MASTER'S THESIS | LUND UNIVERSITY 2015

GPU Usage for Parallel Functions and Contacts in Modelica

Axel Goteman, Vilhelm Roxling

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2015-41

GPU Usage for Parallel Functions and Contacts in Modelica

Axel Goteman axel.goteman@gmail.com
Vilhelm Roxling vilhelm.roxling@gmail.com

The thesis was carried out at Dassault Systemes AB, Lund, for the Department of Computer Science at LTH

September 17, 2015

Supervisors:
Michael Doggett, michael.doggett@cs.lth.se
Hilding Elmqvist, hilding.elmqvist@3ds.com

Examiner:
Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

This thesis investigates two ways of incorporating GPUs in Modelica. The first by automatically generating GPU code for Modelica functions, and the second by using GPU accelerated external code for a contact handling package.

Automatic parallelization of functions is desired, as it can potentially accelerate large simulations significantly. Special patterns of nested for-loops in Modelica code are recognized and translated into CUDA kernel functions. Inline integration allows a broader spectrum of models to take advantage of the parallelization, by reducing CPU-GPU transfers. The prototype has been tested and achieved a speed-up factor of up to five compared to the CPU.

The contact handling package is capable of handling both complex contact behavior between arbitrarily shaped bodies and large DEM-like simulations, something which Modelica is currently lacking. Attempts to accelerate the package with GPUs were made, with partial success for the broad phase. The package uses Morton encoding for the broad phase, and the narrow phase is based on CSG intersection with BSP trees. Contact response is calculated using a volume dependent method, taking friction, damping and multiple contact points into account. The capability of the package was demonstrated by the fact that both complex contact behavior such as the inversion of the Tippe Top toy and tens of thousands of colliding spheres could be simulated.

Keywords: GPU, Modelica, Auto-generation, Collision Detection, Collision Response, Arbitrary Shapes

Acknowledgments
First of all we would like to thank our company supervisor Hilding Elmqvist for initiating the thesis, and for continued support throughout the work. We would also like to thank our leading supervisor Michael Doggett, for helpful input and discussions. A big thanks to Dassault Systemes AB, and the full staff at the Lund office. Special thanks to Hans Olsson, Sven-Erik Mattsson and Toheed Ghandriz for all the helpful discussions and advice. Also thanks to Alexander Pollok and Dirk Zimmer, at Institute of Dynamics and Control, DLR, Germany, for providing test models. Finally, we would like to thank family and friends for all their support and shown interest.

Terminology and Abbreviations

AABB - Axis Aligned Bounding Box
BSP - Binary Space Partitioning
Body - The term used for objects/particles in the contact handling package.
CFD - Computational Fluid Dynamics
CPU - Central Processing Unit
CSG - Constructive Solid Geometry
CUDA - Compute Unified Device Architecture. A GPU programming model developed by NVIDIA.
DEM - Discrete Element Method
Device - Unless otherwise stated, the GPU
DS - Dassault Systemes
Dymola - Dynamic Modeling Laboratory
FVM - Finite Volume Method
GPU - Graphics Processing Unit
Host - Unless otherwise stated, the CPU
IDE - Integrated Development Environment
Interactivity - In this thesis, it refers to the interactivity between different physical domains, unless otherwise stated.
MSL - Modelica Standard Library
ODE - Ordinary Differential Equation
Performance - In this thesis, it refers to performance in terms of computational time, unless otherwise stated.
PDE - Partial Differential Equation
DRAM - Dynamic Random Access Memory
Scene - Refers to the set of bodies present in a simulation with the contact handling package.
SIMD - Single Instruction Multiple Data
SM - Streaming Multiprocessor
SP - Streaming Processor
STL - Standard Template Library

Contents

Part I

Introduction

Chapter 1

Introduction

1.1 Problem Description

Modelica is an equation based programming language, designed for large multi-physics system simulations. Today, models tend to grow large, which naturally increases simulation times. Accelerating such models could e.g. speed-up the process of prototyping new products, thus lower the cost of developing the products. Some models may have the potential to perform better in a parallel execution environment. Dassault Systemes (DS), who is the developer of one of the leading Modelica Integrated Development Environments (IDEs), Dymola, wanted to evaluate the possibility of using GPUs to enhance Modelica. More exactly, they wanted an investigation of the effect of GPU usage in specific real world problems, rather than a theoretical evaluation of general GPU possibilities in Modelica.

DS recognized the lack of efficient contact handling between bodies for Modelica, both for Discrete Element Method (DEM) simulations and for more complex contacts between fewer objects. Since DEM is a problem generally well suited for parallel execution, see e.g. Xiaoqiang et al.[34], a GPU accelerated contact handling package was one of the things they wanted investigated. The other thing they wanted investigated was the possibility of generating GPU code for functions automatically. A good category of problems where functions suitable for parallel execution is fluid simulations.

This naturally divides the thesis into two distinct parts; automatic generation of GPU code for Modelica functions, and developing a contact handling package for Modelica, with potential GPU accelerations.

1.1.1 Automatic GPU Code Generation for Modelica Functions

In order for automatic GPU code generation of functions to be useful in Modelica, it would have to keep Modelica's high-level programming style. This means that the user should only have to suggest with e.g. an annotation, to the compiler, that

a certain function may be suitable for GPU execution. No, or few changes to the language should be made.

1.1.2 Contact Handling in Modelica

As with most Modelica packages, DS wanted this package to be of as general use as possible, which translated into the following specifications:

- The shape of bodies should be arbitrary, e.g. concave and holed bodies should be supported.
- Bodies should be able to have varied shape complexity and size in the same scene.
- It should keep Modelica's high-level, user friendly style, e.g. by supporting the drag and drop feature of bodies.
- It should support interaction with other Modelica models, especially within Modelica's MultiBody library[29].
- It should be able to handle large number of bodies (DEM).

1.2 Foundations

1.2.1 GPU Programming

GPUs today are used more and more for general purpose computations, since harnessing the GPUs parallel execution potential allows for large performance boosts on some problems. Different tools have been developed in the recent years, to make the power of GPUs more accessible.

CUDA was the chosen GPU programming model for this thesis, for several reasons. First of all, CUDA is developed by NVIDIA for their GPUs, and since NVIDIA GPUs where used during this thesis, that would likely induce less problems than another choice. Secondly, the authors had limited prior knowledge in GPU programming, and many sources indicated that CUDA is one of the easier tools to get started with. Since the goal was to investigate possible improvement, and not to evaluate maximum performance, the performance of CUDA compared to other GPU programming models was considered sufficient.

1.2.2 Automatic GPU Code Generation for Modelica Functions

Automatic GPU code generation of Modelica models has been done before by Gebremedhin et al.[1], in the form of a proposed extension to Modelica, called ParModelica. But this extension requires knowledge in parallel computing to use, as it was up to the user to specify which memory to use for the different GPU variables. Large speed-up factors were achieved, but only on simple direct calculations, such as matrix multiplications, something which is not the primary area of usage for Modelica. Since that work never became widely used, it appears that sacrificing ease of

use for speed should be done with restraint, as well as highlighting the importance of testing GPUs on areas where Modelica is likely to be used.

GPU acceleration of fluid simulations, or computational fluid dynamics (CFD), is nothing new, it has been done by e.g. Karlsson [6]. However, a GPU acceleration of a CFD model in Modelica means a GPU acceleration of a system with high interactive possibilities, which would be more unique.

There are a lot of tools for different kinds of flow simulations in the Modelica Standard Library (MSL) already [29]. Models for different kinds of medias, heat sources and other components exists. But the user still has to implement the motion of the fluid, often provided from a discretization of the Navier-Stokes equations. Such a model was sent to DS from another company, in the hopes of a GPU acceleration. This underlines the desire for GPU acceleration of Modelica models, specifically that of discretized partial differential equations (PDEs).

1.2.3 Contact Handling in Modelica

Collision handling has been developed before for Modelica, see [13]. In that package, however, the collision handling itself was made with an external package, leaving restrictions to the developers. Only a limited set of shapes where supported. It was then left to the developers to calculate the physical response, based on the data from the external package, and communicate that with the Modelica MultiBody library. In this thesis, the hope is to develop a package that has full control over the collision detection. This package should be more general in the sense of shapes and scene sizes, easier to use, and more efficient. The remainder of this section focuses on introducing the most important tools used to fulfill the requirements of the desired package.

It was decided to use soft(elastic) contact mechanics, both for numerical stiffness reasons and because there exist a validated method for calculating the contact force, if soft collisions are used. This method [15] uses the volume of the overlapping region between two colliding bodies to determine the resulting force. A tool was then needed to calculate the overlapping region between two colliding bodies. For this, the well known, decently fast, Constructive Solid Geometry (CSG) intersection operator was used, which works well on arbitrary shaped objects [14]. By arbitrarily shaped, a restriction is made to bodies with a triangular meshed surface; no parametric or other representation (other than for spheres) is currently supported. To the authors knowledge, CSG has not been used for contact mechanics before. Since CSG is already widely used, complete open source implementations exists. A JavaScript implementation where binary space partitioning (BSP) trees are used, has been done by E. Wallace [16]. The principles of BSP tree-based CSG is described by Segura et al. [14].

One part of a DEM simulation that is suitable for GPU acceleration is the broad phase, see [40]. The broad phase is the part where potential collision pairs of particles are found, to avoid making unnecessary calculations. A broad phase only exists for optimization purposes. In this package, a method based on the algorithm presented by D. Lavrov [12] is used.

1.3 Workflow

Since neither of the authors had any prior experience in GPU programming, the first phase of the project was a comprehensive literature study of GPU architectures and CUDA. This was followed by an investigation of how to incorporate GPUs in Modelica, through Jos Stam's fluid algorithm [38].

Then more complex models were studied, and through hand coding, instructions for auto-generating Modelica functions for CUDA was compiled.

Once the instructions were finished, and handed to DS, work began on the contact handling package. This continued until DS had time to implement a prototype for the automatic GPU code generation. From this point on, the work continued in parallel on the two tasks.

1.4 Report Outline

This report covers two distinct ways of incorporating GPUs into Modelica. Because of this, the report is divided into four parts to facilitate easier reading. This first part covers introduction and general theory used in both the auto-generation and the contact handling package.

Part two and three pertain to the auto-generation of Modelica functions and the contact handling package respectively. For part two, the first chapter describes the process of arriving at an auto-generation scheme and the how the current auto-generation prototype works. The tests used to evaluate the auto-generation prototype are also presented in that chapter. Part three contains two chapters corresponding to our work on the contact handling package, the first with a description of the package itself and the theory behind the different parts, and the second containing the models we created to test the package. The chapters corresponding to our work (3, 6 and 7) also contains a few results, that are considered necessary for going further in the explanations. Both parts are concluded with **Results** and **Discussion** chapters, presenting the results of the mentioned tests and analyzing them.

The last part is a summary of the report along with final thoughts and conclusion that are related to the entire project and to usage of GPUs for Modelica.

1.5 Contributions

As the thesis is clearly divided into two different parts, it is natural to list our contributions according to what part they belong to.

1.5.1 Automatic GPU Code Generation for Modelica Functions

The major contribution in this part, is that we have analyzed the output of the Dymola compiler and given suggestions on how this should change in order to generate parallel GPU code. It should be emphasized that staff at DS made the changes to the compiler. We also suggested and manually tested the ideas of inline integration,

which was later included in the compiler by staff at DS. We performed all tests and analysis of the prototype.

1.5.2 Contact Handling in Modelica

The main contribution in this part was that we put together tools from different areas into a complete package. Most of them were redesigned and optimized in order fit together and perform well. However, some tools and algorithms we had to develop ourselves. We list those contributions below:

- Parallelization of some parts (step 1 and 3 in section 6.3.2) of the broad phase
- Parallel version of BSP tree traversal in the narrow phase
- Multicore attempts of BSP tree traversal in the narrow phase
- The intersection splitting algorithm
- The rotational damping law
- Additional debug window

All tests were designed, performed and analyzed by us.

1.6 Author Specific Work

The work of this thesis has to a very large extent been carried out in cooperation between the authors, and it is therefore hard to attribute most of the work to a specific author. However, the parts where one author has contributed more than the other, are listed below:

- Axel:
  - Optimizations of the broad phase
  - Original implementation of CSG intersection for the narrow phase
  - Multicore attempts of the BSP tree traversal
  - The intersection splitting algorithm
  - The additional debug window
  - Designed the Tippe Top model
- Vilhelm:
  - Original implementation of the broad phase
  - Reduced the CSG intersection algorithm
  - Parallel version of the BSP tree traversal
  - Implemented the contact response
  - Designed the Bucket Digging model

- Added CSG operations for Modelica model setup

The auto-generation part of the thesis could not be split up into parts where one author contributed more than the other, since most of the work consisted of discussions.

Chapter 2

Theory

2.1 Modelica

In this section, a brief introduction to the Modelica programming language is presented, focusing on the properties that are needed to understand the thesis. A full specification is given in the Modelica language specification [30].

Modelica is an object-oriented programming language invented by Hilding Elmqvist in 1996, based on his earlier PhD studies [3]. Since 2000, the language and its standard library is developed and maintained by the non-profit Modelica Association [39].

Modelica is primarily meant for physical modeling, and it is based on symbolic handling of equations. One of the most important differences between Modelica and usual programming languages is that equality in an expression means *equality* between the left and right hand sides, not an assignment to the left hand side variable.

A Modelica program is built up by classes, or *models*, usually consisting of elements (variables, corresponding to attributes/member variables in Java/C++) and `equation` sections. A class often represents a specific physical component. Variables can be of the built-in "primitive" data types `Boolean`, `Integer`, `Real`(double) and `String`, instances of classes, or arrays of any of those. If a variable is not specified as parameter or constant, it is assumed to depend on time. Equations may contain constants, time dependent variables, or the derivative of time dependent variables. The `der()` operator gives the time derivative of its argument.

A special type of classes are the connector classes, typically containing elements and no equations. Objects that should interact with other objects need connectors as elements. A connection can then be established, using two connectors of the same type. A connection basically means a constraint expressed as a new set of equations to the system. Different classes can have the same type of connectors, making it possible, often through a number of connections, for interaction between different physical domains. This is the reason for Modelica's widely known multi-physical

modeling capabilities.

Modelica models can often be edited in a graphical environment, called schematic or diagram editing. Objects are then represented by symbols, which the user can drag and drop, and draw connections with the cursor, in an intuitive, user-friendly way.

To simulate a Modelica model, a Modelica compiler is needed, which is often combined with software to edit the code and visualize the results. One such tool is Dymola, which will be explained in section 2.2. It is not defined how to simulate a model in the Modelica specification. In most compilers, however, the Modelica code is translated into C code before it is compiled to machine code. But there is an important step in the translation process called *flattening*, resulting in *flattened* Modelica code. The flattening process is basically a transformation and expansion of existing equations in the objects, into a system of differential algebraic equations (DAEs). Index reduction and other complex processes may be needed before the system can be solved by a numerical integration method. Models are only solved and integrated in time, as ordinary differential equation (ODEs). When solving PDEs, spatial (or other) discretizations have to be carried out by the user.

In an equation section, the different equations are not computed in any specific order. In the flattening process, equations may be reordered and transformed in any way needed, to make solving possible. One should only think of equations as the mathematical equations they represent, i.e as equalities that should hold at any point in time (within the simulation interval). When implementing certain algorithms, however, a well defined order may be required. For those cases , `algorithm` sections are used. In an `algorithm` section, assignments are performed with the `:=` operator, and operations are executed in the expected order.

If one wishes to use external calculations, external functions can be used to communicate with e.g. C code. If internal memory is used between function calls, a reference to that memory can be kept by the use of external objects, which are instances of a special kind of classes. With external functions and external objects, variables of the "primitive" types can be passed, and arrays of those (together with the array size).

2.2 Dymola

Dymola, or Dynamic Modeling Laboratory[20], is an IDE for Modelica. This section will introduce the parts of Dymola that are relevant to the thesis, which means it will center around how Dymola compiles Modelica code, and end with a brief overview of integration methods and the specific ones used in the thesis.

2.2.1 Compilation

Since Dymola is not an open-source software, the exact details of how compilations are done are not available. Instead a description of the compilation "flow", and important results will be described.

When Dymola compiles a Modelica model (`.mo` file), it first translates the model into C code (`.c` file). During this translation process, the model gets flattened, and

the equations get manipulated symbolically. Therefore the resulting `.c` file may or may not contain assignments corresponding to the equations stated in the `.mo` file. Arrays may also change during this process, and therefore the indexing in the `.mo` file may not correspond to the `.c` file.

After the translation, the C code will correspond to the equations in the `.mo` file, but only once the entire `.c` file has been run. During a time step, the state of the system of equations is unknown. This means that one cannot identify a translated part of the `.mo` file, since even if one can identify a certain equation, the state of other equations is unknown. This makes direct editing of the translated C code very difficult.

Once the `.mo` file has been translated, the `.c` file is linked with a numerical pre-written solver, into an executable that can be run with Dymola. The `.c` file is then used as input to the solver for finding the next time state. During simulation, this solver will iteratively integrate the model in time, until the simulation time is reached. Tools for visualization of simulation results are also included in Dymola.

2.2.2 Integration Methods

The system of equations solved looks in their most general (implicit) form, as follows:

$$\mathbf{F}(\mathbf{x}, \dot{\mathbf{x}}, t) = \mathbf{0}$$

where $\mathbf{F}$ is a vector valued function of size n_{dof}, $\mathbf{x}$ contains the n_{dof} states, $\dot{\mathbf{x}}$ its derivatives with respect to time, and t is time. *dof* refers to *degrees of freedom*, and n_{dof} is often called the number of degrees of freedom of the system.

However, most solvers need a system in the following, explicit format:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$$

This format can then be inserted into an integration scheme defined by a certain solver. Solvers are often divided into explicit and implicit ones, which is not to be confused with explicit and implicit systems of equations. An explicit solver is explicit in the sense that the next state can be expressed in terms of earlier states, while an implicit solver requires a solution of a system of equations. This difference is easiest seen by looking at the two simplest solvers, often called the explicit and the implicit Euler methods. In the explicit Euler method, the following approximation is made:

$$\dot{\mathbf{x}}_n \approx \frac{\mathbf{x}_{n+1} - \mathbf{x}_n}{dt}$$

where $\mathbf{x}_i$ is the state at step i, and dt is the step size. $n+1$ is the next, unknown state, and n is the last known state. This leads to the following scheme

$$\mathbf{x}_{n+1} = \mathbf{x}_n + dt\dot{\mathbf{x}}_n = \mathbf{x}_n + dt\mathbf{f}(\mathbf{x}_n, t)$$

This scheme is obviously explicit, and the function $\mathbf{f}$ only needs to be evaluated once per time step. How computationally heavy an integration method is, is often measured in the number of function evaluations it requires. In the implicit Euler method, the approximation is slightly different:

$$\dot{\mathbf{x}}_{n+1} \approx \frac{\mathbf{x}_{n+1} - \mathbf{x}_n}{dt}$$

which leads to

$$\mathbf{x}_{n+1} = \mathbf{x}_n + dt\mathbf{f}(\mathbf{x}_{n+1}, t)$$

This system of equations has to be solved for $\mathbf{x}_{n+1}$. If $\mathbf{f}$ is non linear in $\mathbf{x}$, which is most often the case, an iterative method is needed. A commonly used method is Newtons method, which is based on the use of Jacobian matrices, which requires multiple function evaluations (at max n_{dof}) to construct. Implicit methods therefore usually require more computation time than explicit methods. To the expense of computation time, they generally deliver better stability and smaller errors. For *stiff* problems, implicit methods sometimes are required, since the explicit methods get unstable solutions. We will not go into what stiffness actually means in numerical situations, and it is here sufficient to know that it is related to large changes in $\mathbf{f}$. Discontinuities are the extreme case of high stiffness, and implicit methods usually can not get past such points, since they will have problem with convergence in the process of solving the non-linear system of equations. Explicit methods with fixed step size, however, might accidentally "step over" the really stiff parts. Since the forces often get close to discontinuous in collision handling, explicit methods are sometimes of better use.

Some more advanced integration methods often use adaptive step size. That means that the step size is adapted to how stiff the problem currently is. It tries to keep the step size as large as possible, while keeping the error under a chosen tolerance. In collision handling, that means that (at least for few objects) the step size can be large between collisions, and small during collisions.

A short description of the methods most commonly used in this thesis:

- Explicit Euler - explained above.
- Fixed step size, explicit Runge-Kutta methods - More advanced explicit method. Exists with different *orders*, where a higher order means higher order of convergence, and smaller errors. A higher order requires more function evaluations per time step. Those algorithms are in this report often referred to as RkfixX, where X should be replaced by the order.
- Adaptive step size, explicit Runge-Kutta methods - Same as above, but with adaptive step size. Here, a higher order method is used to estimate the error. Referred to as CerkXY, where X should be replaced by the order, and Y=X+1 is the order used for error estimation.
- DASSL - Very advanced, implicit, variable step size, variable order method.

In this thesis, the integration has with only one exception been made with Dymola's integrators.

2.3 Graphics Processing Unit

This section serves as an introduction to graphics processing units (GPUs), including some of the fundamentals, GPU architectures and performance considerations. Facts that are considered important and relevant for this thesis are presented. For a

more thorough introduction of GPUs, see e.g. "*Programming massively parallel processors*"[37].

A GPU is, as the acronym suggests, a processor designed for graphics. Although most GPUs today are capable of more than graphics calculations, their design is still influenced and favors certain types of algorithms. The reliance on very specific types of problems have allowed GPUs to become very fast at processing data, as long as the algorithm is correctly designed.

The most fundamental difference between central processing units (CPUs) and GPUs, is that the CPUs are general purpose processors, designed to perform well on sequential code, whereas a GPU is designed to process massively parallel tasks, only. The large cache memories and control logics of a CPU are not provided for the cores of a GPU. Instead, the manufacturers focus on putting as many cores as possible on the chip, letting the threads share cache and control logic between them. As a result, a GPU today may have thousands of cores. This gives most modern GPUs more raw processing power than modern CPUs. As of 2012, the peak double precision performance gap between most GPUs and CPUs was about a factor 10 [37].

This performance is possible, because GPUs only have to work in a SIMD (Single Instruction, Multiple Data) fashion. That means that, when feeding the GPU a task, that task is the same for every thread. The only difference is which data the task is to be performed on. The SIMD processors are designed to maximize the throughput of a large number of threads, which means that they are designed to process a large amount of threads as fast as possible, even if each single thread is executed slowly. This is an important distinction from a CPU, which strives to maximize the execution speed of a single thread. It also means that there are distinct types of problems where GPUs can perform much better than CPUs, and vice versa. Problems where there are large amounts of identical computations to be done on large data sets are generally done much faster on a GPU, and long complex tasks is usually faster on a CPU.

2.3.1 GPU Architecture

Since NVIDIA[33] GPUs were used in this thesis, the conceptual architecture of NVIDIA GPUs will be described in this section. For further details, see [23].

A GPU consists of several so called streaming multiprocessors (SMs). Each of these SMs has control logic for thread scheduling, memory, memory load/store units (LD/ST), special function units (SFU), and a number of streaming processors (SPs), see figure 2.1. Almost all GPUs also feature a dynamic random access memory (DRAM) separate from the CPU's DRAM.

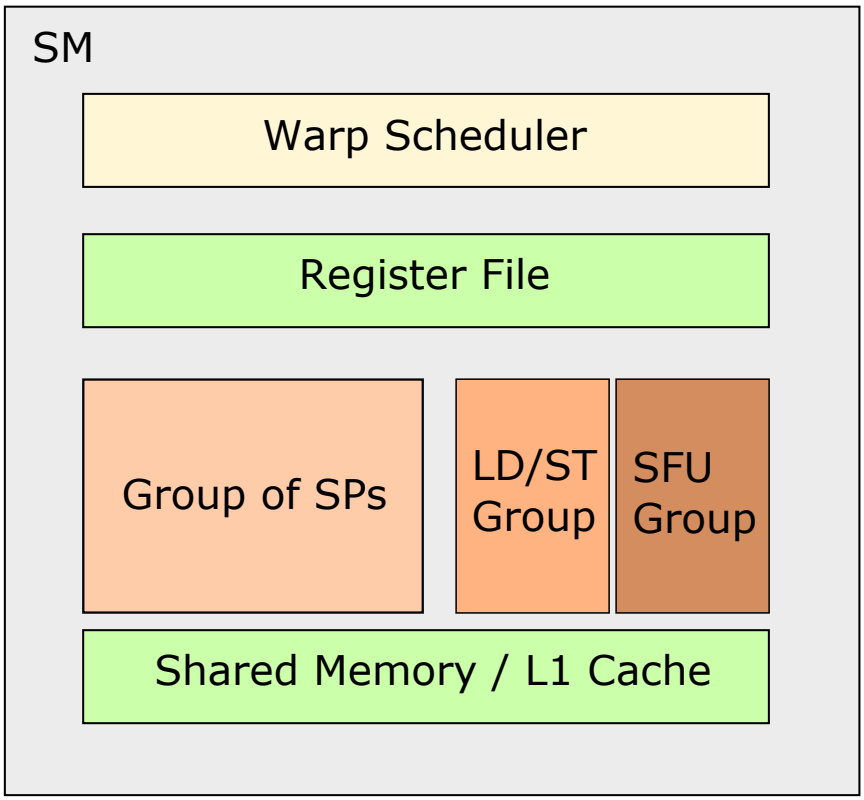


Figure 2.1: Conceptual image of an NVIDIA GPU

The memory is divided into register and cache memory, which will be explained in 2.5. The SPs each consist of an arithmetic logic unit (ALU) and a floating point unit.

In all current NVIDIA GPUs, instructions are issued to a group of threads, called *warps*. This group of threads, which usually numbers 32, has to share the same set of execution instructions. An SM has one or more warp schedulers which selects an available warp and passes one instruction from that warp to either the SPs, the load/store units or the special function units. A warp is available if it is loaded onto the SM, not currently executing and is not waiting for a long latency operation. A long latency operation is most often when memory has to be read from the DRAM memory.

Special notice should be paid to the size of the L1 cache, which is approximately 64 kB. Since this cache is for all threads currently loaded onto the SM, the cache for each thread is very limited. To give an example, the graphics card used for most of this thesis is the NVIDIA Quadro K2100M, which also has 64 kB of L1 cache memory. The K2100M can have 2048 threads loaded onto an SM, which means 64 kB/2048 = 32 bytes per thread, assuming no memory is used to share variables between threads.

2.3.2 Performance Considerations

When programming the GPU, its special architecture requires some special considerations when writing the programs. One of the major ones is directly linked to the

small cache size, while another is the SIMD processing model. The theory presented in this section is mainly based on [37], ch. 3-6.

Memory Latency

Because of the limited cache sizes of most GPUs, a lot of memory has to be fetched from the DRAM. On a GPU, this is a very slow operation, requiring hundreds of clock cycles. To avoid this problem, the GPU is designed to have many times more threads loaded into registers than can concurrently execute. This allows the warp scheduler to avoid some of the performance drawbacks of having such slow memory access. Whenever a warp is idle because of a long latency operation, it is marked as unavailable to the warp scheduler. The scheduler then selects another warp for execution while the first one waits for the operation to finish. Once the long latency operation is finished, the warp is marked as available again and can be selected by the warp scheduler for continued execution. This is to allow for more efficient thread scheduling, by letting the SP do other work while a long latency operation is performed, effectively hiding some of the latency.

When loading an SM with warps, it's preferable to load it with a large number of warps. This increases the chance that the warp scheduler can find available warps for each clock cycle. If the number of warps is too small, the scheduler might not find any available warps, since all warps could for example be waiting for long latency operations. As mentioned earlier, the K2100M chips can load up to 2048 threads per SM. They have three SMs, and can thus accommodate a maximum of 6144 threads. If 6144 threads would be loaded to the SMs during a full kernel execution, that kernel launch is said to have 100 % *occupancy*. This is for numerous reasons very unlikely to happen, one of them being the restricted amount of L1 cache per SM.

The thread scheduling hides some, but not all of the long latency operations. To fully utilize a GPU, you'll want to let the cores work with floating point operations as much as possible. And even if today's chips have a bandwidth to the DRAM of more than 200 GB/s, DRAM accesses have to be considered for good performance. A good way to analyze this is to consider the number of floating point operations per global memory access for a thread, often called the CGMA (Compute to Global Memory Access) ratio. This ratio should be kept as high as possible. If it is too low, there is no way to keep the cores busy, independently of how the threads are scheduled. That is because there is a limited amount of data that can be delivered to the cores per time unit, and if that rate is lower than the rate at which the operations on the data can be executed, the data transferring has become a limiting factor.

The last aspect to bring up in GPU memory performance, is the process of transferring data between the CPU DRAM and the GPU DRAM. It is the main bottleneck of a GPU. The transfer speed is relatively low, and the transfer is often subjected to a lot of overhead work. This implies that there is no point to send work to the GPU, unless there is a lot of it. So if it is possible to avoid memory transfers of this kind, e.g. by not repeatedly transferring constant data, it should be done.

Instruction Divergence

At warp level, no divisions between threads are made. Because of the SIMD design of the GPU, this means that if one thread in a warp needs to execute an instruction, all other threads in that warp will have to wait for that instruction to finish. This can have large impacts on performance. Consider the case where an if-else statement is executed by some threads. With no division made in the instructions for a warp, this means any warp which has atleast one thread executing each of the if-else branches, would have to take both paths. First the if-part is executed by the threads that fulfill the if-condition, while the rest of the threads wait. Then the else-part is executed by the other threads while the if-threads wait, effectively doubling the execution time for the warp, assuming both branches are equally computational heavy. Even for short branches, for example single line if-statements, this can have large execution costs if that one line is a long latency operation. We refer to the threads of a warp that is currently executing (not idle because of e.g. not fulfilling an if-condition) as *active* threads.

Since SIMD only comes in at the warp level of an execution, it is possible to have code branching, if the programmer knows how warps will be issued to the GPU. If careful planning is not done however, a lot of warps can by mistake contain threads going into both branches. Since it can be hard to determine how warps will be arranged, the general rule should be to avoid code divergence if possible.

2.4 Amdahl's law

When optimizing a part of a program, the speed-up of the program is limited by the fraction of the total execution time, that the optimized section is responsible for. For example, if optimization is made on a part of a program that originally takes half of the total execution time, the total speed-up can never be more than a factor of two, since the other, non-optimized part, always will be left unchanged. A general, mathematical description of this fact was developed by Gene Amdahl in 1967, known as Amdahl's law. The law was originally meant for finding theoretical limits of improvement, when parallelizing software. The definition is as follows: Let f be the fraction (in execution time) of the program that is parallelized, and n the number of cores that is used for parallel execution. Then S_{tot}, the theoretical maximal total speed-up factor, is given by

$$S_{tot} = \frac{1}{1 - f + \frac{f}{n}}$$

The law is very intuitive, and it is easy to apply on the example mentioned above. In Amdahls's law it is assumed that perfect parallelization of a problem equals a speed-up inverse proportional to the number of cores working. This may describe the practical limit for optimizing e.g. multi-core CPUs or similar fairly well, but it does not describe the possible speed-ups for using GPUs instead of CPUs very well. To begin with, the cores of CPUs are generally significantly faster than cores of GPUs, and slow transfers of memory are necessary for a parallelization to take place. There are so many other factors limiting possible speed-ups, that it is practically

impossible to even get close to the inverse proportional factor that Ahmdahl's law assumes. In a modern version of the law, n is replaced by S, the speed-up of the optimized part:

$$S_{tot} = \frac{1}{1 - f + \frac{f}{S}}$$

Even if the law is simple and widely known, it is often forgotten. Or rather, the point it shows is forgotten. Programmers often make huge efforts to optimize code, sometimes without a sufficient analysis of the possible gain the optimization might give.

