing his excellent Python script to plot the simulations.

I would also like to thank my *co-kotteurs du Carpe* for their support, for listening to my complaints, and for encouraging me during the final rush.

Finally, I would like to thank my family and friends for their support and encouragement throughout this year. I would not have been able to complete this thesis without them.

AI usage

AI tools were used to assist in the writing of this master thesis. The following tools were utilized:

- **GitHub Copilot:** Assisted in coding and debugging during the implementation phase. It also helped in writing the master thesis, reformulation, correcting grammar and spelling mistakes.
- **Perplexity:** Used for paper research using the web search and scientific research functions.
- **NotebookLM:** Helped to search inside various documents and papers, while citing its sources.
- **ChatGPT:** Helped for tikz figure creations.

List of acronyms

CUDA	Compute Unified Device Architecture
DEM	Discrete Element Method
NSCD	Non-Smooth Contact Dynamics
CPU	Central Processing Unit
GPU	Graphics Processing Unit
GPGPU	General-Purpose computing on Graphics Processing Units
GPC	Graphics Processing Cluster
TPC	Texture Processing Cluster
SM	Streaming Multiprocessor
SFU	Special Function Unit
FP32	32-bit Floating Point
FP64	64-bit Floating Point
ALU	Arithmetic Logic Unit
FPU	Floating Point Unit
INT32	32-bit Integer
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Threads
BVH	Bounding Volume Hierarchy
FPGA	Field-Programmable Gate Array
MigFlow	GPU-accelerated framework for Non-Smooth Contact Dynamics simulations
TDP	Thermal Design Power
CSR	Compressed Sparse Row
LBVH	Linear Bounding Volume Hierarchy
KD-tree	K-dimensional tree
ED	Event-Driven (method)

SOR	Successive Over-Relaxation
CFD	Computational Fluid Dynamics
API	Application Programming Interface
RAM	Random Access Memory
AI	Artificial Intelligence

Contents

1	Introduction	1
1.1	Introduction	1
1.1.1	Context	1
1.1.2	Motivation	1
1.2	Bodies Simulation	2
1.2.1	Bodies shape	2
1.2.2	Methods	2
1.2.3	Contact Detection	4
1.3	GPU	5
1.3.1	History	5
1.3.2	CPU and GPU	6
1.3.3	Languages	7
1.4	Current State of the Art for Body and Multi-Body Simulation on GPUs	8
1.4.1	Discrete Element Method	8
1.4.2	Non-Smooth Contact Dynamics	9
1.5	Current Work and Goal of this Master Thesis	9
1.5.1	Migflow	9
1.5.2	Validation of Results	9
1.5.3	Hardware and Software	9
1.5.4	Goal	10
2	GPU architecture and programming	11
2.1	GPU Architecture	11
2.1.1	Ada Lovelace Architecture	11
2.1.2	Overview	11
2.2	Memory Hierarchy	13
2.3	Programming Model	14
2.3.1	Hardware and Software Equivalence	14
2.3.2	Thread, Block, and Grid	14
2.3.3	Warp	14
2.4	GPU Programming	15
2.4.1	Parallel Programming	15
2.4.2	Parallel Reduction	15
2.5	Optimization	16
2.5.1	Coalescing	16
2.5.2	Occupancy	16
2.5.3	Data Transfer	17

2.5.4	How Our Program Can Be Limited	17
2.5.5	GPU Application Profiling	17
3	Contact detection: Bounding volume hierarchy	19
3.1	BVH Overview	19
3.2	BVH for Contact Detection	19
3.3	Quality of the BVH	20
3.3.1	Criteria	20
3.3.2	Contact Detection Choice	20
3.4	Classical BVH Construction	20
3.4.1	Top-down	20
3.4.2	Bottom-up	21
3.4.3	Online	21
3.5	GPU Adaptation	22
3.5.1	Linear Bounding Volume Hierarchy	22
3.5.2	Migflow Implementation	22
3.5.3	Improvements and Alternatives	26
3.6	Contacts Storage	27
4	Non Smooth Contact Dynamics	28
4.1	Modelisation	28
4.2	Contacts: Local Frame and Global Frame	29
4.3	Time discretization	29
4.4	Contact unilaterality and friction	30
4.5	Contact Resolution	31
4.6	Solving the problem	32
4.6.1	Gauss-Seidel	32
4.6.2	Jacobi method	32
5	Implementation	34
5.1	Data Structure Adaptation	34
5.1.1	MigFlow Data Structure	34
5.1.2	Particles Stucture Implementation	35
5.1.3	Constant Bodies	36
5.2	Contact Detection	36
5.3	Contact Resolution	37
5.3.1	Contact Resolution for Bodies	37
5.3.2	Velocity Update for Bodies	37
6	Results and performance	39
6.1	Simulations presentation	39
6.1.1	Simulation 1: Stacked particles	39
6.1.2	Simulation 2: Tetris	40
6.2	Precision	41
6.2.1	Simulation Comparisons	41
6.2.2	Contact reactions and normals	43
6.2.3	Precision analysis	43
6.3	Convergence	44
6.3.1	Gauss-Seidel vs Jacobi convergence	44

6.3.2	Convergence parameters	44
6.4	Performance	47
6.4.1	Program hotspots	47
6.4.2	Comparison of GPU implementations	48
6.4.3	CPU vs GPU	50
Conclusion		51

Chapter 1

Introduction

1.1 Introduction

1.1.1 Context

Over the past four decades, graphics cards and their GPU chips, originally designed for graphical computations, have evolved into powerful and essential computational units in computer science.

The programmability of GPUs, the introduction of the stream processing paradigm, and Ian Buck's development of the "Brook" programming language [Buck et al., 2004] during his PhD thesis have significantly improved the accessibility of general-purpose processing on graphics processing units (GPGPU). The introduction of CUDA by NVIDIA in 2006-2007 [Nickolls et al., 2008] marked a turning point in the computing world.

The availability of affordable GPU computing power, combined with the stagnation of single-core CPU performance and the physical limitations of CPUs signaled by the end of Dennard scaling [Dennard et al., 1974], as well as NVIDIA CEO Jensen Huang's controversial assertion that Moore's law [Moore, 1965, Moore, 1975] is obsolete¹, highlight the necessity of adapting various algorithms and programs to these highly parallel architectures.

From mechanical engineering to machine learning, GPUs have become indispensable for solving *embarrassingly parallel* problems, which are well-suited for parallelization.

1.1.2 Motivation

In mechanical engineering, the simulation of bodies and multi-bodies is a highly valuable field with numerous applications in both research and industry. Examples include the simulation of granular materials, such as regolith [Schwartz et al., 2014], which is present on the moon and enables the study of regolith erosion during lunar landings [Berger and Hrenya, 2017], as well as applications in agricultural machinery [Zhao et al., 2021]. These simulations allow engineers to test ideas under various conditions, optimize designs, and predict behavior before physical implementation.

¹This claim is controversial; Intel's CEO replied that it's not dead.

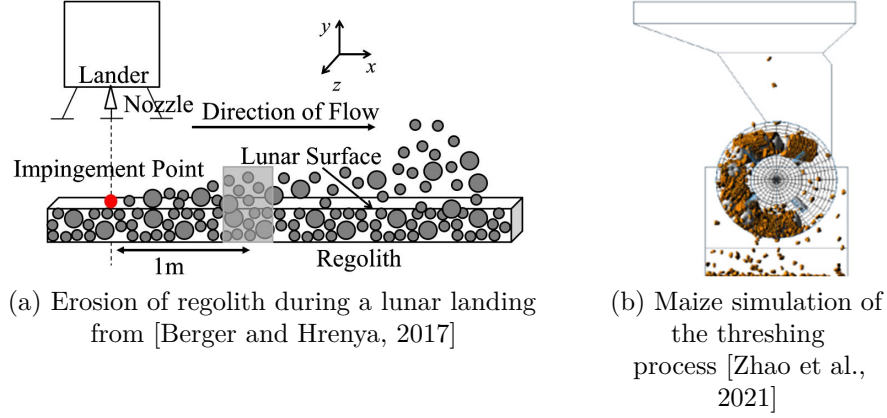


Figure 1.1: Examples of particle simulations.

1.2 Bodies Simulation

1.2.1 Bodies shape

Simple circular or spherical bodies

The simplest way to simulate bodies is to use circular (2D) or spherical (3D) shapes, as shown in Figure 1.2a. In this approach, each body is represented by a single point (its center) and a radius. There is no need to manage orientation, as circular or spherical bodies are invariant under rotation.

Polyhedral bodies

Some applications require more complex shapes. Some papers [Zhao et al., 2021, de Arruda Tino and Tavares, 2022] propose using polyhedra in 3D, such as cubes or pyramids, and in 2D, shapes like triangles or squares [Kafashan et al., 2019].

Multi-Bodies and multi circular/spherical bodies

Multi-bodies are a simple way to simulate complex-shaped bodies by composing a body from multiple interconnected bodies forming a single entity. Multi-circular or multi-spherical bodies consist of several circular or spherical particles.

1.2.2 Methods

There are currently three main approaches to simulating bodies: the ***Discrete Element Method*** (DEM) introduced by [Cundall and Strack, 1979], the ***Event-Driven Method*** [Luding, 2008], and ***Non-Smooth Contact Dynamics*** (NSCD) [Jean, 1999].

Smooth vs Non-Smooth Methods

Simulation methods can be categorized as ***smooth*** or ***non-smooth***. Smooth methods use continuous and differentiable functions to describe body motion, allowing slight overlaps between bodies. In contrast, non-smooth methods treat bodies as perfectly rigid, prohibiting overlap; their reaction functions are discontinuous, and the resulting motion is not differentiable.

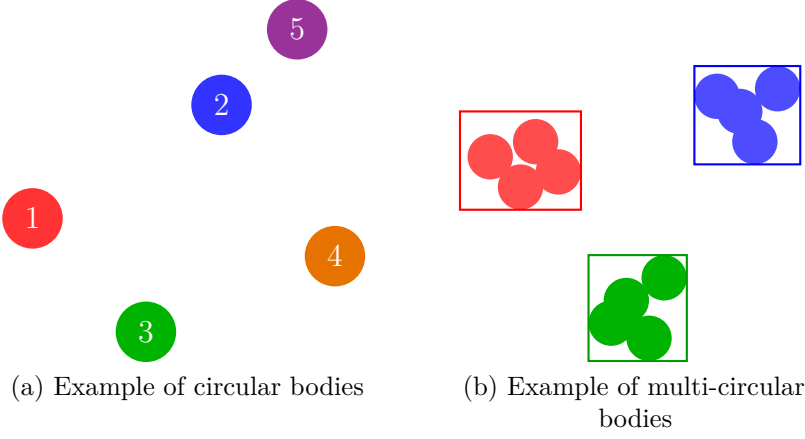


Figure 1.2: Circular bodies vs multi-circular bodies

Discrete Element Method

The Discrete Element Method (DEM) is a smooth, explicit method. As described in [Dubois et al., 2018], DEM numerically integrates position and velocity using previous values, without solving a system of equations.

DEM is a time-driven method: it advances the simulation by a fixed time step Δt , resolving all contacts at each step before updating positions and velocities. Contacts occur when two bodies slightly overlap; the contact force is computed and applied, after which positions and velocities are updated. According to [Cundall and Strack, 1979], the fundamental equations of motion for each body i are:

$$m_i \frac{d^2 \mathbf{x}_i}{dt^2} = \mathbf{F}_i + \mathbf{F}_e \quad I_i \frac{d^2 \theta_i}{dt^2} = \mathbf{T}_i + \mathbf{T}_e \quad (1.1)$$

where m_i is the mass, $\mathbf{x}_i$ the position, $\mathbf{F}_i$ the contact forces, $\mathbf{F}_e$ external forces (e.g., gravity), $\mathbf{T}_i$ the contact torque, $\mathbf{T}_e$ the external torque, and I_i the inertia of body i .

For simple circular or spherical bodies, positions and velocities are updated using numerical integration:

$$\dot{\mathbf{x}}_i(t + \Delta t) = \dot{\mathbf{x}}_i(t) + \frac{\mathbf{F}_i + \mathbf{F}_e}{m_i} \Delta t \quad \mathbf{x}_i(t + \Delta t) = \mathbf{x}_i(t) + \dot{\mathbf{x}}_i(t) \Delta t \quad (1.2)$$

$$\dot{\theta}_i(t + \Delta t) = \dot{\theta}_i(t) + \frac{(\mathbf{T}_i + \mathbf{T}_e)}{I_i} \Delta t \quad \theta_i(t + \Delta t) = \theta_i(t) + \dot{\theta}_i(t) \Delta t \quad (1.3)$$

Contact forces are modeled using the Hertzian contact law, friction by Coulomb's law, and damping is included to represent energy dissipation during contact.

Each iteration is computationally fast, but the time step must be small to maintain stability and prevent excessive overlap, which could cause numerical instability and unrealistic results. The required time step depends on contact rigidity; for rigid contacts, a really small time step is necessary to ensure very small overlaps.

Event-Driven Method

The Event-Driven (ED) method, often considered a type of DEM in the literature, is a non-smooth, event-based approach. Instead of advancing by fixed time steps, the simulation

progresses from one event (contact) to the next: the next event is identified and resolved, then the process repeats. When no collisions occur, velocities remain constant and positions are updated based on velocity and elapsed time:

$$\mathbf{x}_i(t + \Delta t) = \mathbf{x}_i(t) + \dot{\mathbf{x}}_i(t)\Delta t \quad (1.4)$$

When a collision occurs, velocities are updated according to the (inelastic) contact force law.

Non-Smooth Contact Dynamics

NSCD, on the other hand, is an implicit method. It allows for large time steps, but each step is computationally intensive, as it requires solving a system of equations using methods such as Non-Linear Gauss-Seidel, Non-Linear Jacobi, or a queue-based approach as proposed in [Coppin et al., 2022]. NSCD is discussed in more detail in section 4.1.

Optimization-Based Method Some papers formulate NSCD as a convex optimization problem [Anitescu, 2006]. These methods converge faster than classical NSCD but provide less accurate simulations.

Comparison

DEM is very easy to implement and parallelize, but it requires small time steps to ensure stability, especially with more rigid contact parameters. This can result in a large number of iterations for long simulations.

The event-driven method is also simple to implement and can be parallelized. It is efficient for simulations with few contacts, as it only processes events when they occur. However, it is not suitable for simulations with many simultaneous contacts, since it handles contacts one by one. Persistent contacts are also problematic and can result in repeated or even infinite contact detections and resolutions, causing simulation stagnation.

NSCD, on the other hand, is more complex to implement and parallelize. However, it allows for larger time steps and is more stable for rigid contacts. NSCD can handle multiple contacts simultaneously, making it well-suited for multi-body simulations. The main drawback is that it requires solving a system of equations at each step, which can be computationally expensive.

1.2.3 Contact Detection

Contact detection is another important part of the simulation. It is the process of detecting when two bodies are in contact. There are various methods for contact detection. **Grid-based** methods are popular, for instance, using a uniform grid [Lazaveric et al., 2002].

Grid-Based

Uniform Grid One easy way to use spatial partitioning is to use a *Uniform Grid*. It is easy to implement and, most importantly, easy to parallelize. When using a uniform grid, you divide space into a grid of cells of equal size, square cells in 2D or cube cells in 3D, with the size of the cells equal to the largest body in the simulation. As illustrated in Figure 1.3a, to detect contact between two bodies, you only need to check the bodies that are in the same cell as the bodies' centers or in the adjacent cells. This method is very simple and can be parallelized

easily, but it can be inefficient if the bodies are not uniformly distributed, because you have a lot of empty cells. The varying sizes of bodies can also impact performance because there can be many small bodies in the same cell.

Hierarchical Grids To address the problem of bodies having different sizes, there exist hierarchical grid methods [Fan et al., 2011] using different cell sizes for different body sizes.

Tree-Based Structures

A classical tree-based method used in video games is the **Quadtree** for 2D and its 3D adaptation, the **Octree** [Finkel and Bentley, 1974]. The **K-D tree** is also a tree structure often used [Bentley, 1975].

Quadtree As shown in Figure 1.3b, in a Quadtree, you have a tree structure, with each node having 4 children. Each node corresponds to a cell of the same size: one northwest (NW), one northeast (NE), one southwest (SW), and one southeast (SE). For each child of the node, if the number of bodies in the cell is greater than a threshold, you divide the cell into 4 children again and repeat the process until you reach a cell with fewer than the threshold of bodies. For detecting contact, you only need to check the bodies in the same cell. Octrees follow the same idea but in 3D, with each node having 8 children.