The whole idea that the law describes is highly relevant in the context of this thesis, as every parallization is made on a part of the program and this law helps determine what parts to optimize. Due to circumstances around Modelica and Dymola, there are many parts that are not parallelizable at the moment.

For more on Amdahl's law, see e.g. Hill et al. [5].

2.5 CUDA

When starting this thesis, the decision between which GPU programming language to use stood between OpenCL (Open Computing Language) and CUDA C/C++. In terms of performance, there seems to be no advantage in choosing one language over the other, and since the thesis is not about investigating the maximum performance of GPU acceleration in Modelica, performance was not considered that important. OpenCL had the advantage of being cross-platform and compatible with more hardware, while CUDA is locked to specific NVIDIA hardware. However, since CUDA seemed easier to learn, neither of the authors of this thesis had any prior experience with GPU programming, and because the used computers for the thesis were equipped with NVIDIA GPUs, the decision was made to use CUDA.

Compute Unified Device Architecture, or CUDA, is a parallel computing platform and programming model developed by NVIDIA [17]. In this thesis CUDA C/C++ was used, which is an extension to the C/C++ languages. CUDA C/C++ will henceforth be referred to as CUDA. By being an extension to C/C++, CUDA allows for easy CPU programming, while containing some additional keywords and API functions for handling memory transfers and work on/to the GPU. CUDA is also designed with scalability in mind, that is to say, a CUDA program written for one GPU will work on subsequent GPUs with more cores etc. This is possible since explicit thread handling is not done in CUDA, which means changes in available SMs etc wont prevent execution. Another necessary part in the scalability of CUDA is the fact that CUDA uses just-in-time compilation. This means that CUDA code is compiled into something called PTX code. The PTX code is compiled at application launch for the specific GPU and driver configuration it is to run on. One should keep in mind though, that just because CUDA makes code scalable does not necessarily mean it will run as effectively on other GPUs as on the one the program was written for [19].

CUDA is compiled by a specific compiler designed by NVIDIA, called NVCC (NVIDIA CUDA Compiler). NVCC compiles CUDA files which have the ending `.cu`, by separating the standard C/C++ code from the CUDA code. Once separated

NVCC handles the compilation of the CUDA part by itself and passes the rest to another C/C++ compiler. Once finished, the two parts can be linked back together.

2.5.1 The CUDA Programming Model

This section will detail the basics of the CUDA programming model, as well as some of the syntax and ideas of how CPU-GPU programming is done using CUDA. For a complete introduction, see [37] or [19].

Kernels

The basic idea of CUDA builds on something called *kernels*. A kernel consists of a C/C++ function which is defined by the keyword `__global__`. Inside a kernel, the code works a little differently than in ordinary C/C++ code. The kernel is executed by a number of CUDA threads, specified by the programmer. Each CUDA thread that runs this kernel is assigned an id, used to identify the thread and let the programmer decide what the thread should do. Although CUDA itself allows the programmer to specify individual work for each thread using it's thread id, this is generally a bad idea. Because of the SIMD design of GPUs, instruction divergence should be avoided. Instead, the general praxis is to have the same instructions for each thread, but allow the data they are working on to differ with the thread's id. The code below illustrates how this is done.

```
__global__
void exampleKernel(int *array){
   size_t id = threadIdx.x;
   array[id] = array[id] + 1;
}
```

A kernel execution is often referred to as a kernel *launch*.

There are some additional requirements on kernels. They have to have return type `void`, and only a (growing) subset of C/C++ is supported. Only functions with the special keyword `__device__`, designed for GPU execution, may be called from the kernel.

A kernel is launched from the C/C++ code as a function with a special syntax, as shown below:

```
int main(){
   exampleKernel<<<grid, block>>>(pointerToMemoryOnGPU);
}
```

Note the specific <<<>>> signs at the function call, meant to take arguments for how the kernel should be run. The minimal arguments required are the `grid` and `block` variables, which contains information about how the threads running the kernel should be configured. The function input argument inside the parentheses takes a pointer, as seen in the example kernel further up, which should point to GPU memory.

Memory Transfers

CUDA operates under the assumption that the CUDA threads operate on a physically separate *device* from the CPU, or *host*. Furthermore, the CUDA programming model assumes that both the host and the device both have separate DRAM memories, or at least separate spaces in the DRAM memory. This means that memory spaces available to the host is not accessible from the device, and vice versa.

What this means to the programmer is that he has to explicitly tell the program what data to transfer between the host and device memories. To do this transfer, CUDA contains API functions for memory transfers and memory allocations. The two most important functions are `cudaMemcpy` and `cudaMalloc`. CUDA also has some automatic memory transfers built in. For example, arguments to a kernel will be passed by value, as shallow copies. Thus, pointers or arrays need to point to addresses already on the GPU to be usable in a kernel, and their content transferred by some of the available API functions. Another restriction is the size of the arguments passed to the kernel. The exact limit changes depending on hardware, but it is generally not that large.

Since most kernels are only useful if run using large data sets, this means that the general work-flow of launching a kernel consists of the following steps:

1. Allocate memory on the device
2. Transfer data from host to device
3. Launch the kernel, using as arguments pointers to the allocated device memory
4. Transfer data back from the device
5. Free the device memory

In code, this would equate to:

```
__global__ void kernel(int *array);

int main(){
   int *array, *d_array;
   size_t sizeOfArray = 1000;

   array = (int*) malloc(sizeOfArray*sizeof(int));

   //Initiate array, declare and determine the grid and block
   ...
   //Start CUDA
   //1
   cudaMalloc((void**)&d_array, sizeOfArray*sizeof(int));
   //2
   cudaMemcpy(d_array, array, sizeOfArray*sizeof(int),
       cudaMemcpyHostToDevice);
   //3
   kernel<<<grid, block>>>(d_array);
```

```
    //4
    cudaMemcpy(array, d_array, sizeOfArray*sizeof(int),
        cudaMemcpyDeviceToHost);
    //5
    cudaFree(d_array);
    //End CUDA

    free(array);
}
```

`d_` for device is added to the name of device memory pointers. The kernel arguments `grid` and `block` are explained in the next section.

Grid and Blocks

When CUDA issues work to the GPU, it does so by assigning batches of threads called *blocks* to the SMs. A block is a group of threads that is divided into warps, which are all assigned to and leave an SM at the same time. This gives the block some specific properties. It shares an L1 cache, and the threads within it can be synchronized by the warp scheduler. The shared L1 cache allows for quick sharing of variables, while synchronization allows the programmer to define parts of code that all warps must finish, before any warp continues.

Threads are always assigned to the SMs in full blocks, where the block size is passed in the `block` variable at kernel launch. Since no half-blocks are assigned to the SMs, certain care should be given to what block size is chosen. This is to assure that the warp scheduler has the maximum amount of warps to choose from, increasing the possibility of hiding long latency operations. Recall that K2100M can load 2048 threads per SM. If the block size is set to 768, only two blocks and a total of 1536 threads can be loaded to each SM, effectively decreasing the occupancy.

One block is usually not enough for the amount of work to be done on the GPU. Therefore multiple blocks is ordered into a *grid* of blocks, where each block has a unique id. The thread ids are only unique within blocks, but together with the block id, a global id, that is unique over the grid, can be assigned for each thread. The grid is specified at kernel launch in the variable `grid`. This ensures that the programmer can work with a much larger data set easily, by first dividing it into blocks and then into threads. An example below:

```
__global__
void exampleKernel(int *array, size_t n){
    size_t id = blockIdx.x*blockDim.x + threadIdx.x;

    if(id < n)
        array[id] = array[id] + 1;
}
```

This example gives each thread a unique index in the array to work on, while not leaving any gaps. `n` specifies the size of the array to work on. As the blocks are launched whole, it is likely that the last block will contain threads reaching beyond

the array size, explaining the need of the if-statement.

One thing to keep in mind when using the grid is that the blocks of the grid are not necessarily assigned to the same SM. For this reason threads from different blocks cannot access variables shared in the L1 cache, and cannot be synchronized.

When launching a kernel, the size of the grid and blocks are put as arguments inside the <<<>>> mark, following the function name. They are specified by the CUDA struct `dim3`, containing the three `uint` variables `x`, `y` and `z`. This is to simplify kernel launches of multidimensional problems. In code:

```
size_t gridSize = 10, blockSize = 10;

//Will launch 10 blocks, with 10 threads each. The grid and block will be
    implicitly constructed from the size_t arguments with y=z=1.
exampleKernel<<<gridSize, blockSize>>>(pointerToMemoryOnDevice);

dim3 grid{10,10,1};
dim3 block{10,10,10};

//Will launch 10x10x1 blocks each consisting of 1000 threads configured
    as 10x10x10
exampleKernel<<<grid, block>>>(pointerToMemoryOnDevice);
```

Memory Management

Since fetching data from the DRAM, or global memory as it is called in CUDA, is so slow, the programmer is given some control over which memory to use for storage. The user has the ability to store data in what CUDA calls *shared*, *constant*, *register* or *global* memory. Each of these memories has different access speeds, and different thread visibility.

The access speed is generally as follows, fastest to slowest: register, shared, constant and global memory. The visibility of the different memories is visualized in figure 2.2.

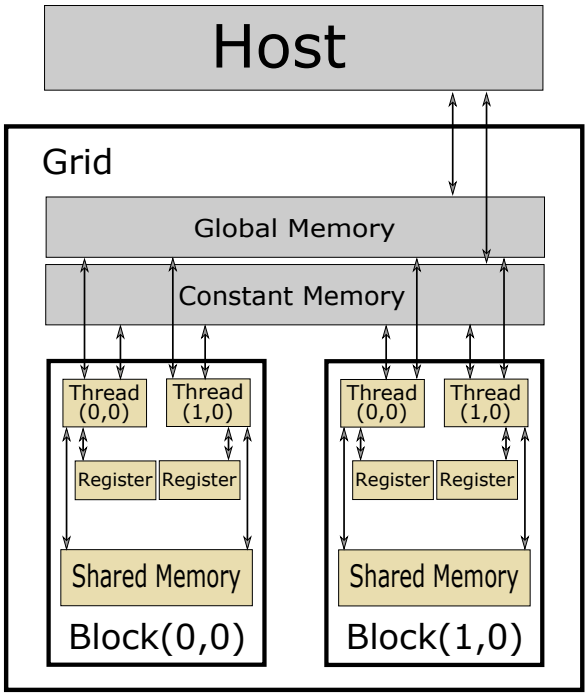


Figure 2.2: Conceptual image of memory visibility

As can be seen in the figure, the register memory is the only memory that is thread specific. Since this is also the fastest memory it's often used for local variables which varies for each thread, or are accessed frequently. The shared memory is visible to all threads in a block, because it resides in the L1 cache (figure 2.1). The constant memory has global visibility and is accessible by all threads. However, the constant memory is a read-only memory during a kernel execution. Global memory also has visibility for all threads, with no further restrictions. It is however significantly slower than any other GPU memory, since it resides in the DRAM.

Since the memories have different access speeds and visibility, the programmer is given some control over where memory is stored. To store variables in the global memory, memory is allocated using the CUDA analogue of `malloc`, `cudaMalloc`. GPU memory can in general only be accessed from a kernel. The shared and constant memory types can be declared by using their respective keywords at declarations:

```
__shared__ int sharedInt;
__constant__ int constantInt;
```

Shared variables are declared inside kernels, while constant memory is declared on file scope.

Last comes the register memory. The programmer cannot explicitly tell CUDA to store something in the register memory, but it is generally the memory used for automatic variables, as long as there is space for it. If an automatic variable is not placed in register memory it is placed in global memory, typically decreasing performance.

The amount of data needed in shared and register memory has a large effect on the number of blocks that can be loaded onto an SM at one time (occupancy).

2.5.2 CUDA Performance Considerations

Performance of code in CUDA is dependent on a large number of factors, some of which are tightly connected with the specific hardware running the code. For this reason this chapter will only bring up some of the factors impacting performance, especially those with large performance impacts which solutions are general for most or all hardware. The reason for this is speed-up solutions catering to specific hardware are not relevant for either automatically generated code, or a library that would potentially be distributed to a large number of clients.

Memory Transfer

One of the largest bottlenecks in GPU programming is the transfer of data from the CPU DRAM to the GPU DRAM. This is due to several reasons, one being the memory transfer itself. Most modern GPUs use PCIe (PCI express, see [35]), which has an optimal transfer speed of about 15 GB/s. While this sounds a lot, the real performance is often hindered by large amounts of transfer overhead. In a test[27], the 15 GB/s PCIe had an actual transfer speed of about 2.3 GB/s and 5.4 GB/s, for two different computers.

The transfer speed can be increased by using pinned-memory for the transfer. Most computers today uses page-able memory as a way to improve DRAM. Page-able memory allows usage of more storage than available on the DRAM. The memory can then swap "page" and load a different piece of memory from disk into the DRAM. Pinned or non-page-able memory is memory that cannot be passed down from the DRAM as a page to disk while other memory occupies the DRAM. Since the transfer between GPU and CPU requires the memory to be in the DRAM, the transfer is done in stages, see figure 2.3. First pinned memory is allocated. Then the data is copied from unpinned to pinned memory before it is copied to the GPU, and the pinned memory is freed. By allocating pinned memory directly, all stages except transfer between GPU and CPU are eliminated. The drawback is that allocations in pinned memory are slow and some of the DRAM on the computer is "locked" until it is freed. Locking some of the DRAM may result in a slow-down of the computer until it is freed, so using pinned memory should be done with restraint. Because of the long allocation times, using pinned memory only results in a performance gain if the data transfers are large or many transfers using the same memory are done [11].

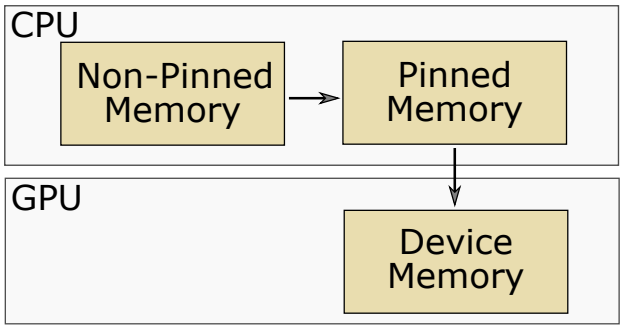


Figure 2.3: Memory transfer from unpinned memory

Another technique to reduce overhead is to batch small memory transfers into one large. Since the overhead remains fairly constant [28], using larger transfers is almost always better.

Memory Accessing

As mentioned, accessing global memory in a kernel is very slow and should be avoided if possible. In some cases this is easy; repetitive usage of some global variable could for instance be saved in an automatic variable, thereby increasing the chance that it is stored in the register memory. Or if the data being read is used by several threads, it can be beneficiary to have one thread load it into the shared memory, so that all threads have faster access.

The loading of variables into shared memory and registers, should be done with care however. The amount of shared and register memory is limited, and using too much of it might have a negative impact on the performance, which is easiest explained by an example. The Quadro K2100M has a maximum of 2048 threads loaded onto an SM, which should share 65 536 bytes of storage. If two blocks with 1024 threads each were loaded onto the SM this would give each thread 32 bytes of shared memory. Now say the programmer changed the program so that each thread used 33 bytes of shared memory. The hardware then responds by reducing the number of threads on the SM to give each thread the 33 bytes it needs. This reduction is done in full blocks, which means the SM would suddenly only load 1024 threads instead of 2048, decreasing occupancy.

To avoid this type of memory overloading, a usual technique is to divide the problem into sub problems, each of which need a fixed amount of shared memory. The kernel can then finish the part currently loaded into shared memory, load a new batch and so on. This is called *tiling*, from 2D cases where it can be visualized as loading tiles of the total data set into shared memory. Tiling is crucial for many applications as it can increase the CGMA ratio significantly.

Memory Coalescing

Because of how DRAM is constructed, multiple, in space consecutive, accesses can be performed at the same time, resulting in a higher transfer speed. This means that when data has to be read from the global memory, care should be taken to ensure the memory addresses read from or written to, are consecutive for the active threads. When CUDA creates warps, the threads will be ordered with `threadIdx.x` as major direction, then `y`, then `z`. For example, warp 0 will consist of threads with `threadIdx.x` 0 to 31, and `threadIdx.y` = `threadIdx.z` = 0, if those threads exists. This is visualized in figure 2.4, where the warps consists of four threads for simplicity.

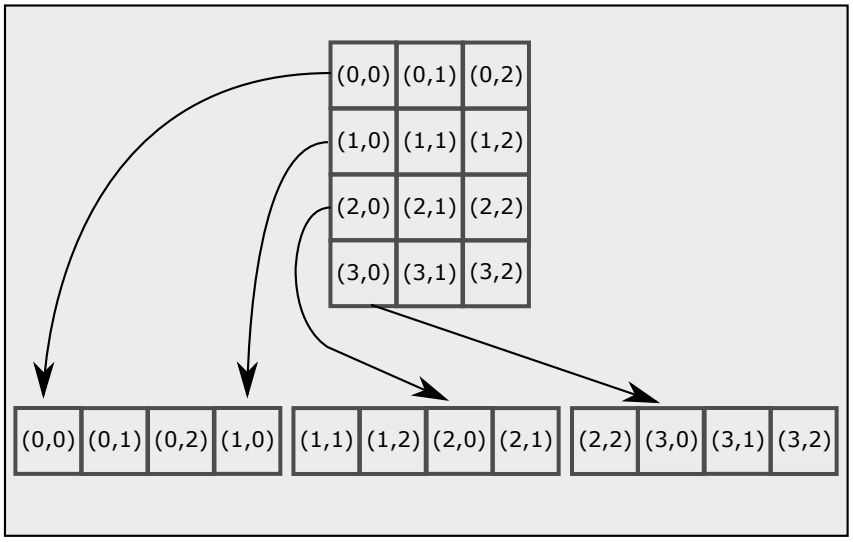


Figure 2.4: The construction of warps from a 2d-thread grid

Instruction Divergence

In section 2.3.1, it was mentioned that if-else statements should be avoided, because they cause instruction divergence. Instruction divergence is not only limited to if-else statements, they are only the most obvious ones. Other instruction divergence sources are for example loops where some condition should be reached, that depends on work done in the loop or on data that is different for each thread. Thread divergence may sometimes be very tricky to outright avoid, and in those cases care should be taken to try to minimize the divergence.

Part II

Automatic GPU Code Generation of Functions

Chapter 3

Approach

Using GPUs for Modelica has as mentioned already been done, but neither DS nor the authors had any knowledge of how to use GPUs for Modelica. This meant that the first step would be testing if and how a GPU could be interactively incorporated in a Modelica model. The next step would be determining how CUDA code for the Modelica models should be formulated. It soon became obvious that this problem would only be solvable by building on DS's already existing auto-generation of C code. If we could determine both how the current translation of Modelica structures are done and how they should be changed to run on the GPU, an auto-generation of CUDA code could be accomplished. In summary, the steps to achieve auto-generation of CUDA code follows below:

1. Determine how GPU code could be interactively incorporated into existing Modelica models.
2. Analyze how functions are currently translated from Modelica to C by Dymola
3. Find a way to reformulate those functions to run on the GPU
4. Try to find optimizations valid for arbitrary models' auto-generated CUDA code

Each of those steps are explained in the following sections, starting from the top. Sections 3.1-3.4 follow the initial workflow of this project (and correspond to steps 1-4 above). Those are followed by section 3.5 that summarizes the conclusions from those sections, applies the new knowledge on a new model, and ends up with parallelization instructions for auto-genereation. Section 3.6 describes the resulting prototype for auto-generation, developed by DS, and section 3.7 presents the tests used to evaluate the prototype.

3.1 Incorporating GPUs in Modelica

To try to incorporate CUDA in Modelica, the decision was made to make a Modelica coupled version of the fluid solver developed by Jos Stam [38], which will be referred

to as Stam's solver, or Stam's algorithm. Stam's solver was chosen as an incorporation test for several reasons. First of all, it served as a simple, real, first example to apply our CUDA knowledge on. It is implemented as a fluid "simulation" program for CPUs, so parallelizing the algorithm was good practice. Secondly, the solver introduced us to CFD as well, which was useful since CFD was one of the areas that would likely benefit from auto-generation of CUDA code. Finally, Stam's algorithm included a window for visualizing the results of the fluid solver, and knowledge of how to incorporate such a window with Dymola would later be necessary for the contact handling package.

3.1.1 Jos Stam's Algorithm

Stam's algorithm is based on solving the incompressible Navier-Stokes equations for velocity and density, by discretizing the equations over a fixed grid of cells. The solution is calculated using a method similar to the ones used in later examples for auto-generating functions as GPU code (see sections 3.2.1, 3.5.1 and 3.7.1). Namely that each cell communicates with nearby cells in order to calculate finite differences, which is used to iteratively update its states. A snapshot from a simulation is shown in figure 3.1.

Figure 3.1: Snapshot from a simulation with Stam's algorithm using a 128×128 grid

As we implemented Stam's algorithm as an external piece of C++ code, we had to make use of Modelica's external object feature, for interactivity. As external C++ code was expected to be required for the contact package, testing the external features with GPUs would be much easier on such a simple problem.

Stam's algorithm was accelerated more than five times, on grid sizes of 256×256 or larger. Most of this speed-up came from parallelization of the Gauss-Seidel relaxation, which is an iterative method for performing matrix inversion. GPU interactivity in Modelica was tested by use of a Modelica model which coupled two separate instances of Stam's fluid simulation. The outflow from one grid was then used as inflow in the other.

As Stam's algorithm was successfully run as a Modelica model with GPU acceleration, it appeared probable that both usage of external objects, as well as acceleration of CFD models, would be possible by using GPUs and Modelica.

3.2 Translated Modelica Structures

A benefit of the approach to rewrite the auto-generated code is that only the output of the Dymola compiler needs to change. This means that most of the underlying structure for identification and equation handling can remain unchanged. In order to analyze how Dymola currently translates Modelica into C, and how it should be translated into CUDA, a CFD model that was sent to DS in hopes of a GPU acceleration was used. The model simulates the airflow in aircraft compartments, and will be referred to as the airflow model. As the model was very complex, and not considered to be important for the reader, only a short introduction is made.

3.2.1 The Airflow Model

The model is used to simulate the flow of air in a domain, defined by the user as a set of cells, each with a range of possible specifications and physical parameters. The model uses the Finite Volume Method (FVM) to discretize the compressible Navier-Stokes equations, and calculate the cell states over time for each cell. The cells contains a series of equations, resulting from the mentioned discretization. Of these equations, six are used to calculate the enthalpy flow from the neighboring cells (one for each direction; east, west, north, south, up and down). These equations were chosen for parallelization, since they were computationally heavy, independent of each other, and contained the communication between cells mentioned in section 3.1.1. It should be emphasized that the parallelization initially only served to help the analysis of how to parallelize auto-generated code, not to achieve a faster model. For that, a parallelization of a larger part of the model would be required.

Because the auto-generated C code is unwieldy to read, and most of the model consist of variables unknown to the reader, conceptual images will be used to explain translation concepts instead of actual code.

Using `algorithm` Sections

When translating the airflow model, the first major problem was the restructuring of equations that was made at translation. By allowing Dymola to restructure as needed, the 6 equations for each cell were reordered and split. Each of the 6 equations for each of the cells could now be split up, working on unrelated data addresses and contain dependencies that had to be run before them in order to get the correct result. This meant that in order to launch them as a kernel, they would all have to be found, all their variables have to be copied individually, and all their dependencies tracked, see figure 3.2.

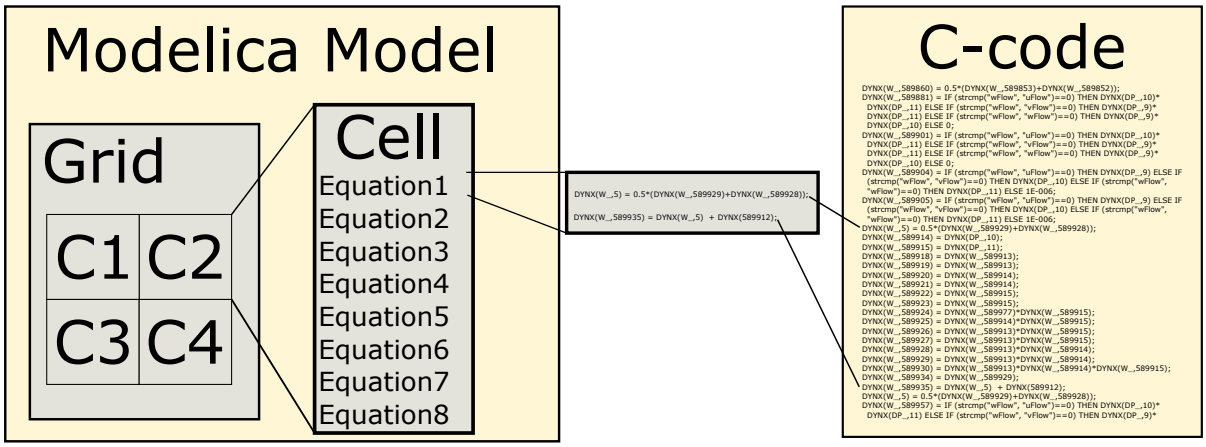


Figure 3.2: Result of translation using equations in each cell

To circumvent this problem, the equations of interest (1-6 in the figures) in the Modelica model were put inside an `algorithm` section with nested for-loops on the scope where all cells are visible, instead of as individual equations inside each cell. Translating the model now would result in nested for-loops over the cells, with the C versions of the Modelica equations intact, see figure 3.3. The Grid in the image refers to a scope where all cells are visible.

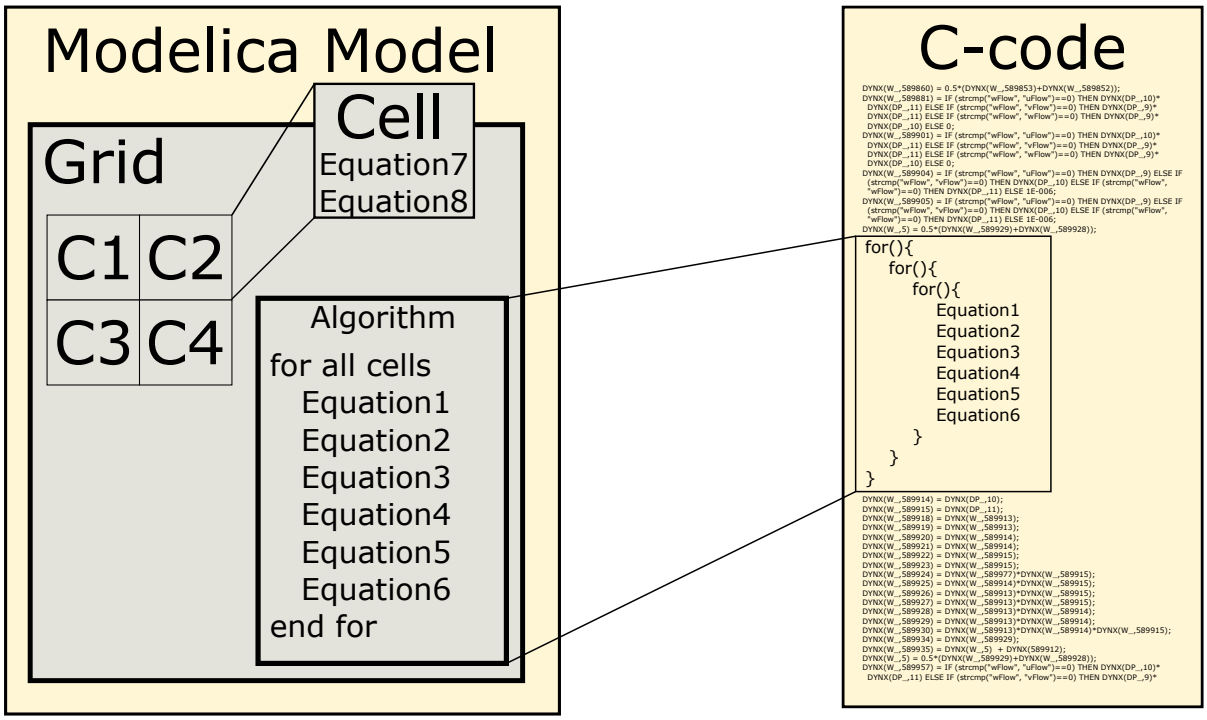


Figure 3.3: Result of translation with an algorithm section for equation 1-6

The nested for-loops in the C code would clearly be easier to parallelize. In the end, the section of code corresponding to the 6 equations in Modelica, got the following (simplified) structure in C:

```
...
for(i = 0; i < max_i; ++i){
   for(j = 0; j < max_j; ++j){
      for(k = 0; k < max_i; ++k){
         struct H_flow answer = calc_H_flow(dyn_arrays, dyn_struct, i, j,
            k);
         set_element(answer.H_flow_east, grid_H_flow_east, i,j,k);
         set_element(answer.H_flow_west, grid_H_flow_west, i,j,k);
         set_element(answer.H_flow_north, grid_H_flow_north, i,j,k);
         set_element(answer.H_flow_south, grid_H_flow_south, i,j,k);
         set_element(answer.H_flow_up, grid_H_flow_up, i,j,k);
         set_element(answer.H_flow_down, H_flow_east_down, i,j,k);
      }
   }
}
...
```

The `dyn_arrays` variable corresponds to arrays (represented by one array here, for simplicity) that are constructed at runtime and contains individual cell data such as pressure, temperature, enthalpy etc., from the previous iteration. The `dyn_struct` comes from the fact that multiple constants and arrays of constants are passed to the Modelica version of the `calc_H_flow` function as a `record`. A record is basically the Modelica version of a struct. The reason that Modelica arrays and records translates to dynamically constructed arrays and structs when written inside an `algorithm` for-loop, is that the variables may be spread out in memory. So in order to access them by indices in a for-loop, Dymola has to *gather* them temporarily so that they are placed consecutively in memory, in a way that allows for accessing using a multidimensional index. This gathering is done using a series of pre-written Dymola functions. Each such function has a pointer to a large chunk of memory, allocated when the simulation starts, and keeps track of how much of the memory is used. That way, data can temporarily be pulled together into one memory space in a way that reconstructs their original multidimensional structure (from Modelica), accessed by index in the for-loop, and then cleared.

3.2.2 Copying Data to and from the Device

Transferring the data in `dym_arrays` from host to device memory is easily done by mapping each array to one `cudaMalloc` and one `cudaMemcpy` call, after it has been gathered on the host. Allocation of memory to store the answers in, corresponding to the `grid_H_flow_east`, `grid_H_flow_west` etc. variables, was done in a similar fashion.

The struct, however, presents more of a problem, since it contains pointers to different sets of data. The reason is that, as mentioned, only a shallow copy is made automatically. To get a deep copy, with pointers to device memory locations, this has to be done manually, with potentially very many calls to `cudaMalloc` and `cudaMemcpy`. It can be done a bit more efficiently by only allocating one large memory space for all arrays within the struct and for the struct itself, and manually

arrange the pointers. A more convenient solution, however, was achieved by rewriting the airflow model in Modelica, passing the content of the record in individual variables instead of as a record.

In Modelica, six arrays were declared specifically to store the answers (`grid_H_flow_east` etc.) of the `algorithm` section. In an `equation` section, those were equaled to the actual variables for enthalpy over the grid. That way, those arrays would be stored consecutively in memory on the host, allowing an easy copy back operation from the GPU after the kernel was done. The arrays are consecutive in memory because of the way Dymola manages memory for variables that are not present in any `der()` operations.

The actual variables for enthalpy are still spread out in memory. But by the above mentioned equality in an `equation` section, the distribution of the answer to the actual enthalpy variables is done outside the `algorithm` section in the C code as well. The gathering of `dyn_arrays` is still done in the `algorithm` section, however. This makes parallelization easier, and turns out, again due to how Dymola manages memory, to result in one less gathering operation per iteration.

3.2.3 The Kernel

To turn the body of the nested for-loops into a working kernel, all of the Modelica functions used in the calculations had to be pre-fixed with the `__device__` keyword. Also, the indexed accesses are done by a function taking the indices as parameters, transforming them to a one dimensional index. This function uses ellipsis arguments in order to allow for handling arrays of different sizes. Since ellipsis arguments is not supported in `__device__` or `__host__` functions in CUDA, the access function had to be redone. The access function was replaced with a macro taking the thread indices and array dimensions as input. The macro looked like this, for three dimensions:

```
#define IX(i,j,k, max_j, max_k) (max_k * (max_j*(i-1) + (j-1)) + (k-1))
```

The downside with using an index macro like the one above is that the dimensional flexibility is lost. It can handle different sizes of the same dimensions, but it cannot be used for arrays with a different number of dimensions. However, since the Modelica model contains the array dimensions, the compiler can use that to generate an index macro for the correct number of dimensions.

By assigning each thread unique indices in the standard fashion, a working kernel was completed.

3.3 Optimizing the Airflow Model Parallelization

The CUDA parallelization of the airflow model presented above is slow. In order for it to become meaningful as a base for auto-generation, an initial speed-up was required. This section explains some different optimizations made to the code, and their limitations. To gain an overview of the state before optimization, pseudo code for the memory transfer and kernel is presented below.

```
...
//Host wrapper function
gather the arrays in dyn_arrays and dyn_struct

for each array in dyn_arrays:
   allocate memory on device
   transfer data to the allocated memory

allocate memory on device for dyn_struct
transfer dyn_struct to allocated memory

allocate answer memory on the device

launch kernel, passing pointers to the arrays of dyn_arrays, dyn_struct
    and answer as arguments

copy answer back to host

free all device arrays of dyn_arrays
free device dyn_struct memory
free device answer memory
...

//Kernel
kernel(dyn_arrays, dyn_struct, answer)
   assign unique indices to thread

   if(indices within simulation grid)
      original for-loop work, with accesses using the IX() macro
      save calculations to answer
```

3.3.1 Allocations and Memory Transfers

By profiling the simulations, it was found that most of the time in the unoptimized CUDA code was spent in allocations and memory transfer operations and not in the actual kernel. This can easily be remedied by only allocating memory on the device the first time the kernel should be launched. This memory can then be reused in every time step, and freed at the end of the simulation. Another optimization is to make one large allocation for all arrays in `dyn_arrays`, and keep track of where each array exists in this data chunk, allowing for fewer transfers between host and device, reducing the transfer overhead. Since all array data needs to be copied in the gathering anyway, having the data in one array requires no extra copying. Obviously this requires the kernel to keep track of what part of the single piece of memory each array occupies. This can be done by keeping track of the order in which the arrays were put in the large array, and the sizes.

Another possible improvement was that large amounts of the data used for the kernel were constants. Things like reference pressure and cell coordinates only need

to be copied initially, reducing the amount of data to be copied for subsequent launches.

Lastly, by allocating pinned memory on the host initially, to gather the arrays of `dyn_arrays` in, copy overhead can be further reduced. Even if a pinned memory allocation is time expensive, the allocated space will be reused for thousands of transfers from host to device, making it beneficiary in the long run.

3.3.2 Kernel Optimizations

An obvious optimization for the kernel was the number of global memory accesses. Since the auto-generated code was for CPUs, each variable from the code was accessed by address from global memory. While this works well on a CPU with its large cache memories, it is likely to result in cache misses on a GPU. By changing the kernel to store the multiple accessed data as automatic variables at their first use, a lot of global memory accesses can be avoided.

As the airflow model is using the FVM method, each cell will use data from the neighboring cells. In a case such as this, common practice with CUDA is to use shared memory and let the threads cooperate to read the data from memory. While this practice might give large performance boosts for problems similar to the airflow model, we decided not to implement it. By reading all the used arrays into shared memory, then obviously the amount of shared memory necessary to run the problem gets model dependent. With a model with large amounts of data in `dyn_arrays`, very few blocks would be executed simultaneously, resulting in a net performance loss, even if each block is executed faster than before. Tiling was not a good idea either, since it is very model dependent as well. The classical tiling method for matrix multiplication, where each thread uses data from the whole grid, is not applicable here, as each cell only needs data from it's neighbors. A similar solution, however, would be to try to identify if the enthalpy calculations could be done in steps, where the quantities could be divided between the steps. As this turned out to be very complex to do even by hand for this single example, it was decided to skip it, since an auto-generation of it capable of handling any scenario most likely would be much harder.

3.4 Airflow Model Complexity

As the now working kernel was tested, it turned out to be significantly slower than the original version. However, when removing the CUDA specific code, the performance was even worse. This indicated that the change of the original model to one using an `algorithm` section was causing the performance dip, not the CUDA code. This is shown by the results in table 3.1, containing numbers on how the `algorithm`-version with CPU and GPU execution related to the original version.

The reason why so many additional function evaluations are done is because no analytic Jacobian can be found by Dymola. An algorithm section is treated like a black box by Dymola during computation. By not knowing if or what mathematical expression could replace this black box, the derivatives necessary for the analytical

Table 3.1: Time and function evaluations for the different versions of the airflow model. Values are factors relative to the original time, and number of function evaluations.

	Original	`algorithm`, CPU	`algorithm`, GPU
Time	1.0	2.48	2.42
Function evaluations	1.0	6.09	6.09

Jacobian can not be found. Instead a numerical Jacobian is used, which requires multiple times more function evaluations to compute.

To verify this theory, the model should be simulated using an explicit method which does not need a Jacobian. This method should then have the same number of function evaluations for both the original and the algorithm versions. Because the airflow model was so stiff, no explicit method that was tried could give a stable result.

Because no signs of total acceleration were seen so far, this was not considered to be enough to base auto-generation on. At least a hint was needed before it was meaningful to change the compiler to produce GPU code. It was therefor decided to make attempts on a less complex model, which could also be simulated using an explicit method. The conclusions from the airflow model could still be used as a starting point for another model. With potential future auto-generation of CUDA, the airflow model could be returned to, to parallelize a larger part of it.

3.5 Changing to a Simpler Model: Shallow Water

Before we move on to the new model, we summarize how the airflow model was changed to run on the GPU:

1. Move what should be parallelized into an `algorithm` section and surround it with (possibly nested) for-loop(s)
2. Connect the assigned variables to the actual, time dependent ones through equalities in an `equation` section
3. Split any record in the parallel part into it's sub parts and arrays
4. In the auto-generated C code, replace the (nested) for-loop(s) with a call to a function that does the following:

```
...
//Host wrapper function
if(first iteration)
   allocate pinned memory for dyn_arrays on the host
   allocate memory for dyn_arrays on the device
   save info about array sizes as constant data
   allocate memory for answer on the device
   allocate memory for constant data on the device. Use constant
      memory if possible.
```

```
gather dyn_arrays into pinned memory

if(first iteration)
   copy all the constant data to the device

transfer dyn_arrays to the device

if(first iteration)
   Set up Grid and Block sizes

launch kernel, passing pointers to device memory of dyn_arrays,
    answer and constant data

copy answer back to host
...
```

5. The body of the (nested) for-loop(s) should be mapped to a kernel according to:

```
//Kernel
kernel(dyn_arrays, answer, constant_data)
   assign unique indices to thread

   Assign pointers to the different arrays in dyn_arrays, using
       sizes stored in constant_data

   if(indices within simulation grid)
      original for-loop work, with accesses using the IX() macro.
          Use automatic variables instead of multiple accesses of the
          same memory location.
      save calculation to answer
```

3.5.1 Shallow Water

The model used as the new test case was developed by the Institute of System Dynamics and Control, DLR, Germany. As this model is much simpler than the airflow model, it is presented below.

For wave power plants or off-shore constructions such as wind-turbines and oil-platforms, as well as for free floating objects such as ships, the interaction with the water surface is a key component for system simulations. For this purpose, the shallow water equations represent a set of PDEs that enables an efficient approximation of the surface dynamics. The PDEs for a 2D-surface in its simplest form is shown

below:

$$\begin{aligned}\frac{\partial h}{\partial t} &= -H(\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y}) \\ \frac{\partial v_x}{\partial t} &= -g\frac{\partial h}{\partial x} \\ \frac{\partial v_y}{\partial t} &= -g\frac{\partial h}{\partial y}\end{aligned} \tag{3.1}$$

where h is the height of the surface, v_x and v_y is the surface velocity of the water, H is a constant and g is the gravitational acceleration. An animation of the model can be seen in figure 3.4. The simulation is made on an axis aligned square, and closed boundary conditions are applied (Naumann conditions), i.e. $\frac{\partial h}{\partial x} = 0$ at the boundary where x is fix, and $\frac{\partial h}{\partial y} = 0$ where y is fix.

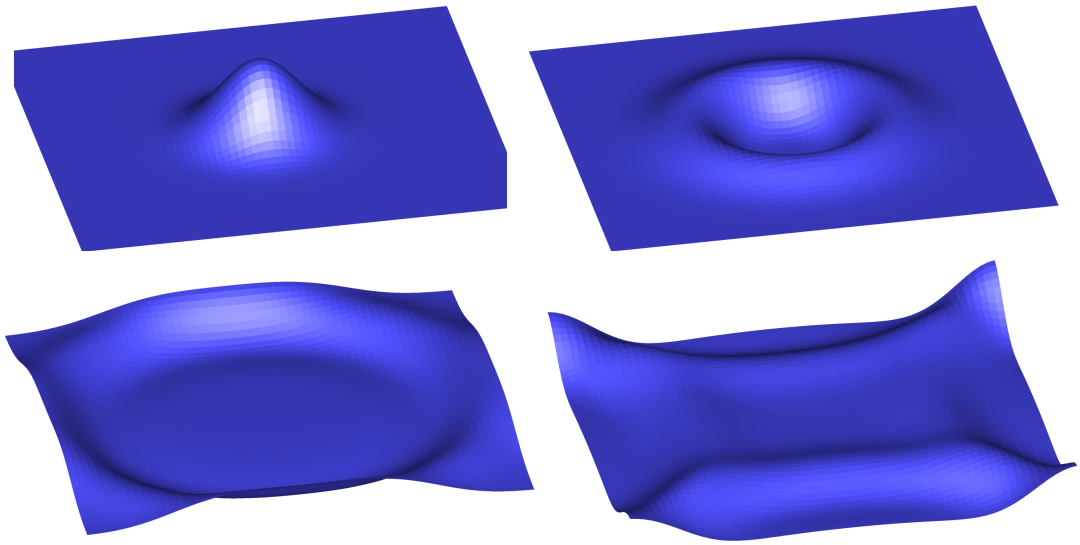


Figure 3.4: The shallow Water model at different points in time

In this model, the velocity of the water flowing within the 2D surface causes a gradient (i.e. a difference between inflow and outflow), which in turn causes the surface height to raise or fall. Spatial gradients in the surface height then cause a counteracting acceleration of the water flow. H is chosen as $L/4$, where L is the length of the side of the surface area, and g is set to 9.81 m/s^2 to model gravity realistically.

This shallow water model was translated into CUDA following the steps described in the beginning of section 3.5, resulting in a working simulation. A full description of the translation exists in appendix A.1. It is there demonstrated, by showing real code, how the Modelica model should be modified and how the by-hand parallelization of the auto-generated C code was done.

With this model, it became apparent that memory accesses in the kernel were not coalesced. A warp would get assigned threads in a row-major fashion, while the data access macro reveals that the data is arranged column-major. To make

the global memory accesses coalesced, the thread id's `x`- and `y`-components could be swapped like

```
size_t iy = threadIdx.x + blockDim.x*blockIdx.x;
size_t ix = threadIdx.y + blockDim.y*blockIdx.y;
```

or the order of the for-loops in Modelica be swapped.

3.5.2 Inline Integration

Memory transfers between the CPU and GPU are as mentioned slow, and therefor they should be avoided as much as possible. Currently, two memory transfers are required in each iteration, to perform the integration in Dymola, store states, possibly interact with other parts of the model, etc.

A lot of these transfers can be avoided by doing the integration directly on the GPU. To keep interaction with other parts of the model, the states could then be transferred back to the host every n_{steps}'th step. To hand code e.g. an explicit Euler or lower order explicit Runge-Kutta method directly in the wrapper function is easy, by only using device-to-device copies.

It could also be a possibility for Dymola to in the Modelica model recognize an outer for-loop, that contains the (nested) for-loop(s), and translate that to the corresponding CUDA code. It would probably be easiest if the outer for loop and the integration method initially was constructed by the user.

3.5.3 Parallelization Instructions

Since the same conversion of C to CUDA code worked for both the airflow model and the shallow water model, those steps could be used to put together instructions for the auto-generation of CUDA code to DS. Those translation instructions are presented below.

- The device code and any CUDA specific functions need to reside in a `.cu` file to compile correctly with Visual Studios. The set of nested for-loops should be translated to a call to an external function in the `.cu` file. This function should be prefixed with the `external "C"` declaration, to prevent name wrangling.

  Note, this function needs the external declaration since compilation has to be done by the NVCC compiler, and therefore cannot be included in the `.c` file. This compilation method has only been tested on Visual Studio 2010, and may or may not be necessary for other compilers.

- All arrays that need to be accessed from the device, i.e. that are placed inside the for-loops, should generate the following:

  – One allocation using pinned memory on the host. If possible, one large allocation should be made for all arrays, to reduce the number of copies per iteration. That would need some way of keeping track of which array that is placed where, and is not so important if all arrays are very large. If

different types are used (such as `double` and `int`), one allocation should be made for each type.

– Corresponding allocation of device memory.

These allocations should if possible be made once, initially. Thus, pointers to those memory locations have to be stored between iterations.

- Device memory for answer arrays should also be allocated once if possible.

- When gathering the arrays for index access they should be gathered in their respective allocated pinned memory.

- Transfer the arrays needed on the device, preferably, as mentioned, as one consecutive set of data.

- Constant data should if possible be recognized, and only be transferred once. Single constants could be passed by value at kernel launch, while larger sets should be stored in global or constant memory.

- The external function should contain a setup of a grid and block sizes. For the sake of simplicity, the following can be used for the 1, 2 and 3 dimensional cases:

  1D: `dim3 block = dim3(1024), grid = dim3((N+1023)/1024);`

  2D: `dim3 block = dim3(32,32), grid = dim3((Nx+31)/32,(Ny+31)/32);`

  3D: `dim3 block = dim3(16,16,4), grid = dim3((Nx+15)/16,(Ny+15)/16, (Nz+3)/4);`

  generating blocks of 1024 threads, which is the limit of currently used hardware. This would of course not be efficient in 2D or 3D, if one of the dimensions is very small, or just larger than the block size. But it could work for a prototype.

- At kernel launch, pass necessary pointers to the data needed, the answer location, and potentially constant memory. If needed, also pass information about array sizes.

- The content inside the (nested) for-loop(s) should be translated to a kernel, i.e. into a function with return type `void`, that is pre-fixed with the `__global__` keyword, which should look like this:

```
int idx = threadIdx.x + blockDim.x*blockIdx.x;
int idy = threadIdx.y + blockDim.y*blockIdx.y; //If 2-dim block
int idz = threadIdx.z + blockDim.z*blockIdx.z; //if 3-dim block

//The if-statement should also change depending on block dimension
if(idx < Nx && idy < Ny && idz < Nz){
   //Body of the (nested) for-loop(s), and if multidimensional, with
       accesses using some macro or function not using ellipsis
       arguments.
}
```

In order to achieve coalesced memory access, `threadIdx.x` should correspond to the innermost for-loop, `threadIdx.y` to the second innermost and so on.

- Any function call from inside the for-loops should in the auto-generated C-code be moved to the `.cu` file, and prefixed with the `__device__` keyword.

Additionally, steps 1-3 from section 3.5 have to be applied on the Modelica model.

3.6 Auto-generation

Using the instructions above, DS started an implementation of automatic CUDA code generation. For completeness sake and to show the interested reader what our analysis lead to, we include a short description of how the current prototype works. For more details, see [8].

Currently, the prototype works by recognizing functions marked with a special annotation. These functions have some specific requirements placed upon them in order to simplify translation.

3.6.1 Variable Allocations

All array variables of the function must either have unknown size (only allowed for inputs), or a size given as a simple arithmetic function of other sizes and integer literals. The unknown array sizes are also propagated to the kernel function. (For performance reasons, and to catch errors, it is good to have as few unknown sizes as possible.) The arrays are allocated on the device, and existing values copied to the device, which allows non-input variables to be assigned a default value in a binding expression. Protected arrays are treated as outputs (answers) of the kernel function.

3.6.2 Loop Patterns

The first pattern for the algorithm is that the entire `algorithm` section is a (possibly nested) for-loop with range `1:size(array, literal)` and inside the loop any algorithmic code. The code inside the for-loop(s) is mapped to a kernel function, and the (nested) for loop(s) are replaced by the corresponding external wrapper function.

3.6.3 Time integration on the GPU

The second pattern (intended to handle time-integration on the GPU) is a for-loop (with arbitrary index) that contains one or more instances of the first pattern. Each instance of the first pattern is then mapped to a kernel function and called at the appropriate place. The rest of the body may contain assignments, and any array assignment is mapped to a device-to-device copy. The outer for-loop then decides how many inline integration steps the device will take before copying the data back to the host. Each loop iteration corresponds to one kernel launch, ensuring thread synchronization before the next time step.

3.6.4 Differences from the Instructions

Below, some notes are made on how the current prototype relates to the instructions that were given.

- As the parallelized part should be placed inside an annotated function, it was possible to remove the need of step 2-3 from section 3.5, which made the feature more user-friendly. The `algorithm` section is still needed, however.
- Allocations are only made once, but they are made separately for each original array, and thus, so is the host-device copies.
- Gathering is done automatically outside the function, also a result of placing the `algorithm` section in a function.
- Constant arrays are not recognized.
- Indexing is generated inline - no macro or other function is needed.

3.7 Additional Test Examples

After DS had implemented a working auto-generation prototype, we tested its performance on the Shallow Water and two additional models; matrix multiplication and another CFD model, which will be referred to as *cold plates*. Matrix multiplication was chosen as it is an area where GPUs are known to perform well. The matrices contained double precision floating point values and no zeros. The matrix multiplication Modelica model and its auto-generated CUDA code are shown in appendix A.2. The cold plates model, presented below, was chosen as it is a more complex model (not as complex as the airflow model, however), closer to what real users of Modelica often simulate.

3.7.1 Cold Plate

Cold plates are, for instance, used to cool power-electronics. The dissipated heat is transported away from the source by conduction and convection. In this two-dimensional example, a single fluid pipe is surrounded by two rectangular conducting plates. For the sake of clarity, a simple monolithic model that can mentally be split

up into three kinds of cells: thermal conduction cells, fluid volume cells, and fluid flow cells, was used, see figure 3.5.

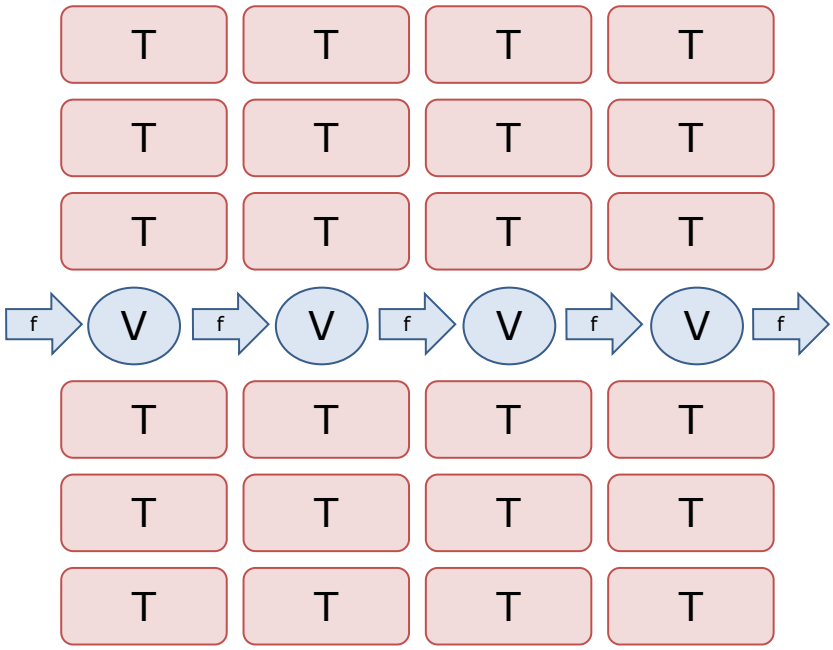


Figure 3.5: Illustration of a simple cold plate model, split into thermal (T), fluid volume (V) and fluid flow (f) cells

The number of cells in both dimensions is configurable, tuning the number of variables and states. In the thermal conduction cells, heat is stored and conducted to the four neighboring thermal cells or fluid volume cells. This is a slight simplification, as the resulting dynamics are anisotropic. In the fluid volume cells, balance equations for mass and energy are established. Fluid is transported between the volume cells by flow-cells, which calculates the mass flow based on the pressure values of the neighboring volume cells.

One thing to note in the cold plate model is that it consists of two functions that are parallelizable, one describing the flow along the central line, and one describing the conduction throughout the grid. These two functions will in the auto-generation correspond to two different kernels.

As initial conditions, the temperature of all cells were set to 295 K. A constant pressure gradient of 0.01 bar was applied, and the inlet temperature was set to 373.15 K, resulting in a heating transient. Since the fluid transport is rather stiff, a small step-size had to be used.

Chapter 4

Results

Full simulation timing was made by us using Dymola's built in timer, and results are from a single simulation unless otherwise stated. A statistical average was not used, since each simulation consists of thousands of iterations. A simulation therefore consists of multiple runs of our code, and the full simulation time will therefore already contain an average. For all simulations the results of the GPU and CPU executions were identical, to machine precision.

The simulations whose results are presented in this chapter were run on computers with the specifications given in table 4.1.

Table 4.1: Hardware specifications

OS:	Windows 7 Enterprise x64
Processor:	Intel(R) Core(TM) i7-4800MQ CPU @ 2.70 GHz
RAM:	16 GB DDR3 1600 MHz
GPU:	NVIDIA Quadro K2100M 2 GB, using PCIe generation 3

4.1 Matrix Multiplication

Table 4.2 shows the simulation time of the auto-generated matrix multiplication model, where n is the size of the square matrices and the speed-up is CPU time/GPU time. Note that CPU-performance of large matrices is sensitive to caches; on the computer used for this thesis, $n = 2^x$ resulted in large CPU performance dips. Therefore, the sizes were chosen to avoid that effect.

4.2 Shallow Water

For this model, an inline integration with the explicit Euler method was easy to implement. Table 4.3 shows the speed-up factor of the auto-generated parallelization, where n_{steps} is the number of inline integration steps, n is the size of the grid, and the values in the table are the CPU time/GPU time ratios. The Rkfix2 method

Table 4.2: Results of the Matrix multiplication model

n	CPU [s]	GPU [s]	speed-up
50	0.000453	0.000438	1.03
100	0.00362	0.0018	2.01
200	0.0275	0.00996	2.76
500	0.506	0.142	3.56
1 000	6.37	1.11	5.74

was used, with a time step that decreased with the grid size, to avoid numerical instability. To get the same number of steps, the simulation time was decreased accordingly. 20 000 steps corresponds to the full simulation being integrated on the GPU, without communication.

Table 4.3: Speedup of the Shallow Water model

$n_{steps} \backslash n$	32	64	128	256
1	0.36	0.77	0.91	1.00
10	0.46	0.94	1.19	1.42
100	0.77	1.89	3.03	3.19
1 000	1.01	3.06	5.30	5.12
20 000	1.09	3.02	5.05	5.20

4.3 Cold Plates

Table 4.4 shows the results of the auto-generated parallelization of the cold plates model, which were generated without inline integration. n_x and n_y are the dimensions along the x and y dimensions respectively, see figure 3.5. The speed-up factors are CPU time/GPU time. The simulations were made using the Rkfix2 method, with $dt = 10^{-7}$ s and $T = 10^{-4}$ s.

Table 4.4: Results of the Cold Plates model

n_x	n_y	CPU [s]	GPU [s]	speed-up
256	256	17.2	12.1	1.42
512	512	91.7	42.9	2.13
500	200	26	18.1	1.43
1000	100	28.6	20.2	1.41
2000	200	103	65.2	1.57
200	2000	104	61.3	1.69

Chapter 5

Discussion

5.1 Matrix Multiplication

The matrix multiplication test shows that in an area where GPUs usually perform well, the auto-generated GPU code does succeed in significantly speeding up the model. While the factors seen in table 4.2 are not as impressive as can be when generally using GPUs for matrix multiplication, the results were achieved on a laptop with non-specialized hardware and with no model specific optimizations. By using shared memory, tiling, and a NVIDIA tesla GPU, the matrix multiplication performance could be improved significantly [37]; but the results would probably be misleading for the average Modelica user, and require specialized knowledge to program.

By outperforming the CPU on matrix multiplication, we can conclude that tasks that are as suited for parallel execution as matrix multiplication, are likely to benefit from this auto-generation scheme, and require no GPU knowledge of the user.

5.2 Shallow Water

The shallow water model without inline integration ($n_{steps} = 1$) does not show any improvement over the CPU execution, see table 4.3. However, when the number of inline integration steps increase, the CPU/GPU runtime factor improves, reaching a maximum of about 5. This indicates that the shallow water model contains too little work in the kernel, compared to the amount of time spent in GPU-CPU memory transfers, which means that even if the kernel performs better than the CPU, the time gained is lost in the transfers. The results also indicate that the transfer time is hidden enough when n_{steps} reaches 1 000, and that shutting off communication completely is not required for a performance boost.

That the shallow water model needs inline integration to show a similar speed-up as the matrix multiplication can be explained by the number of floating point oper-

ations in the kernel compared to the data transferred per kernel call. The number of floating point operations per thread in the shallow water model is constant, while in the matrix multiplication example it is proportional to n. This means that a matrix multiplication kernel does more work per thread, which increases the kernel time in relation to the transfers when scaling up, and therefore results in better performance. By using inline integration, the amount of floating point operations per transfer is increased for the shallow water model, and it receives a similar performance boost.

The drawback of using inline integration, is that it lessens the interactivity in models which are only partly parallelized. The CPU part can only receive new data whenever the states on the GPU are transferred back. This creates a balance between using a large number of inline integration steps to increase GPU performance, and having a high level of interactivity.

While inline integration creates problems for some models, the limited interactivity can actually be a benefit for others. Some models will have one very stiff parallelizable part, while the rest is not so stiff, and runs on the CPU. The inline integration then offers the ability of running the stiff part with a much shorter time step, and only updating the non-stiff CPU part whenever the time steps match. An example of such a model would be a fluid flowing in a pipe, of which the temperature at one point is used to drive a feedback control system, which regulates the energy flow to a boiler in the end of the pipe. The temperature would then change very slowly compared to the fluid movements, and the regulation circuit would only need updates, for example, once every second. The fluid could then be simulated with a time step in for example milli or micro seconds, and update the rest of the system after 10^3-10^6 steps.

5.3 Cold Plates

The cold plates model had the worst performance, reaching a speed-up factor of 2.13 at best, see table 4.4. The results of the simulation times are hard to analyze. On one hand, the problem appears to perform better on the GPU than on the CPU, since all simulations show a CPU/GPU factor larger than one. However, the performance dips when the size grows too large, reaching a peak at $512 \times 512 = 252144$. The probable reason for this would be that either the 1D kernel or the 2D kernel is faster than the other, and when the problem is not squared, the abundance of the "slow" kernel impedes performance. This is not the case however, as if it was, the opposite rectangular grid should perform better, since it would have a larger fraction of the work in the "fast" kernel. If both kernels were equally fast, then the shape of the grid should not matter as much, and performance should not dip when the size is increased. The 1D kernel might suffer bad performance with a too small grid, and therefore decrease the overall performance when n_x is to small. But this dip should then not be present in the 2000×200 grid case.

Conclusively, the performance results are due to some factor other than the kernel speed, a likely candidate being cache misses based on the grid. Unfortunately, time didn't allow for further testing.

5.4 Conclusions

The auto-generation prototype does appear to work, and to give a significant speed-up for sufficiently parallelizable problems. The fact that inline integration is possible allows for a greater amount of problems to be parallelized for a speed up. While none of the speed-ups were equal to what probably would be possible using hand coding, it's still a positive result for a first prototype. That the potential users do not need to know anything about GPUs also makes the prospect look good. An easy speed-up of up to a factor of 5 can be worth more in some cases, than spending hours or days writing parallel code, for the users with no knowledge of GPUs.

That being said, the auto-generation has some drawbacks. One of the larger is the problem when using implicit methods. If a model requires the usage of an implicit method, then the rewrite to a GPU version would introduce the problem with a numerical Jacobian and make the problem unlikely to achieve a speed-up. It also suffers somewhat in the ease of usability in that a user with no knowledge of GPUs might have a hard time determining if a problem would be benefit from being run in parallel or not. For example, the cold plates example appeared to be beneficial to run on the GPU, but was so only barely.

The final remark is that while still very basic, the prototype implies that auto-generation of GPU code that is faster than the CPU code, is possible. Such an auto-generation also appears to be possible with little or no knowledge from the user about GPUs.