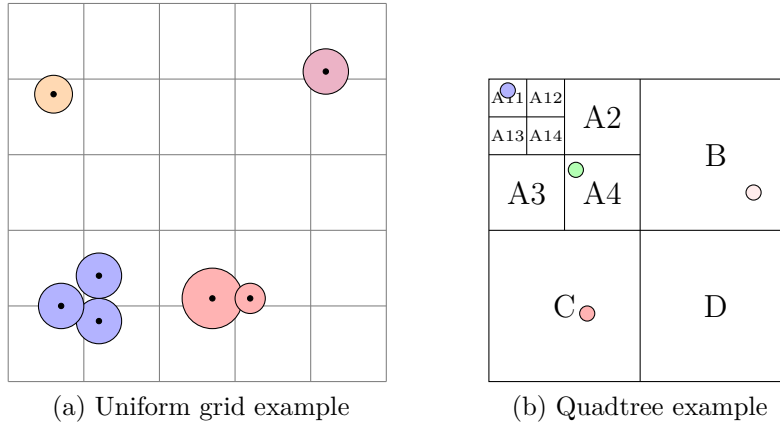


Figure 1.3: Schematics for different contact detection methods

Bounding Volume Hierarchy *Bounding Volume Hierarchy* (BVH) can also be used for contact detection. A GPU algorithm is proposed in [Karras, 2012]. BVH is often used in ray tracing. BVH is explained in more detail in section 3.

1.3 GPU

1.3.1 History

The initial purpose of GPUs was to accelerate graphics rendering. CPUs were and are still slow for graphics rendering. Each pixel of the screen needs to be computed, and the CPU was not fast enough, so GPUs were created to perform graphics rendering in parallel. Initially, they were not programmable; they were based on hardware logic.

In the early 2000s, GPUs evolved and became more programmable [Lindholm et al., 2001], with the introduction of shaders. Ian Buck introduced the Brook programming language [Buck et al., 2004], the first general GPU programming language.

1.3.2 CPU and GPU

GPUs will not replace CPUs, but they are complementary. GPUs can perform computation but need CPUs because most computations require access to I/O devices, and most of the time, these I/O devices can't communicate directly with the GPU. So in general, the CPU will manage I/O, initialize the computation, handle data transfer, and the GPU will perform the computation [Aamodt et al., 2018].

Low Latency vs High Throughput

Another reason these devices communicate with the CPU directly is that the CPU is designed to perform sequential operations, minimizing the latency between instructions, with low-latency memory, while the GPU is designed to achieve high throughput (a high number of operations per second) and uses high-throughput memory but with higher latency. So the CPU is very fast for sequential operations, and on the other hand, the GPU is slower for sequential operations, but it is much faster for parallelized operations. The GPU is useful when the problem is highly parallelizable.

SIMD and SIMT

Most modern CPUs contain SIMD instructions (Single Instruction Multiple Data), which means that a single instruction can be applied to multiple data at the same time. For instance, in a vector addition, the addition is applied to multiple elements of the vector with a single instruction.

SIMT, or Single Instruction Multiple Threads, is an evolution of SIMD. It allows different threads to execute the same program but with a different program flow. GPUs are SIMT systems, and as we will see in the GPU chapter, GPUs use SIMD instructions under the hood within the SIMT architecture.

Integrated vs Discrete GPU

GPUs can be integrated with CPUs, for instance, on the same chip like the Apple M1. This is called an *integrated CPU and GPU*. Alternatively, GPUs can be added as a separate graphics card, a design known as a *discrete GPU*.

In integrated CPU and GPU designs, the CPU and GPU share the same cache and memory, whereas discrete GPUs have their own memory and cache. The main advantage of integrated GPUs is that they consume less power but are generally less powerful than discrete GPUs. For large computations, discrete GPUs cards can be used in parallel.

Performance

In [Sun et al., 2020], the authors compare the FP32 TFLOPS (Floating Point 32 Tera Floating Point Operations Per Second) and FP64 TFLOPS (Floating Point 64 Tera Floating Point Operations Per Second) of various GPUs and CPUs over the years. (FLOPS is not the only factor for performance, but it provides a good idea of the evolution and performance gap. See

the GPU chapters for details.) The collected data show that modern GPUs are much more powerful than CPUs, with a ratio of 2x to 10x for FP32 TFLOPS. For FP64 TFLOPS, 20 years ago the gap was big, but in recent years some GPUs have become very efficient with FP64.

In a recent paper [Cooper-Baldock et al., 2024], the authors compared CPUs and GPUs for CFD (Computational Fluid Dynamics). Although it is not the same type of problem as multi-body simulation, it is still interesting. They used two different types of simulations and found that GPUs (A100, V100) were 80% faster for one simulation and 95% faster for the other.

Power Consumption, Cost

In [Sun et al., 2020], the authors use the TDP (Thermal Design Power) as a measure of power consumption. They admit it is not the best measure but it can give a good approximation. They showed that GPU power consumption halves every 3 to 4 years for the same TFLOPS performance.

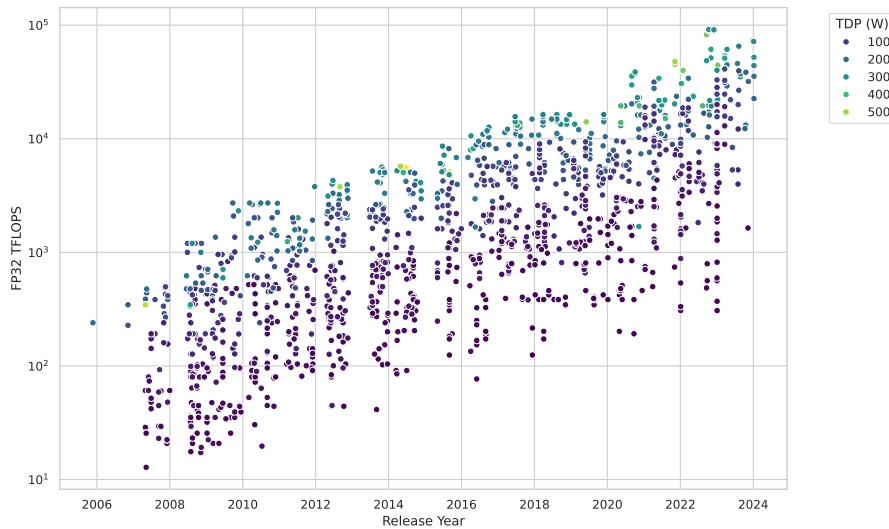


Figure 1.4: TDP of GPUs vs FP32 TFLOPS over the years, data and figure inspired by [Sun et al., 2020]

In [Cooper-Baldock et al., 2024], the authors also observed lower power consumption for GPUs. Power consumption also depends on speed; if the GPU is faster, it will generally consume less power.

1.3.3 Languages

Before the advent of GPU programming languages, GPU programming was limited to a set of very skilled developers with knowledge of the latest graphics APIs and hardware. The programmer had to use shader programming languages like NVIDIA Cg or OpenGL, which were designed for graphics rendering and not for GPGPU programming.

In 2004, Ian Buck, during his PhD thesis at Stanford University, introduced the Brook programming language [Buck et al., 2004], one of the first GPGPU programming languages, using the "stream processing" paradigm. NVIDIA introduced CUDA (Compute Unified Device Architecture) in 2007 [Nickolls et al., 2008], allowing developers to use C++ to write programs that run on the GPU. Nowadays, CUDA is the most used programming language for GPU programming, but there are other languages like OpenCL and AMD ROCm.

GPU Market Share

The current discrete GPU market is shared between NVIDIA and AMD. Intel discrete GPUs represent less than 1% of the market share.

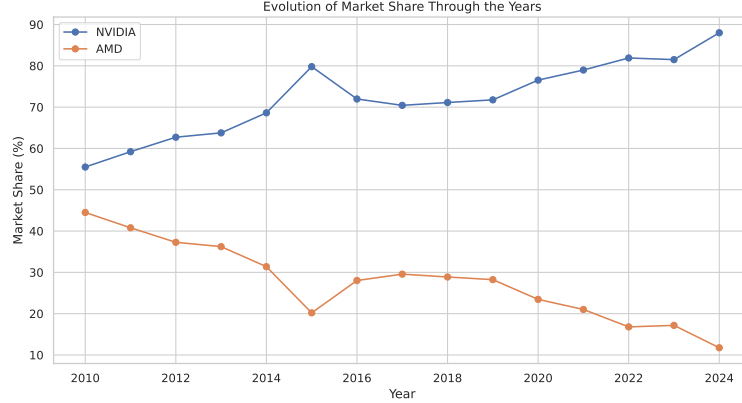


Figure 1.5: Discrete GPU market share, data from [NVI, 2025]

One reason is the **CUDA programming language**, but CUDA is a proprietary language developed by NVIDIA. It is not open source, causing a dependency on NVIDIA and their hardware. Today, a lot of software is developed using CUDA, creating a **monopoly**, and increasing the need for the development of **open-source alternatives**.

Kokkos

[*kokkos*,] is a C++ library developed by the Linux Foundation that abstracts GPU programming. It enables writing **generic code** that is compatible with different GPU programming languages, such as CUDA, OpenCL, and HIP. A nice way to reduce the dependency on CUDA.

1.4 Current State of the Art for Body and Multi-Body Simulation on GPUs

1.4.1 Discrete Element Method

Migflow GPU [Flu, 2025], implement a classic circular DEM solver, contacts are detected using a **Bounding Volume Hierarchy** (LBVH) developed by [Karras, 2012] and stored in a CSR structure as detailed in Chapter 3. Then, a kernel computes the reaction forces and torques for each body, and a final kernel updates their velocities and positions.

Several studies have implemented **multi-body simulation** using GPUs, most of which are based on the **Discrete Element Method (DEM)**. [Dong et al., 2022] propose a method in which all computations are performed on the GPU. The first kernel builds a contact list for each body. To avoid race conditions between cores, they use a **hybrid approach** based on a hierarchical grid that generates a sparse contact list, which is then compressed into a dense contact list. A second kernel then updates the velocities and positions of each body in parallel using their respective contact lists.

1.4.2 Non-Smooth Contact Dynamics

In 2004, [Renouf et al., 2004] proposed a parallel version of NSCD for CPUs by splitting the contact loop within the Gauss-Seidel method into multiple threads. However, since Gauss-Seidel is inherently sequential, this approach is not well-suited for parallelization and may lead to race conditions. On CPUs, this can provide some improvement, but on GPUs, the reliance on atomic operations and mutexes can significantly impact performance. Additionally, the random order of atomic operations can lead to varying results across different runs due to the non-deterministic nature of atomic operations and floating-point arithmetic precision.

Currently, most of the literature on GPUs for *NSCD* focuses on *optimization-based methods*. [Balzer et al., 2013, Ferrari et al., 2020] propose GPU implementations of multi-body simulations using optimization methods such as the one proposed in [Anitescu, 2006].

As mentioned and explained in subsection 1.5.1, *Migflow GPU* [Flu, 2025] has implemented a GPU version of NSCD for circular particles. However, there is currently no GPU implementation of NSCD for multi-circular bodies using the *classical formulation* of [Jean, 1999].

1.5 Current Work and Goal of this Master Thesis

1.5.1 Migflow

Migflow [Flu, 2025] is software developed at *UCLouvain* for *immersed granular flows*. These applications extend beyond multi-body simulation, but multi-body simulation is one of the software’s applications. It contains an implementation of *circular particles* and *multi-circular bodies* using Non-Smooth Contact Dynamics (NSCD) with a Gauss-Seidel solver, a queue-based solver, and a Jacobi solver.

Migflow GPU

Another version of *Migflow* exists for GPUs. It is a port of the CPU version. This GPU version of Migflow uses Kokkos for GPU programming and has already implemented simulations for smooth DEM and NSCD for *circular particles* but not for multi-bodies.

The NSCD algorithm uses the [Karras, 2012] LBVH on GPU and a Jacobi solver running a GPU kernel at each iteration of the Jacobi to solve the contacts. It uses reduction to determine if the Jacobi iteration has converged. A kernel is then launched to update the different velocities. When the Jacobi iteration has converged, a kernel updates the positions in the simulation and advances to the next time step.

1.5.2 Validation of Results

To ensure the *validity* of the GPU implementation results, we will compare the GPU version’s results with the CPU version of Migflow, using *32-bit floating-point numbers* and *64-bit floating-point numbers*. We will also compare performance and convergence between the two versions.

1.5.3 Hardware and Software

The hardware and software used for the different comparisons are shown in Table 1.1.

Component	Specification
Processor	Intel Core i9-14900K
GPU	NVIDIA RTX 4090
Memory	64 GB DDR5
Operating System	Endeavour OS
CUDA Version	12.5
Compiler	clang 19
Kokkos Version	4.0

Table 1.1: Hardware and software used.

1.5.4 Goal

This master thesis aims to implement a GPU version of multi-circular/spherical bodies, using the current BVH for contact detection and NSCD for contact resolution, the Kokkos library with CUDA backend, and compare the results with the CPU version, in terms of performance and accuracy.

Multi-circular/spherical bodies were chosen because they are the simplest case of multi-bodies, can be easily extended from simple spherical bodies, and are used in the original implementation of Migflow. This approach could also be extended to other shapes beyond circular or spherical bodies. NSCD was chosen because it is one of the most accurate and stable methods for multi-body simulation, and there is currently no implementation in the literature of the original NSCD method for multi-bodies on GPU.

Chapter 2

GPU architecture and programming

2.1 GPU Architecture

2.1.1 Ada Lovelace Architecture

This master thesis examines the Ada Lovelace architecture [NVIDIA, 2022], specifically the AD2102 GPU. The Ada Lovelace architecture, succeeding the Ampere architecture, was introduced in 2022. It is designed for a wide range of applications, from gaming to data centers. The GPU used in this study is part of NVIDIA’s RTX 40 series, with the RTX 4090 graphics card employed for testing.

2.1.2 Overview

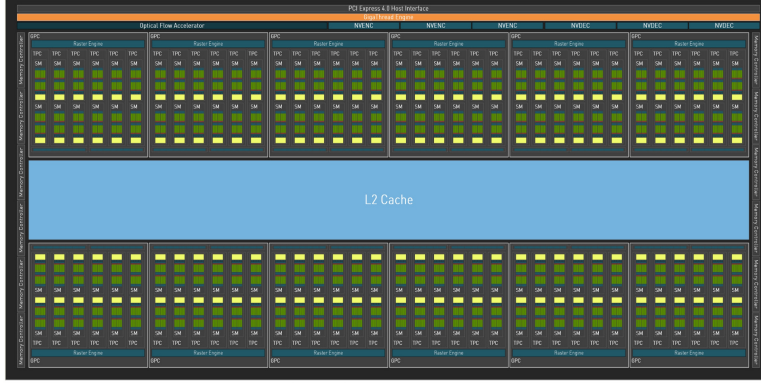
GPC (Graphics Processing Clusters) and TPC (Texture Processing Clusters)

The Ada Lovelace architecture comprises multiple GPCs (*Graphics Processing Clusters*), with 12 GPCs in total. These high-level units are responsible for workload distribution. Each GPC contains 6 TPCs (*Texture Processing Clusters*), which include various units for graphics processing. Among these, the SMs (*Streaming Multiprocessors*) are the most critical for general-purpose GPU (GPGPU) tasks. Refer to Section 2.1.2 for detailed information on SMs.

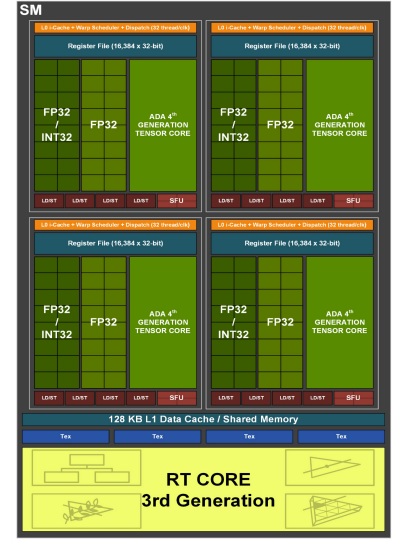
SM (Streaming Multiprocessor)

Each TPC contains 2 SMs (*Streaming Multiprocessors*), resulting in 144 SMs in the Ada Lovelace architecture (12 per GPC). SMs are the primary computational units of the GPU, each comprising various cores (**CUDA cores**, **tensor cores**, etc.). As illustrated in Figure 2.1b, each SM is divided into 4 processing blocks. Each block contains 32 CUDA cores, consisting of 16 FP32 cores and 16 INT32/FP32 cores, totaling 128 CUDA cores per SM.

Each block shares a warp scheduler and a dispatch unit. Additionally, there is an SFU (*Special Function Unit*). Each SM also includes 2 FP64 cores for double-precision operations (not shown in the figure), 4 tensor cores, and 1 ray tracing core for ray tracing tasks. For more details on the cores, see Section 2.1.2.



(a) GPU Global Architecture



(b) SM Architecture

Figure 2.1: Schematic of the Ada Lovelace Architecture (AD21020 GPU) based on the [NVIDIA, 2022] white paper.