5.5 Future Work

The prototype, while functional in the sense that speed-ups were achieved for all tested models, is far from complete. One of the major points that could be improved is that the prototype could have implemented handling of special cases, to introduce use of e.g. shared memory.

It would be possible to develop a couple of different tiling algorithms, for models that need a row, column or matrix style data read in the kernel. These tiling algorithms could then be chosen automatically if the correct access pattern was detected by the Dymola compiler. Another case dependent strategy would be for Dymola to count the amount of variables used in a loop to try to fuse different kernels, or loop iterations. This would allow for more kernels to be effectively run on the GPU.

A problem with the current method of auto-generating inline integration is that only simple explicit methods are going to be possible to implement, as anything else would be far too complex to be user friendly. A far better way, but harder to implement, would be to automatically generate the time integration. Given an explicit set of ODEs, it's quite easy to generate many different integration schemes. Depending on how complex expressions the Dymola compiler can translate into such a system of explicit equations, this would put a requirement on how the kernel could be written.

With the ability to chose which inline integration method to use and automatic synchronization of the inline integration and the rest of the model, a lot more models

would be capable of benefiting from GPU execution.

Part III

Contact Handling

Chapter 6

Package Description

In this chapter the developed contact handling package is presented. After a short DEM introduction in section 6.1, the different parts of the package and the theory behind them are presented in sections 6.2-6.6. Some additional package related details are brought up in section 6.7 and 6.8.

6.1 General DEM Structure

A DEM simulation describes the motion of particles, and the phrase often refers to a simulation of many particles. Our contact handling package, however, does not have a restriction on number, shape or sizes of the particles, and they will be referred to as bodies. The number of bodies may be as few as two, and still be of interest. The model used to describe motion depends on the kind of DEM simulation that is desired, but in classical Newtonian mechanics, each body's motion is usually modeled by the famous equations of motion, here expressed for rigid bodies:

$$\begin{aligned} \mathbf{f} &= m\mathbf{a} \\ \mathbf{t} &= \mathbf{I}\dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times \mathbf{I}\boldsymbol{\omega} \end{aligned} \tag{6.1}$$

Here $\mathbf{f}$ is the force, $\mathbf{t}$ the torque, $\mathbf{a}$ the acceleration, $\boldsymbol{\omega}$ the angular velocity, m the mass, and $\mathbf{I}$ the moment of inertia tensor of the particle. All quantities are taken with respect to the center of mass of the body. In some simulations, of e.g. sphere-shaped objects without friction, rotations may be neglected.

Because the equations are only differentiated in time, most algorithms are based on some integration method, where in each step, the forces and torques of every body are assembled. From here on, unless otherwise stated, "forces" refers to both forces and torques.

In order to determine the forces acting on the bodies, a method is needed to detect collision. Collision detection is often divided into two parts, the *broad phase* and the *narrow phase*. The broad phase is purely an optimization phase, in which potential collision pairs are determined. The selected pairs of bodies are then compared in detail in the narrow phase, to determine if there is a collision, and if so, determine necessary information about the collision, for the force calculations.

Since the equations of motion are only differentiated in time, they are well suited for simulation with Modelica. An advantage of using Modelica and e.g. Dymola for this, is the possibility to use various integration methods. When developing the package, it has been very helpful to, for example, be able to switch between fixed step size methods and adaptive step size methods.

6.2 The Modelica Framework

What is needed for a contact handling simulation in Modelica is a way to represent a scene of three dimensional objects, and to calculate forces for the equations of motion. Integration is taken care of internally by Modelica. As it turns out, the equations of motions need not be implemented, as they already exist in the Modelica MultiBody library. The bodies in the scene can then inherit these from the MultiBody library class `Body`. As those are used, interactivity with other parts of the MultiBody library becomes easier.

Our company supervisor Hilding Elmqvist created a framework in Modelica that was based on the usage of the MultiBody library. In this framework, a scene is represented by an external object `ExternalScene`, containing all bodies of the scene, represented by `ExternalBody` objects. The bodies contain their geometric representation, described by a triangular mesh, with the exception of spheres which have a parametric description. A synchronization is then made so that in each iteration, all bodies does the following in order:

1. Update position
2. Calculate resulting forces
3. Update the old forces

How the synchronization was achieved is explained in Elmqvist et al.[25]. When the forces have been updated, integration is performed, calculating a new state that is used to update positions in the next iteration. "Update position" and "Update the old forces" are merely communications between Modelica and the external C-code.

So our task became to implement those three external functions, and all the necessary supporting functions. Two of them where as mentioned trivial, while the rest of this chapter, with a few exceptions, will focus on the third one. The implementation was mainly done in a CUDA C++ Visual Studio (VS) project, compiled to a `.lib` file.

6.3 Broad Phase

In most cases detailed collision checks are quite computationally expensive, making it more efficient to use *bounding volumes*. If the bounding volumes of two bodies intersect, that pair can be checked in more detail. The most common bounding volume type is the *axis aligned bounding box* (AABB), the smallest axis aligned cuboid enclosing the body, but one can also use e.g. the bounding sphere (centered in the centroid of the body).

As n increases, it gets expensive to even check bounding volumes for all bodies, and this is where the broad phase comes in. A broad phase is used to limit the number of collision checks that a contact simulation has to consider for each time step. Without a broad phase, every body would have to be checked against every other body in the scene, resulting in a time complexity of $O(n^2)$, with n being the number of bodies. Removing this quadratic time dependency is the point of having a broad phase. The most common approaches uses some kind of recursive tree structure into which the objects are placed, resulting in an $O(n\log(n))$ time complexity.

As an example, imagine 1000 balls placed in a straight line, every pair being precisely at contact. Without a broad phase, approximately half a million bounding volume checks would be carried out, assuming that every pair of particles is only checked once. With a good broad phase however, only 999 checks would be needed.

The algorithm used for the broad phase in this package is based on the algorithm presented by D. Lavrov[12]. Before the algorithm is presented, a short explanation of the *Z-order* curve will be made, as it is central for the algorithm.

6.3.1 Z-Order Curve

The Z-order curve is a so called *space filling* curve. Space filling curves are curves that cover every point in space, or in our case, a discretized space. They are useful since they provide the possibility to assign a point in space a single variable, basically how far along the curve the point is placed. This variable then contains the positional information about the point. The Z-order curve arises when doing the following:

Discretize 3D-space into *base cells*, so that the coordinates of each base cell can be described by an integer. Translation of coordinates should also be done, so that the integers are unsigned. Now the idea is to assign a so called *Morton code* to each base cell. The Morton code is determined by interlacing the binary representation of the three coordinates, according to the example in figure 6.1.

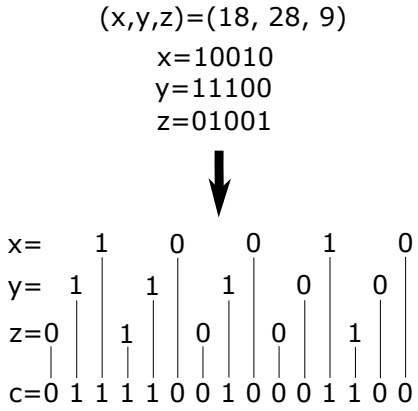


Figure 6.1: Morton encoding

c is the Morton code, and as shown in the figure, starting from the least significant bit, the lowest x-bit comes first, followed by y's lowest, then z's lowest bits. Then the second lowest x-bit, and so on. In the figure, unsigned integers of 5 bits were used for simplicity, which results in a very low resolution (few base cells), only 64x64x64.

In our code, 64 bits where used to store the Morton code. Thus 21 bits could be used for each coordinate in the encoding, resulting in a resolution of 2 097 152 base cells in each dimension.

If the base cells are sorted according to their Morton codes, the order will follow the curve shown in figure 6.2 (in 2D), called the Z-order curve.

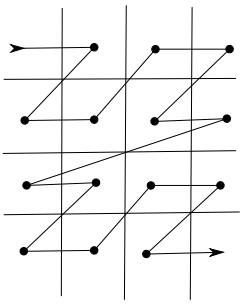


Figure 6.2: Z-curve ordering in 2D

For more details on the Z-order curve and Morton encoding, see e.g. Karras[42].

6.3.2 The Broad Phase Algorithm

The idea is that every body is occupying 1, 2, 4 or 8 Morton code intervals, where the Z-order curve is continuous. The number of intervals each body requires are dependent on how its AABB is placed relative to the Z-order curve. All the Morton code intervals a body generates are of the same length, and that length can easily be determined by looking at the required size for the largest part. This is illustrated in 2D in figure 6.3. The two smaller bodies occupy intervals of length 1, while the larger body occupies intervals of length 4. In 2D, a body's bounding box can occupy 1, 2 or 4 intervals. In the figure, the smallest only needs two intervals, while the other two need four. With some luck, two of the bodies could have been placed so that they only generated one interval each.

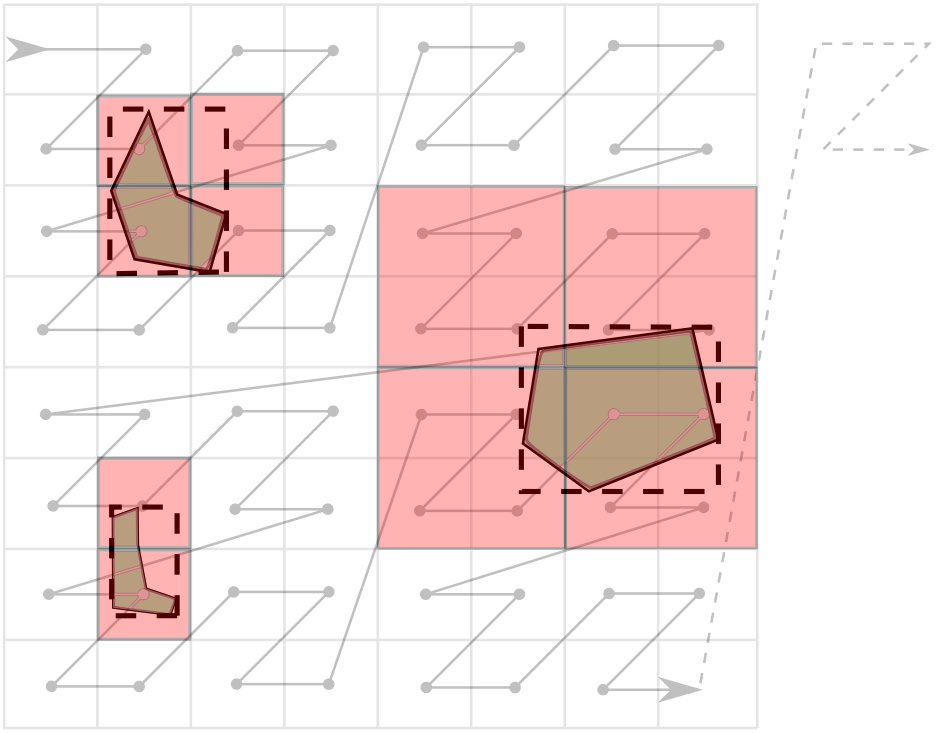


Figure 6.3: 2D illustration of how bodies occupy Morton code intervals

Our implementation of the Morton interval generation is based on D. Lavrov's algorithm[12], from where the code in the `Entry` struct and the `AddAABB_raw` function have been modified to our need. For generating the Morton code of a specific point in space, the "Magic Bits" method presented by J. Baert[31] was used.

The generation of Morton code intervals was used in the following algorithm, as a broad phase:

1. For each body:
    a. Generate its Morton code intervals (1, 2, 4 or 8).
    b. The information should be inserted in a vector, where each element contains the interval, and the body's id.
2. Sort the vector according to the start code of the intervals.
3. For each interval:
    a. Iterate forward in the vector until a start code is found that is greater than the end code of this interval.
    b. For every start code passed within this interval, push a pair containing this intervals id and the others, to another vector.
4. Remove duplicates from the vector with pairs.
5. For each pair left in the vector:
    a. Check the bodies AABBs against each other.
    b. If they intersect, a more detailed analysis is needed (narrow phase).

6.3.3 Optimizations

The broad phase algorithm, as it is, has a time complexity of $O(n\log(n))$ where n is the number of bodies, which comes from the sorting. In fact, two sorts are done, as step 4 (remove duplicates) requires a sorted vector, or at least the standard algorithms do. All other operations have time complexity $O(n)$. Note that n is not the number of bodies for step 4, instead it is the number of pairs generated in step 3. The number of such pairs obviously depends on the state of the scene, but also on how the bodies' sizes compare to the base cells. This section discusses how to optimize steps 1, 2, 3 and 4. Optimization of step 5 will be discussed under narrow phase, section 6.5.3.

Step 2 and 4 (Sorting)
Because of the time complexity, the first optimization done was to use parallel sorting. For this, the Thrust library was used, which is a C++ template library for CUDA, developed by NVIDIA [43]. It performs better on built in primitives, on which it can apply parallel radix sort[26]. Thus, the elements in the vector built in step 1 need to be split up as two vectors; one containing the Morton codes (interval starts) and the other containing end code and ID. Then the vectors could be sorted by `thrust::sort_by_key`, where the Morton codes are the keys. The same goes for the pairs in the second sort; two vectors of unsigned integers (for object ids) were needed, instead of `std::pair`'s or the Thrust equivalent `thrust::pair`. For the first sort this did not matter, but the removal of duplicates had to take key-value split into account. Sorting the possible collision pairs with `std::pair` would sort first by the first body's id, then the second's, resulting in a vector where identical collision pairs would be next to each other. Sorting the same vector according to key-value pairs would sort according to the first body in the possible collision but not the second. So we had to make an algorithm that removes the duplicates of such vectors. The implementation and a brief explanation can be found in appendix A.3.1.

Step 1
The next optimization was to parallelize step 1 with CUDA, by having each thread calculate the Morton intervals of one body. This makes parallelization of step 1 trivial except for one thing; it is not known how many intervals will be generated per body. A general problem when working in the SIMD fashion is to push an unknown amount of data to the end of a structure. Even if enough memory is allocated, so that no reallocations are needed, every thread that pushes something needs to know where to place it's data. For this, a counter is needed, and when threads simultaneously push, they all have to increment the counter, and get a unique index to use. CUDA provides a function for incrementing atomically, `atomicAdd`, but it is serialized, i.e. all active threads in all active warps increments the counter one at a time, making it very slow. CUDA's `atomicAdd` is meant for situations where few threads execute it simultaneously. In step 1, every thread will push at least one element, so atomic operations would not be a good idea. Instead, we can use the fact that a thread at max pushes eight elements, corresponding to the at most 8

cells making up the body. Memory can then be allocated for 8 times the number of bodies. Each thread can then place the intervals corresponding to it's body at index $8i + j$, where i is the body's id, and j is the count of how many intervals that this body has already generated. Considering how warps are organized, $i + jn$, where n is the number of bodies, is preferable to the $8i + j$ mentioned above, for coalesced memory access. By zero initializing the Morton codes, the non-used interval spots will be placed first after sorting. It will then be a trivial matter to remove them. The amount of instruction divergence that will appear depends on the average number of intervals generated per cell. The performance impact sorting and transferring the empty intervals have is also dependent on the average number of generated intervals per cell. The kernel is shown in appendix A.3.2. Note that the same, coalescing, indexing method $(i + jn)$ is used for the AABB data access.

Step 3
For step 3, the main problem was once again that the number of pairs generated per interval is not known. Some intervals may generate none, while others generate many, introducing a certain amount of instruction divergence. The solution used for step 1 is not reusable, as the maximum number of pairs an interval may generate is not know. Again, `atomicAdd` is too slow, as the number of generated pairs per interval is relatively high in most cases. Therefore, two different ideas for parallelizing step 3 were tried.

In the first attempt, a kernel was launched that counted the number of added pairs for every interval, without pushing the pairs. Then, using a method from the Thrust library, the prefix sum of the generated sequence was computed. The prefix sum could then be used in a second kernel launch, where each thread knew how many pairs to push, and where to push them.

In the second attempt, an optimized version of `atomicAdd` was used. The function used is `atomicAggInc`[18], from NVIDIA. It is based on the following idea:

1. Every warp counts it's number of active threads.

2. A single active thread, a *leader*, within each warp is elected, to atomically increment a global counter.

3. From the atomic increment, the leader can share the global index to it's warp, from which the threads should place the data.

4. Each thread uses it's *lane* id, which is it's position within the warp, to compute it's final index. The index will be the offset shared from the leader, plus the number of active threads in the warp with lower lane id than itself.

To count active threads, the parallel CUDA function `__ballot` is used, which returns a 32 bit integer with every bit corresponding to a lane, indicating whether the lane is active or not.

None of the mentioned attempts improved performance of the algorithm. The most probable reasons are that the amount of work per thread is too low, and that it varies a lot, causing high instruction divergence.

Of the optimization of step 5 not much will be said here. Checking the AABBs is too little work to parallelize, considering the amount of copying needed, and optimization of the narrow phase is explained in later sections.

6.4 Special Case: Spheres

To initially test the Modelica Framework, and to test the broad phase, the special case of spheres was used. A parametric description of the spheres is then sufficient, and no triangles are needed. That allows for a very fast narrow phase. In the current prototype, scenes with spheres may contain *only* spheres, as contact between parametric spheres and triangular meshed bodies is not supported.

To detect contact between two spheres whose AABBs are colliding is trivial. Contacts are modeled by a spring-like force, $f = c_s d$, where f is the force, c_s is the spring constant and d is the penetration depth. In 3D, that becomes:

$$\mathbf{f} = c_s \frac{\mathbf{p}_1 - \mathbf{p}_2}{\|\mathbf{p}_1 - \mathbf{p}_2\|}(\|\mathbf{p}_1 - \mathbf{p}_2\| - (R_1 + R_2))$$

where $\mathbf{p_i}$ and R_i are the position of the center and radius of sphere i respectively.

6.5 Constructive Solid Geometry Intersection

This section explains how the CSG intersection operator has been used in the narrow phase. The choice to use BSP tree-based CSG was motivated by:

- Its ability to work on arbitrarily shaped objects.
- The method used for force calculations, explained in section 6.6, needs a set of polygons describing the intersection volume at collision.
- The fact that it is considered to be a relatively fast algorithm.

6.5.1 CSG with BSP Trees

The theory presented in this section is mainly based on the work by Segura et al.[14].

CSG is a technique to, using boolean operations such as *union*, *intersection* and *difference*, construct bodies from geometric *primitives*(typically cuboids, cylinders and spheres). Each body is then represented by its CSG tree (which is not to be confused with a BSP tree, which will be explained later), where every node is an operator, the leaves are primitives, and the result of the root operation is the actual body, see figure 6.4.

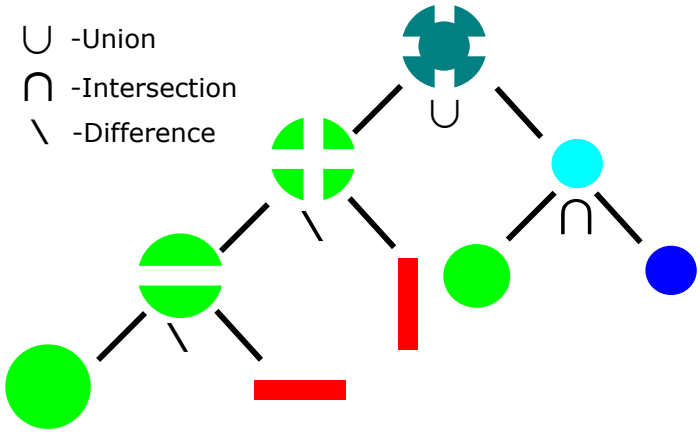


Figure 6.4: Example of a CSG tree in 2D.

However, the shapes of the bodies that are of interest in our package do not necessarily have to be constructed in this way. The only thing that is of interest for this package is the intersect operation, and the way that it can be implemented using BSP trees. We are not interested in the "history" of a body, which is what the CSG tree provides.

In a BSP tree, every node has two children. A node is associated with a plane, and its left child represents one of the half spaces created by this plane, and the right the other half space. We call the child with the half space in the direction of the planes normal the *front* child, and the other one the *back* child. BSP trees are typically used within computer graphics, where the planes might be chosen as e.g. walls in a room, to decide which bodies to render.

When used for CSG operations, the planes are instead chosen as coplanar with the surface polygons of the bodies. Then the front child is outside the body, and the back child is inside, given that the normal points out of the bodies. How the BSP tree of a body looks is not defined. It depends on which order the surface polygons are inserted when constructing the tree. But the functionality is the same, no matter how the tree was created. Let's explain how a tree is constructed, with help of the example shown in figure 6.5.

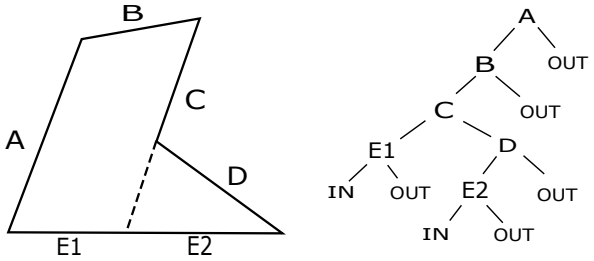


Figure 6.5: Example of a BSP tree construction.

For simplicity, the example is made in 2D, where lines are corresponding to polygons in 3D. The tree is constructed with the polygons $A-E$, placed in a list in that order. The first polygon in the list, A, is chosen as root. Then the rest of the polygons are checked against the plane coplanar to A. All of them are inside A, and are

recursively passed to build the subtree that has A's inside child as root. There, B is chosen as root, and $C - E$ are all passed along to build it's inside subtree, since they are all inside B. For C, all polygons cannot be classified to be on one side of it's plane. E has to be split up into $E1$, inside, and $E2$, outside. $E1$ is then passed along to build C's inside child. Since $E1$ is the only polygon passed, it will be a leaf node. D and $E2$ are passed to build C's outside child. D is chosen as root, and $E2$ will build its inside child, and become a leaf node.

It should be noted that convex bodies result in fully inbalanced, linear, trees. That is obvious, since for every polygon of a convex body, all other points of that body must be behind, or inside, it. This implies that there are better options than BSP-based CSG when dealing with convex bodies only.

With a BSP tree like this, it is possible to decide recursively if another polygon is inside, outside or partially inside this body, which is essential for CSG and contact detection. Beginning at the root, the polygon checks if its inside or outside that node's plane. If it is inside, and the node has an inside child, then the polygon is passed to that. If the polygon is inside, but the node does not have an inside child, the polygon must be inside the body. Accordingly, if it is outside, it is passed to the outside child if that exists, otherwise it is determined to be outside the body. If it at some point a split is necessary, the polygon is replaced by two new polygons, and both have to make their way down the tree. Both could end up with the same classification. Consider e.g. a polygon (line) that is fully inside the body in figure 6.5, crossing the dotted line. That will be split at the node corresponding to polygon C, but both the polygons will end up being inside the body.

By doing the above for every polygon of a body, it is possible to determine what part of that body that is inside, and what part is outside, of the compared body. We will refer to the operation *clipTo*, by an operation that removes everything of a body that is inside another body. To *invert* a body (making the space that was previously outside to inside, and vice versa), the normal directions of the nodes are negated, and their inside and outside children are swapped. By a certain sequence of those two operations, each one of the operations union, intersection and difference can be implemented. This is done in the `csg.js` library that will be explained shortly in the next section.

6.5.2 The Structure of the `csg.js` Implementation

In our implementation, a C++ ported version of the JavaScript library `csg.js` was used as a base. As the structure with the classes is the same in the ported version, we refer to E. Wallace's implementation[16] for a detailed documentation. In our own, modified, version, the classes are slightly changed, and some inheritance is introduced, but the structure is very similar. This section is meant to introduce enough details to the reader, to later be able to understand how the optimizations where carried out.

The most important structs are `Node`, `Polygon` and `Plane`, shown in figure 6.6.

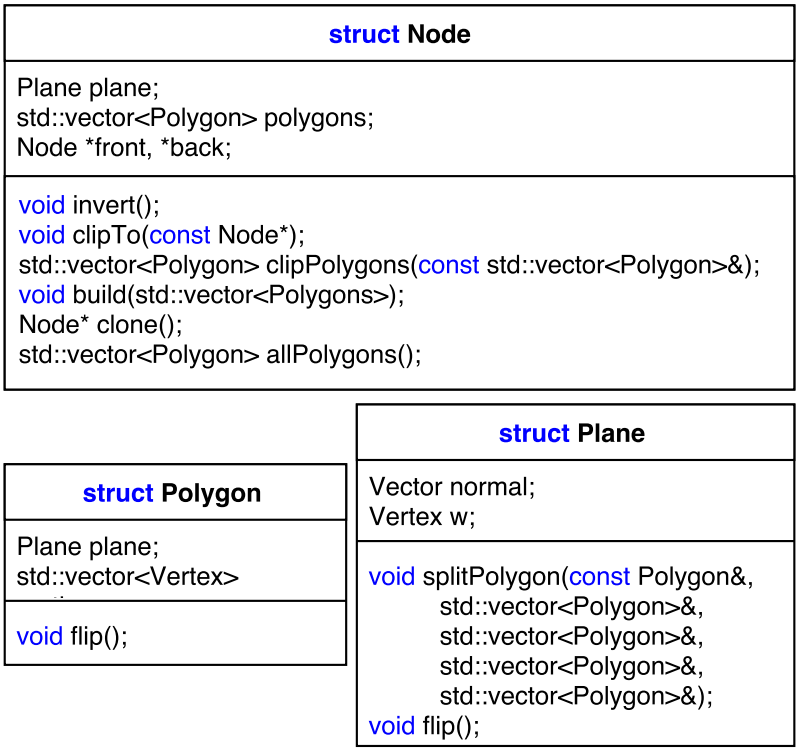


Figure 6.6: Some of the most important data structures of the `csg.js` library.

It shows a simplified definition of the structs, only showing what is considered to be the most important functions and couplings between them. The structs `Vector` and `Vertex` are essentially the same (they are in fact the same type in our implementation), basically consisting of three floating point coordinates. All member functions of `Node` are recursively defined. `invert` flips its polygons and planes, and every polygon that is flipped swaps the order of its vertices, and flips its plane as well. `build` constructs a tree based on a vector of polygons, or extends an existing tree, in the way described by the example of section 6.5.1. At every node, if it is new and does not have a plane, the first polygon's plane is chosen as that nodes plane. `Node::clone` and `Node::allPolygons` does what is expected, returns a copy of the tree and a vector of all its polygons respectively. The `Plane` member function `splitPolygon` is then used on every polygon in the vector, to classify the polygon, and push it to the correct vector. A polygon may be in front of, behind, coplanar with, or spanning (crossing) the plane. If it's spanning, it is split into two polygons that are pushed into the correct vectors.

Now to the most important function, `clipTo`, and its use of `clipPolygons`. We show the implementation, since it is important, and very short.

```
void Node::clipTo(const Node* other){
   polygons = other->clipPolygons(polygons);
   if(front) front->clipTo(other);
   if(back) back->clipTo(other);
}
```

Every node updates its polygons with a call to `clipPolygons` on the other tree. `clipPolygons` will, according to the explanation of the clipTo operation earlier, recursively remove every polygon in `polygons` that is inside `other`, and return that vector. `Plane::splitPolygon` is used at the nodes of `other` to classify every polygon sent to that node.

We are now ready for the implementation of the intersection operator, shown below:

```
inline static Node* intersect(const Node *a1 const Node *b1){
  Node *a = a1->clone();
  Node *b = b1->clone();
  a->invert();
  b->clipTo(a);
  b->invert();
  a->clipTo(b);
  b->clipTo(a);
  a->build(b->allPolygons());
  a->invert();
  Node *ret = new Node(a->allPolygons());
  delete a; a = 0;
  delete b; b = 0;
  return ret;
}
```

The operands are copied, to keep the original trees. After that the following four steps, are simple. By first inverting `a` and then doing `b->clipTo(a)`, everything of `b` that is outside of `a` is removed. Then the same thing is done the other way around. Even though most of the polygons in `b` may be removed when `a->clipTo(b)` is invoked, the nodes still remains, with the correct planes. The reason for the second `b->clipTo(a)` has to do with implementation details of `Plane::splitPolygon` and how to deal with coplanar polygons, and we refer to the documentation [16] for a full understanding of the algorithm. Then the remaining polygons of the trees are combined, and finally a new tree is constructed from them.

6.5.3 Optimizations

The functions `Node::build` and `Node::clipTo` are generally the most computationally heavy among the functions. This is because they require calls to `Plane::splitPolygon`, which contains almost all floating point operations in the algorithm.

It was early realized that the main part of the simulation time was spent on computing the intersections, even for simple geometries. According to Amdahl's law, an optimization of this part would therefore have a large impact on the total performance. Even if it is hard to do, it would probably be more effective work, than trying to optimize easier parts, that corresponded to a minor part of the execution time from the beginning.

In "The Intersection Algorithm" we describe how to completely remove the need of the `build` function during simulation. In "Parallel attempts" and "Multicore CPU attempts", different optimizations of the `clipTo` function are presented.

The Intersection Algorithm

In our initial implementation, the polygons where updated each time step, and then passed to construct the corresponding trees. But since the bodies are rigid, all polygons of a body will have a static relative position to the other polygons. Therefore, the structures of the trees are not changing during simulation. So an easy optimization is to build every tree initially, and then do a recursive update of the polygons and planes in the tree, based on the translation and rotation of the body. Since the Modelica MultiBody library provides the total rotation of a body, the original orientation of the bodies has to be stored. Thus a separate, updated tree is created and passed to `intersect` every time step.

Since the only thing needed from the intersection is the set of polygons, most of the work after the last clipTo operation can be removed. Basically, the only necessary operation after the last clipTo are the inversion of all remaining polygons. The function can then return the full set of polygons by concatenating the result of `a->allPolygons` and `b->allPolygons`.

Parallel attempts

That left the clipTo operations as major time consumers, and the goal became to parallelize the clipTo operation, which would require quite a comprehensive reconstruction of the intersection implementation. The restructuring necessary for parallelization would first be made for the CPU, to allow for easier debugging, before porting it to the GPU. To our knowledge this has never been done before. Traversal of BSP trees in computer graphics has been done for sure (Chambers et al. [36]), and attempts of parallel, general CSG has been made by Deldari et al.[2]. But they are both quite different from this case. To our understanding, traversal in computer graphics is of one tree, while in our case the number of unique trees may be as many as the number of bodies in the scene. And parallel CSG is based on traversal of the CSG tree (figure 6.4), breaking down operations to many operations on the primitives, which can then be carried out more efficiently. As mentioned earlier, this package is not restricted to the use of CSG operations to build bodies, and thus that was not a possibility either.

The basis of the parallel attempt is that, when calling `a->clipTo(b)`, all polygons of `a` are traversing `b` independently. The classification of one polygon has no influence on the other polygons. Thus, all the polygons of `a` could traverse `b` in parallel. In the original implementation, a recursion is done through `a`, starting traversals (recursions) of `b` at every node, resulting in a "double" recursion. To execute efficiently on the GPU, more polygons than those from one body are necessary. So the idea was to make the clipTo operations for all collision pairs in parallel. `b->clipTo(a)` would then mean that all the bodies represented by `b`, which probably includes multiple copies of many of the bodies in a crowded scene, traverse the corresponding trees of the `a` bodies. An illustration of the idea is shown in figure 6.7.

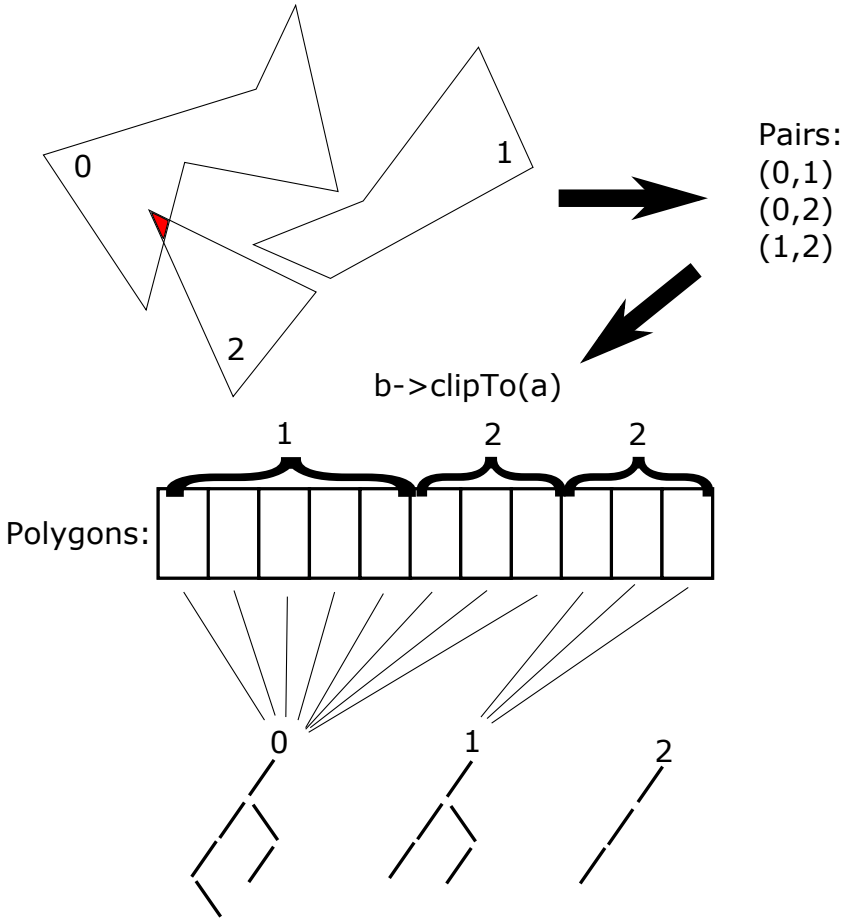


Figure 6.7: A 2D illustration of the parallel tree traversal.

The figure shows a simple 2D example of three bodies. The first operation in the intersection algorithm is `b->clipTo(a)`, where `a` is the first body in the pair and `b` the second. This results in 11 polygons traversing the trees in parallel, 8 of them traversing the tree of body 0, and 3 the tree of body 1.

The threads of a kernel would then each take care of one polygon, deciding whether it is inside or outside its target body. This algorithm would then be capable of handling any type of scene, as both few and complex bodies and many simple ones would generate lots of polygons for parallel traversal.

In the description of how this was implemented we will try to avoid as much details as possible, and instead focus on the ideas. To begin with, it was observed that the bodies could have two representations; 1.) a set of polygons for when their polygons should traverse other trees, and 2.) a tree which other polygons could traverse. Naturally, this separates the vectors of polygons from the trees of the bodies.

Set up

It was noted that trees are only traversed in an inverted state, in the intersect algorithm. Thus the trees could initially be setup and transferred to the GPU in an inverted state. The nodes in these trees did not need to have polygons, only planes, which were updated in every time step. By counting the number of nodes per body's

tree, which is constant during the simulation, the nodes (and their planes) could be stored consecutively in memory.

One vector of polygons is stored per body, and those are the polygons corresponding to the body's initial position. The bodies which might collide in the current time step then have their polygon vectors updated with the current position and rotation. As more than one collision with a single body may occur, the updated polygon vectors are copied to all other collisions where the same body is involved.

Each polygon also has to store information about which body's tree to traverse, and from which body it originates, for the return copy. The polygons are then transferred to the GPU, and after traversal they are marked as being either inside or outside the tree traversed. They are then copied back, and the polygons being outside (because of the inversion) are kept.

Traversal

It is straightforward to implement, except for the fact that polygons may be split. This was solved by pushing one of the resulting polygons to the end of the set of polygons, while the other replace the current polygon. The `AtomicAggInc()` function explained in 6.3.3 was used to push polygons to the end of the current set of polygons.

Because of this split, the traversal would not be done when each thread had traversed its polygon. Instead a new set of polygons has been generated, and the kernel is launched again over those, and so on. At some point, no more polygons are generated, and the traversal is complete.

It is likely that the amount of polygons per launch will decrease and get very small (<100) in the last launches. This motivates a hybrid solution, in which the last launches are replaced by CPU execution. Such a hybrid traversal would require two trees, one on the GPU and one on the CPU, which means the `Polygon` class would need pointers to both.

For some implementation details on the hybrid solution and the traversal, see appendix A.4.1 and A.4.2.

Iterative Version

This parallel traversal will obviously cause instruction divergence. Depending on the type of scene, nearby threads may traverse the same trees, but will most likely take different paths down the trees, and with different depths. The process of splitting polygons will also, naturally, lead to divergence. The amount of automatic variables needed per thread is also high, due to the complexity of `splitPolygonDevice`, which will heavily decrease the occupancy.

These drawbacks can be reduced significantly by changing the kernel slightly. The recursion can be replaced by a while-loop, reducing the required number of automatic variables, and some of the divergence. The implementation is shown in appendix A.4.3.

There is also some data divergence, when accessing nodes. With the while-based version, threads can simultaneously traverse different parts of the same tree, that are not very close in memory. The data divergence of both the while-based and original versions also depends on the type of scene that is present. If bodies are simple, threads within a warp will traverse different trees from the beginning. The

data divergence will therefore increase with every launch, since the order that the polygons have on splits is only kept within the number of active threads in a warp at split.

Iterative Version Variants

A drawback to mention is the transfer of polygons from CPU to GPU. There are many polygons to transfer, each with a size of approximately 100 bytes, and they need to be transferred between each clipTo operation. There may be an alternative way to avoid this, but not with the current structure of the algorithm. One of the transfers however, can be avoided. In the while version, all polygons are copied back after traversal, and are then distributed to their original bodies if they were to be kept. However, only the ones that should be kept need to be transferred. This reduced back-transfer is implemented in appendix A.4.4. Two different approaches to traversal are implemented, using this limited back copying. In one, the threads within a block shares the tasks, allowing threads that are done with their polygon to begin traversal of other polygons that have been pushed from splits. In the other, each thread is instead traversing its own polygon and all polygons that it generates at splits. Both approaches remove the need of multiple launches and hybrid execution. The implementations are motivated by the method used to minimize divergence when traversing the bounding volume hierarchy in T. Karras [41], but did not prove to improve performance.

We will refer to the algorithms presented here collectively as *optimized traversal*, whether they are run on the CPU or on the GPU.

Multicore CPU attempts

The CSG algorithm is quite complex, and even after being optimized, it demanded a significant part of the run time. This, coupled with the independence of each polygon in the optimized traversal, motivated us to try to develop a multicore t version of the narrow phase.

As time was rather limited, the decision was made to only make a multicore version of the narrow phase if it could be done without a major restructuring of the rest of the package. Two main attempts were tried; First OpenMP was used to parallelize the intersection calculation between all the pairs reported from the broad phase. That was in other words based on the version used before optimized traversal was introduced. This approach generated no speed-up and it was assumed to be because the cost of spawning new threads outweighing the performance gain. An attempt was made to work around this problem by using the open source libraries boost [9], to manually spawn a number of CSG worker threads at the start of the program and feeding them whenever work was available. This method did give some performance improvement, however a rare race condition bug caused the program to unexpectedly close at random times during a simulation. As time was short this bug was never found and the multicore version was not included in the final package.

6.6 Force Calculations

In this section, the physical model used to describe contact response is presented. Since the contact region is given from the CSG intersection explained in the previous section, it is given as a set of polygons describing its surface. Those polygons are convex, thus triangulizing the surface is trivial. In fact, in the optimized version, the polygons are divided into triangles directly at split, to keep a static size on polygons. Therefore, all algorithms and equations in this section assume a surface consisting of triangles.

In contact response, the collision is often classified as e.g. edge-to-edge, edge-to-face etc., which is then used to generate the correct contact force and direction. This is not possible here, however, as we have to consider arbitrary shapes. A collision type independent model is needed, and one such model is proposed by Nassauer et al.[15]. In that article, a volume and penetration depth dependent law is derived, based on Hertz model for contacts between spheres. The law assumes that the intersecting volume is small compared to the size of the bodies. The same paper also proposes models for damping and friction, which will, in addition to the normal force, be presented in this section. Additionally, we propose a simple method to describe rotational damping. Since the computation time needed for this type of force calculation is negligible to the time it takes to compute the intersection, no large effort has been made to optimize it.

Additional algorithms that were needed to complete the force calculations are presented in appendix A.5. To be more specific, those are algorithms for the volume, centroid and moment of inertia of arbitrarily shaped polyhedrons.

6.6.1 Communicating with Modelica's MultiBody Library

In Modelica's MultiBody library, equation 6.1 becomes

$$\mathbf{f} = m(\mathbf{a} + \dot{\boldsymbol{\omega}} \times \mathbf{r} + \boldsymbol{\omega} \times (\boldsymbol{\omega} \times \mathbf{r}))$$
$$\mathbf{t} = \mathbf{I}\dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times \mathbf{I}\boldsymbol{\omega} + \mathbf{r} \times \mathbf{f}$$

where all quantities are taken with respect to the *frame* of the object, and $\mathbf{r}$ is the position of the center of mass with respect to the frame. The frame is the origin of the local coordinate system of the body, in which those quantities are expressed. The user can access position, velocity, angular velocity, and what else may be needed, from the frame. The rotation of the object can be communicated in various ways. For the sake of simplicity, the rotation matrix $\mathbf{R}$ has been used, which rotates global coordinates into local ones. Since $\mathbf{R}$ is a rotation matrix, it is orthogonal, and $\mathbf{R}^{-1} = \mathbf{R}^T$. Thus it is trivial to transform local coordinates to global ones as well. Moment of inertia, center of mass and mass are computed initially, and since the bodies are rigid, the force calculation needs only to calculate the force and torque with respect to the frame, and pass them to Modelica in frame local coordinates.

6.6.2 Normal Force

The derivations in the work of Nassauer et al.[15] lead to

$$F = Ek\sqrt{Vd} \tag{6.2}$$

where F is the magnitude of the normal force, E is Young's modulus of the objects, V is the volume of the intersection, d is the penetration depth and

$$k = \frac{4}{3\sqrt{\pi}}.$$

The force is then applied at the centroid of the intersecting volume, and it is shown that this is actually a generalization of Hertz model. To determine the force direction, constant pressure is assumed within the intersection. The direction can then be determined by weighing each triangle's normal with it's area, sum over the triangles from one object (see figure 6.8), and normalize:

$$\mathbf{n}_f = \frac{\sum_j A_j \mathbf{n}_j}{\sum_j A_j} \tag{6.3}$$

A_i and $\mathbf{n}_i$ are the area and normal direction of triangle i respectively. Since this requires the knowledge of which polygon originates from which object, all polygons of one of the objects are marked initially in the intersection calculation.

This results in an elastic behavior, similar to that of an object floating in water. The only difference is the assumption of constant pressure, which, would only be a valid assumption for objects not extending too far down in the water.

The penetration depth is defined as the extension of the overlapping region in the direction of $\mathbf{n}_f$, and can be found as

$$d = max_j(\mathbf{n}_f \mathbf{p}_j) - min_j(\mathbf{n}_f \mathbf{p}_j) \tag{6.4}$$

where $\mathbf{p}_i$ is the position of vertex i and j ranges over all vertices in the overlapping region.

How the normal direction and penetration depth are found is illustrated in figure 6.8.

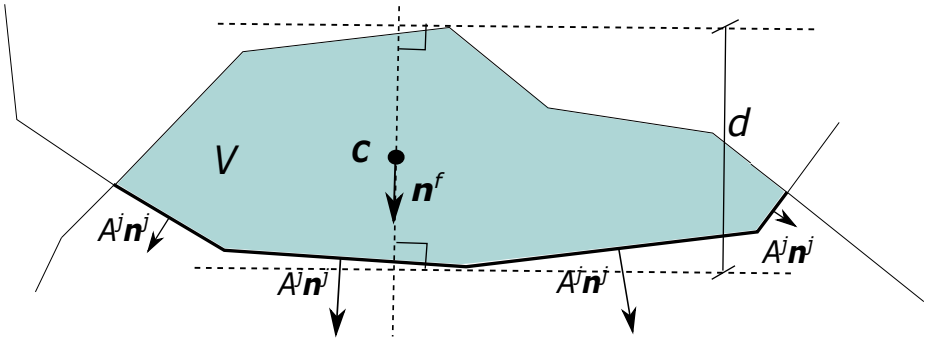


Figure 6.8: A 2D illustration of the determination of normal direction and penetration depth. C is the centroid.

6.6.3 Damping

Energy dissipation can be introduced by modifying equation 6.2 to

$$F = Ek\sqrt{Vd}(1 + c_d v_d)$$

where c_d is the damping constant and v_d is the relative velocity between the bodies in the orientation of $\mathbf{n}_f$. The velocities are calculated at the centroid of the intersecting volume.

6.6.4 Friction

The classical Coulomb friction model turns out to be very hard to implement in DEM-like simulations. There are many reasons for this, the most important being that, in the case of static friction, all other forces acting on the body have to be known.

Instead, the following, velocity dependent, model is used:

$$F_f = ((2\mu_{s*} - \mu_k)\frac{x^2}{x^4 + 1} + \mu_k - \frac{\mu_k}{x^2 + 1})F \tag{6.5}$$

where $x = v_t/v_s$ and

$$\mu_{s*} = \mu_s(1 - 0.09(\frac{\mu_k}{\mu_s})^4).$$

μ_s and μ_k are the coefficients of friction for static and kinetic friction respectively. v_s is the velocity for transition from static to kinetic friction, and v_t is the magnitude of the relative tangential velocity between the bodies. It is chosen as the relative velocity at the centroid of the intersecting volume, and can then be found by projection onto the plane of which $\mathbf{n}_f$ is normal. A plot of equation 6.5 is shown in figure 6.9.

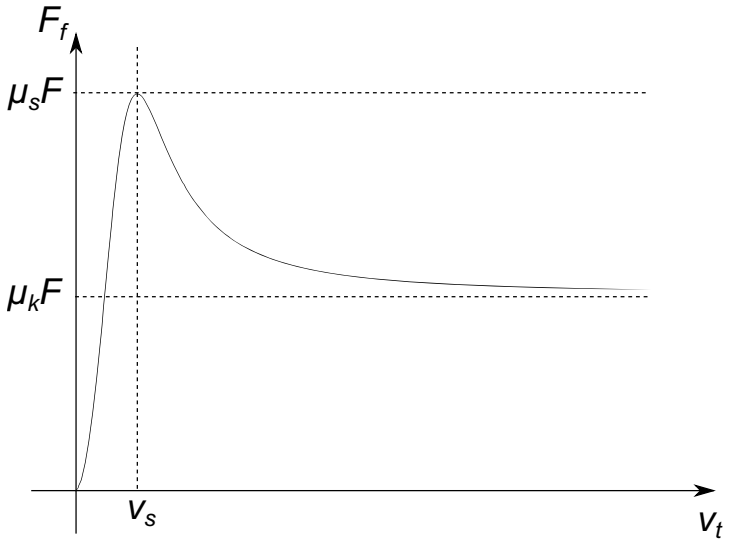


Figure 6.9: The used friction model, v_s is the velocity for transition from static to kinetic friction, and v_t is the magnitude of the relative tangential velocity between the bodies.

As the figure shows, the friction increases fairly linearly with the velocity below v_s, modeling static friction, and then falls off to a constant, kinetic friction. That the friction is continuous is good for numerical reasons.

6.6.5 Rotational damping

To simulate friction caused by relative rotation at the contact point, a model with damping effects is here proposed. It applies a torque in the opposite direction of the relative angular velocity at the centroid of the intersecting volume. This relative angular velocity is taken in the orientation of $\mathbf{n}_f$. The resulting torque should also depend on the normal force, and the area of contact. A simple, linear, model then reads the following:

$$\mathbf{M}_r = -AFc_r(\boldsymbol{\omega}_1 - \boldsymbol{\omega}_2)\mathbf{n}_f$$

where $\mathbf{M}_r$ is the resulting torque, A is the area of the contact region, c_r is a constant for calibration of the model, and $\boldsymbol{\omega}_i$ is the angular velocity of body i. The area can be approximated by projecting the triangles from one body onto the plane of which $\mathbf{n}_f$ is normal, and summarize:

$$A = \sum_j A_j \mathbf{n}_j \mathbf{n}_f$$

The fact that the angular velocity is the same in any point of a rigid body means that the frame's angular velocity can be used. From the theory of coupled forces, it follows that the torque at the contact region can be translated to the frame, without changing its effect.

6.6.6 Multiple Intersecting Volumes in a Single Collision

Since the bodies that are colliding may be concave, there may be multiple disjunctive intersecting volumes within a single pair of bodies. Thus, the intersection given from the CSG operation will consist of many volumes, but they will be given as a set of triangles, without any information about which volume each triangle belongs to. An algorithm is needed, that investigates if there are disjunctive intersection volumes, and if so, divides the intersection correspondingly. Then separate forces and torques are calculated and applied to the bodies, one by one.

The easiest way to realize that such a separation is necessary, is by considering cases when the penetration depth is affected. This is illustrated by an example in figure 6.10. It becomes clear from the figure that the penetration will be too deep, resulting in a contact force that is too large.

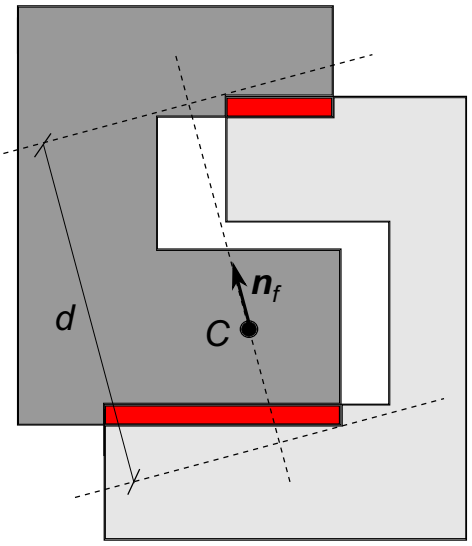


Figure 6.10: Illustration of why detecting multiple contact regions is needed.

Since the number of triangles in the intersection in general is small compared to the number of triangles in a collision, and since not every narrow phase leads to an actual intersection, little effort was made to make the solution efficient. It was suspected that a brute-force solution would be sufficient, and not affect the overall performance too much. A short description of the algorithm is presented below.

The Intersection Splitting Algorithm

The algorithm assumes that the original intersection consists of one or more closed volumes, which are to be represented as a vector of intersections. The algorithm goes through all polygons in the original intersection and checks the polygons vertices against the vertices of each intersection. If no match is found, a new intersection is created. If one match is found, the polygon is added to that intersection. If multiple matches are found, the corresponding intersections are merged, and the polygon added to the result. This is illustrated in figure 6.11.

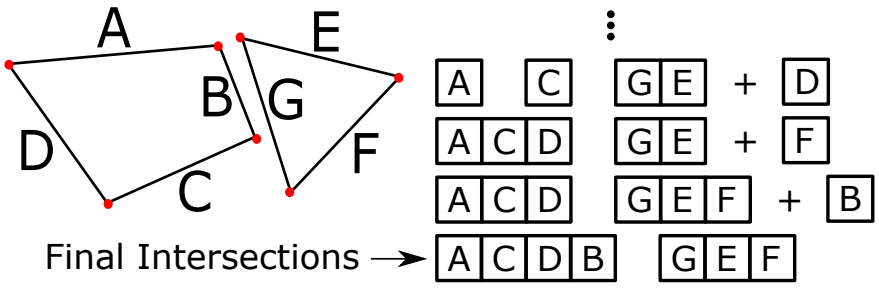


Figure 6.11: 2D illustration of the later stages of the algorithm

In pseudo code, the algorithm looks like this:

```
for polygon in original intersection
   if(nbrOfMatches(polygon) == 0)
      create new disjoint intersection and push polygon
   else if (nbrOfMatches(polygon) == 1)
      push polygon to that intersection
   else
      concatenate all matching intersections
      push polygon to the concatenated intersection
return all disjoint intersections
```

This continues for all polygons in the full intersection, and when all polygons have been compared, the resulting vectors will contain all polygons that make up a disjointed part of the intersection.

6.7 Additional Debug Window

To allow for easier debugging while developing the package, an additional window was developed. This was used to investigate contacts in detail, by giving the opportunities to

- Toggle normal forces, torques, centroids, AABBs and triangular contributions to normal directions.
- Toggle transparent or wired objects and intersections.
- Pause the simulation.
- Navigate in 3D space, to get a convenient view.

A sample from the window is shown in figure 6.12, where the triangular contribution to the normal direction in the intersection between a sphere and a cuboid is viewed.

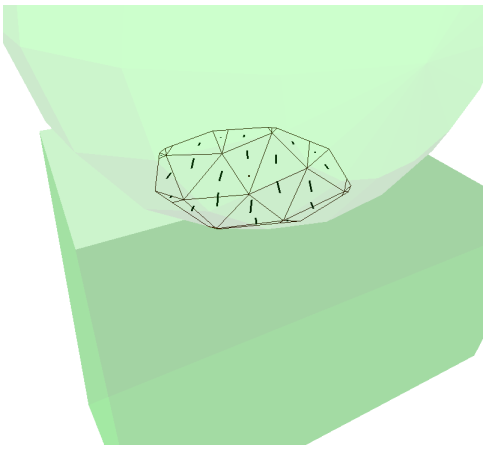


Figure 6.12: Viewing triangular contribution to the normal direction in the intersection between a sphere and a cuboid.

The window was updated after every force calculation, which resulted in another, interesting feature. When using adaptive step size methods for the integration, they are as mentioned trying to increase the step size when the problem is not stiff. In this case, that practically means searching for collisions. This process, of how the integrator is going backward and forward in time, can be visualized with this window.

The window was developed using the OpenGL library FreeGLUT, that implements a simple windowing API for OpenGL[24].

6.8 Additional Package Features

The package has a few additional features, that makes creation of triangular meshed geometries easier for the user:

- Predefined meshes for primitives such as cube, cylinder, prism, sphere and randomized objects (for irregularity).
- CSG operations (union, intersection and difference) between meshes. Those are implemented using the `csg.js` library. This allows for parametric representation of bodies such as the wheels of the Geneva Mechanism.
- Importing of meshes defined in DXF-files.
- Extrusion of 2D meshes into 3D ones.
- Triangularization of concave polygons.

For more details on the full package, see Elmqvist et al.[25].

Chapter 7

Performance and Behavior Tests

In this section, the tests used to evaluate the package are presented. A test is a Modelica model, designed to evaluate some aspect of the package. There were different reasons for the different tests made, some were meant to validate behavior and others to evaluate performance of the broad and narrow phases.

Test models are easy to construct, since objects can be added to a scene without having to worry about object id's or anything similar. Everything is taken care of externally by the package. The objects only need specified geometry and initial conditions, and if desired, be connected to other parts of Modelica. In section 6.8, some information is given on how to define meshes.

For the animations, DS extended the capabilities of their animation window in Dymola. This window generates a user defined number of frames, and only uses actual output states. It also has a connection to the Modelica MultiBody library, and a coordinate system, gravity, and things such as external forces can be visualized.

Unless otherwise stated, E =10 MPa, c_d =0.3, v_s=0.1 m/s, μ_s =0.6, μ_k=0.3, c_r =0.02 and ρ =1000 kg/m^3. When gravity is present, the gravitational acceleration is $g = 9.81$ m/s^2 in the negative y-direction.

7.1 Billiards

To test the broad phase performance, billiard-like simulations were carried out. Since, as mentioned, no interaction between spheres and triangular meshed objects is supported, there is no table in the simulations. The balls are moving free in 3D space, without gravity. To get a relatively dense scene, that is easy to scale, the balls are placed as in the beginning of a game of eightball. The model is constructed so that the only parameter needed to increase the number of objects in the scene is the number of rows, n_{rows}. The number of objects in the scene, n, is then approximately $n_{rows}^2/2$. In the beginning of a simulation, one ball is placed in front of the 2D "pyramid", and given an initial speed to hit it. A picture from a simulation of 100 rows, or 5051 balls, is shown in figure 7.1.

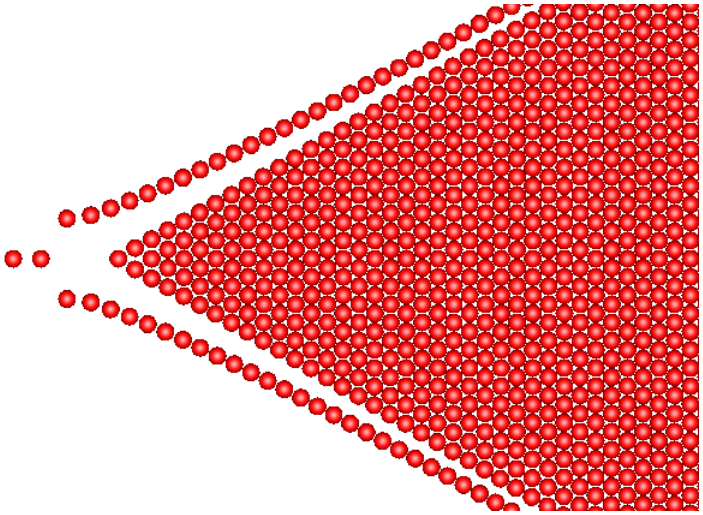


Figure 7.1: Example showing the top of a billiard simulation of 100 layers, or 5051 balls (5050 in the pyramid + 1 in front).

7.2 Simple Contact Tests

To verify that some of the basic mechanics for triangular meshed bodies are working, a very simple example was used, consisting of a cube of size 1 m and m =1000 kg, and a flat ground (large cuboid). The ground is connected to the Modelica `world` object, and thus cannot move, while the cube is affected by gravity.

By dropping the cube from above, with no initial movement, the cube should bounce on the ground, and at some point find rest, due to damping. When the cube is at rest, the force acting on it should be $\mathbf{f} = (0, mg, 0)$. The frame of the cube is positioned in its lower corner, which at rest is coinciding with the origin of the global coordinate system, see figure 7.2.

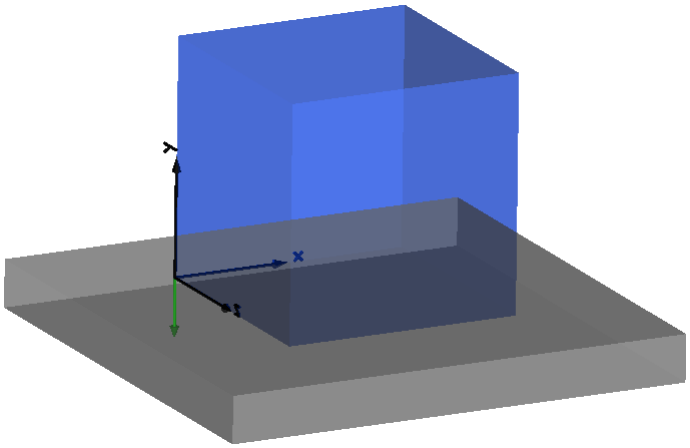


Figure 7.2: The cube at rest. The green arrow shows the direction of the gravitational acceleration.

Thus, since the centroid of the contact region should be placed at $\mathbf{r} = (1/2, 0, 1/2)$, the resulting torque should be $\mathbf{t} = \mathbf{r} \times \mathbf{f} = (1/2, 0, 1/2) \times (0, mg, 0) = \frac{mg}{2}(-1, 0, 1)$.

The friction model can be tested by giving the cube initial velocity in the plane, and the rotational damping by giving it initial angular velocity in the y-direction.

7.3 Multiple Contacts Test

To test multiple contacts within a single collision pair, the test shown in figure 7.3 was used.

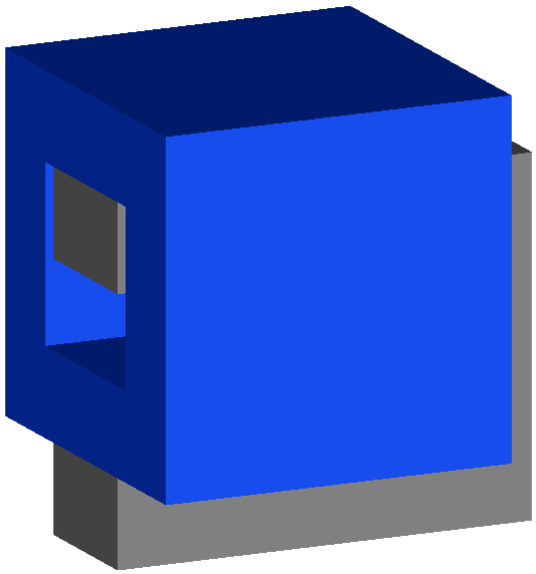


Figure 7.3: The test for multiple contacts

It is based on the example illustrated in figure 6.10. With gravity present, and the "fork" body fixed, the "ring" body should receive a force in the positive y-direction with a magnitude of $f = mg$, where m =750 kg. Without splitting algorithm, the penetration depth would become very large, and the ring body would receive a very large force.

7.4 Pile of Prisms

To test the narrow phase on many triangles with low recursion depth, a pile of prisms was used, see figure 7.4.

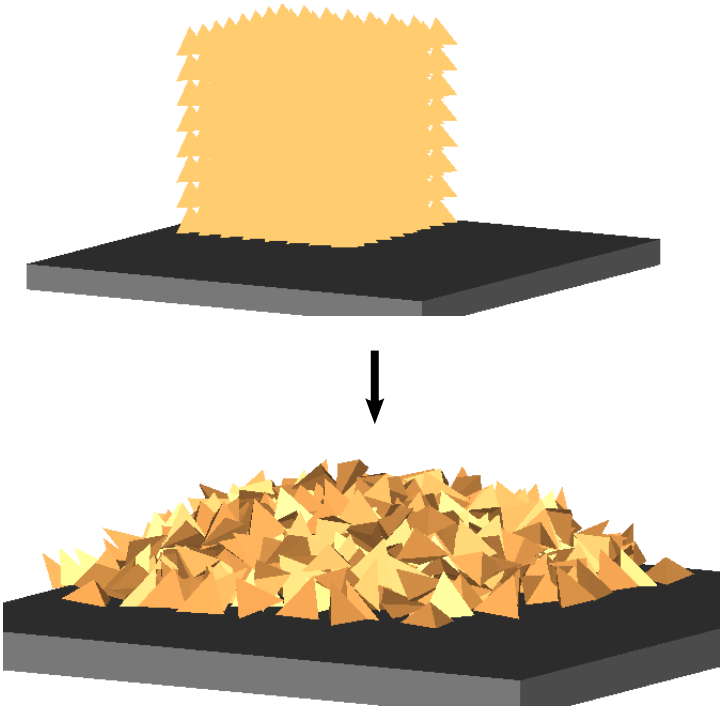


Figure 7.4: A pile of 512 prisms in initial state, and when they have fallen down.