Cores

CUDA cores CUDA cores are of primary interest as they perform basic FP32 and INT32 operations: addition, multiplication, subtraction, and division. A typical CUDA core contains an ALU (*Arithmetic Logic Unit*) composed of an FPU (*Floating Point Unit*) and an Integer unit, both used to perform basic operations on FP32 and INT32. In recent architectures like Ada Lovelace, CUDA cores may consist solely of an FPU unit.

FP64 cores In the Ada Lovelace architecture, each SM contains two FP64 cores to handle FP64 operations. This implies that FP64 operations are slower than FP32 operations, with a ratio of 1:64, indicating that FP64 operations are significantly slower than FP32 operations. In different architectures, the ratio can vary. Some GPU architectures are specialized for FP64 operations, often making them expensive and reserved for high-performance computing and scientific applications, as the general public typically does not require FP64 operations. For instance the volta architecture [NVIDIA, 2017] has a ratio of 1:2

Special Function Units For special functions like square root, sine, etc., SMs contain SFUs (*Special Function Units*), which are used to perform these special functions. The ratio of SFU cores to CUDA cores is 1:32, one SFU per processing block, making special functions slower than basic operations.

Load/Store Units Load/store units are used to load and store data from the different memories of the GPU to/from the register file of the SM. Data is then used by cores as operands and results of the instructions. These units are crucial for hiding latency.

Tensor Cores and Ray Tracing Cores Tensor cores are used for tensor and matrix operations like matrix multiplication, commonly used in deep learning to perform matrix multiplications, on the other hand Ray tracing cores are used to accelerate ray tracing operations

mainly used in video games and rendering applications. Both cores are specialized for specific tasks and are not used in the Migflow application.

2.2 Memory Hierarchy

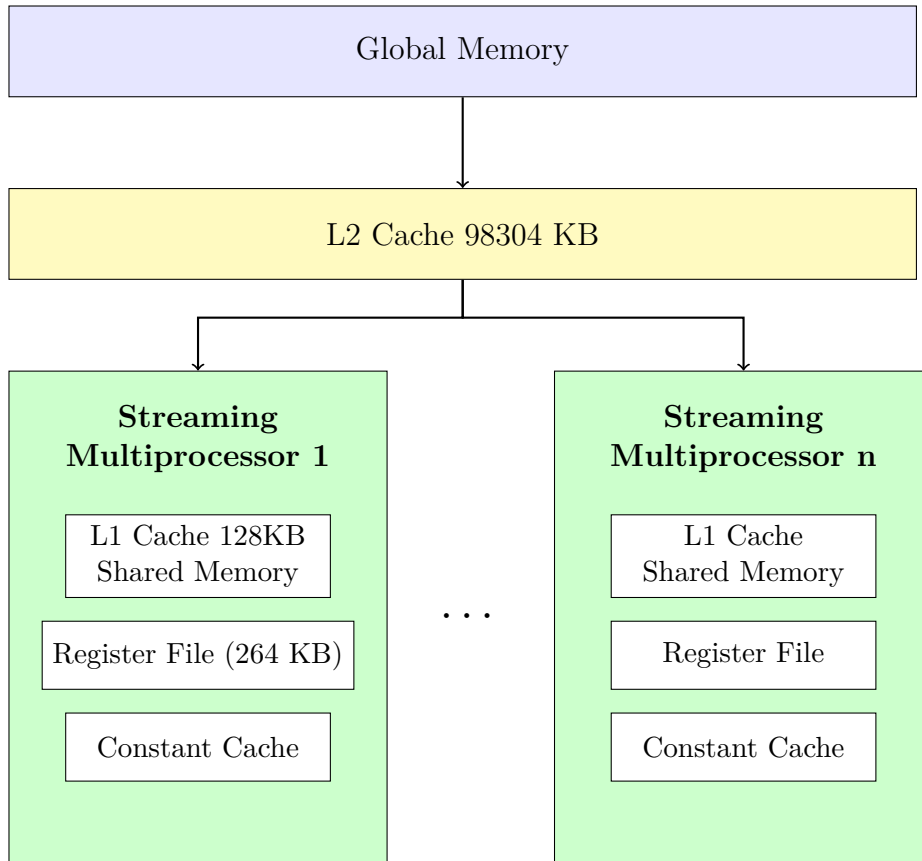


Figure 2.2: Memory hierarchy of the GPU

Main Memory Types

The GPU memory hierarchy consists of several levels, each optimized for specific usage. **Global memory** is the main and largest memory (24 GB on the RTX 4090), accessible by all threads but also the slowest due to its off-chip location and high latency. Each SM has its own **register file** (264 KB), which is the fastest memory and stores operands and results for the cores, with each thread using its private portion. When registers are full, **local memory**—a reserved part of global memory—serves as spill memory for each thread, but with high latency. **Shared memory** is a small, fast on-chip memory accessible by all threads within the same block, enabling cooperation and atomic operations. **Constant and texture memory** are read-only, shared by all threads, and although off-chip, benefit from dedicated caches to accelerate access.

Cache Hierarchy

Caches further accelerate memory access. The **L1 cache**, shared with shared memory (128 KB), speeds up access for threads within the same block. The **L2 cache** (98,304 KB) is shared across the entire GPU and optimizes global data access. A small **constant cache** accelerates access

to constant memory. Finally, the **instruction cache** (L0) stores instructions locally for fast thread execution.

2.3 Programming Model

2.3.1 Hardware and Software Equivalence

The software [NVIDIA, 2025b] to hardware equivalence is as follows:

- Threads $\rightarrow$ executed by cores,
- Thread block composed of threads $\rightarrow$ executed on one SM. Multiple thread blocks can be executed on the same SM.
- Grid of thread blocks (called kernel) $\rightarrow$ executed on the GPU.

2.3.2 Thread, Block, and Grid

Threads all execute the same sequential program in parallel. Threads are grouped in blocks, executed on the same SM. When threads are in the same block, they can cooperate, synchronize, and share data in the shared memory. Thread block sizes are physically limited by the resources of the GPUs. Blocks are grouped in a grid, executed on the GPU, called kernel. Grids execute on different SMs, they do not synchronize, and communication is more expensive.

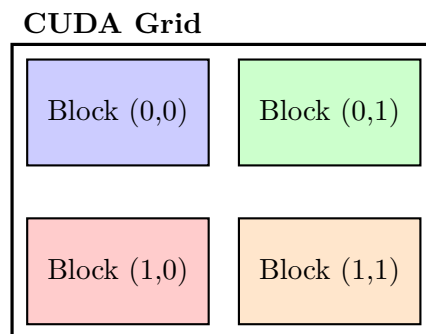


Figure 2.3: Block and grid architecture

2.3.3 Warp

In the GPU, warp schedulers present in the SM (4 in the Ada Lovelace architecture) manage thread blocks in groups of 32 threads called warps. The threads in a warp are chosen by consecutive thread IDs in the block. A warp works in a SIMD style, where each thread in the warp executes the same instruction at the same time (In modern architecture this is not strictly true, **Independent Thread Scheduling** was introduced). This can create what's called warp divergence.

Warp Divergence

Warp divergence [NVIDIA, 2025a] happens when threads in a warp need to take different conditional branches. We have **active** threads that are executing the current instruction and **inactive** threads that are not active in the current branch. Divergence occurs when threads

in a warp take different branches, for instance, a simple if/else statement. When divergence happens, execution is serialized. For example, with an if/else statement, threads in the if branch will execute the instructions while threads in the else branch wait for the if branch to finish before executing the else branch. This means that threads not in the same branch cannot share data or communicate.

Independent Thread Scheduling

Before the Volta architecture, the behavior explained above was the only one possible. Each warp had only one program counter (PC) with a mask called an active mask to indicate which threads were active or inactive. With Volta [NVIDIA, 2017], each thread is fully concurrent, each having its own program counter and call stack. Now, threads can reconverge earlier, and instructions from different branches can be interleaved or executed simultaneously [Anzt and Dongarra, 2018].

Hidden Latency

At each clock cycle, the warp scheduler chooses which warp will execute. Having multiple warps per SM allows the GPU to hide latency. For instance, if a warp is waiting for a memory instruction, the warp scheduler can switch to another warp that is ready to execute. This is called **latency hiding**. All the information of a warp is kept on the GPU during the entire life of the warp, so warp context change has no cost. We can compute the number of warps per block with the following formula:

$$\text{warps per block} = \left\lceil \frac{\text{Threads per block}}{32} \right\rceil \quad (2.1)$$

2.4 GPU Programming

2.4.1 Parallel Programming

The core concept of a GPU is that it can execute many threads in parallel, so we need to parallelize our code to take advantage of the GPU. When writing code for the GPU, this is the first thing to consider.

2.4.2 Parallel Reduction

The problem of reduction is common when applying a function to a set of data, such as finding the sum of a function applied to a set of data, the minimum, the maximum, etc. The classical parallel reduction algorithm, as shown in [Jradi et al., 2017], is a reduction tree. As illustrated in Figure 2.4, the reduction tree is a binary tree. For n elements, we have $n / 2$ threads that will execute the reduction in parallel. Each thread will compute the result of the reduction of its two children and store the result. Then, after syncing, one of the threads becomes the parent recursively until we reach the root of the tree.

With kokkos, we can use the `Kokkos::parallel_reduce` function to perform a reduction, avoiding the need to implement the reduction tree manually.

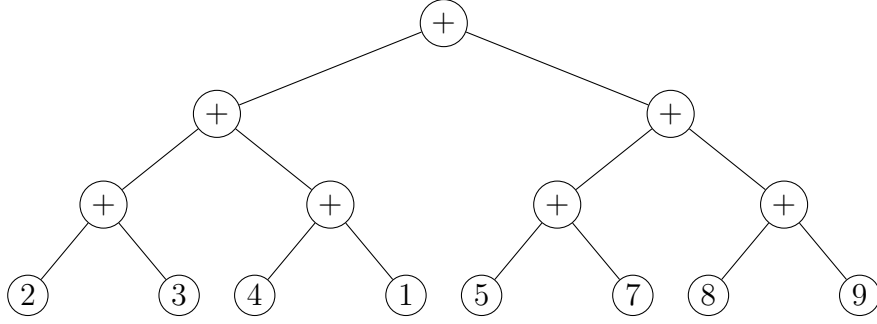


Figure 2.4: Reduction tree, figure inspired by [Jradi et al., 2017]

2.5 Optimization

2.5.1 Coalescing

Memory Access

Global memory is accessed with 32, 64, or 128 bytes at a time, so memory transactions (read/write) are aligned to this size. Memory access is done per all threads in a warp, so when a warp accesses memory, the memory access all the addresses need, if the memory is coalesced meaning done contiguously for all threads in the warp, it will be done in less transactions. For instance, if 32 threads in a warp access 32 consecutive addresses, the memory access will be done in one transaction of 128 bytes. If the memory access is not coalesced, it will be done in multiple transactions, leading to more memory accesses. This is very different from CPU cache access where we try to access memory per thread, as the CPU cache is designed to access memory per thread. In the GPU, we access memory per warp. Coalescing is crucial for performance, as it can lead to a significant speedup in memory access.

Migflow GPU Solution

To optimize memory access, Migflow uses a structure of arrays instead of an array of structures. In the GPU version, each body is stored in a Bodies structure, which contains all the data of the bodies. In the CPU version, we have a body structure stored in a vector of bodies. The GPU version also contains a special array structure for storing the bodies' data next to each other in the GPU memory. This is explained in more detail in the implementation chapter. The GPU design allows us to access the data of the bodies in a coalesced way, as all the data of the bodies is stored in a contiguous memory space.

2.5.2 Occupancy

Occupancy is the ratio of the number of concurrent warps per SM to the maximum number of concurrent warps that can be active on an SM at the same time.

$$\text{occupancy} = \frac{\text{concurrent warps}}{\text{max concurrent warps}} \quad (2.2)$$

Low occupancy can lead to poor performance, as the GPU will not be able to hide latency between warps. For instance, it will be waiting for the memory access to finish before executing a warp. We don't need to reach 100% occupancy to achieve the best performance, and occupancy

is highly dependent on the code. Sometimes a lower occupancy is better; for instance, a kernel with low occupancy will have fewer registers used and can avoid register spilling.

What Can Affect Occupancy

Shared Memory Usage If our kernel uses a lot of shared memory, it can lead to low occupancy because shared memory is limited. If we use too much shared memory, we will not be able to launch many warps on the SM.

Block Sizes and Grid Sizes Block size is very important. The first thing is to have a block size multiple of 32 to avoid wasting threads in a warp. [Woolley, 2013] recommends using 128-256 threads per block and adjusting based on performance. For the grid size, a size with more than 1000 threads is often recommended. If a block needs to use many resources, it limits the number of other blocks on the same SM, thus affecting occupancy.

2.5.3 Data Transfer

For our computation, we need to transfer data from CPU memory (RAM) to GPU. However, transferring data between CPU memory (RAM) and GPU is expensive, so it is important to limit the transfer as much as possible and do it only when needed. For instance, keep the data between different kernels in the GPU memory.

2.5.4 How Our Program Can Be Limited

[Crovella, 2020] cites different limiters that can affect the performance of a GPU.

Memory Bound

TFLOPs are often used to measure the performance of a GPU, but often the performance is not limited by the number of FLOPs but by the memory bandwidth or latency. We consider it memory bound when the GPU memory reaches its theoretical memory performance peak. If our program is memory bound, we can improve performance by optimizing memory usage, such as optimizing caches/shared memory, using specific memory like constant memory, and coalescing memory access.

Compute Bound

On the other hand, compute bound is when the GPU is near its theoretical throughput.

Latency Bound

Latency bound is when our code is not limited by compute or memory. It happens when the GPU occupancy is low, for instance, when the GPU is waiting for memory access to finish. In this case, we can improve performance by increasing occupancy.

2.5.5 GPU Application Profiling

To detect what limits the performance of our code, we can use profiling tools to analyze the performance of our code. These tools provide information about memory usage, compute usage, occupancy, memory access patterns, etc.

Nsight Systems

Nsight Systems is a profiling tool that allows us to analyze the performance of our code. It is the first tool to use to analyze the performance of our code, providing an application-wide view of performance with a timeline of execution. We can see CPU and GPU execution, memory transfers, kernel execution, etc.

Nsight Compute

Nsight Compute is a profiling tool that allows us to analyze the performance of our code at a kernel level. If we want to analyze a specific kernel, we can use Nsight Compute to analyze its performance. It provides information about memory usage, compute usage, SM usage, occupancy, etc.

Chapter 3

Contact detection: Bounding volume hierarchy

As mentioned in Section 1.2.3, there are various methods for contact detection. The one implemented in *miglow GPU* is the *Bounding Volume Hierarchy* (BVH).

3.1 BVH Overview

Bounding volume hierarchy trees, as presented in [Ericson, 2004], are generally binary trees. Each body/particle is present in the leaf of the tree and is bounded by a bounding box. The parent of each node is the bounding box of its two children's bounding boxes. The process is recursive up to the root of the tree. As shown in **Figure 3.1**, the node **B1** has a bounding box containing two particles/bodies **P1** and **P2**. Similarly, **B2** contains **P3** and **P4**. **B0** is the root of the tree with the bounding box containing **B1** and **B2**.

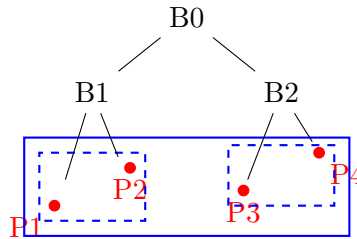


Figure 3.1: BVH tree with two bounding boxes and their particles

3.2 BVH for Contact Detection

In general, contact detection is divided into two steps [Wang et al., 2018]:

- Broad phase: The goal is to find potential contacts between bodies/particles, eliminating those that are too far apart.
- Narrow phase: The goal is to find the exact contact between bodies/particles.

BVH is used for the broad phase. The BVH is built with the bodies/particles, and to detect potential contact with a certain body, we traverse the tree from the root, checking if the body's

bounding box intersects with the bounding box of the nodes recursively. When present in a leaf node, we can perform the narrow phase and check the body with the body in the leaf node to see if there is contact [Ericson, 2004].

3.3 Quality of the BVH

For a set of bodies, there exist many different BVH trees. Some trees are considered high-quality, while others are low-quality based on various **criteria**. In general, there is a trade-off between fast construction time with lower-quality BVH and slow construction time with higher-quality BVH.

3.3.1 Criteria

One criterion could be to minimize the volume of the bounding boxes. For instance, if your bounding boxes are too large, more nodes will be traversed, and the intersection test will be slower. Thus, the most important criterion is to minimize the overall volume of the bounding boxes, with nodes near the top being the most important because they are traversed the most. The tree needs to be as balanced as possible. If the tree is unbalanced, more nodes will be traversed, and the tree will be slower to traverse. There exist cost functions to evaluate the quality of a BVH, as described in [He, 1999].

3.3.2 Contact Detection Choice

In the case of contact detection, the BVH is used for the broad phase, so the quality of the BVH is less important than the speed of construction. The literature prefers fast construction methods over high-quality BVH.

3.4 Classical BVH Construction

There are three classical methods for BVH construction: **top-down**, **bottom-up**, and **online construction**.

3.4.1 Top-down

The top-down construction of the BVH is a recursive algorithm. It starts with the root node and recursively splits the bodies into two sets until there is only one body in each leaf node, as shown in **Algorithm 1**.

Algorithm 1 Top-down construction of the BVH tree

```
function BUILDTree(bodies)
  nBodies  $\leftarrow$  length(bodies)
  node  $\leftarrow$  CreateNode()            $\triangleright$  create a node for children and bbox
  node.boundingBox  $\leftarrow$  ComputeBoundingBox(bodies)
  if nBodies = 1 then                $\triangleright$  if there is only one body, this is a leaf node
    node.body  $\leftarrow$  bodies[0]
    return node
  else
    partition1, partition2  $\leftarrow$  Partition(bodies)    $\triangleright$  partition the bodies into two sets
    node.left  $\leftarrow$  BuildTree(partition1)
    node.right  $\leftarrow$  BuildTree(partition2)
  end if
end function
```

There are various methods for partitioning the bodies into two sets, related to the **criteria** of BVH quality.

3.4.2 Bottom-up

The bottom-up construction, as shown in **Algorithm 2**, works in the opposite way of the top-down construction. It starts with the bodies as the leaf nodes and recursively merges the bodies into internal nodes until there is only one node left, the root node. This method tends to be slower to build, but the quality of the BVH is better than the top-down construction.

There are different optimizations for the bottom-up construction, such as using a priority queue to find the closest nodes. Some methods also use the Z-order curve/Morton order to sort the bodies beforehand, as we'll see in the GPU implementation.

3.4.3 Online

The online construction is a method that builds the BVH by inserting the bodies one by one into the tree. This method is not particularly interesting for us because it is difficult to parallelize.

Algorithm 2 Bottom-up construction of the BVH tree

```
function BUILDTREEBOTTOMUP(bodies)
  nodes = []
  for body in bodies do
    node  $\leftarrow$  CreateNode()  $\triangleright$  create a node for children and bbox
    node.body  $\leftarrow$  body
    node.boundingBox  $\leftarrow$  ComputeBoundingBox(body)
    nodes.append(node)
  end for
  while length(nodes) > 1 do
    node1, node2  $\leftarrow$  FindClosestNodes(nodes)  $\triangleright$  find the two closest nodes according to a
    chosen metric
    newNode  $\leftarrow$  CreateNode()
    bbbox1  $\leftarrow$  node1.boundingBox
    bbbox2  $\leftarrow$  node2.boundingBox
    newNode.boundingBox  $\leftarrow$  MergeBoundingBox(bbbox1, bbbox2)
    newNode.left  $\leftarrow$  node1
    newNode.right  $\leftarrow$  node2
    nodes.remove(node1)
    nodes.remove(node2)
    nodes.append(newNode)
  end while
end function
```

3.5 GPU Adaptation

3.5.1 Linear Bounding Volume Hierarchy

Linear Bounding Volume Hierarchy (LBVH) proposed in [Lauterbach et al., 2009], is a specific method for building very fast BVH trees, of good quality (not the best) because as mentioned earlier in contact detection speed construction is preferred to quality construction. The method is based on Z-order curve/Morton order, the idea is to sort the bodies' centroids according to their Morton code, and then build the BVH tree using the sorted bodies.

Z-order curve/Morton order

Morton order is a way to map 2D, 3D, or nD coordinates to a binary number and keep the spatial order of the coordinates. So as shown in **Figure 3.2**, if points are near in 2D they will be near in 1D. The purpose for the BVH tree is that bodies with the nearest Morton code will be next to each other in the tree, so the bounding boxes will be near to each other, and the tree will be more balanced.

3.5.2 Migflow Implementation

In migflow GPU, the LBVH implementation from [Karras, 2012] is an improved version of the previous one.

The paper identifies different defaults in existing methods for parallel BVH construction:

Binary Radix Tree Binary radix tree is the same as a radix tree but the only values are 0 and 1, the keys are then bitstrings and each node has two children except the leaf nodes.

Tree Properties and Layout

Lexicographic nodes ordering The nodes are ordered lexicographically, so the left child differing value for a internal the 0 and the right child is the 1. Sorting is useful because we can determine the position where the prefix differs and the children are always at the same position in the array, this way we can consider linear ranges of keys $[i, j]$, where keys are increasing. We can define the cover of a range as the number of keys covered between node i and node j ,

Range of keys In the paper tree layout, each leaf is a bitstrings key, and each internal node represents a range of keys with the longest common prefix.

Split Position The range of an internal node is split into two parts at the split position γ , it is the first differing bit in the range of covered keys. For instance with the keys 0000, 0001, 0010, 0111, the first differing bit is the second bit, so the split is at the key 0010, with zero indexed, the split position is $\gamma = 2$, and children are γ and $\gamma + 1$. The resulting children are stored in the array $L_\gamma, L_{\gamma+1}$ or I_γ and $I_{\gamma+1}$ depending if it is a leaf or an internal node. So each range of internal nodes is either the start or the end of the range, and the children are stored in the array L or I depending if it is a leaf or an internal node.

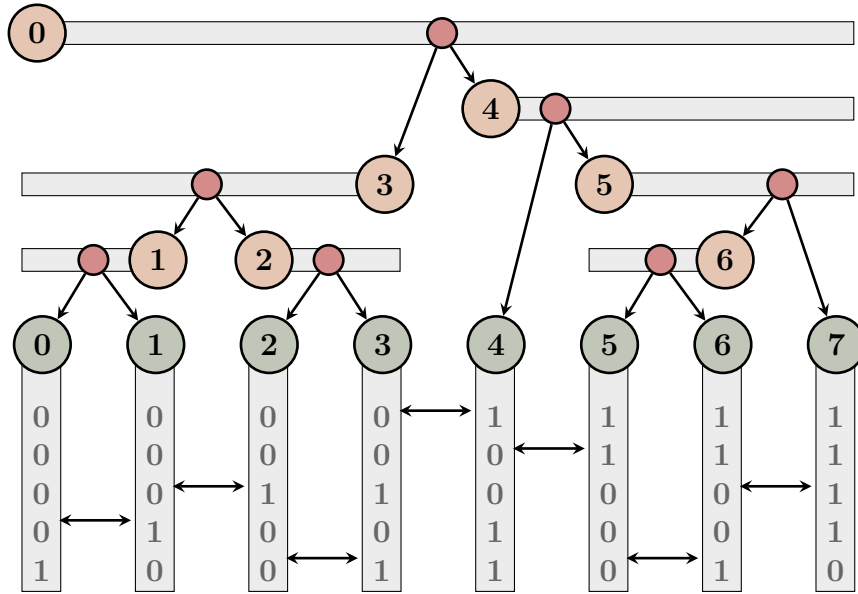


Figure 3.4: Layout of the binary radix tree, figure from [Prokopenko and Lebrun-Grandie, 2024] representing [Karras, 2012]

Algorithm

The very smart layout allows to do the construction in parallel, because the children of a node are known without having earlier results of the tree construction. All the following steps are done in parallel for each internal node i in the tree.

Finding the range of keys By construction each internal node is either the start or the end of the range, the first step is to find the direction of the range, the direction is computed by comparing the cover of the node with its two neighbors, if the node is the start of the range, the cover of the node with its right neighbor is greater than the cover of the node with its left neighbor because the range starts and shares more keys with the right neighbor than with the left neighbor. The neighbor cover also give us a lower bound for the range of keys covered.

Algorithm 3 Find range direction

```

i ← ...
d = cover(i, i + 1) − cover(i, i − 1) > 0 ? 1 : −1
minCover = cover(i, i − d)

```

Then to find the range of keys, the algorithm find an upper bound for the range and then use a binary search to find the start/end of the range.

Algorithm 4 Finding the range

```

upperBound ← 2
while i + upperBound * d < n && cover(i, i + upperBound * d) > minCover do
    upperBound = upperBound * 2
end while
lowerBound ← 0
while lowerBound ≤ upperBound do
    mid ← (upperBound − lowerBound) / 2 + lowerBound
    if cover(i, i + mid * d) > minCover then
        end ← i + mid * d
        upperBound ← mid − 1
    else
        lowerBound ← mid + 1
    end if
end while

```

Finding the split position We then need to find the split position γ to determine node children, binary search is also used as shown in **Algorithm 5**.

Algorithm 5 Finding the splitting point

```

lowerBound ← 0
upperBound ← end
while lowerBound ≤ upperBound do
    mid ← (upperBound − lowerBound) / 2 + lowerBound
    if cover(i, i + mid * d) > cover(i, end) then
         $\gamma$  ← i + mid * d + min(0, d)
        upperBound ← mid − 1
    else
        lowerBound ← mid + 1
    end if
end while

```

Update the range of keys Finally, we can update our array I with the range of keys.

Adapt Binary Radix Tree to BVH

Construction Each body is assigned a Morton code and the bodies are sorted, then the binary radix tree is built using the Morton code as the key.

Bounding Box When all the previous steps are done a new kernel is launched to build the bounding box of the bodies. To add the bounding box of the bodies, we start from each leaf node, then traverse the tree up to the root node, and for each node, we compute the bounding box of the two children and store it in the parent node.

To avoid traversing nodes twice, we keep track of the nodes already traversed and update using atomic operations the node visited.

Algorithm 6 Build the bounding box for each node

```
 $i \leftarrow \dots$  ▷ Leaf node in parallel  
 $parent \leftarrow parents[i]$   
while  $parent \ \&\& \text{atomicAddFetch}(parent, 1) == 0$  do  
     $child1 \leftarrow children[parent][0]$   
     $child2 \leftarrow children[parent][1]$   
     $parent.bbox \leftarrow mergeBbox(child1.bbox, child2.bbox)$   
     $parent \leftarrow parents[parent]$   
end while
```

3.5.3 Improvements and Alternatives

Improvements

[Apetrei, 2014] focus on BVH and propose an improvement of the [Karras, 2012] paper for LBVH construction, the first thing Apetrei does is merging the kernel together by building the ranges in a bottom-up manner. The internal nodes corresponds to the split position γ . Apetrei is way easier to implement and understand than karras and gets faster results than karras.

[Prokopenko and Lebrun-Grandie, 2024] proposes an improve of the Karras method and Apetrei method working for both using the stackless traversal technique, a technique to avoid using explicitly a stack while traversing the tree using a skip connection but does not detail any performance improvement.

Octree and kd-tree

The paper also presents a method to use the presented binary radix tree with octree and kd-tree, which could be interesting for future work, and compare the octree of the CPU version.

BVTT front based

[Wang et al., 2018] use a BVTTT front-based collision that exploit spatio-temporal coherence, it also use a cache-friendly data layout and instead of rebuilding the BVH tree and the BVTT, it restructure the tree and the BVTT, less costly than rebuilding the tree.

Ray Tracing Core Usage

A recent preprint [Mandarapu et al., 2025] proposes a method using ray tracing cores to quickly detect contacts between bodies.

High-Quality BVH

Karras also presents a method for building higher quality BVH trees [Karras and Aila, 2013], but the construction time is slower than the LBVH construction.

3.6 Contacts Storage

Contacts are stored in a compressed sparse row (CSR) format to avoid wasting memory and to allow fast access to the contacts of a body.

Chapter 4

Non Smooth Contact Dynamics

Numerous papers in the literature present NSCD using different notations [Jean, 1999, Dubois et al., 2018, Coppin et al., 2022, RADJAÏ et al., 2010]. In this work, we consider the problem in 2D, with some references to 3D, noting that the approach can be readily extended to three dimensions.

4.1 Modelisation

$$\mathbf{q}_i = \begin{pmatrix} x_i \\ y_i \\ \theta_i \end{pmatrix} \quad \dot{\mathbf{q}}_i = \begin{pmatrix} \dot{x}_i \\ \dot{y}_i \\ \dot{\theta}_i \end{pmatrix} \quad \mathbf{M}_i = \begin{pmatrix} m_i & 0 & 0 \\ 0 & m_i & 0 \\ 0 & 0 & I_i \end{pmatrix} \quad \mathbf{q} = \begin{pmatrix} \mathbf{q}_1 \\ \mathbf{q}_2 \\ \vdots \\ \mathbf{q}_n \end{pmatrix} \quad \mathbf{M} = \begin{pmatrix} \mathbf{M}_1 & 0 & 0 \\ 0 & \mathbf{M}_2 & 0 \\ 0 & 0 & \ddots \\ 0 & 0 & 0 & \mathbf{M}_n \end{pmatrix}$$

Here, $\mathbf{q}_i$ is the position and orientation of body i , $\dot{\mathbf{q}}_i$ its velocity, and $\mathbf{M}_i$ its mass and moment of inertia. Vectors $\mathbf{q}$ and $\mathbf{M}$ represent the stacked positions and mass matrices for all bodies.

According to Newton's law, the acceleration and position of body i are given by:

$$\mathbf{M}_i \ddot{\mathbf{q}}_i = \mathbf{F}_i \quad (4.1)$$

$$\mathbf{q}_i(t) = \mathbf{q}_i(\tau) + \int_{\tau}^t \dot{\mathbf{q}}_i dt \quad (4.2)$$

Generalized to all bodies:

$$\mathbf{M} \ddot{\mathbf{q}} = \mathbf{F} \quad (4.3)$$

$$\mathbf{q}(t) = \mathbf{q}(\tau) + \int_{\tau}^t \dot{\mathbf{q}} dt \quad (4.4)$$

where $\mathbf{F}$ can be decomposed as:

$$\mathbf{F} = \mathbf{r} + \mathbf{F}_e \quad (4.5)$$

Here, $\mathbf{F}_e$ is the external force (e.g., gravity), and $\mathbf{r}$ represents the contact forces.

It is common to express the previous equation using measure theory to account for discontinuous reaction forces:

$$\mathbf{M} d\dot{\mathbf{q}} = \mathbf{F}_e dt + \mathbf{r} dv \quad \mathbf{q}(t) = \mathbf{q}(\tau) + \int_{\tau}^t \dot{\mathbf{q}} dt \quad (4.6)$$

Here, dv is a non-negative measure representing the density of contact force over time dt , considering both continuous and instantaneous (shock) forces.

4.2 Contacts: Local Frame and Global Frame

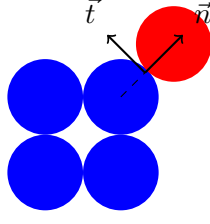


Figure 4.1: Contact between two bodies

Each contact between two bodies can be described both in the global frame (the simulation world frame) and in the local frame of the contact itself. In two dimensions, the local frame is defined by the normal vector $\vec{n}$ and the tangent vector $\vec{t}$ at the contact point. The normal vector is perpendicular to the contact surface, while the tangent vector is perpendicular to the normal.

To apply contact forces to body velocities expressed in the global frame, the forces are transformed from the local to the global frame. The linear application matrix $\mathbf{H}^\alpha(\mathbf{q})$ transforms vectors from the local to the global frame, while $\mathbf{H}^{*\alpha}(\mathbf{q})$ performs the inverse transformation.

For a contact α , $\mathbf{U}^\alpha$ denotes the relative velocity at the contact, with U_N^α the normal component and $\mathbf{U}_T^\alpha$ the tangential component (a scalar in 2D, a vector in 3D). R_N^α and $\mathbf{R}_T^\alpha$ are the normal and tangential reaction forces at the contact. d^α is the distance between the two bodies, with $\dot{d}^\alpha = U_N^\alpha$.

In summary, to obtain $\mathbf{U}^\alpha$ from $\dot{\mathbf{q}}$ and $\mathbf{r}^\alpha$ from $\mathbf{R}^\alpha$:

$$\mathbf{U}^\alpha = \mathbf{H}^{*\alpha}(\mathbf{q})\dot{\mathbf{q}} \quad \mathbf{r}^\alpha = \mathbf{H}^\alpha(\mathbf{q})\mathbf{R}^\alpha. \quad (4.7)$$

4.3 Time discretization

To advance the simulation by a time step Δt from τ to $t = \tau + \Delta t$, discretization is required. Integrating both sides yields:

$$\mathbf{M}(\dot{\mathbf{q}}(t) - \dot{\mathbf{q}}(\tau)) = \int_\tau^t \mathbf{F}_e dt + \int_\tau^t \mathbf{r} dv \quad \mathbf{q}(t) = \mathbf{q}(\tau) + \int_\tau^t \dot{\mathbf{q}} dt \quad (4.8)$$

The impulse $p(t) = \int_\tau^t \mathbf{r} dv$ aggregates all contact forces and shocks, and can be written as $\mathbf{p}(t) = \sum_\alpha \mathbf{r}^\alpha(t)$.