They are initially released above the ground, to fall together into a pile. They are placed in a cube, and the model can be scaled by the side. In the figure, $8 \times 8 \times 8 = 512$ prisms are simulated. The ground is fixed, and gravity is present. Each prism only has four triangles, and four nodes in its tree. As mentioned earlier, since prisms are convex, their trees will be fully unbalanced. There are probably more efficient ways to make this sort of simulations, but this is a test. Further on this example will be referred to as *prisms*.

7.5 Bucket Digging in a Pile of Irregular Stones

To make a more realistic simulation, a pile of irregular stones (Belgian block stones) was created, into which a bucket could be digging. That would demonstrate both the capability of handling many triangular meshed bodies, possibly concave, and the interactive possibilities of the package. By giving the bucket a fixed trajectory, which is possible using the Modelica MultiBody library, the force needed to grab the stones can be calculated, even the in-plane forces. The stones are simply created as cubes with a stochastic perturbation of the corner positions, and they are affected by gravity, while the ground (together with the "railing") is fixed. This simulation is shown in figure 7.5.

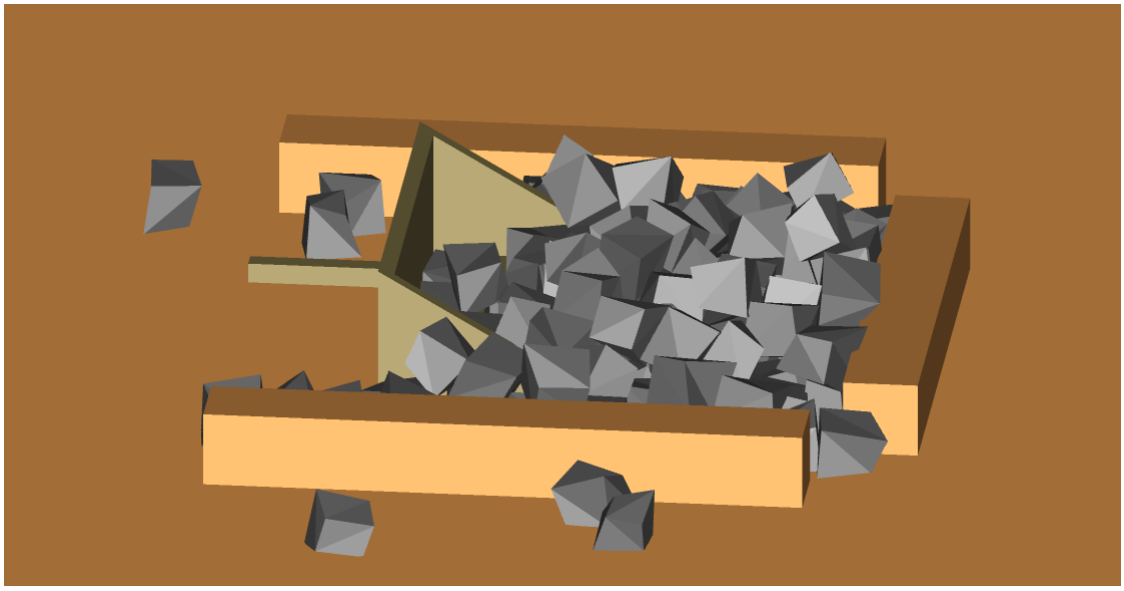


Figure 7.5: Bucket digging in a pile of 125 stones.

7.6 Geneva Mechanism

The Geneva mechanism has been used since long ago to achieve intermittent motion (Bickford, 1972). It is used, for example, in mechanical watches and in film projectors to move the film. A snapshot from a simulation is shown in figure 7.6

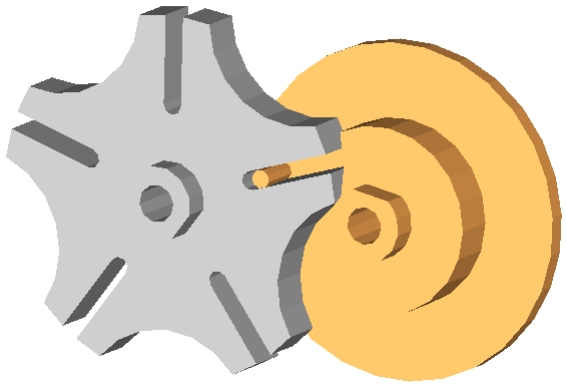


Figure 7.6: The Geneva Mechanism. The bronze colored wheel is the drive wheel, and the gray wheel is the output wheel.

The drive wheel (bronze in the figure) has a pin which rotates the output wheel (gray) when in contact in the slots. When the pin is not in contact, the output wheel lock its locking surface by sliding against the locking ring of the drive wheel.

The dynamics is thus quite complex and there is a fast change in the acceleration when the drive pin enters and leaves the slot. It is also a good example for testing of the narrow phase. Deep trees with many triangles (744 and 564 in the wheels in the figure), and only two objects.

To construct this example, joints from the Modelica MultiBody library were used, giving the necessary constraints on the system.

7.7 The Tippe Top Toy

The Tippe Top is a special toy that has fascinated a lot of physicists for a long time. A detailed analysis is presented in e.g. Rutstam [7], which helped to figure out suitable initial conditions for the simulations. It consists of a hollowed sphere with a stem, as can be seen in figure 7.7. The figure also shows the geometry used in the simulations.

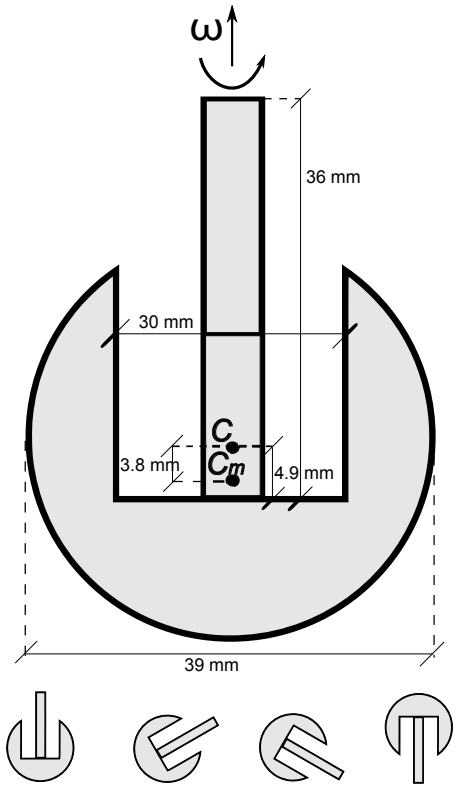


Figure 7.7: Top: The Tippe Top's geometry. C is the center of the sphere, and C_m the center of mass. Bottom: Illustration of inversion process from left to right.

As a result of its special geometry, its center of mass is located below the center of the sphere. When spun, the Tippe Top will invert itself, as a consequence of its geometry and friction, and spin on the stem. Since the behavior of the Tippe Top is quite complex, a successful simulation of it would serve as a good indicator of the potential of this package.

The toy is affected by gravity, and placed above a fixed ground, with a small initial tilt and a high initial angular velocity.

Chapter 8

Results

This section is divided into results regarding behavior and results regarding performance. The simulation statistics from simulations in both sections will be presented, however. That generally consists of simulated time, integration method, time step (if fix step size is used), number of scene force calculations and number of collisions. With scene force calculations we refer to the number of times that an update of forces in the whole scene was performed. How that number relates to the number of time steps heavily depends on the integration method. The number of collisions in a simulation is the total number of intersection calculations which results in an intersecting volume. This means that two bodies might generate more than one count when colliding, if they stay in contact for more than one intersection calculation, which is most often the case. The number of scene force calculations and the total number of collision calculations are rounded to three significant digits.

The simulations for behavior validation in this section were run using the original intersection algorithm unless otherwise stated. The optimized traversal contains a bug that that has not yet been found, which results in physically incorrect behavior in some particular cases.

For a note on timing and the used hardware, see the introduction of chapter 4. For other, more detailed timing purposes, the `QueryPerformanceCounter` function of the Windows API for high precision timing was used [4].

8.1 Behavior Validations

In this section, we present results from tests that were carried out to test behavior and physical correctness of the package. Those tests were made using the bouncing cube, multiple contact, pile of rocks and Tippe Top examples.

8.1.1 Simple Contact Tests

In the first test, the cube was dropped from 1 m above the ground. The simulation time was $T = 3$ s, using the Rkfix4 integration method, with $dt = 0.001$ s. For this, 15 000 scene force calculations were carried out, of which 8 640 had a collision.

Computation time was 0.434 s, using the optimized traversal. The results are shown in figure 8.1.

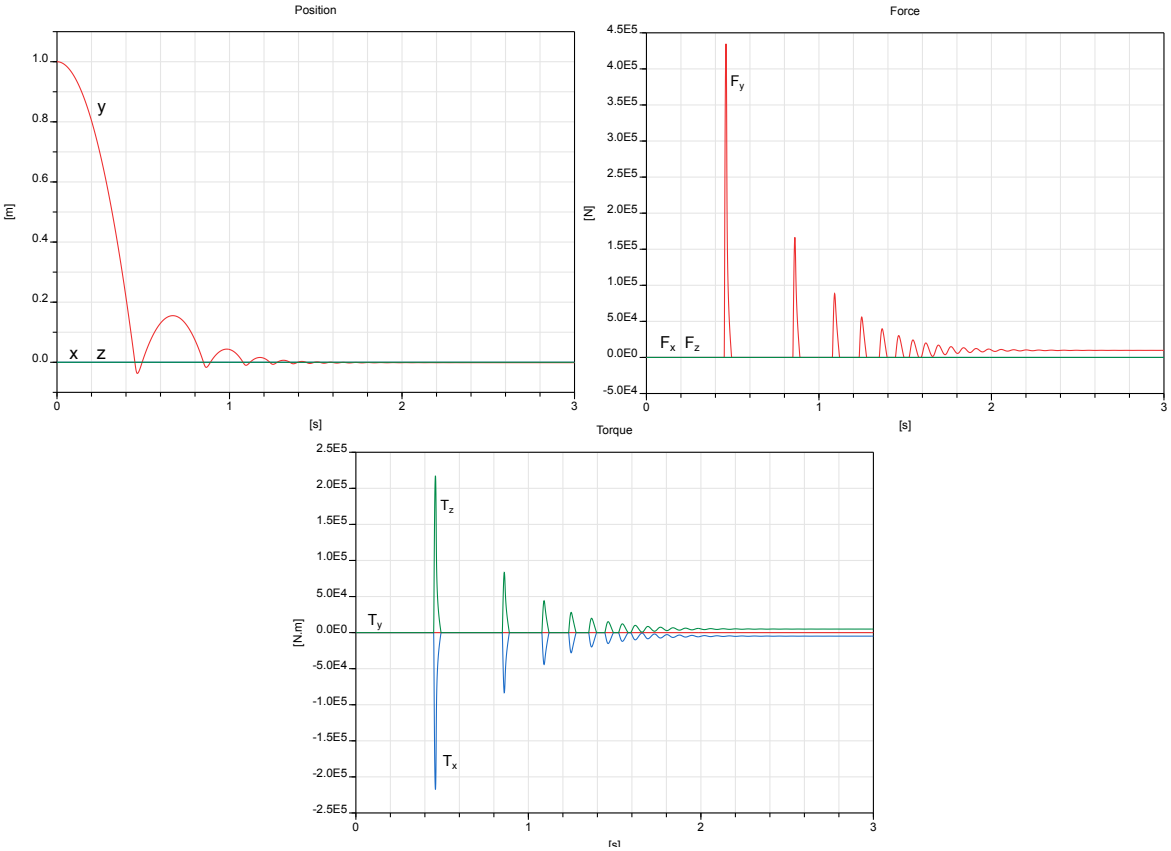


Figure 8.1: The results from a drop test of the cube.

The y-component of the force, top right, stabilizes at 9 810 N, and the x- and z-components of the torque, bottom, stabilizes at $\pm$4 905 Nm.

For the friction test, the cube was placed on the ground, with an initial velocity of 1 m/s in the x-direction. Rkfix4 was used again, with $T = 2$ s and $dt = 0.001$ s. 10 000 scene force calculations were carried out, with 10 000 collisions, which took 0.508 s with optimized traversal. The velocity is shown in figure 8.2.

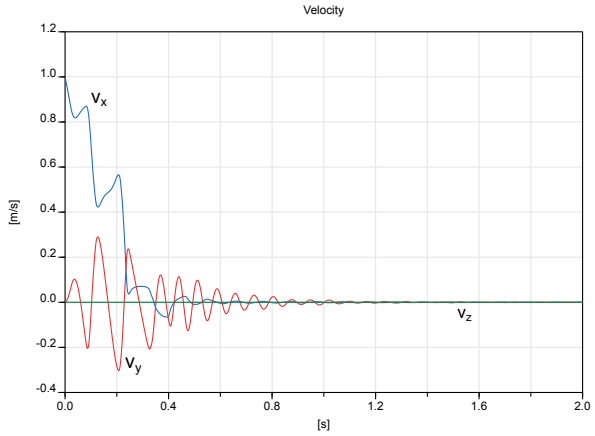


Figure 8.2: The velocity of the cube in the friction test.

To test the rotational damping, the cube was again placed on the ground, but now instead with an initial angular velocity of 1 rad/s in the y-direction. Rkfix4 was used, with $T = 6$ s and $dt = 0.001$ s. 30 000 scene force calculations took place, with 23 000 collisions, which took 1.66 s with optimized traversal. The angular velocity is shown in figure 8.3.

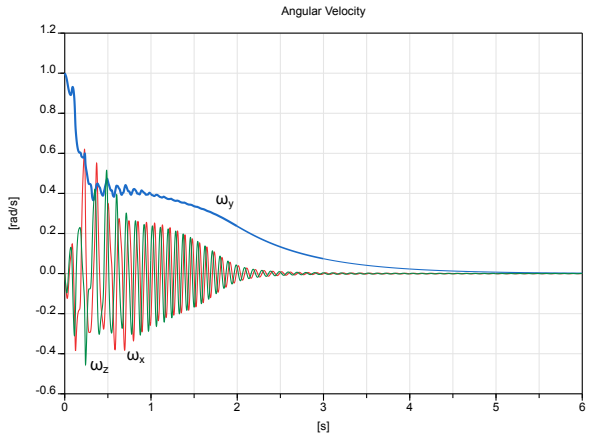


Figure 8.3: The angular velocity of the cube in the rotational damping test.

8.1.2 Multiple Contact Test

The test of the intersection splitting algorithm was performed using the Rkfix4 method, with $dt = 0.01$ s and $T = 2$ s. With splitting enabled, the simulation had 1 030 scene force calculations, with 1 990 collisions, and took 0.437 s. Without the splitting algorithm, the simulation had 1060 scene force calculations, with 158 collisions and took 0.290 s. Results are shown in figure 8.4.

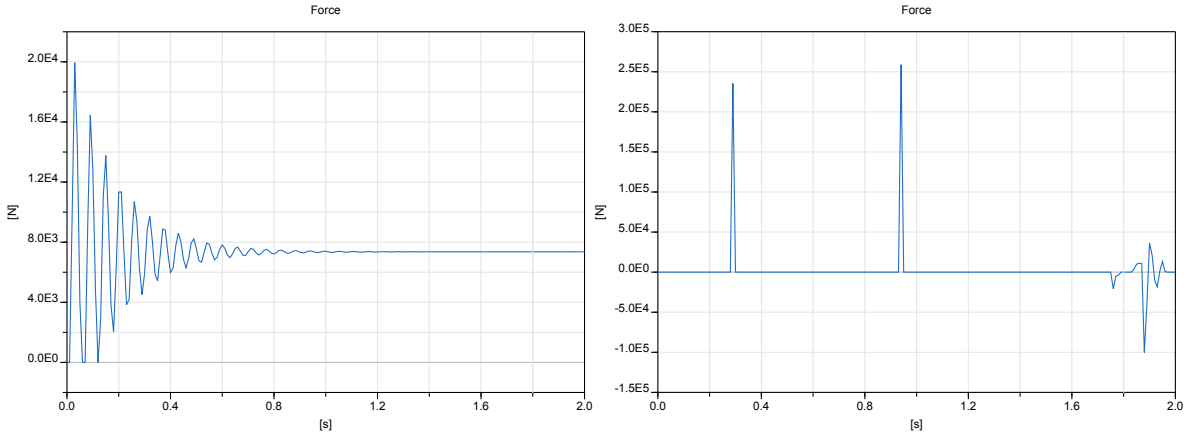


Figure 8.4: The resulting vertical force on the ring body. Left: with intersection splitting. Right: without.

The asymptotic value of the y-component of the force is 7357.5 N when using the splitting algorithm.

8.1.3 Bucket Digging in a Pile of Irregular Stones

The simulation was performed using the Rkfix4 method, with $dt = 0.001$ s and $T = 4$ s. This simulation took 26 min and 30 s, and required 25 700 scene force calculations during which 7 120 000 collisions took place. The resulting forces on the shovel are displayed in figure 8.5.

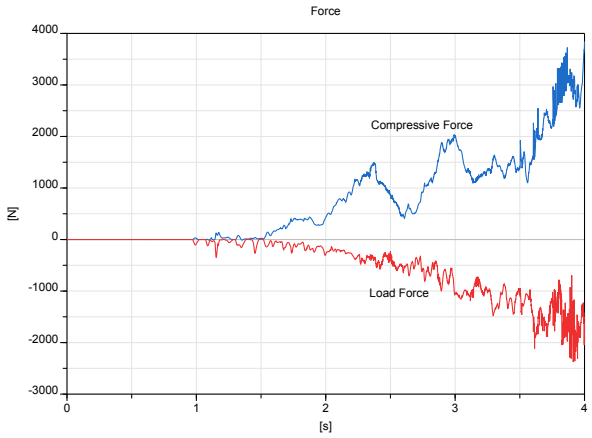


Figure 8.5: The resulting forces when the bucket is pushed into the pile. Compressive force is the horizontal force, and load force is the vertical force.

8.1.4 The Tippe Top Toy

An inversion of the Tippe Top toy was successfully simulated, with the following initial conditions:

- A tilt of 0.1 radians.

- Angular velocity of 180 rad/s in the y-direction.
- Positioned 0.5 mm above the ground.

The simulation was made using the Rkfix4 method, with T =3.3 s and $dt = 0.1$ ms. It made 167 000 scene force calculations, with 75 300 occurring collisions, and took 749 s. Figure 8.6 shows snapshots from the simulation.

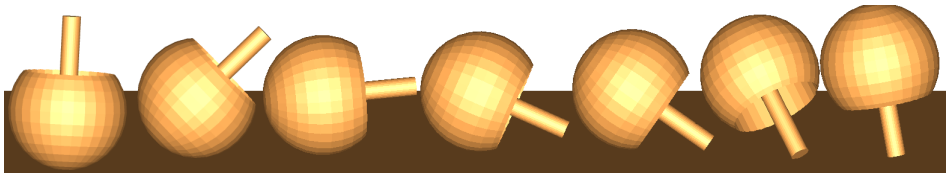


Figure 8.6: Snapshots from the Tippe Top simulation

The y-component of the angular velocity is shown in figure 8.7, expressed in the local coordinate system.

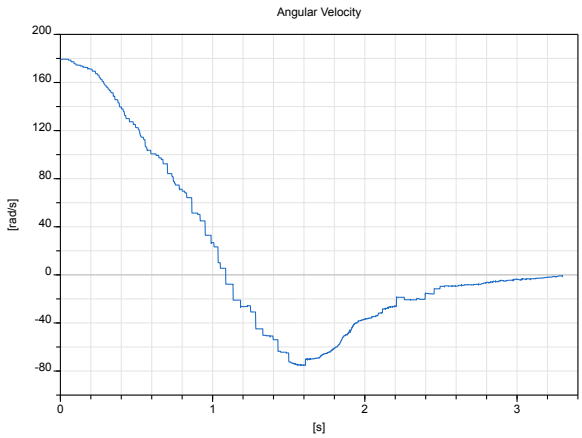


Figure 8.7: Angular velocity of the Tippe Top during inversion

8.2 Performance Tests

8.2.1 Broad Phase

In this section, the billiard example is used to make detailed tests of the broad phase implementation. The prisms example is then used to test it in a triangular meshed case.

Billiards

When presenting the results, we will use the total simulation time for $n_{rows} = 100$ (5 051 bodies), using the Rkfix4 method with T =20 s and $dt = 0.01$ s, as reference. This required 16 000 scene force calculations, with 47 100 000 collisions. Without a broad phase, using an $O(n^2)$ brute-force solution, the simulation time was 44 min and 50 s.

With the first implementation, using cell sizes of 1 m, the simulation time was 4 min and 15 s.

By applying Thrust's sort algorithm, together with the special unique algorithm, the simulation time was further reduced to 2 min and 54 s. The first sort, of intervals, needed more than 160 000 elements before it was advantageous to do it on the GPU. The same number for the second sort, of collision pairs, was 190 000.

After parallelizing the Morton encoding, the simulation time became 2 min and 49 s.

Experiments where carried out to count the number of intervals generated for different sizes of the base cells. The balls had a diameter of 1 m. The average number of intervals per body, for different base cell sizes, is shown in table 8.1.

Table 8.1: Average number of intervals generated per body

Size [m]	Intervals
2	4.5
1	8.0
0.5	8.0
0.1	5.2
0.01	3.2
0.001	7.8

The usefulness of the Morton encoding kernel turned out to heavily depend on those results. Sample results for 100 rows, or 5 051 bodies, are shown in table 8.2, for cell sizes 1 m and 0.01 m. Transfer times are included.

Table 8.2: Timings of the Morton encoding

Size [m]	CPU [ms]	GPU [ms]
1	1.8	1.3
0.01	0.6	1.1

For cell size 0.01 m, the extra sorting took 1.1 ms (1.6 ms sorting 40 000 elements, instead of 0.5 ms sorting 16 000), and removing took about 0.1 ms, which did not occur at all for cell size 1 m. For cell size 1 m (8.0 intervals per body), the GPU performed better on about 1 800 bodies and more, but reached a limit of the speed up factor of about 2.5.

With cell sizes of 0.01 m, and by going back to CPU execution, the simulation time was further reduced to 2 min and 11 s.

For the best version, 25 %, or 33 s, of the total simulation time was spent in the broad phase. In the first implementation, that number was 62 %, or 2 min and 38 s. The rest (narrow phase, force calculations and Dymola integration and storage of variables etc.) then took 1 min and 38 s for the optimized, and 1 min and 37 s for the original. In the brute-force case, 96 %, or 43 min and 10 s, were broad phase, and the rest took 1 min and 40 s. Thus the broad phase has been accelerated by a factor of 4.8 compared to the first implementation, and 78 compared to the brute-force solution.

This also means that the actual speed up of the last steps were larger than they looked, as by Amdahl's law, their total impact cannot be so large when more than 50 % is spent outside the broad phase. Also, some of those correspond to small fractions of the broad phase, which motivates a look at the following measurements.

Table 8.3 shows how the time is distributed between the different tasks for the first implementation, and the fastest.

Table 8.3: Detailed timing of the broad phase, for the original implementation, and the best, optimized version. Numbers within parenthesis are % of broad phase.

Task	Original [ms] (%)	Optimized [ms] (%)
Morton encoding	1.8 (15)	0.6 (24)
First sort	3.3 (28)	0.6 (24)
Finding pairs	0.9 (8)	0.5 (20)
Second sort	5.7 (48)	0.5 (20)
Unique	0.1 (1)	0.3 (12)
Total	**11.8** (100)	**2.5** (100)

The first sort were operating on 40 000 and 16 000 elements for the original implementation, and the fully optimized one, respectively. The same numbers for the second sort were 73 000 and 42 000. For both simulations, the narrow phase, including force calculations, took about 0.5 ms per iteration.

With the final optimization of the broad phase, simulations of larger models, with tens of thousands of balls, could be simulated in reasonable time (in a matter of minutes).

Prisms

To see how much time the broad phase takes for a triangular meshed simulation, where the narrow phase and the contact response are far more complex than for spheres, the prisms example was used. A simulation of 2 197 prisms (13×13×13), with the optimized traversal, was accelerated by a factor of 2.6 by the optimized broad phase compared to the brute-force solution. The distribution of time in the different phases for the two examples is shown in table 8.4.

Table 8.4: Table showing how the computation time was distributed between the different phases in the prisms example, for the brute-force solution, and the fully optimized one.

	Brute force [%]	Optimized [%]
Broad phase	73	8
Narrow phase	25	65
Other	2	27

"Other" is mainly represented by Dymola integration, storage of states, and communication between Dymola and the external library.

8.2.2 Narrow Phase

In this section, performance measurements on the prisms and Geneva mechanism examples will be presented. We begin with some measurements of the general speed up of the narrow phase. Then the results of GPU performance experiments are presented. Last of all, some results from tests on the impact of the intersection splitting algorithm are presented.

General Acceleration

The measurements are carried out on three different versions of the narrow phase. The first is the original version, which basically was the first working implementation, where as much as possible was untouched in the `csg.js` library. Many optimizations were then made on that package, mainly consisting of a reduced intersection algorithm and fitting the data structures better to our need. We refer to that version as the "Reduced intersection". Then there is the last, fully optimized version, which includes the optimized traversal with serial CPU execution. The last two versions also have the multiple contacts algorithm implemented, which may influence their performance negatively compared to the original.

Prisms
The prisms example was run with 512 (8×8×8) prisms, using the Rkfix4 integration method with $T = 1$ s and $dt = 1$ ms. During the simulation, the pile falls for about 0.4 s, and collides during the rest, with a total of 3 330 000 collisions, out of 6 690 scene force calculations. All measurements are made with the best version of the broad phase algorithm. In table 8.5, the simulation times for the three different versions of the narrow phase are presented.

Table 8.5: Simulation times for the three different versions, for the prisms example. Numbers within parenthesis are % of total time spent on the narrow phase (including force calculations).

Narrow phase version	Simulation time (%)
Original	14 min and 5 s (96)
Reduced intersection	4 min and 53 s (87)
Fully optimized	1 min and 56 s (68)

The total speed up factor of the narrow phase, from the original version to the fully optimized, was 10.3. The optimized traversal accelerated the narrow phase with a factor 3.22.

Geneva Mechanism
The Geneva mechanism example was simulated using the Rkfix4 method as well, with $T = 5$ s and $dt = 5$ ms. The angular velocity of the drive wheel is fixed on 1 rad/s, which means it will rotate 5 radians, or 80 % of a full turn, including one "step" for the output wheel. That meant 2 500 collisions out of 5 160 scene force calculations. The simulation times are presented in table 8.6.

Table 8.6: Simulation times for the three different versions, for the Geneva mechanism example. Numbers within parenthesis are % of total time spent on the narrow phase(including force calculations).

Narrow phase version	Simulation time [s] (%)
Original	54.1 (99)
Reduced intersection	21.6 (98)
Fully optimized	3.46 (89)

On this example, the speed up factor of the narrow phase was 2.5 between the original and the reduced. The optimized traversal resulted in an additional speed-up of of 6.9 times, for a total of 17.4. The lack of a multiple intersection algorithm for the original version did not seem to have any impact on the physical results.

The optimized intersection algorithm used only triangular polygons. This was also tested on the reduced intersection algorithm, and it increased the total computation time with 40 % on the Geneva mechanism example.

Parallel Traversal on the GPU

The prisms and Geneva mechanism examples were used for the GPU performance measurements as well, with the same settings as in the previous section.

The prisms example had a total of $512 \times 4 + 12 = 2060$ triangles in the scene. During the end of the simulation, after the pile had hit the ground, the sequence of polygons for traversal in the first clipTo operation (see section 6.5.3) in average looked like this: 15 900, 6 600, 1 000, 80, 3. The values in the sequence are the number of polygons to traverse at each launch, generated from splits in the previous launch. Before the pile hit the ground, due to the regular positioning of the prisms, no splits where made, and the sequence only consisted of 18000 polygons. The best result that was achieved, was by launching kernels when more than 5000 polygons needed traversal, and doing smaller groups on the CPU. The total traversal was then slightly faster, average traversal on the GPU being 3.1 ms, and 3.4 ms on the CPU, for the first clipTo operation. But together with the CPU/GPU transfer which took 1.0 ms, a speed-up was not possible. The time it took to transfer the updated planes of the nodes was negligible. In the simulations, the traversals corresponded to 75% of the narrow phase (force calculations excluded). Of that, the first clipTo operation in average took 75%, the second 20%, and the last 5%. The total simulation times of the prism tests (512 and 2197 polygons) can be seen in table 8.7.

Table 8.7: The simulation times for GPU and CPU execution of prisms

	CPU time	GPU time
Prisms (512)	1 min 56 s	2 min 8 s
Prisms (2197)	9 min 10 s	9 min 39 s

For the "Geneva Mechanism" example, 1308 (744+564) triangles were present. That resulted in an average sequence of 744, 650, 50, 8 in the first clipTo operation. The best GPU performance was achieved with sets larger than 500 being executed on the GPU. To make a test with deep traversal and enough polygons to saturate the GPU, what could be considered as a very artificial test was created. 12 pairs of Geneva mechanisms were simulated simultaneously, resulting in 12 300 triangles in the scene, and an average sequence of 8 930, 7 800, 600, 96. Given the knowledge from earlier measurements on the Geneva mechanism example, it is sufficient to present the total simulation times, which can be seen in table 8.8.

Table 8.8: The simulation times for GPU and CPU execution of Geneva mechanism

	CPU	GPU
Geneva	3.46 s	7.47 s
Geneva ×12	44.1	41.9 s

In the measurements, kernels of size 256 were launched, which appeared to give the best performance.

Intersection Splitting Algorithm

The intersection splitting algorithm was run on two models with bodies of different geometric complexity; the prisms and the Geneva mechanism. The simulations were run using the Rkfix4 method, with $dt = 0.001$ s and $T = 5$ s. Two sizes of the pile of prisms were used: $3 \times 3 \times 3 = 27$ and $5 \times 5 \times 5 = 125$. Results are presented in table 8.9, where the percentage shows how much slower the simulation became.