Applying the implicit Euler method, the equations are discretized as:

$$\mathbf{M}(\dot{\mathbf{q}}^{n+1} - \dot{\mathbf{q}}^n) = \mathbf{F}_e \Delta t + \mathbf{p}^{n+1} \quad (4.9)$$

$$\mathbf{q}^{n+1} = \mathbf{q}^n + \dot{\mathbf{q}}^{n+1} \Delta t \quad (4.10)$$

where $\mathbf{q}^n$ is the position at the current timestep and $\mathbf{q}^{n+1}$ at the next timestep.

$\mathbf{U}^{n\alpha}$ and $\mathbf{U}^{n+1\alpha}$ are approximated for a constant matrix $\mathbf{H}^{*\alpha}$ as:

$$\mathbf{U}^{n\alpha} = \mathbf{H}^{*\alpha}(\mathbf{q})\dot{\mathbf{q}}^n \quad \mathbf{U}^{n+1\alpha} = \mathbf{H}^{*\alpha}(\mathbf{q})\dot{\mathbf{q}}^{n+1} \quad (4.11)$$

If $\mathbf{H}^{*\alpha}$ is not constant, it can be approximated using an intermediate value $\hat{\mathbf{q}}$.

4.4 Contact unilaterality and friction

In NSCD, it is assumed that bodies do not interpenetrate, so the distance between two bodies is always non-negative: $d^\alpha(t) \geq 0$. Additionally, there is no attraction between bodies, so the reaction force is always non-negative: $R_n^\alpha(t) \geq 0$. When bodies are not in contact, the reaction force is zero: $R_n^\alpha(t)d^\alpha(t) = 0$.

These are known as the Signorini conditions:

$$d^\alpha(t) \geq 0 \quad R_n^\alpha \geq 0 \quad d^\alpha(t)R_n^\alpha = 0 \quad (4.12)$$

If velocities are smooth, the Signorini conditions can be expressed in terms of velocity:

$$\dot{d}(t) = U_n^\alpha \quad U_n^\alpha \geq 0 \quad R_n^\alpha \geq 0 \quad U_n^\alpha R_n^\alpha = 0 \quad (4.13)$$

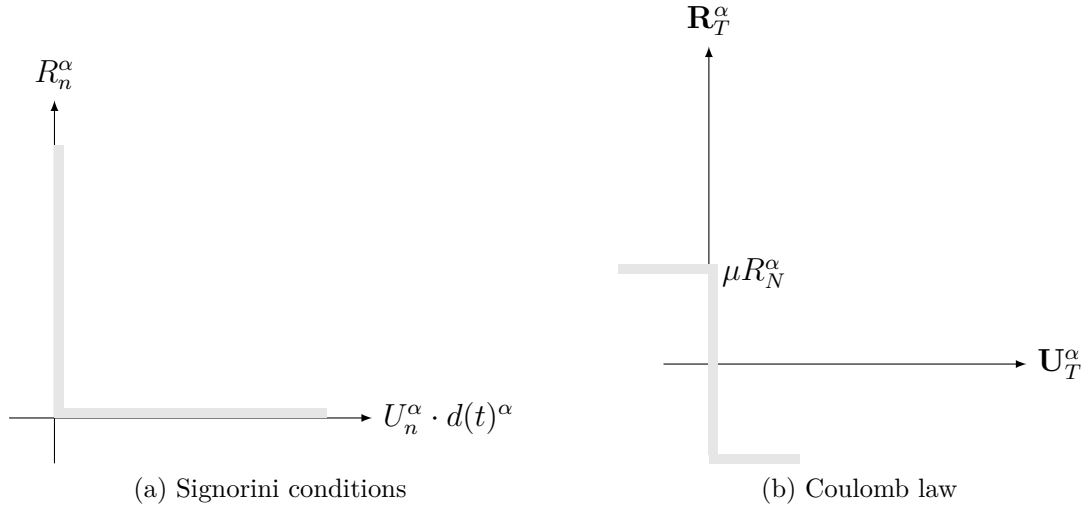


Figure 4.2: Signorini conditions

Friction is described by Coulomb's law [Brendel et al., 2004], which characterizes the interaction between two bodies in two scenarios:

- Sliding/Slipping: When the tangential velocity is non-zero, the tangential force opposes the sliding motion, as defined in Equation 4.15.
- Sticking: When the tangential velocity is zero, the friction force is constrained by Equation 4.14. In this case, friction prevents sliding.

Here, μ is the static friction coefficient:

$$\|\mathbf{R}_T^\alpha\| \leq \mu R_N^\alpha \quad \mu \geq 0 \quad (4.14)$$

$$\mathbf{R}_T^\alpha = -\mu R_N^\alpha \frac{\mathbf{U}_T^\alpha}{\|\mathbf{U}_T^\alpha\|} \quad \|\mathbf{U}_T^\alpha\| \neq 0 \quad (4.15)$$

4.5 Contact Resolution

Based on the previous discretization and linear mappings, each contact α can be considered and solved independently as follows:

$$\mathbf{U}^{(n+1)\alpha} = \mathbf{U}_{free}^\alpha + \sum_{\beta \in \text{Contacts}} \mathbf{W}^{\alpha\beta} \mathbf{P}^{n+1\beta} \quad (4.16)$$

$$\mathbf{U}_{free}^\alpha = \mathbf{H}^{*\alpha}(\dot{\mathbf{q}}^n + \mathbf{M}^{-1}\mathbf{F}_e) \quad \mathbf{W}^{\alpha\beta} = \mathbf{H}^{*\alpha}\mathbf{M}^{-1}\mathbf{H}^\beta \quad (4.17)$$

Here, $\mathbf{U}_{free}^\alpha$ is the relative velocity at contact α in the absence of contact forces. The second term represents the contributions of impulses from other contacts to the relative velocity at α . The Delassus operator $\mathbf{W}^{\alpha\beta}$ represents $\mathbf{M}^{-1}$ in the local frame and is a 2×2 matrix in 2D. The equation can be rewritten as:

$$\mathbf{U}^{(n+1)\alpha} = \mathbf{U}_{locfree}^\alpha + \mathbf{W}^{\alpha\alpha} \mathbf{P}^{n+1\alpha} \quad \mathbf{U}_{locfree}^\alpha = \mathbf{U}_{free}^\alpha + \sum_{\beta \neq \alpha} \mathbf{W}^{\alpha\beta} \mathbf{P}^{n+1\beta} \quad (4.18)$$

For a single contact, the problem reduces to finding $\mathbf{U}$ and $\mathbf{P}$ such that:

$$\mathbf{U} = \mathbf{U}_{locfree} + \mathbf{W}\mathbf{P} \quad \text{Signorini}(U_N, \mathbf{P}_N) \quad \text{Coulomb}_{\mu P_N}(\mathbf{U}_T, \mathbf{P}_T) \quad (4.19)$$

Contact α can be solved by provisionally assigning values to $\mathbf{P}^{n+1\beta}, \beta \neq \alpha$, defining $\mathbf{U}_{locfree} = \mathbf{U}_{locfree}^\alpha$ (computed from the provisional values), $W = W^{\alpha\alpha}$, and treating $\mathbf{P} = \mathbf{P}^{n+1\alpha}$ and $\mathbf{U} = \mathbf{U}^{(n+1)\alpha}$ as unknowns.

This yields a system of four equations and four unknowns (U_n, U_t, R_n, R_t). The solution is unique if the following condition is satisfied:

$$-1 < -\mu \frac{w_{nt}}{w_{nn}} < 1 \quad (4.20)$$

where w_{nn}, w_{nt} , and w_{tt} are defined in 2D as:

$$\mathbf{W} = \begin{pmatrix} w_{nn} & w_{nt} \\ w_{tn} & w_{tt} \end{pmatrix} \quad (4.21)$$

This constrains μ ; if not satisfied, multiple solutions may exist. For further details, see [RADJAI et al., 2010].

$P_{locstick}$, the impulse when the relative velocity is zero, is defined as:

$$P_{locstick} = -W^{-1}\mathbf{U}_{locfree} \quad (4.22)$$

The system's solution depends on the situation:

- If $U_{locfree,N} > 0$: No contact, $P = 0$.
- If $U_{locfree,N} \leq 0$ and $P_{locstick,T} + \mu P_{locstick,N} < 0$: $P_N = \frac{U_{locfree,N}}{(1-\mu \frac{w_{nt}}{w_{nn}})w_{nn}}$, $P_T = -\mu P_N$.
- If $U_{locfree,N} \leq 0$ and $P_{locstick,T} - \mu P_{locstick,N} > 0$: $P_N = \frac{-U_{locfree,N}}{(1+\mu \frac{w_{nt}}{w_{nn}})w_{nn}}$, $P_T = \mu P_N$.
- Otherwise, no sliding: $P = P_{locstick}$.

4.6 Solving the problem

Several iterative algorithms exist. [Moreau, 2000] describes the Gauss-Seidel method, shown in Algorithm 7, which is widely used in the literature due to its favorable convergence properties. Another variant is the Jacobi method, or the approach of [Coppin et al., 2022] using a queue to store unconverged contacts, solving them in a Gauss-Seidel fashion.

4.6.1 Gauss-Seidel

The Gauss-Seidel method iterates until convergence for all contacts. At $it = 0$, all reactions are initialized to zero. In each iteration, all contacts are considered individually, and each contact is solved using the previously described equations, with the current velocities used for computing the impulses from other contacts.

Algorithm 7 Gauss-Seidel solver

```

converged  $\leftarrow$  false
while not converged do
    itConverged  $\leftarrow$  false
    for each contact do
        body1, body2  $\leftarrow$  contact  $\triangleright$  get the bodies involved in the contact
        contribution  $\leftarrow$  solveContact(contact)  $\triangleright$  solve contact according to the equation
        body1.velocity  $\leftarrow$  body1.velocity + contribution
        body2.velocity  $\leftarrow$  body2.velocity - contribution  $\triangleright$  apply the contribution to the
        bodies' velocities
    converged  $\leftarrow$  converged and checkCriterion()  $\triangleright$  check convergence according to the
    chosen criterion
    end for
    if itConverged then
        converged  $\leftarrow$  true
    end if
end while

```

Convergence criterion

A common convergence criterion is to check whether all contributions to the velocities are below a threshold tol :

$$\forall \alpha, \left\| \mathbf{u}^{k+1\alpha} - \mathbf{u}^{k\alpha} \right\| \Delta t < tol \quad (4.23)$$

4.6.2 Jacobi method

The Jacobi method is similar to Gauss-Seidel but is easier to parallelize, though it converges more slowly. It follows the same principle, but the contributions of the contacts to the velocities are stored in an array and applied at the end of the iteration. The convergence criterion is the same as for Gauss-Seidel.

Successive Over-Relaxation

To improve the convergence of the Jacobi method, successive over-relaxation (SOR) can be used:

$$x^{n+1} = (1 - \omega)x^n + x_{n+1}\omega \quad (4.24)$$

where ω is the relaxation parameter, with $0 < \omega < 2$ and $\omega = 1$ corresponding to the standard Jacobi method.

The convergence criterion must be adapted to account for the relaxation parameter:

$$\forall \alpha, \left\| \mathbf{u}^{k+1\alpha} - \mathbf{u}^{k\alpha} \right\| \Delta t < tol \cdot \omega \quad (4.25)$$

Algorithm 8 Jacobi solver

```

converged  $\leftarrow$  false
while not converged do
  itConverged  $\leftarrow$  false
  contributions  $\leftarrow$  empty array  $\triangleright$  array to store the contributions of each contact
  for each contact do
    body1, body2  $\leftarrow$  contact  $\triangleright$  get the bodies involved in the contact
    contribution  $\leftarrow$   $\omega \text{solveContact}(\text{contact}) + (1 - \omega) \text{prevContact}$   $\triangleright$  solve the contact
    according to the equation
    contributions[body1.index]  $\leftarrow$  contributions[body1.index] + contribution
    contributions[body2.index]  $\leftarrow$  contributions[body2.index] - contribution
  end for
  for each body do
    body.velocity  $\leftarrow$  body.velocity + contributions[body.index]
    converged  $\leftarrow$  converged and checkCriterion()  $\triangleright$  check convergence according to the
    chosen criterion
  end for
  if itConverged then
    converged  $\leftarrow$  true
  end if
end while

```

The Jacobi method is well-suited for parallelization, as each contact can be solved independently, and the contributions to the velocities can be accumulated in a temporary array and applied at the end of each iteration.

Previous timestep contact solution

Another way to improve convergence, applicable to both Gauss-Seidel and Jacobi methods, is to use the contact solution from the previous timestep as an initial guess. Applying the previous timestep's contact solution to the current timestep can help accelerate convergence.

Chapter 5

Implementation

5.1 Data Structure Adaptation

In this master thesis, two different implementations of multi-circular bodies were evaluated, based on the existing MigFlow implementation for simple circular particles.

5.1.1 MigFlow Data Structure

Structure of Arrays

As previously explained, when designing data structures for the GPU, the approach is the opposite of the CPU: a structure of arrays is used instead of an array of structures to improve memory access on the GPU.

Kokkos View Kokkos provides an abstraction for vectors called *View*, which allows representing a vector in a way that is compatible with both GPU and CPU. A *View* can reside on the GPU, CPU, or both, and can be resized, copied, and manipulated like a standard vector. It can be easily transferred between the GPU and CPU via a mirror *View*.

VecKD

Current version of MigFlow GPU uses a data structure called VecKD (see Figure 5.1). It is a Kokkos View that stores the data for each body contiguously in memory. This is a 2D vector where the first dimension is the body index and the second is the data index. An enum called *fields* defines the different data, and accessor methods are provided for easy access.

It a certain way, it works like a classic struct, a bit more flexible but it may not be the best datastructure choice for coalescing calls as it mean that data need to be accessed in the same instruction or without saturating the cache.

Data Structure Implementation

MigFlow GPU, as shown in Figure 5.2, uses a *Bodies* structure that stores two VecKD arrays: one for integer data and one for floating-point data. Each VecKD array contains the data of the bodies, such as position, rotation, and velocity. The bodies structure is stored in the global

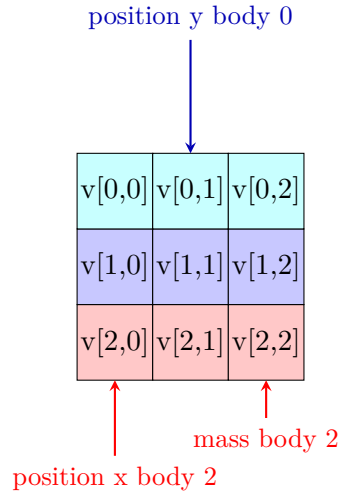


Figure 5.1: VecKD data structure example

memory of the GPU and is copied to CPU memory when needed, for example, when writing results to a file.

Bodies
- data : VecKD<scalar[fields::N_SCALAR]>
- idata : VecKD<int[fields::N_INT]>
+ Bodies(n : int = 0)

Figure 5.2: Original implementation of bodies structure

enum fields integer
MATERIAL,
ID,
TYPE,
SET,
N_INT

(a) Original implementation of fields enum for integer

enum fields float
IMASS = 0,
RADIUS = 1,
IINERTIA = 2,
POSITION = 3,
VELOCITY = POSITION + 2,
EXTERNAL_FORCE = VELOCITY + 2,
OMEGA = EXTERNAL_FORCE + 2
THETA = OMEGA + 1
N_SCALAR = THETA + 1

(b) Original implementation of fields enum for floating point

Figure 5.3: Original implementation of fields enum

5.1.2 Particles Structure Implementation

In the particles structure implementation, there are two separate structures: one containing the bodies and one containing the particles.

The same structure from the bodies structure is kept but with two new pieces of information: the maximum number of child particles and an array containing the child particles for each

Bodies
<pre>- data : VecKD<scalar[fields::N_SCALAR]> - idata : VecKD<int[fields::N_INT]> - children_particles : View<int *> - max_children_particles : int</pre>
<pre>+ Bodies(n : int = 0, max_children_particles_ : int = 1)</pre>

(a) Bodies

Particles
<pre>- ids : View<int *> - vec_parent_bodies : View<int *> - vec_radius : View<scalar *> - vec_relative_position : View<scalar *[dim]></pre>
<pre>+ Particles(n : int = 0)</pre>

(b) Particles

Figure 5.4: Naive implementation of bodies and particles

body. The particles structure contains all the child particles and their information: the relative position to the body, the radius, and the index of the body they belong to.

VecKD particles structure

An other implementation of the particles structure is tested that use the VecKD datastructure to store the particles information, it will allow to test if the VecKD structure is more efficient or less efficient than the other versions.

5.1.3 Constant Bodies

The constant bodies implementation approaches the problem differently: instead of having a structure for the bodies and one for the particles, different types of bodies are stored in a View that keeps a position for each particle. Each body then has a start index and an end index representing a range in the particles views. The end index is a design choice but maybe not the best; it duplicates information for the same bodies but allows different bodies to use the same particle information.

Storing these bodies in constant memory could be faster if all the bodies fit in the constant cache of an SM, but Kokkos does not offer any options for constant memory; pure CUDA code is needed, this option was thus not tested.

5.2 Contact Detection

As explained in Chapter 3, contact detection is performed using an LBVH tree:

1. Each body is assigned a Morton code based on its position, in parallel.
2. The bodies vectors are sorted using the Morton code.
3. The LBVH tree is built and each node's bounding box is computed.
4. Contacts are detected in parallel for each body and counted to resize the contact structure accordingly.
5. Contacts are stored in the CSR contacts structure, in parallel for each body.

Particles structure Variant In the particles structure implementation, detection is performed directly on the particles, so the previous steps are the same but applied to the particles.

Constant Bodies In the constant bodies implementation, the procedure is the same as in the original, but for each body, we need to iterate over its particles during the contact structure count and build. The contact structure is adapted to store the bodies and the specific particles involved in the contact. The bounding box size for the bodies needs to be adapted according to the maximum body radius; currently, it is set manually but could be computed.

The design choice of using the bounding box on the body was made for simplicity, based on the intuition that the cost of iterating over the particles and the larger bounding box is not significant, as the number of particles is small compared to the number of bodies. However, it could be interesting to compare these approaches.

5.3 Contact Resolution

1. Before resolving the contacts, those already solved in the previous timestep are applied to accelerate convergence of the Jacobi method, in parallel for each body.
2. The Jacobi loop then begins:
 - (a) In parallel, each contact is solved using the procedure explained in Chapter 4; each contact checks for convergence, and a reduction is applied to obtain the number of converged contacts.
 - (b) The contributions to velocities are stored in a temporary View during the previous step. These contributions are then applied in a new kernel.
 - (c) If the number of converged contacts is not equal to the total number of contacts, the previous step is repeated; otherwise, the process stops.
3. Positions are updated and external forces are applied to the bodies, in parallel for each body.
4. The next timestep begins.

5.3.1 Contact Resolution for Bodies

For bodies, the computation is slightly more complex than for circular particles; the steps are detailed in Chapter 4.

5.3.2 Velocity Update for Bodies

Particles Structure

For updating the velocities of the particles in the particles structure implementation, two different approaches were tested: an atomic operation and a list of particles.

Atomic Operation In the atomic operation version, parallelization is performed over each particle. For each particle, the contact velocities are found and the velocity of the body it belongs to is updated using an atomic operation. This implementation performed poorly because

atomic operations slow down the kernel and produce inconsistent results with floating point numbers due to the random execution order of threads and floating point precision.

List of Particles In the list of particles version, parallelization is performed for each body. For each body, its list of particles is iterated over and the body velocity is updated with the contribution to velocities from each particle.

Constant Bodies

For the constant bodies implementation, the approach is similar to the list of particles version: for each body in parallel, the algorithm iterates from the start index to the end index and adds the contributions to the body velocity. The temporary View is adapted to store the contributions of each particle.

Chapter 6

Results and performance

6.1 Simulations presentation

To evaluate the different implementations, two distinct simulations were conducted to compare precision, convergence, and performance.

6.1.1 Simulation 1: Stacked particles

The first simulation considers a simple scenario with n bodies, each composed of two particles. This setup is particularly useful for comparing the precision of the GPU implementation with the CPU version. Each body consists of two particles, stacked vertically with a slight horizontal offset. The bodies are subjected to gravity, and the objective is to observe their stacking and bouncing behavior, enabling a comparison between the GPU and CPU implementations.

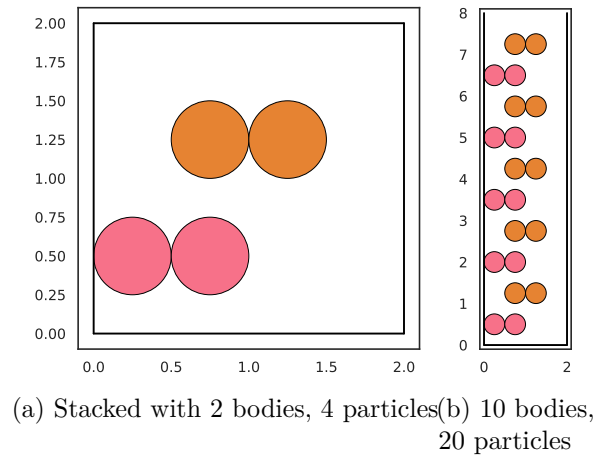


Figure 6.1: Stacked particles simulation with varying numbers of bodies.

Table 6.1: Body properties for Simulation 1

Body	Nb. of particles	Particle radius	Inv. Inertia	Mass
Body 1	2	0.25	0.05432	196.34954

Table 6.2: Simulation 1 default parameters

Δt	Nb. bodies	Ext. Forces	Relax.	Tol.	Implem.	Precision
1×10^{-3}	2	$-9.81 \times \text{mass}$	0.5	1×10^{-6}	Const.	double (64 bits)

6.1.2 Simulation 2: Tetris

The second simulation is a Tetris-like scenario, where each body is composed of four particles, and the total number of bodies is $n = x^2$, distributed evenly in the domain. Six different body types are included. This simulation enables the assessment of how body diversity influences performance and convergence.

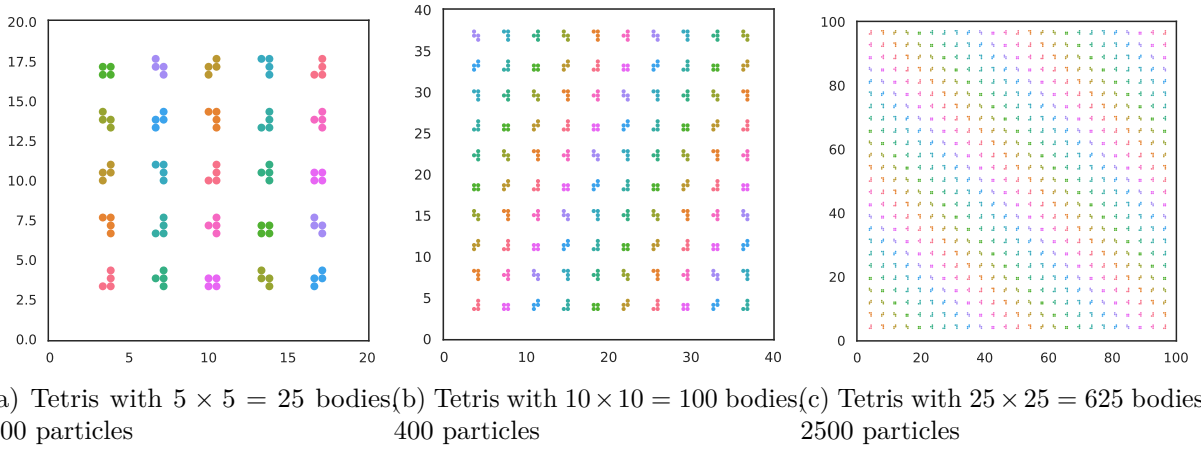


Figure 6.2: Tetris simulation with varying numbers of bodies.

Table 6.3: Body properties for Tetris simulation

Body	Nb. of particles	Particle radius	Inv. Inertia	Mass
Body 1	4	0.25	0.02037	196.34954
Body 2	4	0.25	0.02037	196.34954
Body 3	4	0.25	0.02328	196.34954
Body 4	4	0.25	0.02328	196.34954
Body 5	4	0.25	0.03259	196.34954
Body 6	4	0.25	0.02507	196.34954

Table 6.4: Tetris simulation default parameters

Δt	Nb. bodies	Ext. Forces	Relax.	Tol.	Implem.	Precision
1×10^{-3}	25, 100	$-9.81 \times \text{mass}$	0.5	$10^{-6}, 10^{-4}$	Const.	double (64 bits)

6.2 Precision

6.2.1 Simulation Comparisons

Stacked particles

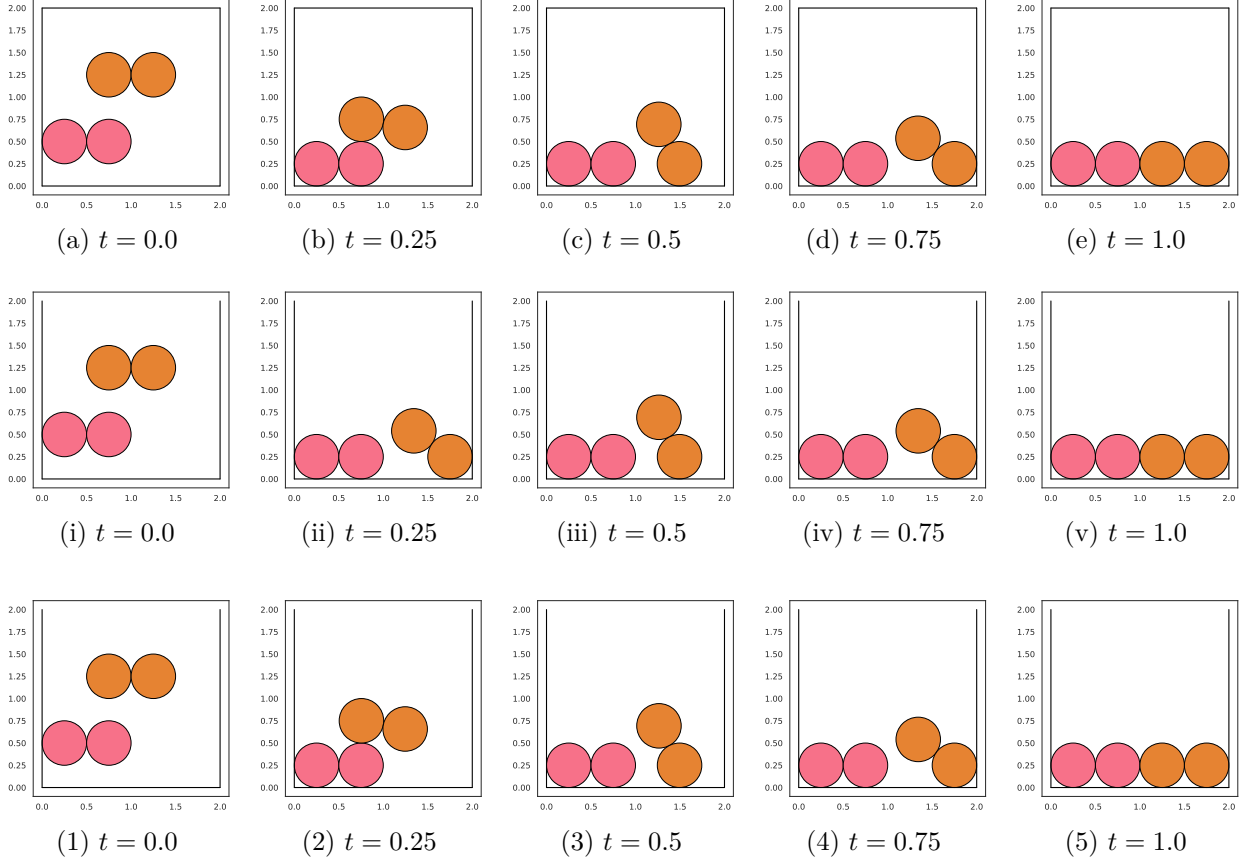


Figure 6.3: Snapshots of the stacked particles $n = 2$ simulation at different times. The first row shows the GPU implementation. The second row shows the CPU Jacobi implementation, and the third row shows the CPU Gauss-Seidel implementation.

For the stacked particles case with $n = 2$, the GPU, CPU Jacobi, and Gauss-Seidel implementations yield visually indistinguishable results. The bodies fall and the second body bounces on the first, aligning horizontally. No perceptible differences are observed between the implementations.

Tetris

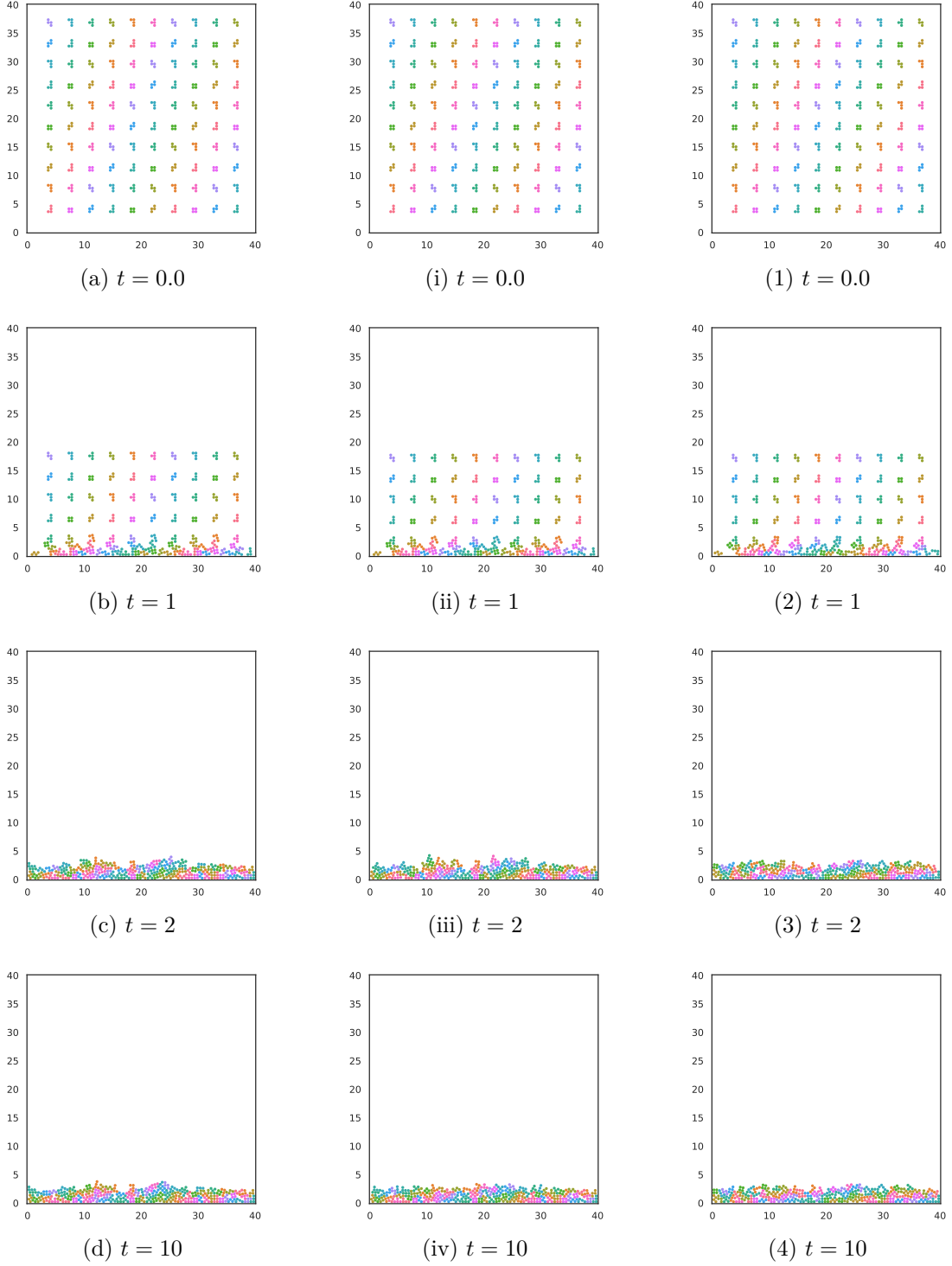


Figure 6.4: Snapshots of the Tetris $n = 10 \times 10 = 100$ simulation at different times. Center is Jacobi on CPU and the left column shows the GPU. The right column shows the same simulation with the CPU Gauss-Seidel implementation.

For the Tetris simulation, the GPU, CPU Jacobi, and Gauss-Seidel implementations produce slightly different results. The bodies fall, stack, and eventually reach a stable state, as observed from $t = 2$ to $t = 10$. During this interval, the bodies remain stable and do not exhibit further

movement. Overall, the GPU implementation closely resembles the CPU Jacobi implementation, which is expected due to their use of the same algorithm. The Gauss-Seidel implementation, while similar, deviates slightly from the two Jacobi implementations.