Table 8.9: Intersection splitting algorithm's effect on simulation times

	Prism Pile 27	Prism Pile 125	Geneva Mechanism
Without splitting	17.4	230	3.51
With splitting	22.8	310	3.48
Slow down (%)	31	35	-

Chapter 9

Discussion

9.1 Behavior Validation

9.1.1 Simple Contact Tests

The dropped cube behaves as expected, see figure 8.1, with correct asymptotic values, equal to the analytic solutions. Due to the linear damping model, the energy dissipates exponentially in time.

That smoothness of dissipation is not present in the friction test. This is natural, since when the friction force is first applied, it makes the cube want to tilt, as it would if it had enough speed, and the friction was high enough. Since the friction gets smaller when the cube rises, it will glide back down, get a high friction force, tend to rise again, and so on. In figure 8.2, it can be seen that v_y changes much (the cube accelerates), when v_x looses speed fastest, which is when it is rising.

Figure 8.3 shows a similar behavior of the angular velocity in the rotational damping test. This is a bit less intuitive however. Both the rotation test and the friction test may be affected by another fact. The cube needs to find a state in its y-position, where its penetration depth and volume corresponds to the asymptotic normal force. Thus, minor oscillations of the position are initially present in the y-direction, before it gets damped out, which may affect the experiments.

Something worth mentioning, even if it may be a performance consideration, is the fact that the friction test is simulated slower than the drop test, even if it only takes 2/3 of the time steps. The reason is that in the drop test, the cube spends a relatively large amount of time (43%) in the air, resulting in no collisions during those steps. The friction test has more collisions, and thus requires more narrow phase executions.

9.1.2 Multiple Contact Test

That the number of collisions is nearly twice as many as the number of scene force calculations indicates that the splitting works as expected. It is not exactly two times the number of scene force calculations, because of a slight bouncing before the vertical movements are dampened.

Figure 8.4 indicates the importance of the algorithm. As expected, the force seems to converge correctly with the algorithm, again approaching the analytically correct values, while an unstable behavior is shown without it.

9.1.3 Bucket Digging in a Pile of Irregular Stones

The results from figure 8.5 shows a behavior that seems reasonable. While the exact behavior is hard to verify, without a real world example to compare to, the results shows no unexpected behavior. The bucket is penetrating further and further into the mass of stones, and as it does so, more and more blocks gets pushed into the stop on the backside. This should correspond to a steadily increasing force, necessary for further forward motion, which is present in the figure (compressive force). The sudden dips in the figure are explained by the motion and shape of the rocks. When the bucket pushes on the rocks, they will rearrange themselves until they are forced into a position where all (or most) rocks are locked by other rocks or walls. The pressure then builds until the rocks have enough force acting on them to overcome the "lock". They will then suddenly rearrange themselves to reduce the potential energy from the pressure. After a little while, the rocks will once again find themselves in a locked position where the pressure will build until the next "restructuring" occurs. The load curve shows the vertical force exerted on the bucket from the rocks. When the bucket is pushed into the pile, more and more rocks gets pushed into the bucket adding to the vertical force. The erratic movements of the red curve is due to the movements of the rocks; bouncing, rolling, falling off, piling on etc.

This test shows that a large number of objects can be successfully simulated and can interact with parts from the Modelica MultiBody library. It also shows that it is possible to mix complex bodies (bucket) with simple ones (rocks), convex with concave, and large with small, with no problems.

9.1.4 The Tippe Top Toy

The simulation is very sensitive in whether it will fully invert or not for different initial conditions. This is part of reality, since it will not invert on every try in real experiments. However, we suspect that the dicretized shape has an impact on this as well, which is indicated by the angular velocity in figure 8.7. It takes rectangular steps, which is a result of a small bouncing, which probably comes from when the Tippe Top turns over it's edges. Those bounces also explains why only 45 % of the scene force calculations have collisions.

It is interesting to see that the angular velocity gets below zero before it stops, which was expected based on other research on the toy. The angular velocity is zero when the stem is horizontal, but this is in the local coordinates, around its stem. In

the global coordinate system, it still rotates around the y-axis. When inverted, the angular velocity has locally changed direction, but since it is inverted, this globally corresponds to a rotation around itself in the same direction as before.

9.2 Performance Tests

9.2.1 Broad Phase

Billiards

The scene had an average of 0.59 actual collisions per scene force calculation and body, which should be considered as dense, since far more AABB-collisions than this should have occurred. The total time spent outside the broad phase for the three versions varied between 97 s and 100 s, which is expected since small variations in the simulation times always occur as the operating system is not idle.

Thrust's parallel radix sort proved very useful, which is strengthened by table 8.3, even if the sorts were applied on different numbers of elements. It is also clear that the extra 0.2 ms from the special unique algorithm that was required, is hidden by the boost in the second sort. It should be noted that the high number of elements required for useful GPU radix sort means that most simulations performed don't use the GPU for sorting.

The results shown in table 8.1, regarding the performance of the Morton encoding was found in a late state of development, and it was for long believed that, probably due to some error in the implementation, the number of generated intervals per body was bound to be close to 8. This means that table 8.3 showed an unfair performance for GPUs on the Morton encoding, having a significant impact on the total performance. But with the new results, it seems that a more varied number of intervals could actually be achieved. However, to find a good cell size may not be trivial, and the GPU version can still be used by measuring the average number of intervals generated each iteration, and activate it when it gets close to 8. To have cell sizes larger than the objects is obviously bad, since it will generate more AABB checks than necessary. The number also seems to grow again when getting very small compared to the body size. With more time, this would be measured and analyzed in more depth. With a statistical basis, it should be possible to automatically scale the cell sizes after the average body size in the scene, to get as few intervals as possible.

The overall results of the broad phase for the billiard simulations are good, meaning in the end it corresponded to a marginal part of the total simulation time. In table 8.3 we can also see how well the work has been distributed between the steps in the algorithm, after the optimizations. There is no major bottleneck. And the accelerations are well motivated by the fact that the broad phase, even after the first broad phase algorithm was implemented, stood for 62 % of the computation time. The narrow phase stood for a minor part of the simulation time, even in the fully optimized version, as expected.

Pile of Prisms

The total broad phase acceleration had an important impact on the prisms example. But with the optimized solution of both broad and narrow phase, the broad phase only corresponds to 8 % of the total execution time, while the narrow phase takes 65 % of it (see table 8.4). Considering that this is about how large the simulations get at the moment, in terms of number of triangular meshed bodies, it becomes clear that the narrow phase is what should be in focus. Since prisms are the simplest polyhedrons that exist, this underlines the fact that the narrow phase is the most computationally heavy part. For the example of a bucket digging in a pile of stones, the broad phase would probably take significantly less than 8 % of the time, even if it was scaled up to the same size. That being said, the broad phase would still be the theoretical bottleneck if it had an $O(n\log(n))$ time complexity, and the narrow phase an $O(n)$. But with the parallel radix sort, the time complexity is actually reduced to being linear (also theoretically, for large n's) for the broad phase as well [22]. However, its impact would be larger (compared to the narrow phase) in less dense scenes.

9.2.2 Narrow Phase

General Acceleration

To begin with, the optimizations of the narrow phase were well motivated, as the narrow phase for both examples originally stood for over 95 % of the total execution time. Even after the accelerations, it is responsible for most of the computation time. That it is so for the Geneva mechanism is not a surprise, since there are only two bodies, so practically no time is spent on the broad phase or on the storage of variables and integration in Dymola. Updating vertex positions is a part of the communication process between Modelica and the external library, and that is probably responsible for most of the 11 % that is not narrow phase, in the fully optimized version.

It is interesting to see that the reduced intersection gave more in the prisms example than for the Geneva mechanism (speed up factors were 3.3 and 2.5). Generally, optimizations are expected to have a larger impact on the more complex geometry. A probable reason for this result is the fact that the trees are not rebuilt in the "Reduced intersection" version. In the Geneva mechanism case, two tree constructions were removed per scene force calculation. In the prisms case, the number of tree constructions were probably a lot more than 512 per scene force calculation in the original version. It would be interesting to measure that number, but the feature has not yet been implemented. But the number of actual collisions per scene force calculation was $3328048/6686 \approx 500$, and the number of colliding AABBs is naturally higher than that number, especially in such a crowded scene.

We should also discuss the fact that the optimized traversal gave such a large speed-up in serial execution, which was not expected. With one exception, mentioned below, the same work is being done. No in-depth analysis has been made on the fact, but some possible reasons come to mind. Except from general effects that the required restructuring might have made from being a more efficient implemen-

tation, some possibly contributing factors are listed below.

- When using the optimized version polygons became restricted to triangles as opposed to general shapes, to have a static number of vertices per polygon. It has earlier been mentioned that only triangles are used. At splits this results in more polygons, as a triangle, when split, almost always generates one four-vertex polygon. Knowing that only triangles are used reduces the need of vectors, and thus the number of dynamic memory allocations/deallocations. This can also have a negative effect on the performance, since more vertices are created that need storage. The higher number of polygons also results in more new traversals, and possibly more polygons to work on in the force calculations. In section 8.2.2 it was noted that only using triangles results in 40 % slower execution time for the Geneva mechanism with the reduced intersection algorithm.

- Bodies were separated into trees and vectors of their polygons. Before, each node had its own vectors of polygons. The whole clipTo traversal was then based on replacing that vector with concatenated vectors that where created during the traversal. Thus, both dynamic memory allocations/deallocations, and copying has been reduced a lot.

- As mentioned earlier, a "double" recursion was performed in the original version. In the optimized parallel version, that has been reduced to a single recursion, one polygon at a time. On the GPU, the recursion was even fully removed.

- In the process of separating trees and polygons, a way of keeping track of which bodies' polygons had been updated was implemented. That made it possible to only update the polygons of a body once, and then if that body occurred in multiple contact pairs, the updated polygons could be copied from that one's. This should of course have been possible to implement in earlier versions as well, but it was then not considered important enough. With the optimized version however, it turned out to give a small performance boost, for crowded scenes. But for the Geneva mechanism example, that did not have an impact, since only one collision pair can exist. Still, that example had the largest speed up measured, indicating that previous points are of greater importance.

Parallel Traversal on the GPU

First of all, the total simulation times serve as good indicators on the traversal performance for the Geneva mechanism. This can be confirmed since according to table 8.6, the narrow phase makes up almost all of the execution time. And the traversal corresponded to 75 % of the narrow phase, for the much simpler traversals, in the prisms example.

That the Geneva mechanism simulation did not benefit from GPU execution did not come as a big surprise. It plainly had too few polygons. When scaled up however, very similar performance to that of the prisms simulation was achieved.

That was very surprising, and is most probably a coincidence, since the problems are so different.

In the prisms case, each tree is as mentioned fully unbalanced. They are also as shallow as an unbalanced tree can be, with only four nodes. This should generate quite small divergence in terms of how deep threads have to traverse. Recall that divergence in terms of threads taking different paths down the trees was removed in a loop-based solution. However, a split most likely causes divergence, and the time a split takes in relation to a full traversal is of course larger if the traversals are shallow.

That the traversal for the prisms example was distributed as 75/20/5 % for the three clipTo operations also sounds reasonable, since after the first traversal, all the collision pairs which are not colliding are not further analyzed.

In the Geneva mechanism example more work per polygon is needed. This potentially hides the transfer times better, and even if the trees of those concave wheels are better balanced, the divergence in terms of traversal depth is probably larger. Even if the Geneva mechanism needed more splits, and thus more atomic operations per polygon, those numbers (the sequences) are fairly similar. Note that the number of splits per polygon in general decreases in each launch:

$$\frac{6600}{15900} > \frac{1000}{6600} > \frac{80}{1000} > \frac{3}{80}$$

which probably is a result of the fact that polygons from splits start their traversals somewhere (a bit down) in the trees (not in the root), decreasing the probability of new splits.

Those speculations motivate more detailed investigations of e.g. average recursion depth, divergence in terms of recursion depth, the influence of splits in execution time, performance on other tree structures (where the actual tree structures are visualized), and so on. But since it was realized that this solution most probably would not generate any useful results without a huge investment of work, and due to a limited amount of time, that was never done.

Intersection Splitting Algorithm

As can be seen in table 8.9, the intersection splitting has some performance effects for very simple geometries, which disappears when the geometric complexity increases. This is to be expected from the formulation of the algorithm, which has the time complexity $O(n^2)$, where n is the number of vertices in the intersection. Since the number of vertices in an intersection is proportional to the number of polygons in the intersection, the algorithm time is dependent on how many polygons are in the intersection. As intersections most often are geometric primitives, or at least simple in relation to the two bodies colliding, the time the splitting algorithm takes should for most geometries be negligible. The exceptions are very simple geometries, as prisms, where a splitting algorithm is not needed, as they are convex.

9.3 GPU Conclusions

9.3.1 Broad Phase

For the number of bodies present in current simulations, the broad phase proved to have little use of GPU execution. GPU accelerated sorting only starts giving a large impact when the number of bodies is larger than what is currently practical to simulate using Dymola and Modelica. It is possible that another broad phase algorithm would be better suited for parallelization on GPUs. But the implemented algorithm was chosen because of its simplicity, and a faster algorithm would not affect the total simulation time much. This follows from Amdahl's law and the fact that less than 25 % is spent in the broad phase with the current implementation, even in the special case of spheres.

9.3.2 Narrow Phase

A meaningful speed-up was not achieved with any tried solution, on the GPU. It should then be noted that the task from the beginning is very hard, as an acceleration that should cover both DEM simulations and complex contact mechanics between few objects is sought.

Tests were made on the two extreme types of scenes, and similar results were achieved - the GPU performed on the same level as the CPU, given a scene with enough polygons. It may be possible to make a better implementation of this idea, but it is more probable that a slightly different approach is needed. For example, the data transfer could be decreased by porting more of the work to the GPU. It may be possible to do the whole intersection on the GPU, without transferring data back between the clipTo operations. Operations such as inversion of polygons would not be a problem. The main obstacle that we see, is that each body can appear in multiple collision pairs. That will require a lot of device-to-device copying, or a completely different structure of the solution.

Another slight change that could be tried out, given the time, would be to stop threads that need to split their polygons. Since splits are computationally heavy, they are probably one of the major limiting factors, when they appear in divergence. Splits could then be made in a second, separate kernel, where the fact that the maximum number of generated polygons per split is two, could be used to remove the need of atomic operations. Little divergence would appear in such a kernel, and it would mainly occur at construction of the new triangles. The calculations of the new vertices, which is the heavy part, would be made without divergence.

9.4 Future Work

As the package is only a prototype, there is much left to do before it can be considered as finished. Other than bug fixing and general increased robustness of the package, below is a list of some things that we think should be implemented. Some of them are trivial changes, both theoretically and practically.

- As mentioned in section 9.2.1, a statistical analysis of the cell size used for Morton encoding should be used for automatic, scene-dependent, scaling.
- A better contact model for spheres would be easy to implement. Friction, damping and rotations of spheres can be done with the same principles that are now used for the triangular meshed bodies. Also, a better force law for spheres could be used, e.g. Hertz model: $F = \frac{4}{3}E\sqrt{R}d^{3/2}$ where E, R and d are effective values.
- It should be possible to make simulations with a mix of triangular meshed bodies and parametric spheres. An easy solution would be to have the triangular meshed surfaces of the spheres calculated originally, and then apply them when spheres are colliding with triangular meshed bodies. This would be most beneficial in cases where the majority of the collisions are sphere-sphere, such as spheres in a bowl/box. The simulation time of a real billiard simulation would probably be dominated by the contact between all the balls and the table, so that the fact that sphere-sphere collisions are parametric does not give a significant boost. (This could in practice be solved by not having a table bottom, only table walls, since no out of plane forces should occur if the balls are not rolling)
- Optimize communication between Modelica and the external library. For example, triangles in Modelica are stored as indices into a vector of vertices, which is the most data efficient representation. Such a representation may be possible to use for the triangles in the external library as well. At the moment, every triangle has its own vertices, which are updated when needed. This makes splitting easier, for example. Otherwise, one has to make some kind of distinction between original triangles and the resulting triangles from splits, or gradually insert new vertices into the vector.
- The resulting meshes of CSG operations contains more triangles than necessary, due to all the splitting. For better performance on those bodies, and possibly for other bodies as well, retriangulation of the mesh could reduce the number of triangles, making intersection calculation faster. This is connected to the next idea.
- Currently, the tree of each body is built initially, as mentioned, with the triangles stored in the tree. Using optimized traversal, the triangles are then extracted and stored in a vector, while the tree structure is saved for traversals. There is no need for the triangles to be split according to their own body's geometry, as they are only used when traversing trees of other bodies. The triangles should be stored in their initial way, and the tree be built on those, but only storing planes from the beginning. That would speed up usage of concave bodies when using optimized traversal, since splitting only occurs then.
- Multithreading of the narrow phase, including force calculations, should be entirely possible, and useful in scenes with more than one pair of bodies with colliding AABBs per scene force calculation.

- It should be possible for the user to define more physical constants, and make it possible for such physical constants to vary between objects. E.g. Young's modulus and friction coefficients.
- The model for rotational damping, proposed in section 6.6.5, does not take into account the fact that the friction force is most probably not uniformly distributed across the contact area. A better solution would be to integrate over the contact area, and apply the friction model from section 6.6.4 on the triangles. F should then be replaced by the corresponding local contribution to the normal force, and v_t by the tangential component of $\boldsymbol{\omega} \times \mathbf{r}_i$, where $\mathbf{r}_i$ is the position of triangle i relative to some fixed point, e.g. the centroid of the contact region.

9.4.1 Alternative Methods

A possible acceleration of the package could be to detect whether a scene only has convex bodies, or possibly, in a mix of concave and convex bodies, make that check for every collision pair. This would remove unnecessary intersection splitting, and perhaps using a completely different algorithm to faster determine the intersecting volume between convex bodies. With BSP trees, it is trivial to check whether a body is convex, by checking if the tree is fully unbalanced, i.e. linear tree, which would only have to be done once.

Also, alternative implementations of the CSG intersection operation should be looked for, or even other methods for the narrow phase, since it still takes the majority of the simulation time for all simulations. Even if some ideas for how to improve the performance on the current algorithm is presented in section 9.3.2, there may be other methods that are better suited for parallel execution.

Part IV

Summary and Final Words

This thesis has detailed two ways to incorporate GPUs in Modelica; by automatically generating GPU code for Modelica functions, and by using external functions and objects in a contact handling package. Both subjects culminated in papers (Hilding et al. [8] and [25]) for the 11th International Modelica Conference, September 21-23, 2015, Paris, and will be presented there. The contact handling package showed enough promise to be proposed as a starting point of a new part of the MSL for contact handling and DEM simulations. DS future plans involve combining the contact handling package with flexible body capabilities [25] and a newly developed 3D simulation setup environment [44]. A short summary of the the two parts:

For the automatic GPU code generation, a prototype was created, that achieved accelerations of up to 5 times. By investigating auto-generated code from Modelica models, and how such code should be changed to run in parallel on the GPU, it became possible to compile instructions on how the output from the Dymola compiler should be changed. The prototype uses `algorithm` sections containing (possibly nested) for-loop(s) in order to recognize parallelizable parts, as the algorithm section remains intact in the generated C code. The contents inside the for-loop(s) are then translated into a kernel, that is launched from a generated wrapper function. The wrapper is based on the range(s) of the for-loop(s), and the contents of the kernel, and manages memory allocations, launch parameters, data transfers and other necessities. An important performance limiter was the fact that states had to be transferred from GPU to CPU memory and back at each iteration, for integration in Dymola. This were partly avoided by allowing translation of an inline integration routine written in a certain format. In that way, the model could be integrated locally on the GPU, allowing for fewer transfers. This made it possible to accelerate more models, that would otherwise not have achieved any performance boost. As manually writing the inline integration may be very complex, it should only be a temporary solution for this prototype. This should instead be automatically generated, in order to maintain Modelica's high-level programming style. It is definitely still an early prototype, but the fact that speed-ups are possible without any knowledge in GPU programming or model specific optimization, is promising.

A prototype contact handling package was created, capable of handling anything between few, very complex bodies, and thousands of simple ones. The contact model was capable of handling complex mechanical behavior, when this was tested. A Modelica framework for synchronization, communication and interaction between an external library and the rest of Modelica was developed. This was built on the Modelica MultiBody Library, with new bodies that were assigned triangular meshes, describing their geometries. The idea was that the external package in each iteration calculates the forces and torques acting on each individual body, and then letting Modelica use this information to integrate the equations of motion for the system. The forces and torques were calculated using a force law depending on the intersecting volume between two colliding bodies, and models for friction, damping and rotational damping. To find the required intersection, the CSG intersection operation based on BSP-trees, was used. To allow for large numbers of particles, a broad phase algorithm was implemented, based on the use of Morton codes. Much time has been spent on optimizing the narrow and broad phases. Among those optmizations, were attempts at GPU accelerations. GPU acceleration was possible

in the broad phase, on large numbers of bodies, while no significant speed-ups were achieved on the narrow phase. In the end, the limiting factor is the narrow phase, except for spheres, which were specifically handled by parametric representations.

Bibliography

[1] **A Data-Parallel Algorithm Modelica Extension for Efficient Execution on Multi-Core Platforms**
Gebremedhin M., Hemmati Moghadam A., Fritzon F., Stavåker K. 2012.

[2] **A Skeleton for Parallel CSG with a Performance Model**
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.16.2166
Accessed 23/7-15
H. Deldari, J.R. Davy, P.M. Dew. 1997.

[3] **A Structured Model Language for Large Continous Systems**
http://www.control.lth.se/Publication/elm78dis.html
Accessed 25/6-15
Hilding Elmqvist. 1978.

[4] **Acquiring high-resolution time stamps**
https://msdn.microsoft.com/en-us/library/windows/desktop/dn553408(v=vs.85).aspx
Accessed 27/7-15
Microsoft Corporation. 2015.

[5] **Amdahl's Law in the Multicore Era**
http://research.cs.wisc.edu/multifacet/papers/ieeecomputer08_amdahl_multicore.pdf
Accessed 4/7-15
Mark D. Hill, Michael R. Marty. 2008.

[6] **An Incompressible Navier-Stokes Equations Solver on the GPU Using CUDA**
Niklas Karlsson. 2013.

[7] **Analysis of Dynamics of the Tippe Top**
http://www.diva-portal.org/smash/get/diva2:602220/FULLTEXT01.pdf
Accessed 27/7-15
Nils Rutstam. 2013.

[8] **Automatic GPU Code Generation of Modelica Functions**
To be published:
https://www.modelica.org/publications/ModelicaConference

Hilding Elmquist. Hans Olsson. Axel Goteman. Vilhelm Roxling. Dirk Zimmer. Alexander Pollok. 2015.

[9] **Boost C++ Libraries**
http://www.boost.org/
Accessed 5/8-15
Boost Community. 2015.

[10] **Calculating the Volume and Centroid of a Polyhedron in 3D**
http://wwwf.imperial.ac.uk/~rn/centroid.pdf
Accessed 24/7-15
Robert Nürnberg. 2013.

[11] **Choosing Between Pinned and Non-Pinned Memory**
http://www.cs.virginia.edu/~mwb7w/cuda_support/pinned_tradeoff.html
Accessed 6/7-15
University of Virginia.

[12] **Collision Detection**
http://dmytry.com/texts/collision_detection_using_z_order_curve_aka_Morton_order.html
Accessed 26/6-15
Dmytry Lavrov. 2014.

[13] **Collision Handling for the Modelica MultiBody Library**
https://www.modelica.org/events/Conference2005/online_proceedings/Session1/Session1a4.pdf
Accessed 25/7-15
M. Otter, H. Elmqvist, J. Díaz López. 2005.

[14] **Constructive Solid Geometry Using BSP Tree** https://www.andrew.cmu.edu/user/jackiey/resources/CSG/CSG_report.pdf Accessed 26/6-15
Christian Segura, Taylor Stine, Jackie Yang. 2013.

[15] **Contact Forces of Polyhedral Particles in Descrete Element Method**
http://link.springer.com/article/10.1007\%2Fs10035-013-0417-9
Accessed 25/6-15
Benjamin Nassauer, Meinhard Kuna. 2013.

[16] **csg.js**
http://evanw.github.io/csg.js/docs/
Accessed 25/6-15
Evan Wallace. 2011.

[17] **CUDA**
http://www.nvidia.com/object/cuda_home_new.html
Accessed 24/6-15
NVIDIA. 2015.

[18] **CUDA Pro Tip: Optimized Filtering with Warp-Aggregated Atomics**
http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/
Accessed 15/7-15
Andrew Adinetz. 2014.

[19] **CUDA Programming Guide**
http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html\#axzz3ekPbb4YQ
Accessed 2/7-15
NVIDIA. 2015.

[20] **Dymola 2016**
http://www.Dymola.com
Accessed 24/6-15
Dassault Systemes. 2015.

[21] **Explicit Exact Formulas for the 3D Tetrahedron Inertia Tensor in Terms of it's Vertex Coordinates**
http://docsdrive.com/pdfs/sciencepublications/jmssp/2005/8-11.pdf
Accessed 24/7-15
F. Tonon. 2004.

[22] **Fast 4-way parallel radix sorting on GPUs**
http://vgc.poly.edu/~csilva/papers/cgf.pdf
Accessed 3/8-15
Linh Ha, Jens Krüger, Cláudio T. Silva. 2009.

[23] **Fermi**
http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
Accessed 1/7-15
NVIDIA. 2009.

[24] **FreeGLUT**
http://freeglut.sourceforge.net/
Accessed 24/7-15
Maintainers: John F. Fay, John Tsiombikas, Diederick C. Niehorster. Used release: 5 April 2013.

[25] **Generic Modelica Framework for MultiBody Contacts and Discrete Element Method**
To be published:
https://www.modelica.org/publications/ModelicaConference
Hilding Elmqvist, Axel Goteman, Vilhelm Roxling, Toheed Ghandriz. 2015.

[26] **High Performance and Scalable Radix Sorting: A Case Study of Implementing Dynamic Parallelism for GPU Computing**
Duane Merrill, Andrew Grimshaw. 2011.

[27] **How to Optimize Data Transfers in CUDA C/C++**
http://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/
Accessed 6/7-15
Mark Harris. 2012

[28] **Memory Transfer Overhead**
http://www.cs.virginia.edu/~mwb7w/cuda_support/memory_transfer_overhead.html
Accessed 6/7-15
University of Virginia.

[29] **Modelica Standard Library**
http://modelica.github.io/Modelica/help/Modelica.html
Accessed 24/6-15

[30] **Modelica, A Unified Object-Oriented Language for Systems Modeling. Language Specification, Version 3.3**
https://www.modelica.org/documents/ModelicaSpec33.pdf
Accessed 24/6-15
The Modelica Association. 2012.

[31] **Morton encoding/decoding through bit interleaving: Implementations**
http://www.forceflow.be/2013/10/07/morton-encodingdecoding-through-\\bit-interleaving-implementations/
Accessed 14/7-15
Jeroen Baert. 2013.

[32] **Numerical Methods for Shallow-Water Flow**
C.B. Vreugdenhil. 1994.

[33] **NVIDIA**
http://www.nvidia.com/page/home.html
Accessed 30/6-15
NVIDIA. 2015.

[34] **Parallelization of a DEM Code Based on CPU-GPU Heterogenous Architecture**
http://link.springer.com/chapter/10.1007/978-3-642-53962-6_13
Accessed 26/6-15
Xiaoqiang Yue, Hao Zhang, Congshu Luo, Shi Shu, Chunsheng Feng. 2014.

[35] **PCI Express**
http://www.nvidia.com/page/pci_express.html
Accessed 7/6-15
NVIDIA.

[36] **Practical Parallel Rendering**
Alan Chambers, Timothy Davis, Erik Reinhard. 2002.

[37] **Programming Massively Parallel Processors, second edition**
David B. Kirk, Wen-mei W. Hwu. 2013.

[38] **Real-Time Fluid Dynamics For Games**
http://www.intpowertechcorp.com/GDC03.pdf
Accessed 13/7-15
Jos Stam. 2003.

[39] **The Modelica Association**
`https://www.modelica.org/`
Accessed 25/6-15

[40] **Thinking Parallel, Part I: Collision Detection on the GPU**
`http://devblogs.nvidia.com/parallelforall/thinking-parallel-part-i-collision-detection-gpu/`
Accessed 26/6-15
Tero Karras. 2012.

[41] **Thinking Parallel, Part II: Tree Traversal on the GPU**
`http://devblogs.nvidia.com/parallelforall/thinking-parallel-part-ii-tree-traversal-gpu/`
Accessed 23/7-15
Tero Karras. 2012.

[42] **Thinking Parallel, Part III: Tree Construction on the GPU**
`http://devblogs.nvidia.com/parallelforall/thinking-parallel-part-iii-tree-construction-gpu/`
Accessed 13/7-15
Tero Karras. 2012.

[43] **Thrust documentation**
`http://docs.nvidia.com/cuda/thrust/\#axzz3flIKdbQu`
Accessed 13/7-15
NVIDIA. 2015.

[44] **Unification of Schematics, 3D assembly, Exploded View and Animation for Gamification of Modelica Dynamic Modeling**
To be published:
`https://www.modelica.org/publications/ModelicaConference`
Hilding Elmqvist, Alexander D. Baldwin, Simon Dahlberg. 2015.

Appendix A

A.1 Shallow Water Model Translation

Equation 3.1 is transformed into an ODE by spatial discretization using finite differences, resulting in the following Modelica model:

```
parameter Integer n = 128;
parameter Real L = 32;
parameter Real dx = L/n;
parameter Real H = L/4;
parameter Real g = 9.81;

Real h[n,n];
Real vx[n+1,n];
Real vy[n,n+1];

equation
  for ix in 1:n loop
    for iy in 1:n loop
      der(h[ix,iy]) = H*(vx[ix,iy]-vx[ix+1,iy]+vy[ix,iy]-vy[ix,iy+1])/dx;
    end for;
  end for;

  for ix in 2:n loop
    for iy in 1:n loop
      der(vx[ix,iy]) = g*(h[ix-1,iy] - h[ix,iy])/dx;
    end for;
  end for;

  for ix in 1:n loop
    for iy in 2:n loop
      der(vy[ix,iy]) = g*(h[ix,iy-1] - h[ix,iy])/dx;
```

```
    end for;
  end for;
```

The reason the for-loops cover different ranges is that a staggered grid is used, which is for numerical reasons common in CFD. However, that makes the example interesting, since it, as shown in the code below, puts some extra requirements on the kernel.