6.2.2 Contact reactions and normals

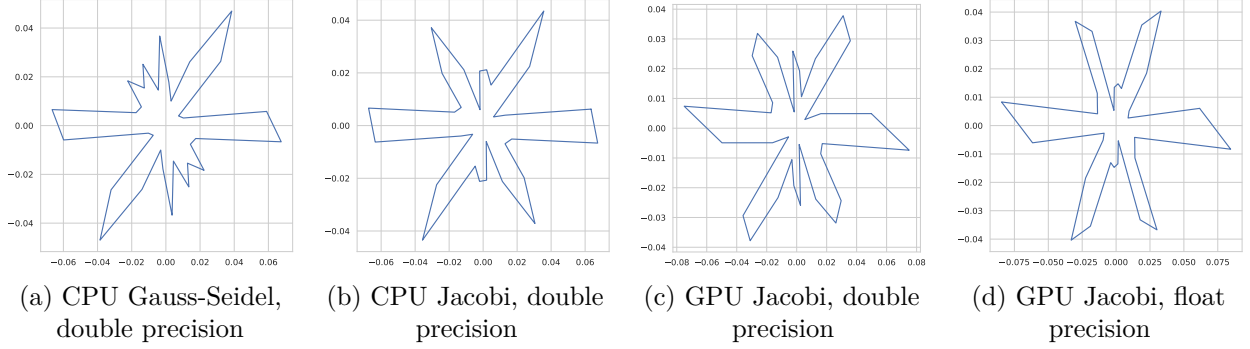


Figure 6.5: Contact normal histograms (32 bins) for the Tetris $n = 5 \times 5$ simulation for different implementations and precisions, bodies-segment contacts omitted

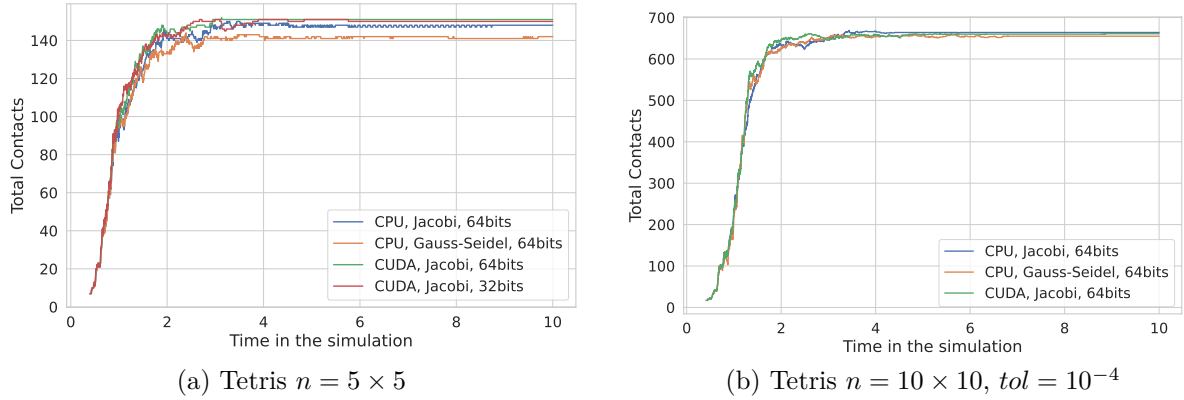


Figure 6.6: Number of contacts over different simulations.

6.2.3 Precision analysis

Jacobi vs Gauss-Seidel

As [Renouf et al., 2004] explains, Gauss-Seidel can give different acceptable solutions depending on the order of contact resolution. This explains the difference between the Jacobi and Gauss-Seidel methods: they solve the contacts differently, both solutions are acceptable depending even if they differ a bit. In the figure, we can clearly see that the contact normals are different but similar following the same pattern, and the difference is small enough to be acceptable for the simulation.

Floating point precision

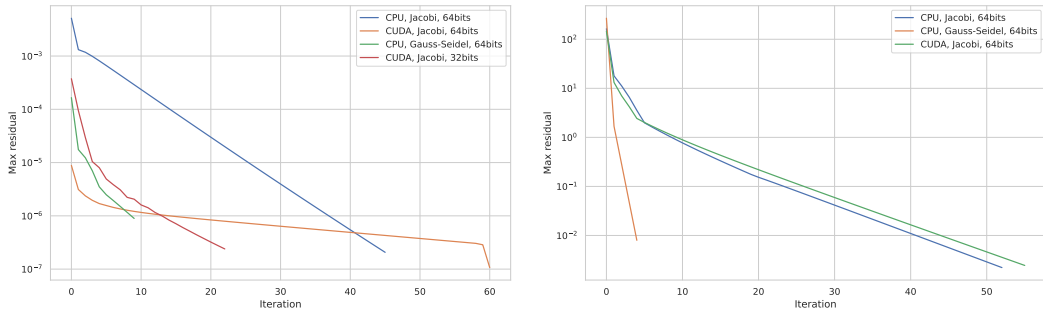
Order of operations CPU Jacobi and the GPU Jacobi results differ a bit, this is due to floating point precision and the order of operations; changing the order of operations change the

results, error accumulating throughout the simulation and thus the results of the simulation.

Float and double Another factor is the use of float (32-bit) or double (64-bit) precision for the simulation. Double is more precise and is used for the CPU version. On GPU, we tested the impact of using float or double, and we can see that double gives results closer to the CPU version. However, as mentioned earlier, most commercial GPUs have fewer FP64 units than FP32, so performance could be affected by the use of double.

6.3 Convergence

6.3.1 Gauss-Seidel vs Jacobi convergence



(a) For tetris $n = 5 \times 5$ at $t = 2$

(b) For tetris $n = 10 \times 10$, $tol = 10^{-4}$ at $t = 1$

Figure 6.7: Average number of iterations for jacobi over the simulation for different dt parameters. (The convergence criterion for jacobi include the relaxation parameter, which explain the different threshold for convergence.)

It is well-established that the Gauss-Seidel method have better convergence property than the Jacobi method. This is because the Gauss-Seidel method uses the most recently computed values for the next iteration, while the Jacobi method relies on values from the previous iteration. This behavior is clearly illustrated in our simulation with the figure 6.7: for various simulations, the Gauss-Seidel method consistently converges faster than the Jacobi method.

6.3.2 Convergence parameters

Jacobi method's convergence is really sensitive to the choice of parameters. Currently, there is limited research on parameter selection for NSCD simulations in the literature. The tuning of parameters is highly dependent on the simulation, including factors like the number of bodies and the number of inter-contacts.

In general, simulations with a high number of inter-contacts between bodies tend to diverge with a too large Δt . Precision also impacts convergence; different implementations of the same algorithm may converge or diverge due to floating point precision, which can be counterintuitive.

An example from the collected data shows that the simulation tetris $tol = 1e - 6$, $\Delta t = 1e - 3$ with $n = 40 \times 40$ diverges, while the simulation with $n = 50 \times 50$ converges. This can be attributed to floating point precision and slightly different solutions that converge, this kind of behavior is not uncommon in numerical simulations.

Table 6.5: Example of convergence with different parameters and methods

Simulation	Parameters		Methods	
	Tol.	Δt	Gauss-Seidel	Jacobi
Tetris 40×40	10^{-4}	1×10^{-4}	Converges	Diverge
Tetris 50×50	10^{-4}	1×10^{-4}	Converges	Converges
Tetris 100×100	10^{-4}	1×10^{-4}	Converges	Diverges
Stacked $n = 50$	10^{-4}	1×10^{-4}	Converges	Diverges

Previous contacts application

As explained in 5.3, the Jacobi method (and Gauss-Seidel) can be improved by applying previous contacts to the next iteration. This is a significant improvement that can greatly reduce the number of iterations needed for convergence. Tests have shown a noticeable reduction in the number of iterations required for convergence when using previous contacts. One notable reason is that in the final state of the simulation, the contacts are stable and do not change much, so applying previous contacts will converge very quickly.

Simulation	Wo prev. contacts	W. prev. contacts
Tetris $n = 5 \times 5$	222	16
Tetris $n = 10 \times 10$	151	4

Table 6.6: Average number of iterations for the Jacobi method with and without previous contacts application from $t = 0$ to $t = 10$ for the Tetris simulation.

Relaxation parameter

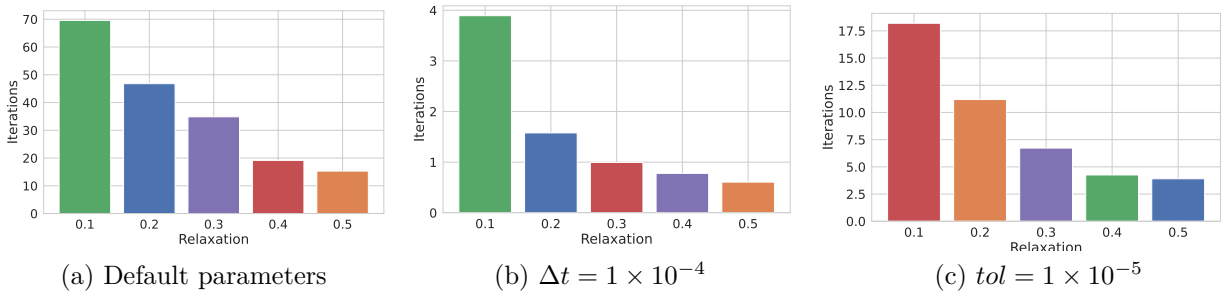


Figure 6.8: Average number of iterations for Jacobi over the simulation for different relaxation parameters, tolerance, and Δt in the Tetris $n = 5 \times 5$ simulation.

Choosing the appropriate relaxation parameter for the SOR method 4.6.2 is crucial: a value that is too small can lead to slow convergence as the reactions update slowly, while a value that is too large can cause the simulation to diverge.

From the results, the best parameter for the relaxation parameter is around 0.5, more it diverges and less it converges slowly.

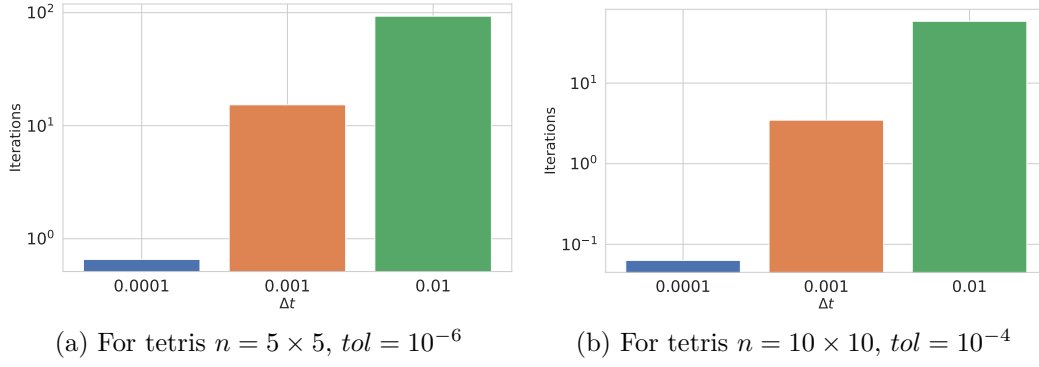


Figure 6.9: Average number of iterations for Jacobi over the simulation for different Δt parameters.

Δt selection

The choice of time step Δt also impacts convergence. A time step that is too small can result in a lengthy simulation with the Jacobi doing few iterations, while a time step that is too large may lead to non-convergence of the simulation or the Jacobi method converging only after a prolonged period.

Tolerance

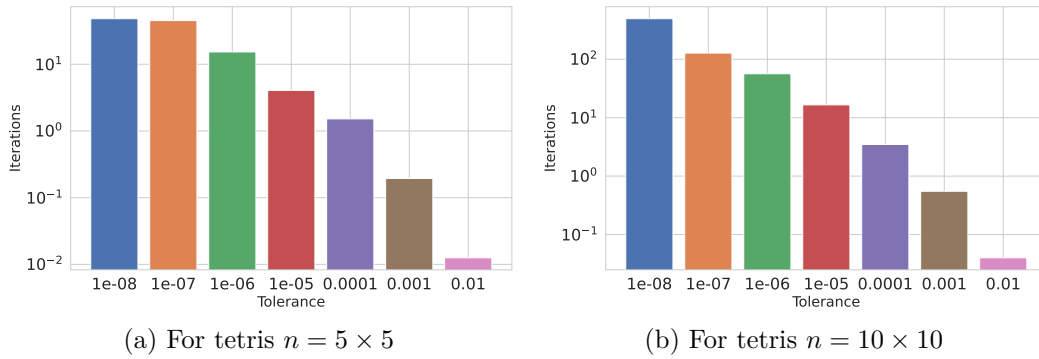


Figure 6.10: Average number of iterations for Jacobi over the simulation for different tolerance parameters.

Selecting the appropriate tolerance is critical: a tolerance that is too small can lead to a prolonged simulation with the Jacobi method converging very slowly, while a tolerance that is too large can result in unrealistic simulations with significant interpenetration. Therefore, the choice of tolerance involves a trade-off between precision and performance.

Conclusion on parameters

In conclusion, the choice of parameters is very important for the convergence of the simulation. The relaxation parameter, time step Δt , and tolerance are both parameters that can greatly affect the convergence of the simulation.

There exists a trade-off choice between precision and performance. If our simulation diverge, we can choose to decrease the time step Δt increasing simulation running time or decrease the

tolerance, leading to a less precise simulation.

If Δt becomes too small, NSCD may no longer be the best choice. In such cases, DEM—with strict conditions on interpenetration and small timesteps—could be a better option, as explicit methods are very fast in this regime. Alternatively, one can consider the convex version of NSCD for improved convergence, or use the sequential Gauss-Seidel method, which generally offers much better convergence properties and can operate with larger Δt values.

6.4 Performance

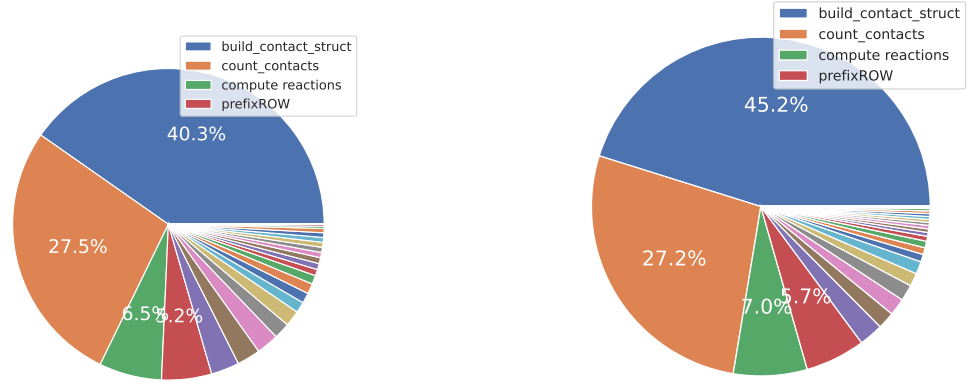
Performance is a primary motivation for this work, with the objective of achieving fast and accurate simulation of contacts between bodies. This section presents performance results for the various GPU implementations, as well as comparisons with the CPU Migflow framework.

Performance is evaluated using two main metrics: the time spent in the kernel and the total execution time, which includes both kernel time and host code execution. Kernel time and total execution time for the GPU are measured using *simpleKernelTimer* from [Kok, 2025]. For the CPU, the time is measured using *std::chrono::high_resolution_clock*; hardware details are provided in section 1.5.3.

6.4.1 Program hotspots

Identifying program hotspots is essential for optimization. The time spent in different kernels is highly dependent on the simulation (number of contacts, contact density) and parameters (tolerance, time step). In the present case, contact detection and contact list construction are more computationally expensive than contact resolution. This is primarily because, at the start of the simulation, there are few contacts, so tree construction dominates, while in the final state, many contacts exist, making contact detection costly. However, contact resolution becomes faster due to the application of previous contacts 6.3.2 and the stability of the contact set. The use of a BVTT front-based collision approach, as explained in section 3.5.3, for contact detection could potentially address this bottleneck.

As contact detection is not the focus of this thesis, only a brief overview of current bottlenecks is provided. The remainder of the performance section focuses on contact resolution performance.



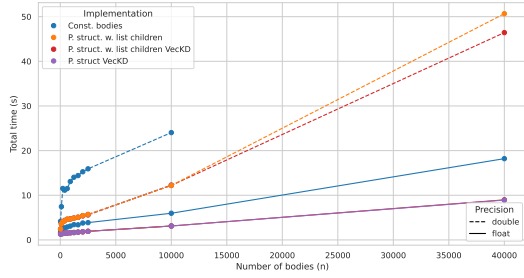
(a) Tetris $n = 50 \times 50$

(b) Tetris $n = 100 \times 100$

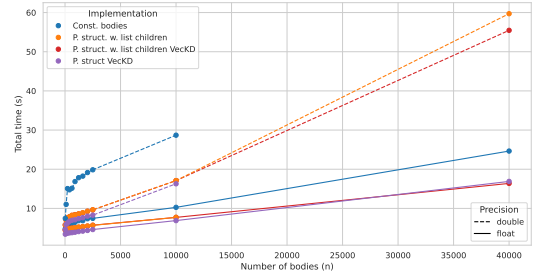
Figure 6.11: Kernel time distribution for Tetris simulation with $tol = 10^{-3}$ and $\Delta t = 10^{-4}$, over simulation time $t = 10$ for the Particles struct with children implementation.