Following the steps in the beginning of section 3.5, the equations were moved into an algorithm section, and wrapped in for-loops. Variables that are not inside any `der()` expressions (the `tempX` variables in the code below) are used in the `algorithm` section, and then equaled the actual, time dependent variables in an `equation` section (which happens to be derivatives). No record was used in the shallow water model, and therefore that step wasn't necessary. The resulting Modelica model now looked like

```
parameter Integer n = 128;
parameter Real L = 32;
parameter Real dx = L/n;
parameter Real H = L/4;
parameter Real g = 9.81;

Real h[n,n];
Real vx[n+1,n];
Real vy[n,n+1];
Real tempH[n,n];
Real tempVx[n + 1,n];
Real tempVy[n,n + 1];

algorithm
  for ix in 1:n loop
    for iy in 1:n loop
      tempH[ix, iy] = H*(vx[ix, iy] - vx[ix + 1, iy] + vy[ix, iy] -
          vy[ix, iy + 1])/dx;
      if (ix > 1) then
        tempVx[ix, iy] = g*(h[ix - 1, iy] - h[ix, iy])/dx;
      end if;
      if (iy > 1) then
        tempVy[ix, iy] = g*(h[ix, iy - 1] - h[ix, iy])/dx;
      end if;
    end for;
  end for;

equation
  for ix in 1:n loop
    for iy in 1:n loop
      der(h[ix, iy]) = tempH[ix, iy];
      if (ix > 1) then
        der(vx[ix, iy]) = tempVx[ix, iy];
      end if;
```

```
      if (iy > 1) then
        der(vy[ix, iy]) = tempVy[ix, iy];
      end if;
    end for;
  end for;
```

Note that the for-loops have been merged into a single one, with if-statements to get the ranges right. Since the point is that the contents of each set of nested for-loops ends up as a kernel, this means that more work can be done in one kernel, at the cost of some instruction divergence.

When this Modelica model is translated, it results in the following C-code.

```
{
    int i_0_;
    int ix0_0_0;
    for(i_0_ = 0, ix0_0_0 = 1;ix0_0_0 <= 32;ix0_0_0 += 1 , ++i_0_) {
      {
        int i_1_;
        int iy0_0_0;
        for(i_1_ = 0, iy0_0_0 = 1;iy0_0_0 <= 32;iy0_0_0 += 1 , ++i_1_) {

        //Corresponds to calculation of tempH
          SetRealElement(DYNX(W_,4)*(RealElement( RealArrayArray ( 33,
            RealTemporaryDense( &DYNX(W_,5), 1, 32), RealTemporaryDense( &
            DYNX(X_,1024), 1, 32), RealTemporaryDense( &DYNX(X_,1056), 1,
                32),
            RealTemporaryDense( &DYNX(X_,1088), 1, 32),
                RealTemporaryDense( &
            DYNX(X_,1120), 1, 32), RealTemporaryDense( &DYNX(X_,1152), 1,
                32),
                                          •
                                          •
                                          •
            DYNX(X_,2998), DYNX(X_,2999), DYNX(X_,3000), DYNX(X_,3001),
            DYNX(X_,3002), DYNX(X_,3003), DYNX(X_,3004), DYNX(X_,3005),
            DYNX(X_,3006), DYNX(X_,3007), DYNX(W_,132))),
                (SizeType)(ix0_0_0),
            (SizeType)(iy0_0_0+1))),
             RealTemporaryDense( &DYNX(W_,133), 2, 32, 32),
             (SizeType)(ix0_0_0), (SizeType)(iy0_0_0));
          PopAllMarks();

          //Corresponds to calculation of tempVx
          if (ix0_0_0 > 1) {
            SetRealElement(RealElement( RealTemporaryDense( &DYNX(X_,0),
                2, 32, 32),
               (SizeType)(ix0_0_0-1), (SizeType)(iy0_0_0))-RealElement(
              RealTemporaryDense( &DYNX(X_,0), 2, 32, 32),
                 (SizeType)(ix0_0_0),
```

```
            (SizeType)(iy0_0_0)), RealTemporaryDense( &DYNX(W_,1157), 2,
               33, 32),
             (SizeType)(ix0_0_0), (SizeType)(iy0_0_0));
          PopAllMarks();
        }

        //Corresponds to calculation of tempVy
        if (iy0_0_0 > 1) {
          SetRealElement(RealElement( RealTemporaryDense( &DYNX(X_,0),
             2, 32, 32),
             (SizeType)(ix0_0_0), (SizeType)(iy0_0_0-1))-RealElement(
            RealTemporaryDense( &DYNX(X_,0), 2, 32, 32),
               (SizeType)(ix0_0_0),
             (SizeType)(iy0_0_0)), RealTemporaryDense( &DYNX(W_,2213), 2,
               32, 33),
             (SizeType)(ix0_0_0), (SizeType)(iy0_0_0));
          PopAllMarks();
        }
      }
    }
  }
}
```

Assignments to the `tempX` variables are made by `SetRealElement(value, pointer, indices)`, where

1. `value` is the value to assign, calculated by the corresponding arithmetic expression. The gathering is made using the `RealTemporaryDense` function, and `RealElement` is used to extract a value from the gathered vector.

2. `pointer` is a pointer pointing to the corresponding answer vector

3. `indices` corresponds to the indices needed to access `pointer` correctly

The reason why the section corresponding to the `tempH` array is so much longer than the other two is because it's data is more fragmented (the result of an arithmetic expression of v_x and v_y), and thus requires a more complex gathering.

In order to convert the above into something that complies to the pseudo code of point 4 in the list in section 3.5, a few things have to be noted.

The sizes for the allocation of pinned host memory, the corresponding device memory, and device memory for answers, has to be determined. Since h has a value in n^2 points, and v_x and v_y in $n(n+1)$ points each, the total amount of memory needed for the three quantities is $n^2 + 2n(n+1)$. The same amount of memory is needed for the answer. The only constant being H, it was deemed better to just pass it along as an argument to the kernel. If constant terms can be determined by the Dymola compiler, then it should be easy to keep track of the amount of constant data, making a decision, of whether to use arguments or allocate memory, possible.

The nested for-loops in the auto-generated C-code become a function call to an external function, defined in a `.cu` file. This function would then contain the following code:

```
/*
Wrapper function replacing the nested for-loops in the auto-generated C
    code. Manages memory and kernel launch. W_ and X_ are pointers to the
    memory Dymola uses to store all variables used in this simulation.
*/
extern "C" //Used to avoid name wrangling, since NVCC will compile
    function names as C++, and Modelica as C.
   void wrapperFunction(double *W_, double *X_){
     static bool first = true;

     if(first){
       //Allocate pinned memory on the host
       cudaMallocHost((void**) &data, (2*N*(N+1) + N*N)*sizeof(Real));
       //Allocate memory for device data
       cudaMalloc((void**)&d_data, (2*N*(N+1) + N*N)*sizeof(Real));
       //Allocate memory for device answer
       cudaMalloc((void**)&d_answer, (2*N*(N+1) + N*N)*sizeof(Real));
     }

     //Memory gathering of tempVx
     memcpy(data, &W_[5], N*sizeof(Real));
     memcpy(&data[N], &X_[(N*N)], ((N*N)-N)*sizeof(Real));
     memcpy(&data[(N*N)], &W_[N+5], N*sizeof(Real));

     //Memory gathering of tempVy
     for(int i = 0;i < N; ++i){
       data[(N*N)+N + (N+1)*i] = W_[(2*N+5) + 2*i];
       memcpy(&data[(N*(N+1)) + (N+1)*i + 1], &X_[2*(N*N) - N +
           (N-1)*i], (N-1)*sizeof(Real));
       data[(N*(N+1)) + (N+1)*i + (N-1) + 1] = W_[(2*N + 5) + 1 + 2*i];
     }

     //Memory gathering of tempH
     memcpy(&data[2*N*(N+1)], X_, N*N*sizeof(Real));

     //Set up a block and grid
     if(first){
       block = dim3(32,32,1);
       grid = dim3((N + 31)/32, (N + 31)/32, 1);
     }

     //Transfer data to the device
     cudaMemcpy(d_data, data, (2*N*(N+1) + N*N)*sizeof(Real),
         cudaMemcpyHostToDevice);

     //Launch the kernel. The last argument is H
     kernel<<<grid,block>>>(d_data, d_answer, W_[4]);
```

```
    //Copy back the answers. &W_[133] can be seen in the C code above as
       the location to save answers on (third last argument in
       SetRealElement)
    cudaMemcpy(&W_[133], answer_d, (2*N*(N+1) + N*N)*sizeof(Real),
       cudaMemcpyDeviceToHost);

    if(first)
      first = false;
}
```

The reason why the gathering looks different in the CUDA code compared to the C-code is that the gathering location was changed. When hand coding the gathering, it's much easier to look for the gathering patterns, and assemble uniquely for this example, than to rewrite the auto-generated gathering. Note that hard coded numbers have been identified from the auto generated code, to get the correct addresses.

Finally, the contents of the nested for loops were written as the kernel `kernel`:

```
#define IX(i,j, max_j) ((i)*(max_j) + (j))

__global__
  void kernel(const double *data, double *answer, const double H){
    size_t ix = threadIdx.x + blockDim.x*blockIdx.x;
    size_t iy = threadIdx.y + blockDim.y*blockIdx.y;

    if(ix < N && iy < N){
      answer[IX(ix,iy, N)] = H*(data[IX(ix,iy, N)]-data[IX(ix+1,iy, N)]
         + data[(N*(N+1)) + IX(ix,iy, N+1)]-data[(N*(N+1)) + IX(ix,
         iy+1, N+1)]);

      if (ix > 0)
        answer[(N*N) + IX(ix,iy, N)] = data[2*N*(N+1) + IX(ix-1, iy,
           N)] - data[2*N*(N+1) + IX(ix,iy, N)];

      if (iy > 0)
        answer[N*(N+1)+N*N + IX(ix,iy,N+1)] = data[2*N*(N+1) +
           IX(ix,iy-1, N)] - data[2*N*(N+1) + IX(ix,iy, N)];
    }
}
```

A.2 Matrix model

The matrix multiplication model used for testing the auto-generation of CUDA code is presented below. Note that the only new element compared to the Modelica 3.3 standard is the annotation `gpuFunction=true`.

```
function Multiply
  input Real A[:,:];
  input Real B[size(A,2),:];
```

```
  output Real C[size(A,1),size(B,2)];
  protected
  Real temp;

  algorithm
  for i in 1:size(A,1) loop
    for j in 1:size(B,2) loop
      temp := 0;
      for k in 1:size(A,2) loop
        temp := temp + A[i, k]*B[k, j];
      end for;
      C[i, j] := temp;
    end for;
  end for;
annotation(gpuFunction=true);
end Multiply;
```

The two outer for-loops maps to the following wrapper function:

```
extern "C"
void Multiply(double const * A, size_t Adim0, size_t Adim1, double const
    * B, size_t Bdim1, double * C)
{
  /* GPU Memory declaration */
  static double * AGPU=0;
  static size_t AGPUS=0;
  size_t AGPUs;
  static double * BGPU=0;
  static size_t BGPUS=0;
  size_t BGPUs;
  static double * CGPU=0;
  static size_t CGPUS=0;
  size_t CGPUs;
  /* GPU Memory size */
  AGPUs=Adim0*Adim1;
  BGPUs=Adim1*Bdim1;
  CGPUs=Adim0*Bdim1;

  /* GPU Memory allocation */
  if (AGPU&&(AGPUS<AGPUs))
    {AGPUS=0; cudaFree(AGPU); AGPU=0;}
  if (!AGPU)
    {AGPUS=AGPUs;cudaMalloc((void**)&AGPU, AGPUS*sizeof(double));}

  if (BGPU&&(BGPUS<BGPUs))
    {BGPUS=0; cudaFree(BGPU); BGPU=0;}
  if (!BGPU)
    {BGPUS=BGPUs;cudaMalloc((void**)&BGPU, BGPUS*sizeof(double));}

  if (CGPU&&(CGPUS<CGPUs))
```

```
    {CGPUS=0;cudaFree(CGPU);CGPU=0;}
  if (!CGPU)
    {CGPUS=CGPUs;cudaMalloc((void**)&CGPU, CGPUS*sizeof(double));}

  /* GPU Memory copy to */
  cudaMemcpy(AGPU, A,AGPUs*sizeof(double), cudaMemcpyHostToDevice);
  cudaMemcpy(BGPU, B,BGPUs*sizeof(double), cudaMemcpyHostToDevice);
  cudaMemcpy(CGPU, C,CGPUs*sizeof(double), cudaMemcpyHostToDevice);
  /* Call GPU function */
  dim3 block=dim3(32, 32, 1);
  dim3 grid=dim3((Adim0+31)/32, (Bdim1+31)/32, 1);
  GPUfunction_cuda<<<grid,block>>>(AGPU, Adim0, Adim1,BGPU, Bdim1,CGPU);
  /* GPU Memory copy from */
  cudaMemcpy(C, CGPU,CGPUs*sizeof(double), cudaMemcpyDeviceToHost);
}
```

and it's contents to the following kernel:

```
#include <stddef.h>
__global__
void Multiply_cuda(double const * A, size_t Adim0, size_t Adim1, double
    const * B, size_t Bdim1, double * C){

        double temp;
        temp=0;
        int i = 1+threadIdx.x + blockDim.x*blockIdx.x;
        int j = 1+threadIdx.y + blockDim.y*blockIdx.y;
        if ((i<=Adim0) && (j<=Bdim1)) {
           temp = 0;
           {
              int end_ = Adim1;
              int k;
              for(k = 1; k <= end_; k += 1) {
                 temp = temp + A[(i-1)*Adim1+(k-1)] * B[(k-1)*Bdim1+(j-1)];
              }
           }
           C[(i-1)*Bdim1+(j-1)] = temp;
        }
   return;
}
```

A.3 Broad Phase

A.3.1 uniqueSplit

```
/*
```

```
Returns true if the pair (collId1[i], collId2[i]) is unique in the
    sequence [0,i], false otherwise, with the assumtions on collId1 and
    collId2 explained at uniqueSplit().
*/
inline bool checkUniqueBack(const thrust::host_vector<unsigned> &collId1,
    const thrust::host_vector<unsigned> &collId2, size_t i){
   int j = i - 1; //Index of pair to compare with.
   while(j >= 0 && collId1[i] == collId1[j]){ //Only needs to compare
       with pairs of same first value, as those are sorted.
      if(collId2[i] == collId2[j--]) //Found a duplicate
         return false;
   }
   return true;
}

/*
Removes duplicates of pairs of the vectors collId1 and collId2, where a
    pair is the pair of values that are placed at the same index in the
    two vectors. The vectors are assumed to have the same size, and are
    assumed to be sorted according to the values of collId1. That is, the
    order of the values of collId2 for a sequence of equal values in
    collId1 is undefined.
*/
void uniqueSplit(thrust::host_vector<unsigned> &collId1,
    thrust::host_vector<unsigned> &collId2){
   size_t uniques = 1; //Number of uniques found.
   for(size_t i = 1; i < collId1.size(); ++i){
      if(checkUniqueBack(collId1, collId2, i)){ //Unique
         if(uniques != i){ //If all have been unique, nothing has to be
             done.
            //Write over the old vectors with guarantied unique pairs
            collId1[uniques] = collId1[i];
            collId2[uniques] = collId2[i];
         }
         ++uniques;
      }
   }
   if(collId1.size()){ //Only keep the unique pairs.
      collId1.erase(collId1.begin()+uniques, collId1.end());
      collId2.erase(collId2.begin()+uniques, collId2.end());
   }
}
```

A.3.2 addAABBSepKernel

```
/*
Kernel adding Morton code-based intervals for every object in a scene. It
    is placing start codes and the rest of the information in separate
```

```
    memory locations, to allow for Thrust using radix sort. The AABBs of
    the objects are placed in AABBs, spaced with nBodies, the number of
    objects, between each corner.
*/
__global__ void addAABBSepKernel(EntrySep *entries, uint64_t *starts,
    double *AABBs, size_t nBodies){

   size_t i = threadIdx.x + blockDim.x*blockIdx.x; //Object id.

   if(i < nBodies){
      unsigned lx, ly, lz, hx, hy, hz, d, szp, sz2, mask, count;

      //lx, ly and lz are discretized coordinates of lower corner, hx, hy
          and hz are of higher corner.
      lx = (unsigned)(resolution*(AABBs[i]+maxDim)+97151);
      ly = (unsigned)(resolution*(AABBs[i+nBodies]+maxDim)+97151);
      lz = (unsigned)(resolution*(AABBs[i+2*nBodies]+maxDim)+97151);
      hx = (unsigned)(resolution*(AABBs[i+3*nBodies]+maxDim)+97151);
      hy = (unsigned)(resolution*(AABBs[i+4*nBodies]+maxDim)+97151);
      hz = (unsigned)(resolution*(AABBs[i+5*nBodies]+maxDim)+97151);

      d = thrust::max(hx-lx, thrust::max(hy-ly, hz-lz));

      if(hx>2097151)hx=2097151;
      if(hy>2097151)hy=2097151;
      if(hz>2097151)hz=2097151;

      for(sz2=1, szp=0; sz2<d; sz2<<=1, szp++);

      mask=sz2-1;
      lx&=~mask;
      ly&=~mask;
      lz&=~mask;
      hx|=mask;
      hy|=mask;
      hz|=mask;

      count=0;
      for(unsigned iz=lz; iz<=hz; iz+=sz2){
         for(unsigned iy=ly; iy<=hy; iy+=sz2){
            for(unsigned ix=lx; ix<=hx; ix+=sz2){
               starts[i+count*nBodies]=mortonEncodeMagicBits(ix,iy,iz);
               entries[i+count*nBodies]=EntrySep(szp,i);
               ++count;
            }
         }
      }
   }
}
```

To lower divergence, the nested for-loops in the end where in different attempts reformulated, so that the combinations that where true came first. This, however, did not turn out to give better results. The only reason we could see is the extra register usage required for the reformulations.

A.4 Parallel Traversal

A.4.1 hybridTraversal

```
/*
Kernel for parallel traversal of end-start polygons. count contains the
    number of total polygons, and is potentially increased in
    clipPolygonDevice, at splits. This kernal is used in hybridTraversal,
    see below.
*/
__global__ void traverseKernel(HybridPolygon *polygons, size_t *count,
    size_t start, size_t end){
   size_t i = start + blockDim.x*blockIdx.x + threadIdx.x;
   if(i < end)
      polygons[i].inside =
          polygons[i].d_treeToTraverse->clipPolygonDevice(i, polygons,
          count);
}

/*
Function for hybrid GPU-CPU traversal of polygons. It is assumed that the
    function is only called with an initial number of polygons to be
    traversed that is greater than GPU_LIM, so that at least one launch
    will be made. The number of initial polygons is stored in nbrOfTasks
    on CPU memory, and in d_count on GPU memory. The polygons are
    initially stored in d_polygons on GPU memory only. Afterwards, the
    polygons are stored in polygons, and the total number of tasks is
    stored in nbrOfTasks.
*/
void hybridTraversal(HybridPolygon *polygons, size_t &nbrOfTasks,
    HybridPolygon *d_polygons, size_t *d_count){
   size_t start = 0, diff = nbrOfTasks; //diff - Currently known
       remaining number of tasks.
   bool copiedBack = false;
   while(diff){
      size_t end = start + nbrOfTasks;
      if(diff > GPU_LIM && !copiedBack){
         traverseKernel<<<(diff+(BLOCK_SIZE-1))/BLOCK_SIZE,
             BLOCK_SIZE>>>(d_polygons, d_count, s, end);
         cudaMemcpy(&nbrOfTasks, d_count, sizeof(size_t),
             cudaMemcpyDeviceToHost);
      }else{
         if(!copiedBack){
```

```
            cudaMemcpy(polygons, d_polygons, end*sizeof(HybridPolygon),
                cudaMemcpyDeviceToHost);
            copiedBack = true;
          }
          for(size_t i = s; i < end; ++i) //CPU traversal. A reference of
              nbrOfTasks is passed, so it can be increased.
            polygons[i].inside =
                polygons[i].treeToTraverse->clipPolygon(i, polygons,
                nbrOfTasks);
        }
        diff = nbrOfTasks - end;
        start = end;
    }
    if(!copiedBack) //In case no small set were launched. Very unlikely.
        cudaMemcpy(polygons, d_polygons, nbrOfTasks*sizeof(HybridPolygon),
            cudaMemcpyDeviceToHost);
}
```

A.4.2 clipPolygonDevice

The `clipPolygon` and `clipPolygonDevice` functions of the `Node` class are different, as they only handle one polygon, and returns if it was inside or outside. We present `clipPolygonDevice` here (`clipPolygon` essentially looks the same).

```
__device__ bool GPUNode::clipPolygonDevice(size_t i, HybridPolygon
    *polygons, size_t *count){
    bool inside = plane->splitPolygonDevice(i, polygons, this, count);
    if(front && !inside) return front->clipPolygonDevice(i, polygons,
        count);
    if(back && inside) return back->clipPolygonDevice(i, polygons, count);
    return inside;
}
```

First, the polygon is classified according to this node in `splitPolygonDevice`. There, new polygons may be pushed as a result of splits, explaining the need of passing `this` (a new polygon will begin its traversal on this node or some of its children) and `polygons`. Then, if the polygon was outside, and there exists a `front` child, or inside with an existing `back` child, further investigation of this node is needed. Otherwise, the traversal for this node is complete, and `inside` is it's result.

A.4.3 traverseKernelNonRecursive

```
/*
Non-recursive version of clipPolygonDevice(..), called from
    traverseKernelNonRecursive(..), see below.
*/
```

```
__device__ GPUNode* GPUNode::clipPolygonDeviceNonRecursive(size_t i,
    HybridPolygon *polygons, size_t *count){
   bool inside = plane->splitPolygonDevice(i, polygons, this, count);
   if(front && !inside) return front;
   else if(back && inside) return back;
   else{
      polygons[i].inside = inside;
      return nullptr;
   }
}

/*
Kernel for non-recursive parallel traversal of end-start polygons. count
    contains the number of total polygons, and is potentially increased
    in clipPolygonDevice, at splits.
*/
__global__ void traverseKernelNonRecursive(HybridPolygon *polygons,
    size_t *count, size_t start, size_t end){
   size_t i = start + blockDim.x*blockIdx.x + threadIdx.x;
   if(i < end){
      GPUNode *ttt = polygons[i].d_treeToTraverse;
      while(ttt = ttt->clipPolygonDeviceNonRecursive(i, polygons, count));
   }
}
```

A.4.4 Other Approaches of Parallel Traversal

In this section, the implementation of two alternative traversal methods is shown. Both of them uses the `clipPolygonDeviceNonRecursive` function and the idea implemented in the `traverseKernelNonRecursive` kernel, see appendix A.4.3. They are also transferring the polygons to be kept to another part of memory, for which the `outsideCount` variable is used.

First, the "shared" version. A limiting factor is the size of polygons. `MAX_SHARED_POLYGONS` must have a large margin to how many polygons that may in total be generated from the threads of a block. This reduces the number of threads per block, and also the amount of cache memory per thread, since a large part of the L1 cache is used as shared memory.

```
__global__ void traverseKernelShared(GPUPolygon *polygons, size_t
    nOriginalPolygons, size_t *outsideCount){
   __shared__ GPUPolygon sharedPolygons[MAX_SHARED_POLYGONS];
   __shared__ size_t taskCount; //Number of existing tasks in the block.
   __shared__ size_t tasksDone; //Number of begun tasks in the block.
   size_t workingInd = threadIdx.x; //Where this thread is working wihtin
       sharedPolygons.
   size_t i = workingInd + blockIdx.x * blockDim.x;
   if(i < nOriginalPolygons){
      sharedPolygons[workingInd] = polygons[i]; //Push the original
```

```
        polygon.
      taskCount = tasksDone = blockDim.x;
      GPUNode *ttt = sharedPolygons[workingInd].treeToTraverse;
      while(true){
        ttt = ttt->clipPolygonDeviceNonRecursive(workingInd,
            sharedPolygons, &taskCount);
        if(!ttt){ //Traversal of current polygon completed.
          if(!sharedPolygons[workingInd].inside){ //This polygon should
              be kept, and is therefor copied to the separate memory
              location.
            polygons[atomicAggInc2((int*)outsideCount)] =
                sharedPolygons[workingInd];
          }
          if(taskCount - tasksDone){ //There are more tasks to be done.
            workingInd = atomicAggInc2((int*)&tasksDone);
            if(workingInd < taskCount) //This thread gets a new tree to
                traverse.
              ttt = sharedPolygons[workingInd].treeToTraverse;
            else{ //There where more active threads than new tasks. Only
                some of them gets work. Correct tasksDone.
              tasksDone = taskCount;
              break;
            }
          }else
            break;
        }
      }
    }
}
```

An alternative could be to use global memory for `sharedPolygons`, as was the case in the next approach. It is very similar to the previous approach, except here, every thread takes care of the polygons it generates. Since this may vary a lot from thread to thread, a large amount of memory is needed per thread, and this has to be global memory. The need of atomic pushes at splits are completely removed in this approach.

```
__global__ void traverseKernelLocal(GPUPolygon *polygons, size_t
    nOriginalPolygons, size_t *outsideCount, GPUpolygon *localPolygons){
  size_t i = threadIdx.x + blockIdx.x * blockDim.x;
  localPolygons += i*MAX_LOCAL_POLYGONS;
  if(i < nOriginalPolygons){
    localPolygons[0] = polygons[i];
    GPUNode *ttt = localPolygons[0].treeToTraverse;
    i = 0;
    size_t taskCount = 0;
    while(true){
      ttt = ttt->clipPolygonDeviceNonRecursive(i, localPolygons,
          &taskCount);
```

```
        if(!ttt){
           if(!localPolygons[i].inside){
              polygons[atomicAggInc2((int*)outsideCount)] =
                  localPolygons[i];
           }
           if(taskCount - i){
              ++i;
              ttt = localPolygons[i].treeToTraverse;
           }else
              break;
        }
     }
  }
}
```

As they are, none of the two presented approaches proved to be of good use, as they were too slow. It was just ideas that we wanted to try out, and many of their drawbacks have already been mentioned. It may be possible to combine the approaches, or change them in some way to perform better, but they where for now left in this state, due to a lack of time for more experiments.

A.5 Additional Algorithms for Polyhedrons

We begin with some well known facts about triangles, that are used frequently in our calculations. Their normal, $\mathbf{n}$, is given by $\hat{\mathbf{n}}/\|\hat{\mathbf{n}}\|$, where

$$\hat{\mathbf{n}} = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})$$

and their area is given by

$$A = \frac{\|\hat{\mathbf{n}}\|}{2}$$

where $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ are the vertices of the triangle, ordered clockwise.

A.5.1 Volume and Centroid

Nürnberg[10] presents algorithms for computing the volume and centroid of an arbitrarily shaped polyhedron. If a polyhedron has a surface consisting of N triangles A_i, $i = 0, ..., N-1$, all with vertices $(\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i)$ and non-normalized normal $\hat{\mathbf{n}}_i$, it's volume V is given by

$$V = \frac{1}{6} \sum_{i=0}^{N-1} \mathbf{a}_i \hat{\mathbf{n}}_i$$

By denoting unit vectors of the x, y and z axes in a 3D Cartesian coordinate system $\mathbf{e}_j$, $j = 0, 1, 2$, it's centroid $\mathbf{c}$, has coordinates

$$c_j = \frac{1}{48V} \sum_{i=0}^{N-1} \hat{\mathbf{n}}_i \mathbf{e}_j([(\mathbf{a}_i + \mathbf{b}_i)\mathbf{e}_j]^2 + [(\mathbf{b}_i + \mathbf{c}_i)\mathbf{e}_j]^2 + [(\mathbf{a}_i + \mathbf{c}_i)\mathbf{a}_j]^2)$$

A.5.2 Moment of Inertia

An algorithm to compute the inertia tensor of the bodies is also needed. This can be done for an arbitrary polyhedron (with triangle faces) in the following way:

Pick a point, e.g. the point around which the inertia tensor is needed. We pick the frame. If another point is chosen, a simple translation to the desired point has to be carried out afterwards.

The integration over the body's volume can be transformed to a sum of integrals over tetrahedrons. Those tetrahedrons are the ones created by connecting the chosen point with the triangles of the body surface. Now, for one such tetrahedron, if the normal of the triangle from the body's surface points into the tetrahedron, the contribution from this tetrahedron is to be subtracted, otherwise added, to the total inertia tensor.

What remains is to actually calculate the moment of inertia of arbitrary shaped tetrahedrons. Tonon[21] presents expressions for this in terms of the vertex coordinates. As the expressions takes about one page to present, the interested reader will have to follow the link in the reference.

GPU Accelerated Simulations

Popular Science Summary
Authors: Axel Goteman
Vilhelm Roxling

Supervisors: Hidling Elmqvist (Dassault Systemes)
Michael Doggett (Dep of CS, LTH)
Examiner: Flavius Gruian

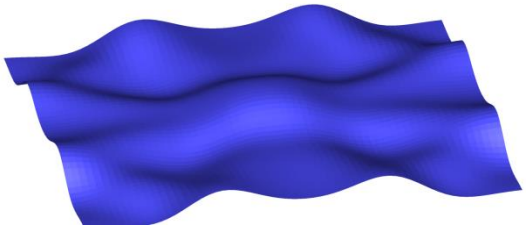
One of the key components in our modern society is the ability to simulate. By simulations, the industry can design new cars, phones, aircrafts etc., without having to go through prototype after prototype, allowing us the cheap and high-tech products most of us rely on in our day-to-day life. However, good as simulations are today there are still severe limitations on what can be simulated. Two of the largest limiting factors are how hard simulations are to design and how long they take to run. The first of these problems are tackled by the Modelica programming language, which is designed for easy set-up of simulations. We have attacked both these problems by using GPUs to speed-up Modelica simulations more than 5 times, and extending the capability of Modelica to handle collisions between objects.

A modern GPU (graphics card) has more processing power, that is, can make more calculations per second, than a modern CPU. They are not used often however, as it is much harder to write programs that utilizes the capabilities of the GPU. This is because they are designed for effective processing of graphics, meaning doing the same operation for the thousands of pixels that makes up the screen. Any program that wishes to utilize the GPU needs to mimic this working method, i.e. doing thousands of identical operations, one for each element in a set.

GPUs in Modelica

Modelica works by having the user set up a number of equations, giving the rules for what to simulate and how it should behave. Our GPU version extracts parts from these rules that have a structure which fits the GPU needs mentioned above. These parts are then separated from the rest of the simulation by being run on the GPU. The GPU and CPU then communicate necessary information during the simulation.

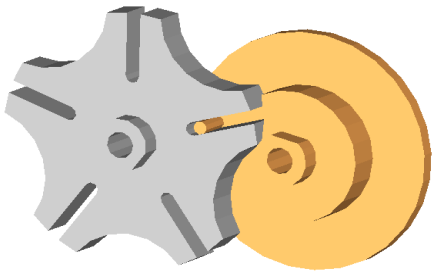
Typical problems where GPU usage is beneficial would be fluid/flow simulations. Wave simulation is such a problem, which can be useful in creation of for example wave power plants. When testing our prototype on such a problem, the simulation time decreased to less than a fifth of the original simulation time.



Wave simulations done using our prototype

Collisions in Modelica

One of the areas where Modelica has previously been lacking was collision simulations. Our prototype handles objects of any shape and size, by using an algorithm for contact reaction that isn't based on the shape of the object. Instead it allows very tiny penetrations between objects and uses this penetration to calculate the correct collision response. The prototype is also capable of handling anything from zero to tens of thousands of objects.



Our prototype simulating a part of a watch