6.4.2 Comparison of GPU implementations

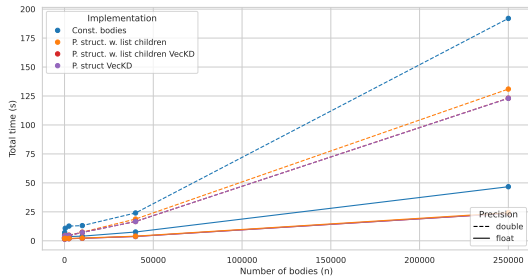
Contact Resolution



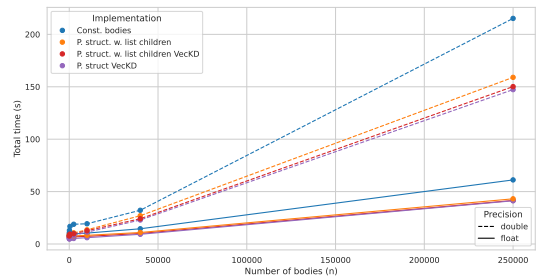
(a) Total time in kernel for contact resolution ($tol = 10^{-3}$)



(b) Total time for NSCD ($tol = 10^{-3}$)



(i) Total time in kernel for contact resolution ($tol = 10^{-2}$)



(ii) Total time for NSCD ($tol = 10^{-2}$)

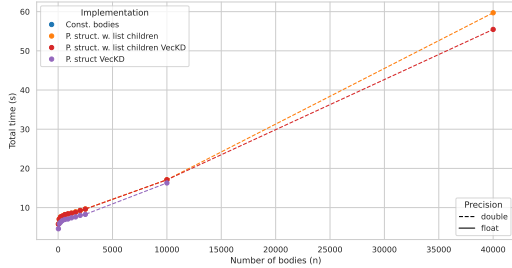
Figure 6.12: Contact resolution kernel and NSCD time for Tetris simulations. The first row corresponds to $tol = 10^{-3}$ and $\Delta t = 10^{-4}$. The second row corresponds to $tol = 10^{-4}$ and $\Delta t = 10^{-4}$. If no data point is present, the simulation diverged.

A notable observation is that double precision is significantly slower than float precision, as expected and discussed in 2.1.2. The difference between the list children and atomic versions

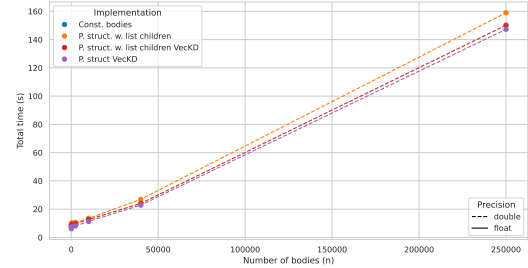
is minimal; the time spent in the kernel for contact resolution is nearly identical, as shown in 6.12a. The overall speed is slightly slower for the list children version, which can be explained by the update velocity kernel running on each body rather than each particle, and the atomic operations not impacting performance as much as anticipated. The atomic version is slightly faster than the list children version, but the difference is not significant.

The constant bodies implementation is the slowest, as it does not benefit from the constant cache due to Kokkos limitations 5.1.3, instead using global memory, which reduces memory access efficiency.

Note that some implementations diverged for $tol = 10^{-4}$. To run the same simulations with this tolerance, the time step would need to be reduced, increasing the simulation time. Therefore, $tol = 10^{-2}$ is used in the following for its faster convergence to run large simulations up to $n = 250\,000$ with 1 million particles. In practice, such a high tolerance would not be used, as it would not yield realistic results.



(a) $tol = 10^{-4}$

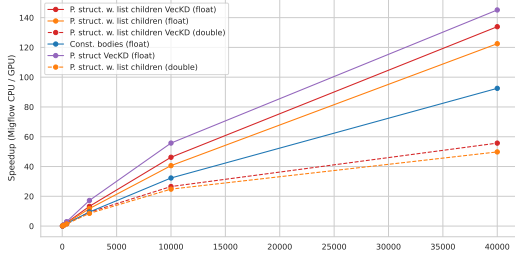


(b) $tol = 10^{-2}$

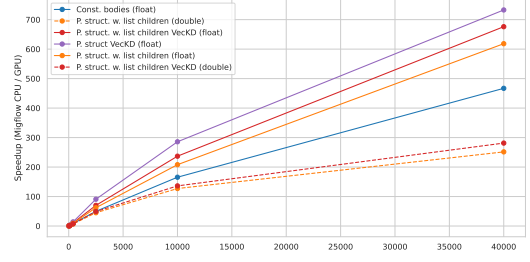
Figure 6.13: VecKD vs Non-VecKD, NSCD total time for different tolerance parameters

VecKD vs non VecKD The difference between VecKD and non-VecKD usage for particles becomes more noticeable with large simulations. Interestingly, the trend observed is that the non-VecKD version is slightly slower than the VecKD version 5.1.1, which is the opposite of what was initially expected. For further work, to be certain that VecKD is the right choice, it would be useful to implement a non-VecKD version of bodies to fully compare the two approaches.

6.4.3 CPU vs GPU



(a) NSCD kernel speedup



(b) Total program speedup

Figure 6.14: (a) Speedup of the GPU NSCD part compared to the Gauss-Seidel CPU NSCD part for the Tetris simulation with $tol = 10^{-2}$ and $\Delta t = 10^{-4}$. (b) Total program speedup (including all steps) for the same simulation.

Figure 6.14a (noting that results depend heavily on the hardware used, but provide a general idea) shows that the GPU implementation achieves a significant speedup compared to the CPU Gauss-Seidel implementation, with up to $140\times$ speedup for the Tetris simulation with $n = 500 \times 500$ and 1 million particles. The speedup increases less rapidly for FP64 than for FP32, as expected. The total program speedup is even more impressive, reaching up to $700\times$ for the same simulation, as shown in Figure 6.14b. This is primarily because the GPU implementation can run contact resolution in parallel, leveraging the GPU's parallelism, while the CPU implementation is sequential.

Conclusion

Multiple implementations of multicircular bodies were evaluated in this thesis, based on the existing MigFlow implementation for simple circular particles. The results show that significant performance improvements can be achieved by using GPU acceleration, with a substantial speedup compared to the CPU implementation. However, the Jacobi solver has reached its limits in terms of convergence, making it difficult to tune the parameters for an optimal balance between performance and precision. Achieving high-precision simulations requires very small time steps, which raises questions about the suitability of NSCD in such cases. In these scenarios, DEM or a convex optimization-based NSCD formulation may be a better alternative. Furthermore, the use of double-precision (FP64) arithmetic, which is often necessary for high-precision, is slower than single-precision (FP32) on most commercial GPUs due to the much lower number of FP64 units compared to FP32 units, further impacting performance but still much faster than CPU implementations.

In conclusion, the Jacobi GPU implementation of NSCD for multicircular bodies is a promising approach, but it requires further research to improve its convergence. Future work could focus on optimizing the Jacobi solver or exploring alternative methods for solving the system of equations.

Bibliography

- [Flu, 2025] (2025). fluidparticles / MigFlow · GitLab.
- [Kok, 2025] (2025). kokkos/kokkos-tools. original-date: 2016-02-11T00:49:46Z.
- [NVI, 2025] (2025). NVIDIA vs. AMD Discrete GPU Market Share (2010 to 2025) PCviewed.com. Section: Insights.
- [Aamodt et al., 2018] Aamodt, T., Wai Lun Fung, W., and Rogers, T. (2018). General-Purpose Graphics Processor Architecture.
- [Anitescu, 2006] Anitescu, M. (2006). Optimization-based simulation of nonsmooth rigid multi-body dynamics. *Mathematical Programming*, 105(1):113–143. Publisher: Springer Verlag.
- [Anzt and Dongarra, 2018] Anzt, H. and Dongarra, J. (2018). A Jaccard Weights Kernel Leveraging Independent Thread Scheduling on GPUs. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 229–232. ISSN: 1550-6533.
- [Apetrei, 2014] Apetrei, C. (2014). Fast and Simple Agglomerative LBVH Construction. In *Computer Graphics and Visual Computing (CGVC)*, page 4 pages. The Eurographics Association. Artwork Size: 4 pages.
- [Balzer et al., 2013] Balzer, M., Kleinert, J., and Obermayr, M. (2013). Parallel implementation of the non-smooth contact dynamics method for large particle systems.
- [Bentley, 1975] Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517.
- [Berger and Hrenya, 2017] Berger, K. J. and Hrenya, C. M. (2017). Predicting Regolith Erosion during a Lunar Landing: Role of Continuous Size Distribution. *Journal of Aerospace Engineering*, 30(5):04017027. Publisher: American Society of Civil Engineers.
- [Brendel et al., 2004] Brendel, L., Unger, T., and Wolf, D. E. (2004). Contact Dynamics for Beginners. In Hinrichsen, H. and Wolf, D. E., editors, *The Physics of Granular Media*, pages 325–343. Wiley, 1 edition.
- [Buck et al., 2004] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. (2004). Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786.
- [Cooper-Baldock et al., 2024] Cooper-Baldock, Z., Almirall, B. V., and Inthavong, K. (2024). Speed, power and cost implications for GPU acceleration of Computational Fluid Dynamics on HPC systems. arXiv:2404.02482 [cs].

- [Coppin et al., 2022] Coppin, N., Constant, M., Lambrechts, J., Dubois, F., and Legat, V. (2022). Numerical analysis of the drag on a rigid body in an immersed granular flow. *Comp. Part. Mech.*, 9(3):393–406.
- [Crovella, 2020] Crovella, B. (2020). GPU Performance Analysis.
- [Cundall and Strack, 1979] Cundall, P. A. and Strack, O. D. L. (1979). A discrete numerical model for granular assemblies. *Géotechnique*, 29(1):47–65. Publisher: ICE Publishing.
- [de Arruda Tino and Tavares, 2022] de Arruda Tino, A. A. and Tavares, L. M. (2022). Simulating breakage tests using the discrete element method with polyhedral particles. *Comp. Part. Mech.*, 9(4):811–823.
- [Dennard et al., 1974] Dennard, R., Gaensslen, F., Yu, H.-N., Rideout, V., Bassous, E., and LeBlanc, A. (1974). Design of ion-implanted MOSFET’s with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268.
- [Dong et al., 2022] Dong, Y., Yan, D., and Cui, L. (2022). An Efficient Parallel Framework for the Discrete Element Method Using GPU. *Applied Sciences*, 12(6):3107. Number: 6 Publisher: Multidisciplinary Digital Publishing Institute.
- [Dubois et al., 2018] Dubois, F., Acary, V., and Jean, M. (2018). The Contact Dynamics method: A nonsmooth story. *Comptes Rendus Mécanique*, 346(3):247–262.
- [Ericson, 2004] Ericson, C. (2004). Real-Time Collision Detection: | Guide books | ACM Digital Library.
- [Fan et al., 2011] Fan, W., Wang, B., Paul, J.-C., and Sun, J. (2011). A Hierarchical Grid Based Framework for Fast Collision Detection - Fan - 2011 - Computer Graphics Forum - Wiley Online Library.
- [Ferrari et al., 2020] Ferrari, F., Lavagna, M., and Blazquez, E. (2020). A parallel-GPU code for asteroid aggregation problems with angular particles. *Monthly Notices of the Royal Astronomical Society*, 492(1):749–761. arXiv:1912.04197 [astro-ph].
- [Finkel and Bentley, 1974] Finkel, R. A. and Bentley, J. L. (1974). Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9.
- [Fredkin, 1960] Fredkin, E. (1960). Trie memory. *Commun. ACM*, 3(9):490–499.
- [He, 1999] He, T. (1999). Fast collision detection using QuOSPO trees. In *Proceedings of the 1999 symposium on Interactive 3D graphics*, I3D ’99, pages 55–62, New York, NY, USA. Association for Computing Machinery.
- [Jean, 1999] Jean, M. (1999). The non-smooth contact dynamics method. *Computer Methods in Applied Mechanics and Engineering*, 177(3):235–257.
- [Jradi et al., 2017] Jradi, W., Nascimento, H. d., and Martins, W. (2017). A Fast and Generic GPU-Based Parallel Reduction Implementation. arXiv:1710.07358 [cs].
- [Kafashan et al., 2019] Kafashan, J., Wiącek, J., Abd Rahman, N., and Gan, J. (2019). Two-dimensional particle shapes modelling for DEM simulations in engineering: a review. *Granular Matter*, 21.
- [Karras, 2012] Karras, T. (2012). Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. Artwork Size: 5 pages ISBN: 9783905674415 ISSN: 2079-8679 Pages: 5

- pages Publication Title: Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics.
- [Karras and Aila, 2013] Karras, T. and Aila, T. (2013). Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference, HPG '13*, pages 89–99, New York, NY, USA. Association for Computing Machinery.
- [kokkos,] kokkos. Kokkos documentation.
- [Lauterbach et al., 2009] Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., and Manocha, D. (2009). Fast BVH Construction on GPUs. *Computer Graphics Forum*, 28(2):375–384.
- [Lazaveric et al., 2002] Lazaveric, D., Dvornik, J., and Kesimir, F. (2002). Contact Detection Algorithm for Discret Element Method.
- [Lindholm et al., 2001] Lindholm, E., Kilgard, M. J., and Moreton, H. (2001). A user-programmable vertex engine. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques, SIGGRAPH '01*, pages 149–158, New York, NY, USA. Association for Computing Machinery.
- [Luding, 2008] Luding, S. (2008). Introduction to discrete element methods: Basic of contact force models and how to perform the micro-macro transition to continuum theory. *European Journal of Environmental and Civil Engineering*, 12(7-8):785–826.
- [Mandarapu et al., 2025] Mandarapu, D. K., James, N., and Kulkarni, M. (2025). Mochi: Fast \& Exact Collision Detection. arXiv:2402.14801 [cs].
- [Moore, 1965] Moore, G. E. (1965). Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35.
- [Moore, 1975] Moore, G. E. (1975). Progress in digital integrated electronics [Technical literature, Copyright 1975 IEEE. Reprinted, with permission. Technical Digest. International Electron Devices Meeting, IEEE, 1975, pp. 11-13.]. *IEEE Solid-State Circuits Society Newsletter*, 11(3):36–37.
- [Moreau, 2000] Moreau, J. J. (2000). Contact et frottement en dynamique des systèmes de corps rigides. *Revue Européenne des Éléments Finis*, 9(1-3):9–28.
- [Morrison, 1968] Morrison, D. R. (1968). PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4):514–534.
- [Nickolls et al., 2008] Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable parallel programming with CUDA. In *ACM SIGGRAPH 2008 classes*, SIGGRAPH '08, pages 1–14, New York, NY, USA. Association for Computing Machinery.
- [NVIDIA, 2017] NVIDIA (2017). volta-architecture-whitepaper.
- [NVIDIA, 2022] NVIDIA (2022). NVIDIA ADA GPU Architecture Whitepaper.
- [NVIDIA, 2025a] NVIDIA (2025a). CUDA C++ Best Practices Guide.
- [NVIDIA, 2025b] NVIDIA (2025b). CUDA C++ Programming Guide.
- [Prokopenko and Lebrun-Grandie, 2024] Prokopenko, A. and Lebrun-Grandie, D. (2024). Revising Apetrei’s bounding volume hierarchy construction algorithm to allow stackless traversal. Technical Report ORNL/TM–2024/3259, 2301619.

- [RADJAÏ et al., 2010] RADJAÏ, F., Dubois, F., and Collectif (2010). *Modélisation numérique discrète des matériaux granulaires*. Hermès science. Google-Books-ID: 1kZCEQAAQBAJ.
- [Renouf et al., 2004] Renouf, M., Dubois, F., and Alart, P. (2004). A parallel version of the non smooth contact dynamics algorithm applied to the simulation of granular media. *Journal of Computational and Applied Mathematics*, 168(1-2):375–382.
- [Schwartz et al., 2014] Schwartz, S. R., Michel, P., Richardson, D. C., and Yano, H. (2014). Low-speed impact simulations into regolith in support of asteroid sampling mechanism design I: Comparison with 1-g experiments. *Planetary and Space Science*, 103:174–183.
- [Sun et al., 2020] Sun, Y., Agostini, N. B., Dong, S., and Kaeli, D. (2020). Summarizing CPU and GPU Design Trends with Product Data. arXiv:1911.11313 [cs].
- [Wang et al., 2018] Wang, X., Tang, M., Manocha, D., and Tong, R. (2018). Efficient BVH-based Collision Detection Scheme with Ordering and Restructuring. *Computer Graphics Forum*, 37(2):227–237. _eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13356>.
- [Woolley, 2013] Woolley, C. (2013). GPU Optimization Fundamentals.
- [Zhao et al., 2021] Zhao, H., Huang, Y., Liu, Z., Liu, W., and Zheng, Z. (2021). Applications of Discrete Element Method in the Research of Agricultural Machinery: A Review. *Agriculture*, 11(5):425. Number: 5 Publisher: Multidisciplinary Digital Publishing Institute.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl