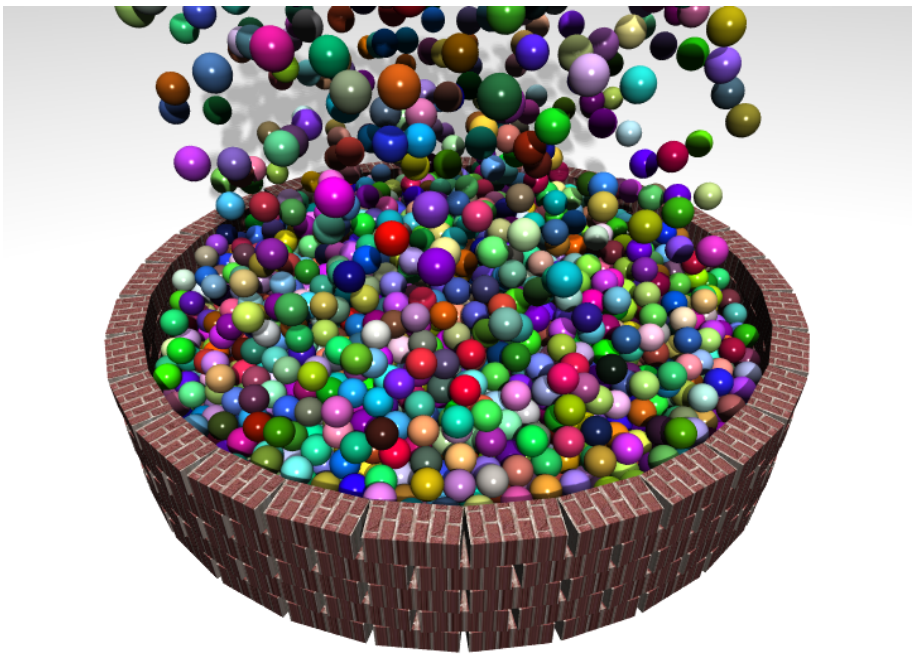
FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

GPU-Based Rigid Body Dynamics

Severin Strobl



Diplomarbeit

GPU-Based Rigid Body Dynamics

Severin Strobl

Diplomarbeit

Aufgabensteller:	Prof. Dr. Ulrich Rüde
Betreuer:	M. Sc. Klaus Iglberger
	Dipl.-Inf. Tobias Preclik
Bearbeitungszeitraum:	02.03.2009 – 02.09.2009

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 2. September 2009 .....

Abstract

The simulation of rigid body dynamics is an area of interest both for scientific research and engineering applications. The accurate numerical solution of the underlying complementarity problems is complex. The use of the computational power of current graphics processing units (GPUs) to solve these systems seems promising. Using *NVIDIA* 's CUDA technology for General-Purpose computation on Graphics Processing Units (GPGPU), an object-oriented iterative solver using a Jacobi method for the efficient simulation of rigid body dynamics is developed. The solver is then integrated into an existing physics engine as one of multiple possible methods for the collision response. Special care is taken to provide an easy to use interface for the synchronization between the host system and the graphics card. The final part of the thesis is focused on the performance evaluation of the implemented solver.

Contents

1 Introduction

The numerical simulation of all kinds of physical effects using high performance computers is a considerably growing field in both computer science and engineering. With todays complex technologies it is imperative to run simulations prior to building and testing the actual systems. The modeling of systems composed of rigid bodies, i.e. bodies that are not deformable at all, along with the interactions between these bodies is the domain of rigid body dynamics. The bodies can be of different shapes and sizes, be freely moveable according to the acting forces or be linked together to form sophisticated structures like robot arms. The simulation of such systems is a problem arising in many areas of engineering. The possible applications range from robotics over machine tools to the study of granular material flows. For these tasks accurate simulations considering external forces like gravity or internal forces like friction between single bodies are required.

Recently a completely different use has emerged: computer games. As common hardware becomes more and more powerful and multi-core systems gain a foothold in standard PCs, the changes in computer games are not restricted to new visual effects. Rigid body dynamics are used to model the physics of game worlds in a more realistic way. This development took place simultaneously to an other evolution related to computer games. The graphics processing unit (GPU) changed from a simple device for rendering graphics into a powerful, massively parallel co-processor, usable for all types of computations. Not long after the appearance of these two new factors, efforts were made to combine them. At the latest as *NVIDIA* took over the physics engine *PhysX*[1], rigid body dynamics on GPUs became mainstream. While the *PhysX* engine is focused on real-time effects for computer games, the underlying hardware is not restricted to this. For several years General-Purpose Computing on Graphics Processing Units (GPGPU), the usage of GPUs for many different tasks aside from graphics processing, has been a resource increasingly used for scientific computing. The applications ported to GPUs include but are not limited to finite element methods, linear algebra tools and fluid dynamics. The reason for this is the extreme increase in performance of GPUs over time. Current GPUs by *NVIDIA* or *AMD* using single precision (32 bit floating point format) reach a performance of over 1 000 GFlops (10^9 floating point operations per second). Combined with a high memory bandwidth of typically over 100 GB/s, they leave common CPUs well beyond. The GPU vendors of course have noticed the increasing interest in GPGPU and began to actively support this trend by the release of programming interfaces and additional libraries for general access to the GPU hardware. Especially *NVIDIA* started to promote its *Compute Unified Device Architecture* (CUDA) and the community around it has grown considerably.

There are some traditional physics engines like *Bullet*[2] which already make use of the computational power of GPUs. For scientific applications a lot of work has been done by Tasora and Negrut [22]. They showed that it is possible to perform accurate rigid body dynamics completely on graphics cards. Inspired by their work, this thesis examines the development of a GPU port of the physics engine *pe* [3], a rigid body simulation framework developed at the Chair for System Simulation at the University of Erlangen-Nuremberg. It is focused on large scale simulations while being highly object oriented and configurable. The GPU port uses CUDA to interface with *NVIDIA* graphics cards and tries to achieve high performance and at the same time make use of object-oriented concepts. This ensures a seamless integration into *pe* and also ensures that the CUDA code is flexible and easily extendible.

[1] http://www.nvidia.com/object/physx_new.html
[2] http://www.bulletphysics.com
[3] http://www10.informatik.uni-erlangen.de/~klaus/

The thesis is split into five major parts. The first section deals with the very basics of GPGPU. Here the actual hardware is described as well as the special features and challenges that may arise during the development of CUDA accelerated projects. Aside from that the corresponding programming model and some tools are introduced. Section 3 provides a short overview of the *pe* engine and its features, but is centered upon the mathematical and physical foundation of rigid body dynamics. The problems solved in the implementation phase and the frame for the simulation are described. The next section focuses on the implementation details, namely the very basics like data structures and data exchange between CPU and GPU. In this context some optimization techniques were adapted for the use with CUDA and GPUs in general. The following section, Section 5, presents the actual implementations of the time integration and collision response phase introduced in Section 3. Next some of the results of the GPU implementation are illustrated along with some performance measurements and an concluding evaluation.

2 GPU Programming

To be able to understand the potential and also the special problems of GPU programming, a closer look at the underlying hardware and programming models is of some help. This section is not intended to provide a complete discussion of all the hardware internals and different programming possibilities. The basic concepts of GPU hardware are described for example in [15, chap. 29, 30] and with focus on current *NVIDIA* hardware in [14]. Here only the concepts necessary to understand the implementation details are presented.

2.1 Evolution of GPUs

As the name suggests, graphics processing units where initially intended for the fast rendering of 2 and 3D graphics. The first GPUs supported only 2D graphics and where rather simple devices. But with the upcoming support of 3D features, driven mainly by the computer games industry, GPUs quickly became more complex and since some years ago even outperform common CPUs in terms of floating point performance and memory bandwidth. A long time the GPU hardware was highly specialized. There were units for the transformation of vertices, pixels, special visual effects, and for rasterization. Some of these units (called *shader*s) supported the execution of user supplied programs. This shader programming was the start of the General-Purpose computation on Graphics Processing Units (GPGPU). It was now possible to use the processing power of common GPUs for all sorts of applications. Based on shader programming the first toolkits like BrookGPU and Sh (now RapidMind) where developed. Using these tools it was possible to develop GPU accelerated programs without going to the low level of shader programming. With the next step in the evolution of GPUs the concept of *unified shaders* was introduced. Unified shaders combined all programmable shading units (vertex shader, fragment shader, later geometry shader) into one single unit. The step was necessary, as GPU hardware became more and more complex and all the separated shader units had to access common parts like the device memory. With unified shaders it is also possible to utilize the hardware in a optimal way, as the single shading jobs are all scheduled in one unit capable of executing them all. Before that it was not uncommon that one shading unit was overloaded with work, while others where idle. The dedicated shading units and later the unified shaders share the concept of parallel execution. As the input data like vertices or pixels are mostly independent, they can be processed in parallel without much synchronization. The unified shaders extended this concept by adding more and more parallel processing units and nowadays GPUs arrived at the level of several hundred parallel processors integrated into one single chip. As the concept of unified shaders prevailed, the two major developers of dedicated GPUs started to actively support the GPGPU efforts. *NVIDIA* presented the *Compute Unified Device Architecture* (CUDA), while *ATI* (now *AMD*) developed *Close to Metal* (CTM, now *Stream SDK*). With the hardware vendors SDKs and APIs, GPGPU became attractive to software developers unfamiliar with the 3D graphics programming. Since then a large number of very different applications from scientific simulations to image processing or cryptography has been ported to GPUs.

2.2 GPU Hardware

The implementation for this work makes use of the *NVIDIA* CUDA toolkit. Therefore the focus here will be on *NVIDIA* hardware, as CUDA does not support third-party hardware. Since the *GeForce 8* series, all *NVIDIA* GPUs share a common architecture named *Tesla* [see 7]. An extremely extensive description also of the low-level functions can be found in an online article by Kanter [6], on which this overview is based.

The essential component of the Tesla architecture are single *Streaming Processor*s (SPs) arranged into groups of eight, forming a *Streaming Multiprocessor* (SM). Depending on the actual hardware generation, two (GeForce 8) or three (GeForce 200 series) SMs are combined into a *Thread Processing Cluster* (TPC) with some shared units.

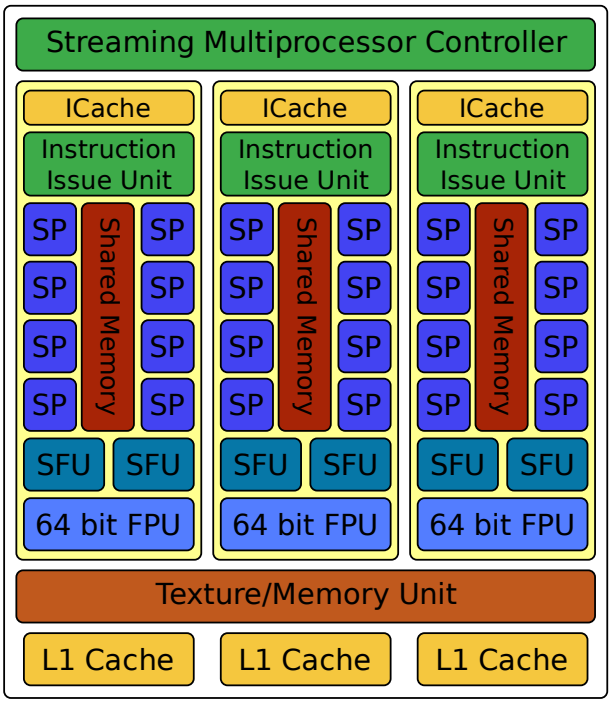


Figure 1: Diagram of a Thread Processing Cluster (TPC) used in current *NVIDIA* hardware

Figure 1 shows a diagram of one TPC with three SMs as used for example in the GeForce GTX 275. The single SPs have access to 16KB of on-chip shared memory. All SPs can read and write arbitrary regions of the shared memory, although bank conflicts [see 12, chap. 5.1.2.5] may occur for certain access patterns. However in typical cases, access to the shared memory is almost as fast as to each SPs individual registers. The single SPs are capable of 32 bit fixed and floating point operations, including advanced operations like fused multiply-add (MAD). Each SM contains two SFUs (*Special Function Units*), which are used for less common computations like transcendental functions, reciprocals, and square roots. All SPs in one SM execute the same instructions based on the decisions of the *Instruction Issue Unit*. It is a form of Single Instruction, Multiple Data (SIMD), often described in this context as Single Instruction, Multiple Thread (SIMT), as the units scheduled are threads and not packed data types. The GPUs of the newest generation (Compute Capability 1.3) also include a single 64 bit floating point unit shared among all 8 SPs. It is able to perform double precision fused MADs, compliant with the IEEE 754R floating-point specification [9]. As there is only one 64 bit FPU in each SM, the floating point performance using double precision is at least a factor of 8 lower than for single precision. However as double precision calculations are preferred in scientific computing future, hardware designs might address this problem. The TPC combines the SMs together with 8KB cache for each SM. The logic for actual memory accesses is part of the TPC and is controlled by the *Streaming Multiprocessor Controller*. Here also typical graphics related functions like the texture pipeline and render output units are located. The L1 cache of the TPCs is only used for constant data like global constants or textures. The hardware has no protocols for cache coherency, therefore the global read-write device memory can not be cached. As the shared memory is almost as fast as the hardware registers of the

SPs, it is often used as a software managed cache to overcome this difficulty. This approach requires an efficient use of the very limited local memory of merely 16 KB. The single SMs already support a high number of threads (768 or 1024), of which of course only 8 are executed at the same time. The hardware manages the threads in groups of currently 32, a unit which is called a *warp*. Warps are only used in the actual GPU hardware, on the software side only single threads and on a larger scale thread blocks are visible. For certain cases like global memory access, the developer has to take the warps into account.

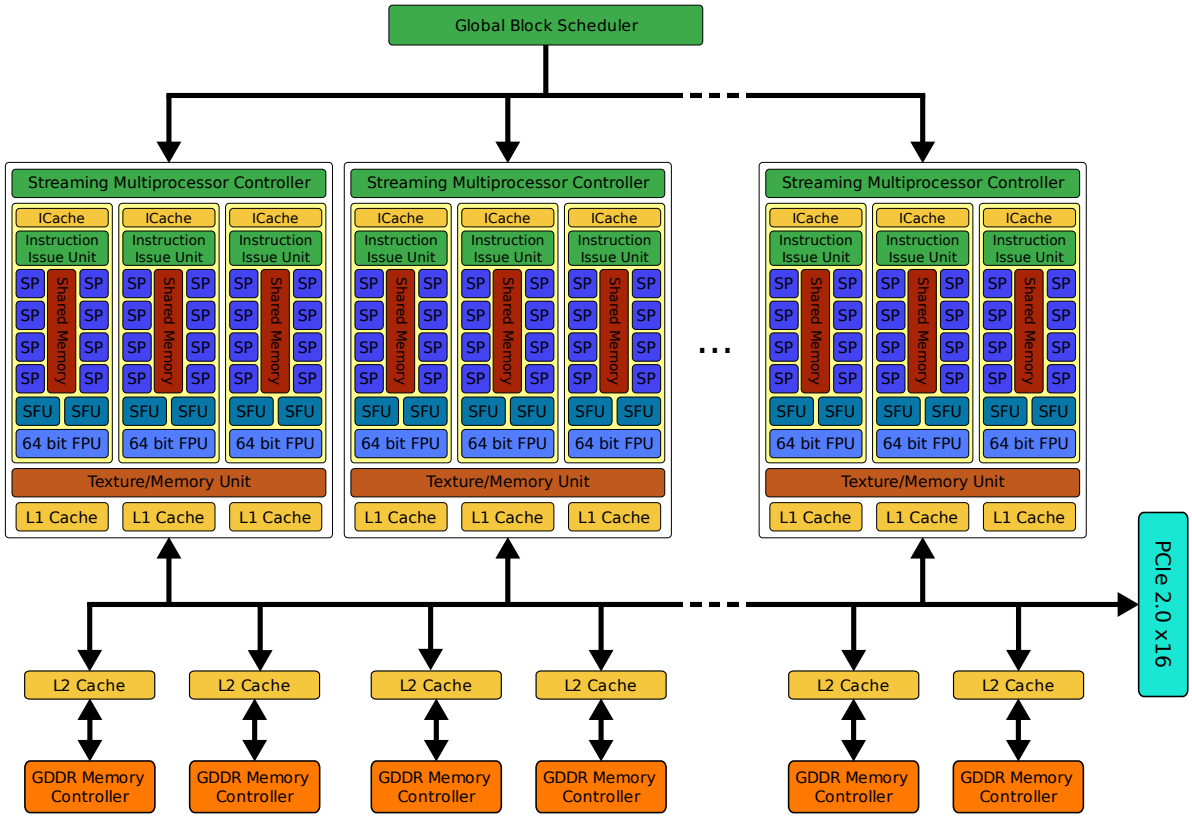


Figure 2: Diagram of a *NVIDIA* GeForce 200 series GPU

The Tesla architecture is scalable by the variable number of TPCs included in one GPU and by the clock rates of the various parts like SPs and memory. This simplifies the development of code using CUDA, as the underlying hardware architecture and the supported features stay mainly the same. Figure 2 shows a diagram of a GeForce 200 series GPU with multiple TPCs. The TPCs are linked to the global device memory by a shared bus. The reading of constant data and textures is cached by an additional L2 cache. The connection between the GPU and the host system is established by a PCI Express link.

In the course of this thesis, three different *NVIDIA* GPUs were used. These were two GeForce series 8 models: a GeForce 8600 GT and a GeForce 8800 Ultra. The most recent GPU which was also used for the performance evaluations was a GeForce GTX 275. The GPUs vary among other things in the supported features, the number of streaming processors and the amount of device memory. Table 1 gives a overview of the various hardware specifications. As only the GeForce GTX 275 is of Compute Capability 1.3, implementations using the double precision floating point format are limited to this device.

	GeForce 8600 GT	**GeForce 8800 Ultra**	**GeForce GTX 275**
Streaming Multiprocessors	4	16	30
Streaming Processors	32	128	240
Core Clock Rate (MHz)	540	612	633
Shader Clock Rate (MHz)	1180	1500	1404
Peak Performance (GFlops)	114.2	576.0	1010.9
Global Memory (GDDR3)			
Amount (MB)	256	768	896
Clock Rate (MHz)	700	1080	1134
Bus Width (bit)	128	384	448
Bandwidth (GB/s)	22.4	104	127
Resources per Streaming Multiprocessor			
Registers	8192		16384
Shared Memory (KB)	16		
Constant Memory Cache (KB)	8		

Table 1: Hardware specifications of used GPUs

2.3 *NVIDIA* CUDA Programming

The *Compute Unified Device Architecture* uses a thread-oriented programming paradigm. As described above, the hardware is capable of executing several hundred threads at the same time. Additionally a even much larger number of threads can be managed and scheduled accordingly. The current series of *NVIDIA* GPUs support up to several 10 000 threads on one single GPU. Clearly this is a very different concept than used in CPU programming where one may use multiple threads so make use of multi-core processors, but not in this extend. As the streaming processors are extremely simple compared to nowadays CPUs, CUDA programming differs strongly from serial programming. The first task the developer has to face, is the subdivision of the problem at hand into many smaller problems which can be mapped to different threads. This is not possible for all problems, but parallel algorithms are a well explored field in computer science. Working with such large numbers of threads requires some sort of grouping or blocking of threads. CUDA uses so-called *thread block*s, which is a one to three dimensional block of threads. These thread blocks can be composed of up to 512 single threads. The thread blocks themselves are organized in a one or two dimensional grid.

Figure 3 shows a grid with 3×3 thread blocks each containing $4 \times 4 \times 4$ threads. In practice the number of threads in a block and the number of thread blocks in the grid is of course much higher. The single threads can be identified by a index in their block (`threadIdx`) and by an index describing the position of the superordinate block (`blockIdx`). Using these indices it is possible to calculate for example array offsets depending on the thread index. One important property of the CUDA thread model concerns the communication between threads. As all SPs in a streaming multiprocessor share a common region of local memory, it is possible to exchange data between those threads. To deal with possible problems arising from concurrency, CUDA provides the `__syncthreads()` primitive. This adds a barrier to a CUDA program all threads in one block are synchronized at. It is implemented in hardware and if no waiting on single threads is necessary can be performed in a few clock cycles. A synchronization between threads in different blocks is not possible and can only be achieved implicitly after all thread blocks of a CUDA program are finished. In newer hardware versions some additional features were added. One is the possibility of atomic operations in global memory, however it has a serious

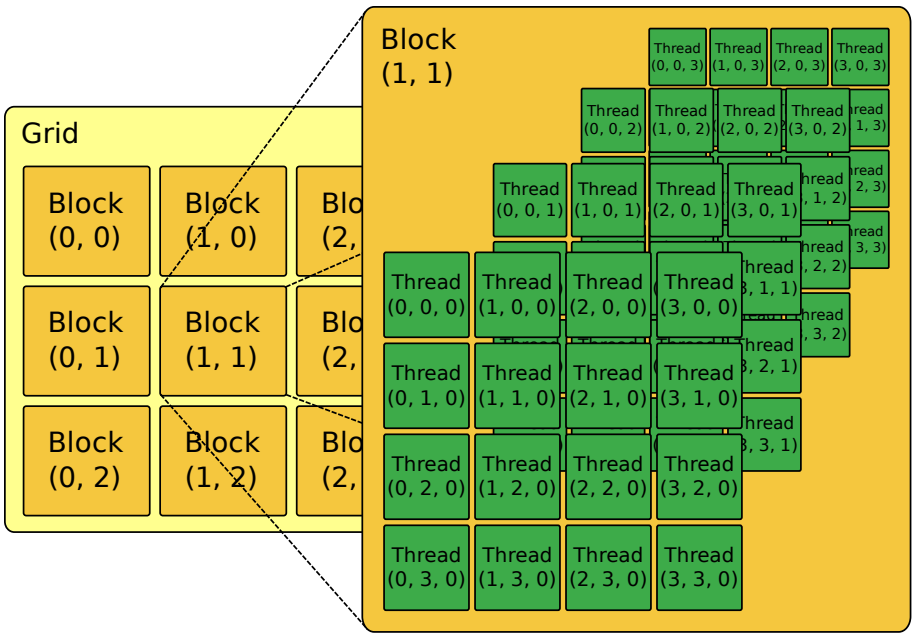


Figure 3: CUDA thread model with a grid of 3×3 thread blocks with $4 \times 4 \times 4$ threads

performance penalty and its use should be carefully assessed. Another new functionality are the warp voting functions `__any()` and `__all()` which can be used to evaluate predicates that may hold for an arbitrary or for all threads in a thread group.

The large number of threads is only one of some rather unfamiliar concepts employed by CUDA. The other one is the memory model. The memory hierarchy is quite different from the one common to current CPUs. The CUDA hardware has no concept of a stack of any form. This means all local data has to be held in registers or written to memory explicitly. Because of this there are also no function calls, in fact all calls are automatically inlined by the *NVIDIA* compiler. This explains the relative high number of registers in one SM. However the registers are not only used by the running threads, but are assigned to all threads in all blocks allocated to the streaming multiprocessor. This means that typically 100s of threads have to share the 8 or 16K registers. The same applies to the shared memory of which only 16KB are available in one SM. The CUDA hardware needs to maintain a much higher number of threads than can actually be run on one SM to hide the latency of global memory accesses. Although the bandwidth to the device memory is impressive compared for example to the host systems RAM, the latency is quite high with 400 to 600 clock cycles. This is amplified by the fact that global memory can not be cached, so the full latency occurs with every memory access. This problem is met by the scheduling in the SM. If one group of threads performs a memory access, they are put to sleep and another group of runnable threads begins execution. This can solely work if there is a high number of threads assigned to each SM. This poses some kind of dilemma: one the one hand there have to be many threads to hide the memory latency, but on the other hand more threads mean less registers and shared memory are available per thread. Table 2 explains this with a few examples of different thread counts in one block.

Aside from the already mentioned memory types register, shared memory, and global memory some others exist. If the number of registers is not sufficient to run all assigned threads or if the developer restricts the maximum number of registers per thread at compile time, registers might be spilled to a special memory region called *local memory*. Despite its name the local memory is a part of the global device memory and as such has the same

Resource	Compute Capability	Number of threads				
		1	128	256	512	786
Registers	1.0 – 1.1	8196	64	32	16	10
	≥ 1.2	16384	128	64	32	21
Shared Memory (Bytes)	any	16384	128	64	32	21

Table 2: Distribution of a SM's hardware resources depending on the number of threads

restrictions concerning bandwidth and latency. A memory type typically used in the context of graphics programming is the texture memory. It is read-only memory which is automatically cached. The hardware provides some special operations for texture handling, mainly related to sampling and filtering. The last memory type supported by CUDA is constant memory. This memory is intended for constants needed by all threads and is also automatically cached by the hardware. The amount of constant memory is however quite limited, only a total of 64KB is available. Table 3 provides an overview of the different memory types and from where they are accessible.

Memory Location	Thread	Threads in Same Block	Threads in Different Blocks	Host System
registers	r/w	–	–	–
shared memory	r/w	r/w	–	–
local memory	r/w	–[a]	–[a]	–
constant memory	r	r	r	w
texture memory	r	r	r	w
global memory	r/w	r/w	r/w	r/w

Table 3: Memory types supported by CUDA and the according access scope

[a] Theoretically an access would be possible, however the compiler should never generate such code

The essence is that threads in the same block are able to access and exchange data via the shared memory. For threads in different thread blocks this is not possible. If those threads need to communicate, they have to do so by using the global memory. An important characteristic of the CUDA architecture concerns the global memory accesses. As described earlier the global memory has a quite a high latency, compared to the local memory and registers, a drawback which is increased by the missing caching. However the GPU memory controller overcomes this by accessing not single elements from global memory, but blocks of 64 or 128 bytes. This means for each memory access a thread performs, a whole block is loaded. This is only efficient, if the threads next to the one requesting the read also read from a near by location. For devices with Compute Capability below 1.2, this implies that all blocks in one half-warp (16 threads) have to access consecutive memory regions. This is the ith thread in a half-warp accesses the ith element. Only then the memory operation is *coalesced* and the memory controller is able to perform it in one single request. If this is not the case, several memory accesses are performed, each with the full latency. The most current devices have a somewhat optimized memory controller. It is able to re-sort the memory requests of the threads in a half-warp. To even if they access the elements for example backwards, only one memory operation is necessary. Common to both device types are two additional conditions: The first address to be read has to be aligned to a multiple of 16 times the size of the accessed element. The second requisite is that all threads access elements of the same data type. If all

those requirements are fulfilled, then the CUDA-capable hardware reaches its full potential.

After this description of the hardware and programming model, some remarks on the actual development using CUDA should be added. The basis needed to be able to execute CUDA-enabled applications is the driver component, which is integrated into the *NVIDIA* display driver since some revisions. Atop of the driver lies the CUDA runtime API. The driver and runtime API support in principle the same operations, but the runtime API provides a notably higher level of abstraction. The runtime API is part of the so-called CUDA toolkit. Aside the runtime libraries it includes the *NVIDIA* CUDA compiler *nvcc*, which is rather a compiler driver than a single compiler. It uses the system's C++ compiler for the parts of the code running on the host. This implies that the host code can use all the features the installed compiler supports. The code targeted for execution on the GPU (device code) which is basically C/C++ code with some extensions is passed on to the *nvopencc*, a optimizing compiler based on the open source *Open64*. This step transforms the C/C++ code into an intermediate language named *Parallel Thread Execution* (PTX [13]). The output, which is comparable to a simple assembly language is transformed into the code executable on the GPU by the *ptxas*. At this point the PTX instructions are mapped onto the instruction set of the specified hardware identified by the Compute Capability.

The CUDA toolkit contains also some higher level libraries for linear algebra (CUBLAS) and Fast Fourier Transformations (CUFFT). As these were not used in this thesis, they will not be discussed here. Additionally the CUDA toolkit includes the documentation for the APIs and libraries.

3 Rigid Body Dynamics

The very basis for rigid body dynamics is classical mechanics as established by Isaac Newton. It uses positions and their derivatives to describe the state of bodies. The forces acting follow Newton's second law:

$$\vec{f} = \frac{\mathrm{d}}{\mathrm{d}t}(m\vec{v}) \tag{1}$$

Based on this law, all the forces acting between the various bodies can be determined. How this forces influence rigid bodies and how they can be calculated is the focus of this section. Prior to this however, a closer look at the *pe* is taken to clarify some of the concepts in the context of rigid body simulation.

3.1 *pe* Physics Engine

The *pe* is a framework for rigid body dynamics simulations developed at the Chair for System Simulation at the University of Erlangen-Nuremberg. It is entirely written in C++ and is fully object oriented. Currently the geometric primitives box, capsule, plane, and sphere are implemented. These can be combined into unions to generate more complex shapes. The engine is aimed at providing both physically accurate and fast simulations, to be fit for both scientific applications and game development. The simulation of the dynamics follows the typical loop of a physics engine as shown in Figure 4.

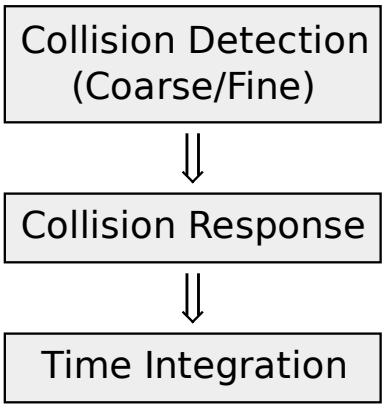


Figure 4: Main simulation loop of the *pe* physics engine

During the collision detection phase, the contacts between the bodies in the system are detected. This is done in a two step fashion. First the coarse collision detection is performed, which determines the bodies that might interact at all. After this the fine collision detection calculates the exact contact points. These contacts are processed in the next step, the collision response. Here forces are applied to the bodies involved in contacts with the aim to prevent interpenetrations. In this step also the friction between bodies is taken into account. The last step for each time step is the time integration. In this phase the single bodies are moved based on the forces calculated in the collision response as well as external forces like gravity. The *pe* includes a multitude of collision response solvers, among them projected Gauss-Seidel and projected conjugate gradient methods as well as a in-exact fast frictional dynamics solver. These solvers are interchangeable and this should be maintained by the CUDA solver developed.

3.2 Equations of Motion

The state of the bodies in the simulation system is represented by various properties. The position in the global world frame is determined by the location of the center of mass of the body. The rotation around this point is described either by a 3×3 rotation matrix or by a quaternion. Quaternions are an extension of complex numbers and are composed of four elements $\in \mathbb{R}$. The first three elements of a quaternion is the rotation axis, the last element describes the rotation. Quaternions are often used in 3D graphics to represent rotations, as they avoid problems like singularities which may arise from rotation matrices. In rigid body dynamics additionally a simplified concept for rotations with three parameters ϕ, θ, and ζ is utilized. The rotations in this form are especially useful during the phase of the collision response. For this case the representation is valid, as the rotations are small and local. After this the updated rotations are transformed to the more general form of rotation matrices or quaternions. For the calculation of the collision response, the generalized coordinates $\vec{q}(t)$ combining the position and the orientation of a body are used. For the 3D case the degrees of freedom for each body are 6, so $6n$ coordinates are needed to parametrize a system of n bodies leading to $\vec{q}(t) \in \mathbb{R}^{6n}$. Based on these generalized coordinates the generalized velocities $\vec{v}(t) \in \mathbb{R}^{6n}$ can be formulated, combining the linear and angular velocities of the bodies. For a simulation of the dynamics of a rigid body also the mass m and the inertia tensor $\mathbf{I}$ are needed. The inertia tensor of a body is determined by the mass distribution and describes the way the body reacts to rotational forces. Its role for rotations is comparable to the one of the inertial mass m for linear movements, hence the name *inertia tensor*. The inertia tensor is a real, symmetric matrix which is also positive definite. In addition to the generalized coordinates and velocities, a generalized mass matrix $\mathbf{M}(q)$ is composed of the masses and inertia tensors of the single bodies. The structure of the block diagonal mass matrix $\mathbf{M}(q)$ is as follows:

$$\mathbf{M} = \begin{pmatrix} m_1\mathbf{E} & & & & \\ & \mathbf{I}_1 & & & \\ & & \ddots & & \\ & & & m_n\mathbf{E} & \\ & & & & \mathbf{I}_n \end{pmatrix}$$

As mentioned in the introduction to this section, the movement of rigid bodies follows Newton's second law $\vec{f} = \frac{\mathrm{d}}{\mathrm{d}t}(m\vec{v})$. In the case of rigid body dynamics the force $\vec{f}$ is often a combination of different external forces acting on the bodies. The discretized equation of motions for a single body are:

$$\vec{q_i}(t + \Delta t) = \vec{q_i}(t) + \vec{v_i}(t) \cdot \Delta t + \frac{1}{2}\vec{\dot{v}_i} \cdot \Delta t^2$$

The calculation of the linear acceleration $\vec{\dot{v}_{i,l}}$ is straightforward using 1 and the fact that the mass of a body is constant over time:

$$\vec{f_i} = \frac{\mathrm{d}}{\mathrm{d}t}(m_i\vec{v_{i,l}}) = m_i \cdot \vec{\dot{v}_{i,l}}$$

$$\Rightarrow \vec{\dot{v}_{i,l}} = \vec{f_i} \cdot m_i^{-1}$$

However the angular acceleration $\vec{\dot{v}_{i,a}}$ (called $\vec{\omega}$ from here on) is somewhat more difficult to derive. The corresponding counterpart in angular movement for a force in linear movement is the torque ($\vec{\tau}$). The torque is defined as $\vec{\tau} = \vec{r} \times \vec{f}$, where $\vec{r}$ denotes the vector from the rotation axis to the point the force $\vec{f}$ is applied. The complete deduction of $\vec{\omega}$ can be found in [2, app. C.2]. Only the basic principles will be discussed here. As the inertia tensor $\mathbf{I}$ is only

valid for the body in its unrotated state, the inertia tensor has to be transformed if the body rotates. From the parallel axes theorem one can derive that for a rotation $\mathbf{R}$ the corresponding transformed inertia tensor $\mathbf{I}'$ is defined as:

$$\mathbf{I}' = \mathbf{R}\mathbf{I}\mathbf{R}^T$$

An additional fact has to be taken into account: the current angular velocity $\vec{\omega}$ of a body. In a special case even when the torque is zero, $\vec{\omega}$ can lead to an angular acceleration, namely if the body rotates not around an axis of symmetry. These combined effects lead to the somewhat complex term for the angular acceleration:

$$\dot{\vec{\omega}} = \mathbf{R}\mathbf{I}^{-1}\mathbf{R}^T(\vec{\tau} - \vec{\omega} \times (\mathbf{R}\mathbf{I}\mathbf{R}^T\vec{\omega}))$$

From this the calculation of the new angular velocity of each body for the current time step can be done effortless:

$$\vec{\omega}_i(t + \Delta t) = \vec{\omega}_i(t) + \dot{\vec{\omega}}_i \cdot \Delta t$$

The vector of forces $\vec{f}$ and the torque $\vec{\tau}$ are typically composed of external forces like gravity and internal forces resulting from collisions and friction. The internal forces are determined in the collision response phase, a process which is described in the next section.

3.3 Collision Response Calculation

The collision response phase is the most complex part in rigid body dynamics if it intends to be physically accurate. Collisions are described by contacts between two bodies, as determined during the collision detection step. A complete discussion of the contact modeling can be found in [17], from which only the basics are outlined here.

As contacts involve two different bodies b_a and b_b and the forces determined during the collision response are acting on the contacts, there has to be a way to map these forces to the bodies of each contact. For the linear forces acting only on the center of mass, nothing has to be done. But the torques are generated depending on the point where the forces are applied to the bodies. The vector from the center of mass of a body b to the contact point is denoted by $\vec{r}$. Using this vector the torque acting on a body is $\vec{\tau} = \vec{r} \times \vec{f}$. As we have not only one but N contacts in our system, the single contacts are assigned an individual index j. For each contact j a single force acting on both bodies is calculated. However depending on the setup of the contact this force acts positive on one and negative on the other body or the other way round. This is determined by the way the local coordinate system of the contact is chosen. Here by definition the contact normal points from the second (b_b) to the first body (b_a). Using the matrix representation $\mathbf{r}^\times$ for the cross product in the torque calculation, the transformation of the force $\vec{f}_{(j)}$ at the jth contact to the bodies b_a and b_b can be written as:

$$\begin{aligned} \begin{pmatrix} \vec{f}_{a,(j)} \\ \vec{\tau}_{a,(j)} \end{pmatrix} &= \begin{pmatrix} \mathbf{E} \\ \mathbf{r}^\times_{a,(j)} \end{pmatrix} \vec{f}_{(j)} \\ \begin{pmatrix} \vec{f}_{b,(j)} \\ \vec{\tau}_{b,(j)} \end{pmatrix} &= -\begin{pmatrix} \mathbf{E} \\ \mathbf{r}^\times_{b,(j)} \end{pmatrix} \vec{f}_{(j)} \end{aligned} \tag{2}$$

These transformations are not restricted to mapping the forces for all N contacts in the system. Instead it is possible to construct the matrix $\mathbf{J}(\vec{q})$ (see Eq. (3)), which maps any vector from the relative contact system to the according bodies. As there are n bodies and N contacts, it follows that $\mathbf{J}(\vec{q}) \in \mathbb{R}^{6n \times 3N}$. By applying the transpose of $\mathbf{J}(\vec{q})$, it is possible to translate properties from the single bodies back to the contact system.

$$\mathbf{J}_{ij} = \begin{cases} \begin{pmatrix} \mathbf{E} \\ \mathbf{r}^{\times}_{a,(j)} \end{pmatrix}, & \text{if } i = b_{a,(j)} \\ -\begin{pmatrix} \mathbf{E} \\ \mathbf{r}^{\times}_{b,(j)} \end{pmatrix}, & \text{if } i = b_{b,(j)} \\ \mathbf{0}, & \text{else} \end{cases} \qquad \text{for } i = 1 \dots n,\ j = 1 \dots N \tag{3}$$

The main purpose of the collision response phase is to prevent interpenetrations of bodies. It is important to note that contacts between bodies are not exclusively generated if bodies actually touch, but if they are close enough to each other. Only in doing so it is possible to resolve possible collisions. To model this, a function $\Phi(\vec{q})$, often called *gap function*, is introduced. $\Phi(\vec{q})$ describes the distance between the two contact points as show in Figure 5. To define the gap function, it is necessary to create a coordinate system local to the contact. The first component is the normal vector $\vec{n}_{(j)}$ at the contact point of the second body pointing towards the contact point of the first body. Orthogonal to $\vec{n}_{(j)}$ and each other, two tangential vectors $\vec{t}_{(j)}$ and $\vec{o}_{(j)}$ are chosen, which span a plane in the contact point. Combining this system into $\mathbf{Q}_{(j)}$ and using the transformation $\mathbf{J}$ it is possible to express the distance between the contact points $\vec{p}_{a,(j)}$ and $\vec{p}_{b,(j)}$ as:

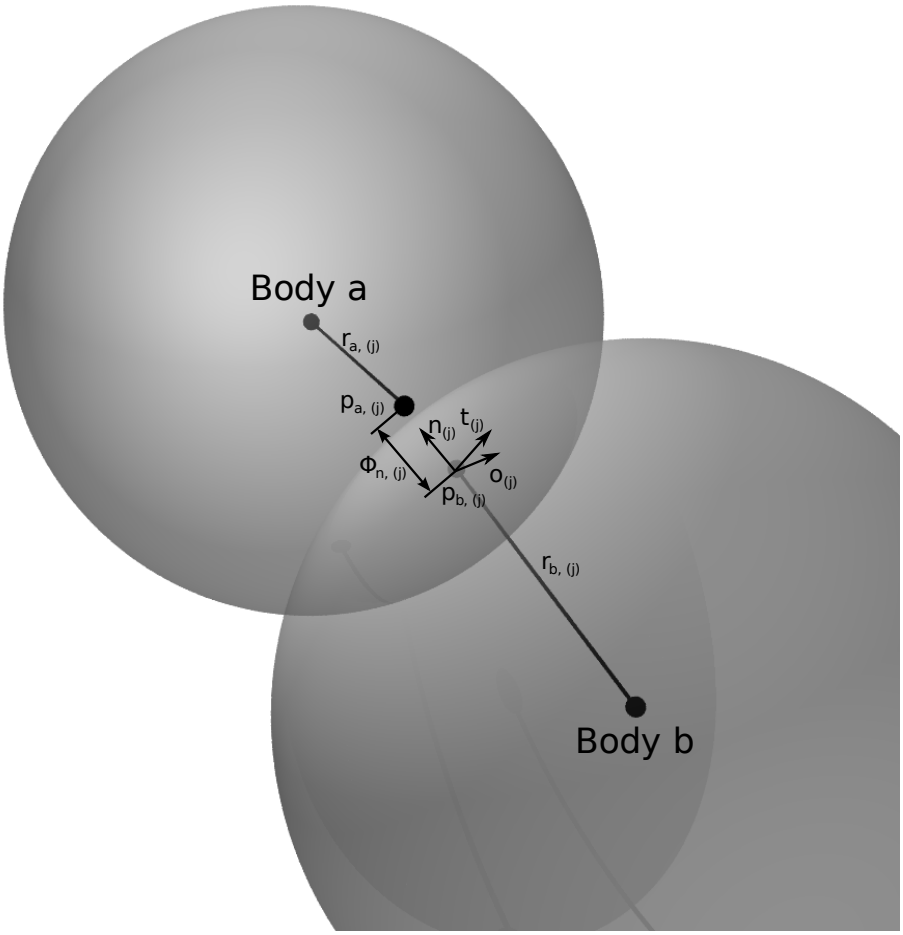


Figure 5: Contact j between two spheres with the normal vectors $\vec{n}_{(j)}$, $\vec{t}_{(j)}$, and $\vec{o}_{(j)}$

$$\Phi(\vec{q})_{(j)} = \begin{pmatrix} \vec{n}_{(j)}^T \\ \vec{t}_{(j)}^T \\ \vec{o}_{(j)}^T \end{pmatrix} (\vec{p}_{a,(j)} - \vec{p}_{b,(j)}) = \mathbf{Q}_{(j)}^T (\mathbf{J}^T \vec{q})_{(j)} \tag{4}$$

For the whole system instead of individual contacts this results in:

$$\Phi(\vec{q}) = \mathbf{Q}^T \mathbf{J}^T \vec{q} : \mathbb{R}^{6n} \to \mathbb{R}^{3N}$$

From here on, the focus will be on the normal component of the gap function, $\Phi_n(\vec{q})$. It describes the distance in normal direction and therefore the state the two bodies involved in a contact are in. For $\Phi_n(\vec{q}) > 0$, the bodies are still separated. If $\Phi_n(\vec{q})$ is negative, the bodies are interpenetrating each other. This is the case the collision response tries to prevent. In the case that $\Phi_n(\vec{q})$ is zero, the distance between the bodies is zero and they are actually in contact. This is also the only case where friction between the bodies is possible. From this a basic complementarity condition can be formulated. If $f_n(t)$ denotes the normal forces of all contacts the following condition has to hold:

$$f_n(t) \geq \vec{0} \perp \Phi_n(q(t)) \geq \vec{0} \tag{5}$$

The complementarity condition expressed by $\perp$ has to be satisfied for each component of the vectors. The interpretation of this complementarity condition is as follows: If the bodies are in contact, that is if $\Phi_n(q(t))$ is zero, there has to act a force $f_n(t)$ to prevent the bodies from interpenetration. However if $\Phi_n(q(t))$ is positive, the bodies do not even touch, so there cannot be any forces acting between them.

From Eq. (5) Preclik derives in [16] the following linear complementarity problem (LCP) for the frictionless case:

$$\begin{aligned} \mathbf{N}^T \mathbf{J}^T \mathbf{M}^{-1} \mathbf{J} \mathbf{N} \vec{x} + \mathbf{N}^T \mathbf{J}^T (v(t) + \Delta t \mathbf{M}^{-1} \vec{f}_{ext}) \geq \vec{0} \perp \vec{x} \geq \vec{0} \\ \text{with } \vec{x} = \Delta t f_n(t) \end{aligned} \tag{6}$$

$\mathbf{N}$ is similar to $\mathbf{Q}$, however it contains only the normal components. For contacts with a simplified friction model, Eq. (6) can be reformulated to:

$$\begin{aligned} \mathbf{Q}^T \mathbf{J}^T \mathbf{M}^{-1} \mathbf{J} \mathbf{Q} \vec{x} + \mathbf{Q}^T \mathbf{J}^T (v(t) + \Delta t \mathbf{M}^{-1} \vec{f}_{ext}) \geq \vec{0} \perp \underline{x}(\vec{x}) \leq \vec{x} \leq \overline{x}(\vec{x}) \\ \text{with } \vec{x} = \Delta t f_n(t) \end{aligned} \tag{7}$$

Here the matrix $\mathbf{Q}$ is used again, as the friction takes effect in the plane spanned by $\vec{t}_{(j)}$ and $\vec{o}_{(j)}$. For a pyramidal friction cone [17], the lower ($\underline{x}(\vec{x})$) and upper limits ($\overline{x}(\vec{x})$) are defined as:

$$\begin{aligned} \underline{x}(\vec{x}) &= (0, -\mu_{(1)} x_1, -\mu_{(1)} x_1, \ \ldots, 0, -\mu_{(N)} x_{3N-2}, -\mu_{(N)} x_{3N-2})^T \\ \overline{x}(\vec{x}) &= (\infty, \mu_{(1)} x_1, \mu_{(1)} x_1, \ \ldots, \infty, \mu_{(N)} x_{3N-2}, \mu_{(N)} x_{3N-2})^T \end{aligned} \tag{8}$$

The $\mu_{(j)}$ in this equation are the friction coefficients for the corresponding contacts. The tangential forces $\vec{f}_{t,(j)}$ at the jth contact point are restricted by the respective normal force $\vec{f}_{n,(j)}$ to $|\vec{f}_{t,(j)}| \leq \mu_{(j)} |\vec{f}_{n,(j)}|$.

An important property of the term $\mathbf{Q}^T\mathbf{J}^T\mathbf{M}^{-1}\mathbf{J}\mathbf{Q}$ from Eq. (7) has to be noted. The resulting matrix is sparse and is composed of 3×3 blocks. Because of the unstructured nature of this matrix, it is ordinaryly stored in a compressed matrix format like compressed row storage (CRS). However as such data structures are not well suited for GPU programming, the implementation developed in the course of this thesis never explicitly sets up the system matrix. Instead only the 3×3 blocks are stored in the corresponding contact object. This is adequate as for the used Jacobi method the complete system matrix is not needed, but only the inverse of the diagonal elements.

4 Infrastructure Implementation

Rigid body dynamics simulation uses certain data structures for the relevant elements in a system. The bodies have properties like position, velocity, and mass and the contacts hold pointers to the involved bodies and information about the contact's position relative to the bodies' centers of masses. The bodies and contacts are often kept in a list or vector and the implementations of the single phases like collision response or time integration iterate over these structures. This is perfect for serial execution. However, for GPU programming this layout is not adequate. For once it is very ineffective to copy all the small chunks of memory containing the body and contact information to the GPU one by one. Aside from this, the CUDA architecture achieves its maximum memory performance only if successive threads access close by (Compute Capability 1.3) or even successive memory locations (see Section 2.3). This is impossible to guarantee with many small memory sections stored in a vector. So for a successful port of the rigid body dynamics code to current GPUs it is crucial to develop optimized data structures both for data transfer between host and device and for the computation on the GPU.

4.1 *GPUsolve* Library

The *GPUsolve* library was developed by the author to address the requirement of data transfers between host and device as described above. The library initially was designed in the context of porting stencil based solvers for systems of linear equations arising from finite element methods to GPUs [21]. From there on *GPUsolve* was extended and adapted to the needs of rigid body dynamics simulations. It offers various vector-like data structures, along with optimized routines for data exchange between the CPU and GPU side. Aside from this a collection of small utilities are included to simplify tasks like accurate performance measurements of CUDA kernels or the management of system wide settings. The basic components of *GPUsolve* are described in the following sections.

4.1.1 Memory Types

For the different requirements in developing GPU enabled applications, three types of memory are supported:

- `HostMemory`
- `DeviceMemory`
- `ConstantDeviceMemory`

Common to all these memory types is the general implementation. The memory structures act only as a wrapper around the underlying basic data types like `int` or `float`. By having overloaded operators for `new()` and `delete()`, they can be used like any data type. The type safe access to the actual data element is enabled via the `operator()`.

`HostMemory` is intended for memory regions in the main memory on the host. Currently there are two different allocators implemented. The default (`StandardAllocator`) version makes use of the `operator new()` of the actual C++ implementation. Besides that, the `CUDAHostAllocator` is available, which uses the CUDA runtime API to allocate page locked memory.

Page locked (or *pinned*) memory is a memory region whose pages are marked by the operating system so they are never swapped out to background memory like disks. On POSIX systems this can be performed using the `mlock()` system call. If the locking is successful, the

corresponding pages are ensured to never be paged out, so they remain in physical RAM all the time. The fast Direct Memory Access (DMA) transfers supported by all current systems and graphics cards can only be performed to or from page locked memory. The reason lies in the way the hardware executes these DMA transfers. DMA by contrast to Programmed IO (PIO) is not carried out by the systems CPU, but by a separate entity, the DMA controller. This controller has no knowledge of the activity of the memory subsystem in the operation system. So it cannot determine which memory pages are paged to the swap area. Because of that the operating system has to make sure that pages involved in DMA transfers are never removed from RAM. The way the operating system does that is by allowing DMA transfers only to locked pages.

Pinned memory regions are the best choice for host to device and vice-versa copies, as the DMA transfers can be executed directly on the corresponding memory, without the need for an explicit buffer and an additional copy operation. Especially in the case of frequent data exchange between the host and device, it is advisible to use page locked memory for the involved data structures.

The `DeviceMemory` type is similar to the `HostMemory`, except of course the memory is allocated in the GPU memory. As the device memory is a completely different address space, the default C++ memory allocation operators cannot be used. Only by using the CUDA API it is possible to register memory on the GPU directly, as this operation is highly hardware and driver dependent. The corresponding allocator in *GPUsolve* is named `CUDADeviceAllocator`.

`ConstantDeviceMemory` is different, as these memory regions are not allocated dynamically. Instead this memory type serves as a wrapper around variables declared with the `__constant__` keyword, which instructs the CUDA compiler to place this variable in constant device memory. The usage of an own memory type however has the benefit of being able to offer type safe operators to write data to those constant parts of the GPU memory.

4.1.2 Data Structures

Based on the memory types described above, *GPUsolve* offers several vector classes. The vectors named `VectorD1`, `VectorD2`, and `VectorD3` are part of the namespace `gpusolve::expr` as they support basic expression templates, a concept described in depth in Section 4.2.1. The vector classes are implemented as C++ templates and are highly customizable. As there are a larger number of template parameters, a technique called *named template parameters* [23, chap. 16.1] was integrated. This allows the user to specify template parameters out of order and even access them by name. An example later in this section demonstrates these benefits. The aspects controllable by the template parameters of the vector classes are:

- data type (`int`, `float`, `double` etc.)
- device (CPU, GPU)
- memory allocator
- memory layout
- compile-time policy

The data type determines the type of the single elements in the vector. For the CPU there are virtually no constraints, but for vectors used in CUDA kernels not all data types are feasible. The types that are guaranteed to work are the build-in primitive data types like `int`, `float`, `double` (on hardware with Compute Capability 1.3) and types composed of these.

The device template parameter is used to identify vector instances and select the appropriate operators. The current implementation of the vector classes uses two specialization of the base templates, one for each of the supported devices. This strict separation allows the integration of code parts that only make sense for one device. For example it does not make much sense to provide a `print()` function to output the contents of a vector on the GPU, as no direct output from CUDA kernels is possible and the host cannot access the underlying memory directly.

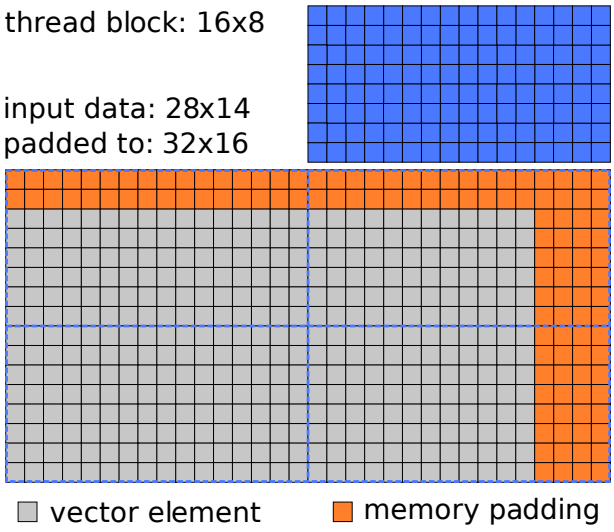


Figure 6: A `gpusolve::expr::VectorD2` is padded to multiples of the size of the processing thread block

The memory allocator may be one of the previous described or based on a user supplied implementation. The memory layout of the vector is controlled by an own policy, also selected via a template parameter. There exists a simple default policy `MemoryLayoutBasePolicy` which is used if the user does not override this parameter. The purpose of the memory layout policy is to provide a simple interface to control the memory allocation and element placement. As specified in Section 2.3 the CUDA architecture requires certain patterns for memory accesses to achieve maximum performance. Particularly important in this context is the alignment of the first element accessed by a half-warp. The address of this element has to be aligned to a multiple of 16 words of 32 bit. This becomes important in the case of the 2D and 3D vector. For these types the memory is linear for the first dimension x. The higher dimensions are mapped to a linear memory layout using column-major order. However, using this simple scheme memory accesses may not be properly aligned any more if the size of the vector in x direction is not a multiple of 16. Here the memory layout policy may be of help. The vector classes use the class specified via the template parameter to determine the padding of the memory for each dimension. For the alignment issue padding in one direction would be sufficient, but there are cases where padding in all directions is necessary. Many algorithms ported to GPUs process data in blocks. These blocks have a certain static size, mostly based on the dimensions of the thread blocks used. In these cases it is often much easier to load and write only full blocks from or to memory, instead of having special cases to handle the boundaries. Here the padding in the remaining dimensions can be employed. By implementing a small class doing the calculations needed for the memory padding, the user has full control over the memory layout of the vectors. Figure 6 shows a sample `VectorD2` padded in both dimensions for use with the depicted blocking scheme. Here blocks of 16 by 8 threads are used to process the input in form of a 2D vector. As the algorithm is optimized to load full blocks the benefit of faster memory access outweighs the disadvantage of some wasted memory for padding the `VectorD2` to full block sizes.

The code listing below shows some of the features of the *GPUsolve* data types. First two vectors are created, one on the host, one on the device. Next the data from the vector on the host is copied onto the device. A third vector of `int`s is allocated and placed in the constant memory of the GPU. As the actual data in the vector is referenced by a pointer in the `VectorD1`, only the pointer is placed in constant memory. The data itself is located in normal device memory and can be accessed as any other data in GPU memory. The benefit of this approach is that the pointer and the other members like sizes can be accessed using

the fast and cached constant memory, while the data remains in device memory as usual.

```
1  using namespace gpusolve;
2  using namespace gpusolve::expr;
3  using namespace gpusolve::policy;
4
5  // create 2D vector on host using a special memory layout
6  VectorD2<double, device::cpu, MemoryLayout_is<MyMemoryLayout> > a(1000, 3);
7
8  // create 2D on gpu using same memory layout and debugging support
9  VectorD2<double, device::gpu, MemoryLayout_is<MyMemoryLayout>,
       CompiletimePolicy_is<compiletime::DebugPolicy> > b(1000, 3);
10
11 // copy data from host to device
12 b = a;
13
14 // create 1D vector on the GPU and place it in constant memory
15 VectorD1<unsigned int, device::gpu> c(200);
16 __constant__ memory::ConstantDeviceMemory<VectorD1<unsigned int, device::gpu>
       > cConst;
17
18 cConst = c;
```

Listing 1: Various features of the *GPUsolve* data structures

By using the compile-time policy it is possible to control some features mainly useful for debugging. *GPUsolve* uses no `#define`s to differ between a performance optimized and a more debugging friendly code version. Instead policies are used to control these aspects on a per instance basis. For example it is possible to enable checks of array indices for just one instance of a vector, but keep the optimized version for all others. This approach is far superior to the traditional preprocessor based code selection. Not only does it reduce the probability of errors, but it is also very easy expandable and maintainable. As checks and evaluations of the compile-time policy classes are done by the compiler, there is no performance loss.

Apart from the compile-time policies, also the runtime behavior of many aspects in the library is manged by policies. Currently two classes of runtime policies are implemented, the `PerformancePolicy` and the `InfoPolicy`. The `PerformancePolicy` controls the amount of measurements *GPUsolve* conducts automatically to help the user evaluate the performance of various aspects like memory transfers or CUDA kernel calls. The `InfoPolicy` is focused on the generation of additional information on the data structures. The vector classes for instance support the generation of messages describing the performed memory alignments or assignments involving copy operations if instructed to do so by a runtime policy. The policies are upgradeable and each facet can be enabled or deactivated individually.

4.1.3 Mapping of Data Types

The efficient mapping of the common data types for rigid body dynamics to the optimized *GPUsolve* structures was an important step in the development. Only if the involved overhead is small enough, the full object oriented approach down to the CUDA kernel implementation can be successful. Many properties of the bodies or contacts are vectors. The position or speed of a body, the normals of contacts are all vectors with three components. Similar omnipresent are the 3×3 matrices and the quaternions. All these types can be easily stored in the data structures *GPUsolve* provides. Figure 7 shows how n single `Vector3`s as used by the *pe* are mapped into a single `gpusolve::expr::VectorD2` with a size of n by 3. The reason for the

separation of the vector elements of the `Vector3` lies in the CUDA architecture. All threads of a half-warp are executed in a single instruction multiple data (SIMD) fashion, meaning that all 16 threads perform exactly the same operations at the same time. As an example the CUDA kernel in Listing 2 should be considered.

```
1  using namespace gpusolve;
2
3  __global__ void VectorD2Access(expr::VectorD2<float, device::gpu>* input) {
4     // calculate the index of the current thread
5     const unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
6
7     // local variable
8     Vector3<float> x;
9
10    // load data from global memory to registers
11    for(int i = 0; i < 3; ++i) {
12       x[i] = (*input)(index, i);
13    }
14
15    // perform some calculations
16    // ...
17 }
```

Listing 2: Loading data from a `VectorD2` to a `Vector3` in a CUDA kernel

All threads load three elements determined by their respective thread index from global memory. As the threads are executed in parallel, the memory accesses to the ith element in the input vector occur at the same time. To achieve the maximum bandwidth, the accessed elements should be located next to each other. If the elements of a single `Vector3` were not split up, this memory access would waste $\frac{2}{3}$ of the memory bandwidth, as two unused elements had to be loaded for each accessed element.

The mapping for the quaternions and the 3×3 matrices is performed exactly the same way, although `VectorD2`s with a size of $y = 4$ and $y = 9$ are used of course. A further optimization for the access of the vectors in global device memory is described in the next section.

4.2 CUDA Math Module for *pe*

The *pe* provides the common data types like vectors and matrices of fixed sizes in its own math module. However these types could not be used in conjunction with CUDA, as they make heavy use of advanced C++ features like type traits and also require some parts of the *boost*[4] libraries to compile. The *NVIDIA* compiler is currently not able to work with the *pe* math types, so a special CUDA math module has been added to the namespace `pe::cuda`. These implementations are optimized to work both with CUDA and *GPUsolve* and also employ some advanced features described below.

4.2.1 Expression Templates

The calculations involved in rigid body dynamics make extensive use of basic elements like vectors, matrices, quaternions, and the corresponding operations. Often up to five different vectors or matrices are involved in one single expression. This is no problem in typical numerical computations on CPUs, however when ported to GPUs using architectures like CUDA, things become more difficult. As described in Section 2.3, the number of registers available to all threads running on a multiprocessor is strictly limited. As there exists no such concept as a stack known from CPU programming, all local variables have to be held in registers or

[4] http://www.boost.org/

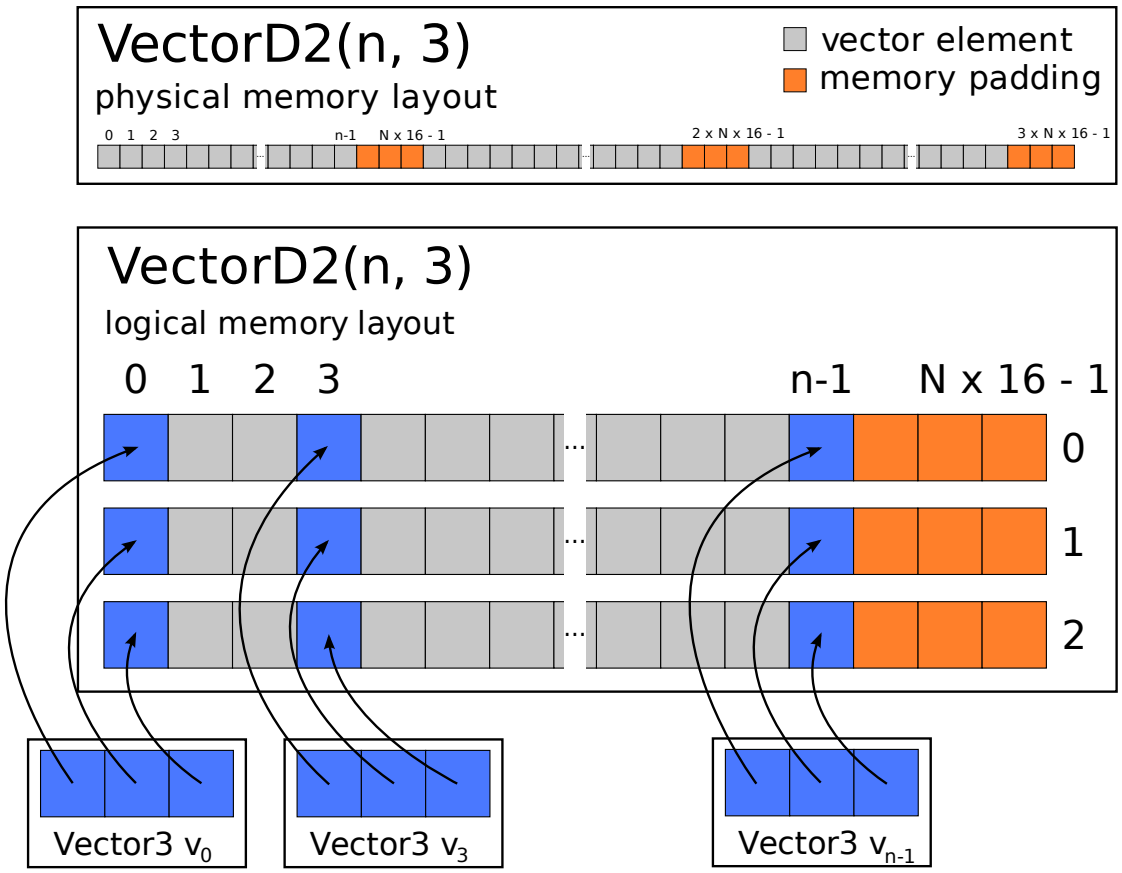


Figure 7: Mapping of n `Vector3`s to a `gpusolve::expr::VectorD2(n, 3)` with memory padding to blocks of 16 elements

be spilled to the local memory, if the number of registers is insufficient. This poses a great difficulty if one aims to use object oriented concepts in the development of GPU code. As an example consider the quite simple update of the linear speed of a rigid body:

$$v(t + \Delta t) = v(t) + (\vec{f} m^{-1} + \vec{g})\Delta t$$

In this term three vectors are involved ($\vec{v}$, $\vec{f}$, $\vec{g}$), as well as two scalar values (m, Δt). So if each of the vectors with its three components needs three registers, all in all 11 registers are needed just to store the variables. When using object oriented operators, further registers may be needed to store temporary objects. This will sooner or later become a problem as simply to many registers are used by each thread. The result is decreased performance as either the number of threads able to run in parallel is reduced or, if the number of registers per thread is limited by the compiler and registers therefore are spilled to local memory, more global memory accesses have to be performed.

The presented implementation takes a two step approach to overcome these difficulties. Firstly, a group of interface classes to the underlying data structures of the vectors, matrices, and quaternions was added. This means for each of the `Vector3`, `Matrix3x3` and `Quaternion` classes there is another version suffixed by `Global` which interfaces to the previously presented vector types from *GPUsolve* . The values are however never copied into the interface classes; only when single elements are accessed, the operation is delegated to the *GPUsolve* data structures. The interface classes themselves store a reference to the *GPUsolve* vector and an index. The index is used to select the offset in the `VectorD2` based on the actual thread index. Aside from this, a `gpusolve::expr::VectorD2Offset` object is contained. This is more or

less a array of offsets of the first entry in each block in y direction of a `VectorD2`. It is used to reduce the amount of offset calculation each thread has to perform. As in typical applications all vectors have the same size, for example the number of contacts or bodies, it would be redundant to calculate the offset for each vector access again. As the *GPUsolve* vectors used for the data exchange all have fixed sizes in y direction (3, 4 or 9), the corresponding offsets can be stored together in a `VectorD2Offset` and placed in constant device memory for fast access.

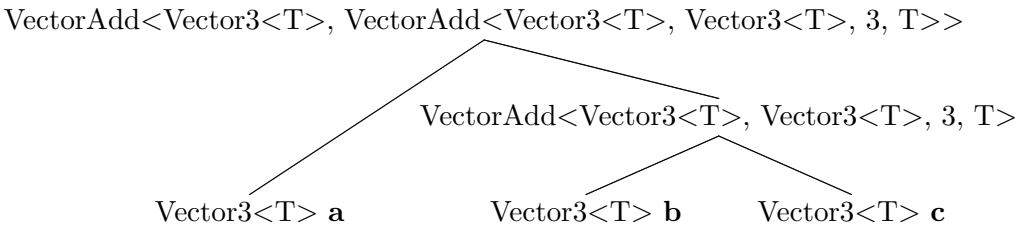


Figure 8: Addition of three `Vector3` objects **a**, **b**, and **c** using expression templates.

Secondly, a technique called *expression templates* was integrated into the linear algebra data types. A profound description and analysis thereof is rather complex and not the focus of this work, but can be found in [3]. Here only the most basic properties are discussed. The idea behind expression templates is the avoidance of temporary objects generated by typical object oriented programming. For example an overloaded `operator+` in a vector class normally returns a new vector with the sums of the two operands as elements. This has two implications. On the one hand, a new object is created, leading to higher memory usage. However this is not a real problem with today's highly optimizing compilers, which are able to eliminate such temporary objects. On the other hand, (and more important in this context to evaluate the sum) all elements of the two involved operands have to be available in registers. Expression templates can solve both these drawbacks of traditional operators. Instead of a new vector object containing the result of the computation a form of meta-vector is returned. This object has references to the involved operands and has the ability to perform the requested operation. For this an access operator is implemented using the same signature as the one in the vector class. The real work is done in the assignment operator of the vector. This operator calls the access operator of the given object, which in turn performs the necessary calculations. Of course this procedure can be used recursively. Figure 8 shows the expressions generated from two vector additions. For clarity the wrapping `VectorExpr` instances have been left out. The evaluation of this expression is sketched in Listing 3.

```
1  template<typename T>
2  class Vector3 : public VectorExpr<Vector3<T>, 3, T> {
3      T v_[3];
4
5      [...]
6      // evaluate expression
7      template<typename E>
8      Vector3& operator=(const VectorExpr<E, 3, T>& rhs) {
9          const E& v(rhs);
10
11         v_[0] = v[0];    // v_[0] = a[0] + b[0] + c[0]
12         v_[1] = v[1];    // v_[1] = a[1] + b[1] + c[1]
13         v_[2] = v[2];    // v_[2] = a[2] + b[2] + c[2]
14
15         return *this;
```

```
16      }
17  };
18
19  // operator+
20  template<typename A, typename B, size_t N, typename T>
21  class VectorAdd : public VectorExpr<VectorAdd<A, B, N, T>, N, T> {
22      private:
23          A a_;
24          B b_;
25
26      public:
27          VectorAdd(A a, B b) : a_(a), b_(b) {
28          }
29
30          T operator[](size_t index) const {
31              return a_[index] + b_[index];
32          }
33  };
34
35  template<typename A, typename B, size_t N, typename T>
36  inline VectorAdd<A, B, N, T> operator+(const VectorExpr<A, N, T>& a,
37       const VectorExpr<B, N, T>& b) {
38
39      return VectorAdd<A, B, N, T>(a, b);
40  }
```

Listing 3: Evaluation of an expression from Figure 8 in the assignment operator of `Vector3`

Expression templates are typically implemented to store references to the operands in the evaluating classes. However as line 23 and 24 of Listing 3 show, this is not the case in this implementation. This is a result of the way the *NVIDIA* compiler works. It could not optimize the references to the operands away, so for each expression, two pointers were stored. This resulted in a huge overhead due to temporary expression objects and was clearly not desired, so after much experimentation a quite simple solution was found. If the operands were stored as a copy in the expression classes, the compiler was able to optimize them away.

By combining expression templates with the interface classes a possibility arises to save a serious amount of local memory, i.e. registers. As for the computation of each entry in the resulting vector only the corresponding entries from the three input vectors are required. There is no need to allocate registers for all elements in these. Instead it is sufficient to load one entry from each vector and perform the summation. So for the evaluation of one element of the resulting vector, only four registers are needed to store data. Of these four registers three can be reused in the computation of the next element. So all in all only $3+3 = 6$ registers are needed to hold the values of the variables. Even less if the results are written directly to memory after the computation. In practice, the number of registers used will be rather higher, as additional registers are needed for the computation of offsets in the global data structures. Listing 4 shows the relevant part of `Vector3Global`, making use of the expression templates and the interfacing to the *GPUsolve* data types.

```
1   template<typename T, gpusolve::SolverDevice Device>
2   class Vector3Global : public VectorExpr<Vector3Global<T, Device>, 3, T> {
3       private:
4           typedef typename gpusolve::expr::VectorD2<T, Device> VecD2;
5           typedef gpusolve::expr::VectorD2Offset<3> VecD2Offset;
6
7           VecD2& v_;
8           VecD2Offset& vOffset_;
9           const size_t index_;
10
```

```
11      public:
12          // interface to gpusolve::expr::VectorD2
13          __host__ __device__ explicit inline Vector3Global(VecD2& v,
                VecD2Offset& vOffset, size_t index) : v_(v), vOffset_(vOffset),
                index_(index) {
14          }
15
16          // access operators
17          __host__ __device__ inline T operator[](size_t index) const {
18              return v_(vOffset_(index_, index));
19          }
20
21          __host__ __device__ inline T& operator[](size_t index) {
22              return v_(vOffset_(index_, index));
23          }
24
25          // evaluate expression
26          template<typename E>
27          __host__ __device__ inline Vector3Global& operator=(const VectorExpr<
                E, 3, T>& expr) {
28              const E& rhs(expr);
29
30              (*this)[0] = rhs[0];
31              (*this)[1] = rhs[1];
32              (*this)[2] = rhs[2];
33
34              return *this;
35          }
36
37          // [...]
38  };
```

Listing 4: Combination of the interface nature and expression templates in `Vector3Global`

The concept of expression templates of course is not restricted to vectors. All involved basic data types can benefit from it. Therefore three types of expressions were created: `VectorExpr`, `MatrixExpr` and `QuaternionExpr`.

The approach described above worked very well in general, however there were some disadvantages. As the interfaces do not cache accesses to the underlying data structures and global memory is not cached in the CUDA architecture, successive reads of the same elements resulted in additional memory accesses. This is mainly a problem with expressions containing for example the same vector twice. Also certain operators like the vector cross product read the single elements of the involved vectors repeatedly. For these cases and also for temporary results required in more than one calculation, it would be nice to have a traditional vector with its elements stored as member variables. So the simple `Vector3`, `Quaternion`, and `Matrix3x3` implementations were reactivated and extended to work with the expression templates. However it turned out that the concept of storing copies of the operands in the expression classes did not work well with these basic objects. The compiler copied the actual members for each single expression, leading to an extreme memory bloat. However the compiler performed the corresponding optimizations and eliminated the temporaries if only objects of one type were used. This also worked for expression classes using references to the operands. Based on this observation a quite elegant solution was found. The idea was to use copies of the operands for the `Global` types and references for the non-`Global` types. The selection of the corresponding method was implemented using C++ type traits. Listing 5 shows the base template and a specialization to select the correct way to store the operand based on the type of the vector. Only for `Vector3` a `const` reference is used, all other expressions use a copy of the operand.

The implementation of the expression classes does not have to be adapted, as the correct types for the stored operands are supplied by template parameters.

```
template<typename T, typename Data_Type>
class VectorExprType {
    public:
        typedef T Type;
};

template<typename Data_Type>
class VectorExprType<Vector3<Data_Type>, Data_Type>  {
    public:
        typedef const Vector3<Data_Type>& Type;
};
```

Listing 5: Implementation of the type trait to determine the type of the operands in an expression

A sample operator using these type traits would be implemented like this:

```
template<typename A, typename B, size_t N, typename T>
__host__ __device__ inline VectorAdd<typename VectorExprType<A, T>::Type,
    typename VectorExprType<B, T>::Type, N, T> operator+(const VectorExpr<A,
    N, T>& a, const VectorExpr<B, N, T>& b) {

    return VectorAdd<typename VectorExprType<A, T>::Type, typename
        VectorExprType<B, T>::Type, N, T>(a, b);
}
```

Using the presented technique, the *NVIDIA* compiler is able to perform all necessary optimizations to eliminate the temporary objects created by the expression templates.

4.2.2 Evaluation of Expression Templates

In order to test and validate the expression templates, some simple benchmarks were performed using both the expressions described previously and a more traditional alternative implementation. The following two approaches were analyzed:

- `pe::cuda` types *without* expression templates
- `pe::cuda` types *with* expression templates

The first sample was a modification of the well-known vector triad $\vec{a} = \vec{b} + c \cdot \vec{d}$ with $\vec{a}, \vec{b}, \vec{d} \in \mathbb{R}^n, c \in \mathbb{R}$. In the CUDA kernel vectors of size $n = 3$ were used, which were mapped to `gpusolve::expr::VectorD2`s for each single thread. The constant factor c was also individual to each thread, managed by a global `VectorD1`.

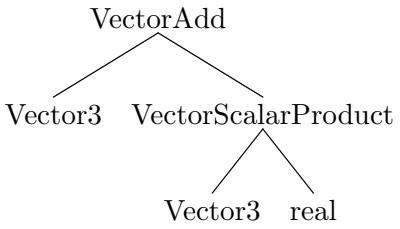


Figure 9: Expression tree of the vector triad $\vec{a} = \vec{b} + c \cdot \vec{d}$

The common part for each of the two implementations is shown in Listing 6, the relevant code of the different versions in Listings 7 and 8. For the implementation using the optimized `pe::cuda` data types, a common instance of `gpusolve::expr::VectorD2Offset` was used, as described in Section 4.2.1. The version with plain C++ may look inefficient due to the loop to read the input and write the result back to global memory. However the presented code proved to

be optimal, as the *NVIDIA* compiler unrolls the loops and does not allocate additional registers. To compare the code the *nvcc* compiler generated, decuda[5], a third party disassembler, was used to decode the actual device code. The assembly code from this step was processed by a python script to determine the number of memory accesses, operations performed and resources used.

```
1  using namespace gpusolve;
2  using namespace gpusolve::expr;
3
4  class MemoryLayoutPolicy : public policy::compiletime::MemoryLayoutBasePolicy
       {
5      private:
6          static const size_t blockSizeX_ = 16;
7
8      public:
9          static const MemoryPadding padding = gpusolve::padding::x;
10
11         static inline size_t sizeXPadded(size_t sizeX) {
12             return util::math::ceil<blockSizeX_>(sizeX);
13         }
14 };
15
16 typedef VectorD1<real, Device, MemoryLayout_is<MemoryLayoutPolicy> > VecD1;
17 typedef VectorD2<real, Device, MemoryLayout_is<MemoryLayoutPolicy> > VecD2;
18
19 __constant__ memory::ConstantDeviceMemory<Types<device::gpu>::VecD2 > aC;
20 __constant__ memory::ConstantDeviceMemory<Types<device::gpu>::VecD2 > bC;
21 __constant__ memory::ConstantDeviceMemory<Types<device::gpu>::VecD1 > cC;
22 __constant__ memory::ConstantDeviceMemory<Types<device::gpu>::VecD2 > dC;
23 __constant__ memory::ConstantDeviceMemory<VectorD2Offset<3> > offsetC;
24
25 __global__ void VectorTriad() {
26     const uint32_t index = blockIdx.x * blockDim.x + threadIdx.x;
27
28     Types<device::gpu>::VecD2& a = aC.getReference();
29     Types<device::gpu>::VecD2& b = bC.getReference();
30     Types<device::gpu>::VecD1& c = cC.getReference();
31     Types<device::gpu>::VecD2& d = dC.getReference();
32
33     if(index >= a.sizeX())
34         return;
35
36     // code of the different versions begins here
37     [...]
38 }
```

Listing 6: Common code of the vector triad $\vec{a} = \vec{b} + c \cdot \vec{d}$ test cases using the `pe::cuda` types

[5] http://www.cs.rug.nl/~wladimir/decuda/

```
cppmath::Vec3 aL, bL, dL;

const int sX = a.memSizeX();
int off = 0;
for(int i = 0; i < 3; ++i, off += sX
    ) {
    bL[i] = b(offset + index);
    dL[i] = d(offset + index);
}

aL = bL + c[index] * dL;

offset = 0;
for(int i = 0; i < 3; ++i, off += sX
    )
    a(offset + index) = aL[i];
```

Listing 7: CUDA kernel for vector traid using plain C++ types

```
// handle to offset object in
    constant
// memory
VectorD2Offset<3>& offset = offsetC
    .getReference();

etmath::Vec3G aL(a, offset, index);
etmath::Vec3G bL(b, offset, index);
etmath::Vec3G dL(d, offset, index);

aL = bL + c[index] * dL;
```

Listing 8: CUDA kernel for vector triad using `pe::cuda` types with expression templates and `VectorD2Offset<3>`

The results of this step are summarized in Table 4. It is apparent that activating the expression templates reduces the number of registers used by the kernel by $\frac{1}{3}$ for both single and double precision floating point types. This may not seem too dramatically, but one has to keep in mind the limited number of registers available for one single thread. Especially for memory bound CUDA kernels like they are likely to occur in rigid body dynamics, the architecture requires a high number of active threads to be able to hide the relatively high latency of global memory. Only if the single threads do not consume too much registers, it is possible to launch several blocks of 100s of threads on one multi processor.

Data Type	**Vector Type**	**Registers**	**Global Loads/ Stores**	**Constant Memory Loads**	**Flops**
`float`	plain C++	9	7 / 3	16	3
	ET	6	7 / 3	14	3
`double`	plain C++	15	7 / 3	16	3
	ET	10	7 / 3	14	3

Table 4: Resource usage of the CUDA kernels for the vector triad test case with and without expression templates (ET)

In Listings 9 and 10 the interesting parts of the assembly code for the two versions using single precision are shown side by side. Registers are addressed with `$r` followed by the number of the register, global memory by `g[0xabcd]` with `0xabcd` being the address. Shared memory which is not used in this case is accessed like global memory but with `s[...]` instead of the `g[...]`. There are multiple constant memory regions. The one for global constants (used here) is addressed by `c0[...]`. The registers are 64 bit wide, but only 32 bit operations are used here. In both code versions the floating point multiplication and addition are transformed into a fused multiply-add (`mad`).

```
1  mov.u32 $r5, g[$r2]
2  add.u32 $r2, $r1, c0[0x0160]
3  mov.u32 $r4, g[$r2]
4  add.u32 $r2, $r1, c0[0x0100]
5  mov.u32 $r2, g[$r2]
6  add.u32 $r0, $r0, $r6
7  shl.u32 $r3, $r3, 0x00000002
8  shl.u32 $r0, $r0, 0x00000002
9  mad.rn.f32 $r8, $r4, $r2, $r5
10 add.u32 $r4, $r3, c0[0x00b0]
11 mov.u32 $r7, g[$r4]
12 add.u32 $r4, $r3, c0[0x0160]
13 mov.u32 $r6, g[$r4]
14 add.u32 $r4, $r0, c0[0x00b0]
15 mov.u32 $r4, g[$r4]
16 add.u32 $r5, $r0, c0[0x0160]
17 mov.u32 $r5, g[$r5]
18 add.u32 $r1, $r1, c0[0x0050]
19 mov.u32 g[$r1], $r8
20 mad.rn.f32 $r6, $r2, $r6, $r7
21 add.u32 $r1, $r3, c0[0x0050]
22 mov.u32 g[$r1], $r6
23 mad.rn.f32 $r1, $r2, $r5, $r4
24 add.u32 $r0, $r0, c0[0x0050]
25 mov.end.u32 g[$r0], $r1
```

Listing 9: Disassembled CUDA device code for vector triad using plain C++ types

```
1  mov.u32 $r0, g[$r0]
2  add.u32 $r2, $r4, c0[0x00b0]
3  mov.u32 $r2, g[$r2]
4  add.u32 $r3, $r4, c0[0x0160]
5  mov.u32 $r3, g[$r3]
6  add.u32 $r5, $r1, c0[0x0178]
7  mad.rn.f32 $r3, $r3, $r0, $r2
8  add.u32 $r2, $r4, c0[0x0050]
9  shl.u32 $r4, $r5, 0x00000002
10 mov.u32 g[$r2], $r3
11 add.u32 $r2, $r4, c0[0x00b0]
12 mov.u32 $r2, g[$r2]
13 add.u32 $r3, $r4, c0[0x0160]
14 mov.u32 $r3, g[$r3]
15 add.u32 $r5, $r1, c0[0x0180]
16 mad.rn.f32 $r2, $r0, $r3, $r2
17 add.u32 $r1, $r4, c0[0x0050]
18 shl.u32 $r3, $r5, 0x00000002
19 mov.u32 g[$r1], $r2
20 add.u32 $r1, $r3, c0[0x00b0]
21 mov.u32 $r1, g[$r1]
22 add.u32 $r2, $r3, c0[0x0160]
23 mov.u32 $r2, g[$r2]
24 mad.rn.f32 $r1, $r0, $r2, $r1
25 add.u32 $r0, $r3, c0[0x0050]
26 mov.end.u32 g[$r0], $r1
```

Listing 10: Disassembled CUDA device code for vector triad using expression templates

One can see clearly the benefit of the evaluation of the expression templates in the assignment operator without the use of temporary vectors. In the plain C++ version the stores of the result values occur together in lines 19, 22, and 25. At this point all computations have been completed and the results have been held in registers up to this point. The version with expression templates interleaves the stores with the calculations of the corresponding value. After all data has been loaded for the evaluation of the first element of the result vector the `mad` is performed in line 7, the offset in global memory calculated in line 8 and the store takes place in line 10. The register used for the result of this vector operation (`$r3`) is then reused in line 13. This is one of the reasons for the reduced register usage of the vectors with expression templates. An other factor is the way the evaluation of the second and third element in the plain C++ version is done. While the first element is computed quite early in line 9, all other operands are loaded into registers before the next two `mad`s are performed.

While the vector triad was a nice basic test to analyze, it is still a quite trivial expression. During rigid body dynamics simulation more complex terms are omnipresent. Needless to say that the expression templates should also be tested using a not so basic term. As an example a part of the calculation of the gap function (see Section 3.3) was used: the evaluation of the expected position of the contact point $\vec{p}_a$ of body b_a based on the current velocities.

$$p_a(t + \Delta t) = p_a(t) + \vec{v}_a \cdot \Delta t + \vec{r_a} - (\vec{r_a} \times \vec{\omega_a}) \cdot \Delta t \tag{9}$$

This is a noticeable harder case, as it contains five different vectors (assuming the updated position is stored in a separate vector). Additionally the vector $\vec{r}_a$ is used twice, so multiple loads of the same values are to be prevented. On top of that a vector cross product (`operator%`)

is included. The difficulty with the cross product lies in its evaluation. For each partial result two values from each of the operands have to be combined. The complete expression tree for this test case named *vector expression* is presented in Figure 10. The variable Δt is common to all threads and supplied by a kernel parameter, which is stored by the CUDA architecture in shared memory for fast access.

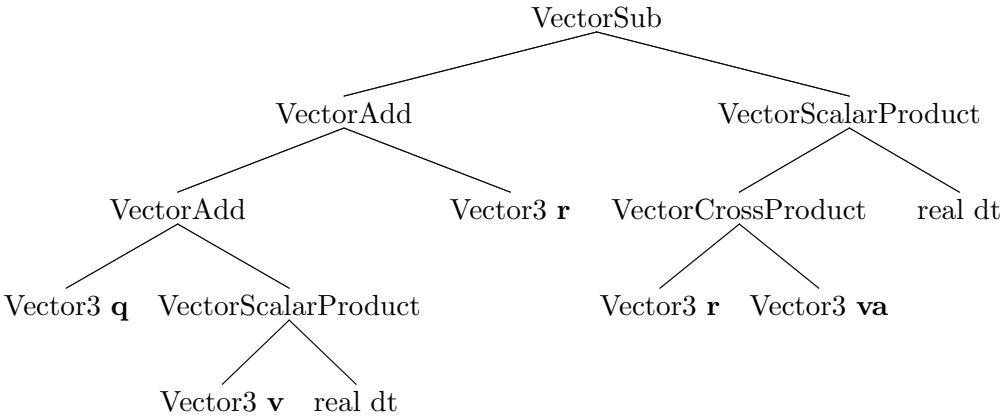


Figure 10: Expression tree for the evaluation of Eq. (9) using `Vector3`s

As this problem could benefit from the caching of the vectors **r** and **va**, two versions both using expression templates were examined. The first version used only the `Vector3Global` type described above. The results are of course not optimal as can be seen in Table 5. Although one (`float`) respectively five (`double`) registers less are needed, the number of loads from global memory is much higher as for the version using plain C++. The number of loads from constant memory is reduced, but there is no gain, as the constant memory is automatically cached and is thus as fast accessible as registers. To overcome this problem with the expression templates, the second version was implemented. Here `Vector3Global`s for **q** and **v**, but for **r** and **va** `Vector3`s are used. Because of this, **q** and **v** are practically cached in registers and no additional memory reads have to be performed. As the results show, the explicit caching clearly pays out well. The number of registers used is reduced while the number of global memory accesses stays on the optimal level. This test case also demonstrates how the `Global` interface classes and the traditional types can be used together depending on the specific requirements of the expression.

Data Type	**Vector Type**	**Registers**	**Global Loads/ Stores**	**Constant Memory Loads**	**Flops**
`float`	plain C++	14	12 / 3	28	13
	ET	13	21 / 3	21	15
	ET (cached)	11	12 / 3	21	13
`double`	plain C++	25	12 / 3	28	15
	ET	21	21 / 3	21	15
	ET (cached)	21	12 / 3	21	15

Table 5: Resource usage of the CUDA kernels for the vector expression test case with and without expression templates (ET)

5 Rigid Body Dynamics Implementation

The development of the *GPUsolve* library and the GPU optimized expression templates provided the means to tackle the actual problem at hand: rigid body dynamics. The corresponding parts of *pe* were not ported all at once, as this would make debugging and testing almost impossible.

5.1 Time Integration Step

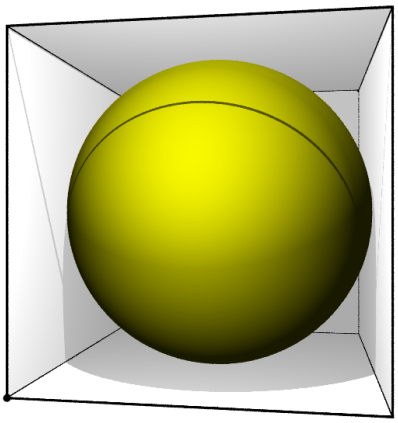
The first part of the simulation loop to be transformed into a GPU accelerated form was the time integration step. As described in Section 3.2, in this portion of the time step the forces calculated by the collision response are applied to the bodies in the system and the bodies are moved accordingly. Aside from this also the bounding boxes needed for the collision detection step are updated. This tasks offered a good starting point for parallelization, as all the bodies can be updated independently in parallel. There are no data dependencies and using the GPU optimized math module with expression templates the development of the CUDA kernel was straightforward.

5.1.1 CUDA Kernels for Time Integration

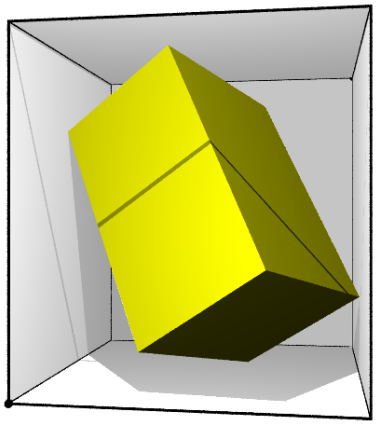
The update of the linear and angular movement was separated into two kernels, as there are no common sections and the CUDA compiler has its difficulties with reusing registers. So these two calculations were split up into the `MoveLinearKernel` and the `MoveAngularKernel`. After the movement of each body, the corresponding bounding box has to be updated as well. This proved to be somewhat more difficult, as the calculation of the axis-aligned bounding box (AABB) *pe* uses is different for each type of body. For the CPU version this is implemented using dynamic binding and virtual functions in each of the different body types. So for all the bodies in the system the corresponding function depending on its actual run-time type is called. As CUDA kernels achieve their maximum performance only if all threads in a warp follow the same execution path, it was not possible to maintain these different functions to determine the new AABB. In theory one could try to group all bodies of one type together, so most consecutive threads work on bodies of the same type. However this would require a complete reordering of all bodies prior to the kernel launch and before the synchronization of the results back to the host. Clearly this is not feasible as the overhead would undo any possible speedup of the parallel calculation. So the only remaining option was to adapt the calculation of the AABBs. For this a radius is introduced for each body and the calculation of the AABB handled each body as if it was a sphere. Figure 11 (a) shows the AABB of a sphere, as calculated both by *pe* and the CUDA code. In Figure 11 (b) the optimal axis-aligned bounding box for a cuboid is depicted. For the update of the AABB using the CUDA kernel, the box is enclosed in a sphere and the bounding box of this sphere is used instead (Figure 11 (c)). Using this bounding sphere, the update of the AABBs was greatly simplified and can be performed independently of the actual type of body. As the size of the bounding sphere is independent of the body's current orientation, the AABB only has to be moved according to the body's updated position. The kernel responsible for that task is the `UpdateBodyKernel`, which also performs some additional duties like keeping track of the current motion of the body.

5.1.2 Synchronization of Body Properties

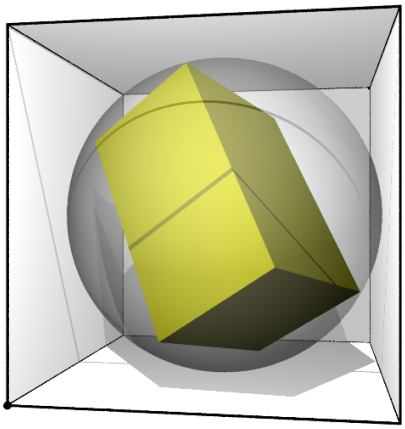
As the time integration step is only called once for each time step and in itself is not that computationally costly, is was most important to keep the overhead of the synchronization



(a) Axis aligned bounding box (AABB) of a sphere



(b) Optimal AABB of a box



(c) Wrapping bounding sphere of a box with the AABB of the sphere

Figure 11: Different types of bounding boxes as used in *pe* with GPU acceleration

between the host and device data structures low. This is especially important as long as not all parts of the rigid body simulations are performed on the GPU. If some day all steps in the simulation loop are completely ported to the GPU, this may change.

For the time being and of course to be able to test all parts of the engine independently, it was necessary to provide a system to keep the CPU and GPU data structures synchronized. For this purpose the `cuda::BodyManager` was added. It is responsible for managing the underlying data structures holding the actual body properties and for the copying of those between the host and device memory. At the core are the two instances of `cuda::BodyProperties`, one on the host and one on the device respectively. The `cuda::BodyProperties` class contains all the properties of the rigid bodies in the system, along with some global world settings like gravity. Listing 11 shows the class `pe::cuda::Types`, which provides some type definitions needed for all parts of the GPU port. In Figure 12 the class `pe::cuda::BodyProperties` is detailed, along with the purpose of its member variables. In addition to the show code, it provides some functionality to be resized and copied as needed.

```
namespace pe {
namespace cuda {

typedef float real;

template<gpusolve::SolverDevice Device, typename Data_Type = real>
class Types {
    public:
        // VectorD1
        typedef gpusolve::expr::VectorD1<Data_Type, Device, gpusolve::expr::
            MemoryLayout_is<MemoryLayoutPolicy> > VecD1;

        // VectorD2
        typedef gpusolve::expr::VectorD2<Data_Type, Device, gpusolve::expr::
            MemoryLayout_is<MemoryLayoutPolicy> > VecD2;
};

} // namespace cuda
} // namespace pe
```

Listing 11: Types used in the `pe::cuda` module

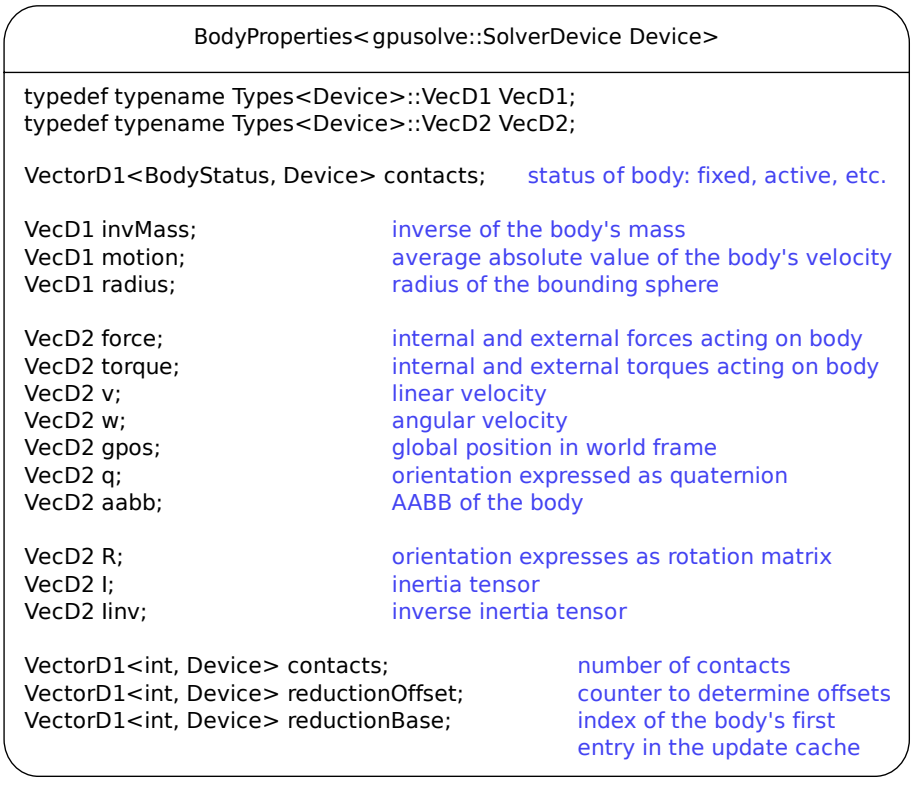


Figure 12: The `pe::cuda::BodyProperties` structure holding the data of all bodies in the system

The `cuda::BodyManager` provides the functions `syncDevice()` and `syncHost()` to keep the GPU data structures up to date and to copy the results of the kernel calls back to the host. The synchronization to the device can be performed in two modes: The default `syncDevice()` copies only the updated body properties like status and the forces and torques to the device. If however new bodies were added to the system, then `syncDeviceFull()` is called, which takes all body settings into account. Here some optimizations are performed to reduce the number

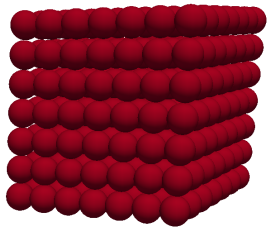
of updates. If the new bodies were added at the end of the CPU list of bodies, the previously existing bodies are oversteped. Also, if bodies are removed from the system, they are merely marked as removed. Only when a certain configurable threshold of removed bodies is reached, all data structures are rebuilt from scratch, as it is much cheaper to perform some unneeded computations for already removed bodies than to rebuild all body properties each time. To add or remove a body, the `cuda::BodyManager` provides the `bodyAdded(BodyID body)` and `bodyRemoved(BodyID body)` functions, where the BodyID is a pointer to the corresponding body. It is important to note that these two functions do not manipulate the data structures at each call. Rather they keep track of the added and removed bodies and only by the time a `syncDevice()` is performed, the updates are applied. This reduces the number of rebuilds of the `cuda::BodyProperties` structures dramatically, if a larger number of bodies is added or removed.

5.2 Collision Response Calculation

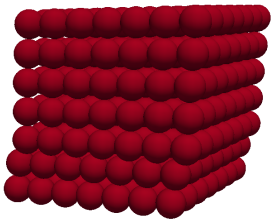
The calculation of the collision response is the most complex part in rigid body dynamics. If there are many contacts in the system, the collision response becomes the limiting factor for the overall performance of the physics engine.

To simplify the development of a CUDA accelerated collision solver, Tobias Preclik was kind enough to provide a simple rigid body dynamics simulator named *ballpark* after the initial test case. This code uses an elementary collision detection algorithm and supports only spheres as rigid bodies. The simulation world can be enclosed in a box to provoke collisions with walls and the floor. The collision response is calculated using a projected Gauss-Seidel method to solve the LCP from Eq. (7). Although *ballpark* is a system without many fancy features, it was perfect to develop a GPU based collision response solver.

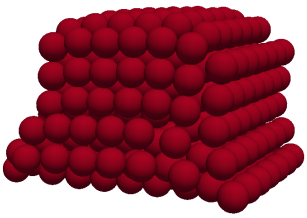
Parallel Reduction Upon examination of the main loop of the Gauss-Seidel solver it became apparent that it was impossible to port this solver to the GPU without major changes. The serial version of the solver iterates over all the contacts in the system and finds a new approximate solution for the LCP problem by adjusting the velocities of the two bodies involved in the single contacts. As the velocities of the bodies are updated in place and typically one body is part of more than one contact, the Gauss-Seidel nature of this solver is clearly visible. For the GPU version to work efficiently, the processing of the contacts has to be performed in parallel. While the contacts share no data between one and another, a single body is very likely part of several contacts. So if the solver was to be ported unchanged, the concurrency of the velocity updates would become a problem, as the different threads would be accessing the same memory location in an arbitrary order for both read and write operations. A solution for this is presented in detail in [22]. Instead of writing the velocity updates back instantly, all updates are collected in a sort of cache. This cache, named `pe::cuda::GPUUpdateCache` has two entries for each contact in the system, so each velocity update can be stored without interference. After each iteration all updates are summed up and added to the corresponding body. The summation of the cache entries is quite trivial if performed serial, but this is not an option on the GPU. Here a more complex approach is needed: The cache entries are grouped together by the body they belong to. This means if body b_a is involved in 5 different contacts, there will be 5 consecutive entries in the update cache reserved for b_a. In addition to the real data in the cache, each entry has a counter (called `reductionCount`), which indicates the offset of the entry's position to the position of the first entry for the corresponding body. These offsets as well as the indices for the cache are contained in each contact for both bodies. The calculation however has to be done on the CPU, as this is a problem, which is absolutely unfit for parallel execution on a GPU. But the introduction of the update cache has another



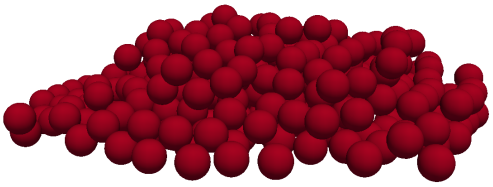
(a) Initial simulation setup



(b) Simulation after 250 time steps



(c) Simulation after 500 time steps



(d) Simulation after 1 000 time steps

Figure 13: Test case of *ballpark* with GPU support using a block of $7 \times 7 \times 7$ slightly misaligned spheres

implication. As all velocity updates are now collected and not applied at once, the solver lost its Gauss-Seidel nature and became a Jacobi solver.

GPUContact<gpusolve::SolverDevice Device>

```
typedef typename Types<Device>::VecD1 VecD1;
typedef typename Types<Device>::VecD2 VecD2;

typename Types<Device, uint32_t>::VecD1 bodyA, bodyB;                          indices of the bodies
typename Types<Device, uint32_t>::VecD1 reductionOffsetA, reductionOffsetB;    offsets in reduction cache
typename Types<Device, uint32_t>::VecD1 reductionCountA, reductionCountB;      number of entries in cache

VecD1 mu;                                          friction coefficient
VecD1 dist;                                        distance between contact points

VecD2 n, t, o;                                     normal and tangentials at contact point
VecD2 p;                                           approximation of the solution of the LCP
VecD2 rA, rB;                                      vectors from centers of bodies to contact points
VecD2 diagblock;                                   3x3 block of the system matrix
VecD2 phi;                                         value of the gap function at current time step

VectorD1<int, real> residual, residualReduced;     buffers for residual reduction
```

Figure 14: The `pe::response::cuda::GPUContact` structure manages the data of all contacts

Reduction of Velocity Updates The summation of the single cache entries, called reduction from here on, is performed by the GPU in parallel and the performance depends mainly on the maximum number of entries for a single body. The problem can be found under the name of *parallel segmented reduction* in the literature, as it is a reduction using the summation as an operator working on independent segments for the single bodies. The solution presented in [22] and implemented in this work is by no means optimal, however an efficient CUDA implementation of a parallel segmented reduction is rather difficulty but possible [see 20]. The kernel which performs the reductions is called `ReduceVelocitiesUpdatesKernel()` and is presented in Listing 12.

```
1  __global__ void ReduceVelocitiesUpdatesKernel(uint32_t s) {
2      const uint32_t index = blockIdx.x * blockDim.x + threadIdx.x;
3
4      GPUUpdateCache<device::gpu>& cache = constUpdateCache.getReference();
5      gpusolve::expr::VectorD2Offset<3>& offsets = cache.offsets_;
6
7      if(index >= cache.size())
8          return;
9
10     const uint32_t reductionCount = cache.reductionCount_[index];
11     const uint32_t offset = (0x1 << (s-1));
12     const uint32_t nextSum = index - offset;
13
14     if(!(reductionCount & offset))
15         return;
16
17     Vec3G(cache.deltaV_, offsets, nextSum) += Vec3G(cache.deltaV_, offsets,
           index);
18     Vec3G(cache.deltaW_, offsets, nextSum) += Vec3G(cache.deltaW_, offsets,
           index);
19     cache.reductionCount_[index] = 0;
20 }
```

Listing 12: CUDA kernel to perform the parallel segmented reduction of the velocity updates

The kernel uses several levels or steps, identified by the parameter s. The number of steps is determined by the maximum number of contacts in any body. If c_{max} is the maximum number of contacts any body has, then the number of necessary steps $s = \lceil \log_2 c_{max} \rceil$. The current implementation performs s calls to the reduction kernel with $n = 1 \dots s$. The CUDA kernel uses one thread for each entry. In every step the current offset o for each thread is calculated as $o = 2^{n-1}$. Only if the `reductionCount` of the thread's entry is a uneven multiple of o, the actual reduction is performed and the entry is marked as processed by setting the `reductionCount` to 0. After s steps, all the velocity updates have been summed up in the cache entry identified by the `reducionBase` for each body. Figure 15 shows the reduction steps for the cache entries of five distinct bodies. The number of contacts the bodies are involved in is diverse. The first body is only involved in one contact, while the third is participating in six contacts and thus has six entries in the update cache. As the maximum number of contacts over all bodies is 6, $s = \lceil \log_2 6 \rceil = 3$ reduction steps are needed.

By looking at this simple example one can see, that the reduction of the velocities became a significant part in the solver. For each Jacobi iteration there are s reduction steps which have to be performed. In typical cases s is well below 10, however even then there are a high number of additional kernel calls necessary. After each reduction step, the velocity updates are still contained in the `GPUUpdateCache`. To apply them to the actual bodies, another kernel is needed which is called `UpdateVelocitiesKernel()`.

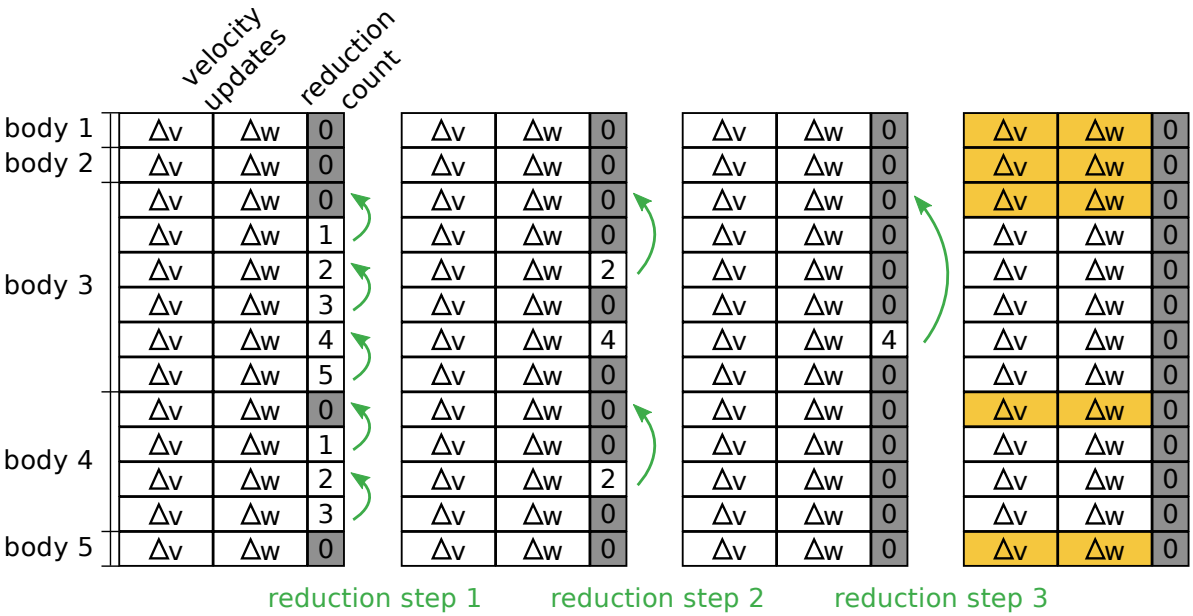


Figure 15: Simple example showing the reduction of the velocity updates for 5 bodies in 3 steps

Under-relaxation Upon the first tests of the solver with the *ballpark* code it became apparent that the solver was not stable in many cases. While it worked fine for very simple setups, it diverged for slightly more complex test cases. After making sure there was no mistake in the solver implementation and verifying each single calculation, a solution was found. If a dampening factor ω is included in the Jacobi method, the solver turned out to be significantly more stable. The grade of under-relaxation necessary depends on the specific problem. Various tests showed that depending on the setup, a factor ω between 0.7 and 0.2 was required. Theoretically it should be possible to detect any divergence at runtime and adjust the dampening factor accordingly, but this has not yet been implemented. The under-relaxation also reduces the convergence rate of the Jacobi method, which is not the best to begin with. This is no implementation dependent problem. For a better convergence behaviour, a different or at least modified solver would be needed.

Reduction of Residual The initial instance of the CUDA solver had just one break condition: the maximum number of iterations. This was fine for the testing of the various components, but for the actual use as a solver in *pe* it was inapt. The ideal solution would be to compare the residual to some kind of threshold ϵ. But due to the parallel processing of the contacts a similar problem as with the velocity updates arise. How should the residual not only of one thread but for all contacts be determined? The answer was again a reduction. This time however the reduction operator was not an addition, but the maximum. With this it is possible to compute the maximum residual across all contacts. The result can be used to determine whether further iterations of the solver are necessary. The implementation of the residual reduction is based on the general solution presented by Harris in [4]. There a highly optimized solution using all features of the CUDA architecture like shared memory is described. Using this as a basis, a efficient reduction of the residual was implemented. The residual calculation is not intended to be performed after each time step. Currently the solver checks the residual only every 50 iterations. This seems to be a good trade-off between too many iterations and too much overhead due to the residual reduction.

```
1  for 0 ≤ it < maxIterations do
2  |  CalcNextPhiKernel();
3  |  ResolveCollisionsKernel();
4  |  for 1 < n ≤ s do
5  |  |  ReduceVelocityUpdatesKernel(n);
6  |  end
7  |  UpdateVelocitiesKernel();
8  |  if ReduceResidualKernel() < ϵ then
9  |  |  break
10 end
11 UpdateBodiesKernel();
```

Algorithm 1: CUDA kernels for the collision response calculation

Jacobi Solver Loop The whole algorithm for the collision response step is sketched in Alg. 1. First the current values for the gap function Φ are determined in the CUDA kernel `CalcNextPhiWFKernel()` depending on the current velocities. This kernel processes the list of contacts and accesses data from both the contact and body data structure. After that the corresponding collision response for both bodies in each contact is computed in the `ResolveCollisionsKernel()`. This is the main part of the solver. All data structures (bodies, contacts, and update cache) are needed for the calculation. The updates of the velocities are stored in the update cache, which is processed in the loop by subsequent calls to `ReduceVelocityUpdatesKernel()`, which adds up the single updates for each body. After this step, the updates are summed up in the first entry of each body, from where `UpdateVelocitiesKernel()` reads them and stores them in the appropriate entry in the body data structure. If desired the residual can be calculated at this point using `ReduceResidualKernel()`. This should not be done every iteration, but only every so often. Now one iteration of the Jacobi solver is complete and the updated velocities can be used in the next step. When the maximum number of iterations or adequate convergence is reached, the solver loop is quit.

5.3 Integration of the Solver into *pe*

To integrate the projected Jacobi solver developed using *ballpark* into *pe* , some adjustments were necessary. The synchronization of the rigid bodies between host and GPU was used from the time integration step as described in Section 5.1.2. The contacts required some more work. The data structures optimal for the presented solver differ to some extend from the ones used in *pe* . However the contact data structures are only build once during each time step and the contacts themselves change from one time step to the next. So it is admissible to perform a rebuild of the CUDA optimized data structures using the ones of *pe* as a basis prior to the launch of the solver. In this step also the diagonal block of the system matrix is computed and stored with the corresponding contact.

This was not the only required change. The *pe* and its previous solvers are working with forces. This means the collision response does not examine and change the bodies velocities, but rather the forces acting on the bodies. These forces were applied in the time integration step together with the external forces like gravity. To be compatible with the developed CUDA solver, a additional step had to be inserted into the simulation loop. Prior to the calculation of the collision response, all body related data is synchronized to the GPU. After that the corresponding forces are applied to the bodies resulting in updated velocities. Figure 16 shows the modified simulation loop with the integrated GPU solver. The collision detection is performed completely on the CPU, after that the data structures containing the body properties are synchronized between host and device. The rest of the simulation loop is now ported to the GPU. After the time integration, the updated body properties like position, velocity and bounding box are copied back to the host. This concludes the simulation of one time step.

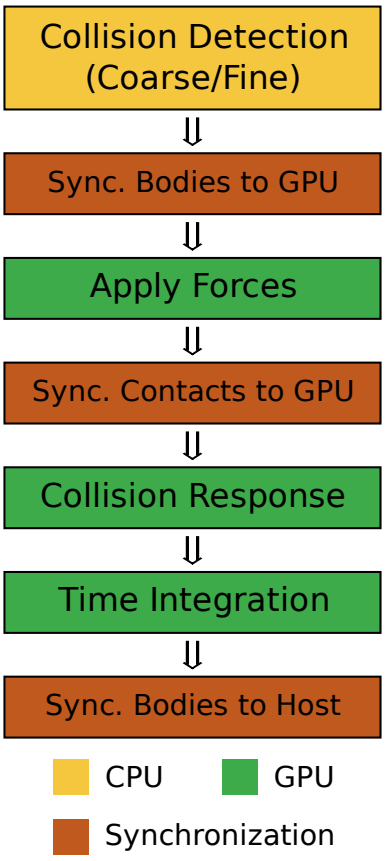


Figure 16: Main simulation loop of *pe* with CUDA support

6 Results

To validate the collision response solver and the time integration step, a series of tests using various examples from *pe* were used.

6.1 Reference Setup

To test the Jacobi solver developed using CUDA, a modified test case derived from the *well* example in *pe* was used. It is composed of a round well placed on a plane with spheres falling in from above. The well consists of 150 boxes placed in five levels to form the well. The boxes are fixed in their location and can not move. The spheres are inserted above the well in a random manner, but always so that they do not overlap. The test case was run with a time step of $\Delta t = 0.0025$ for 20 000 simulation steps. The input of new spheres each with radius 1.0 was already stopped after 18 000 time steps to ensure the solution was stable in the static case. At this point 3 900 spheres had been added to the simulation. The termination condition for the solver was $\epsilon = 5e^{-3}$, the maximum number of iterations it_{max} was limited to 2 000 while the residual was evaluated every 50 time steps. The ϵ is quite high, whereas the it_{max} is rather low. This is based on the observation that the convergence rate of the Jacobi solver is quite bad. If it did not reach a residual lower than ϵ in the first 2 000 time steps, it seldom did so at all. To make sure that the approximated solution was good enough nevertheless, the maximum interpenetration of bodies was constantly monitored. For this setup the interpenetration was always below 0.05 and did not build up over the several thousand time steps.

As a second test the *granular flow* example provided with *pe* was used. In this application a hopper filled with spheres is simulated. As the bodies are not in contact at first, a kind of warm-up phase of 5 000 iterations is used to let the system develop a stable state. After this an outlet is opened at the bottom of the hopper. The difficulties which already became evident in the previous test intensified. The solver was not able to solve the LCP accurate enough to prevent prevent interpenetrations of bodies. The result was that over the course of several 1 000 time steps the interpenetrations grew and the solver was not able to revert the system to a valid state.

6.2 Performance Evaluation

To identify the computationally most complex CUDA kernels, statistics of all the implemented kernels were generated. Table 6 shows the resource usage of the kernels for the application of forces, collision response and time integration. The `ApplyForcesAngularKernel` stands out from the first group. The reason for the high number of global memory reads and floating point operations is the complex calculation of $\vec{\omega}$ as described in Section 3.2. But as the number of registers used in this function is limited to 32 and no local memory is used, it poses no real bottleneck. Additionally the forces are applied only once every time step, so the kernel is used quite seldom. As one could expect, the `ResolveCollisionsKernel` is quite costly. The large number of involved variables like contact normals, the bodies masses, and inertia tensors lead to a high number of global memory accesses. The `ReduceResidualKernel` is the only one using local memory. The amount used (528 Byte) however is so low it does not restrict the number of blocks or threads in any way. Interestingly the `MoveAngularKernel` uses local memory, although the number of registers is not so high, that register spilling would be required. This effect is caused by the use of the transcendental functions sine and cosine during the calculation of the quaternion describing the rotation of the bodies.

To be able to actually determine the amount of time spend in the various parts of the simulation, a number of timers were added. As a test case the *well* example was used with the setup as described above. All test were run using single precision data types on the GTX 275

CUDA Kernel	Registers	Global Reads	Global Writes	Extra	Flops
ApplyForcesLinearKernel	11	8	3	–	6
ApplyForcesAngularKernel	32	34	3	–	63
CalcNextPhiKernel	16	26	3	–	30
ResolveCollisionsKernel	25	66	17	–	73
ReduceVelocitiesUpdatesKernel	11	13	7	–	6
UpdateVelocitiesKernel	10	15	6	–	6
ReduceResidualKernel	6	2	1	528B Shared	8
MoveLinearKernel	9	7	6	–	6
MoveAngularKernel	20	8	16	20B Local	80
UpdateBodyKernel	14	15	14	–	16

Table 6: Resource usage of the CUDA kernels implemented

presented in Section 2.2. The collision response alone accounted for 99% of the time spent in GPU related functions.The synchronization was next with below 1% and the application of forces and the time integration both were in the region of 1‰.

Another test was run to determine the memory throughput and floating point performance of the kernels part of the solver using the hopper test case. The results in Table 7 were measured during the setup phase when the bodies get into contact. The 9 857 bodies in the simulation where involved in about 26 000 contacts. With the CUDA solver using blocks of 128 threads slightly over 200 blocks were handled. This resulted in at least 6 blocks per streaming multiprocessor. So the maximum performance should be reached using this setup. As it is not trivial to determine the number of operations and memory accesses performed by the reduction kernels, no exact values concerning memory bandwidth and floating point performance are available.

CUDA Kernel	Memory Read	Memory Write	Memory Combined	Floating Point Performance
CalcNextPhiKernel	17.7 GB/s	2.1 GB/s	19.8 GB/s	5.5 GFlops
ResolveCollisionsKernel	16.5 GB/s	4.2 GB/s	20.7 GB/s	4.5 GFlops
UpdateVelocitiesKernel	23.5 GB/s	9.4 GB/s	32.9 GB/s	2.5 GFlops

Table 7: Resource usage of the CUDA kernels implemented

By looking at the results and the theoretical achievable performance, a clear discrepancy is obvious. The kernels did neither max out the memory bandwidth nor the floating point performance. The reason for this is can be found in the data access patterns. `CalcNextPhiKernel` needs data from the bodies as well as from the contact it is working on. Only the accesses to the contact data structures are coalesced and reach high performance. The picture is the same with the other two kernels. They all have to access different data structures, while only one of them is optimal for the current access pattern. This can not be changed too easily, as it lies in the principle of the problem. However the memory performance of the CUDA kernels is still well beyond the level of current CPUs, so a speedup is very well possible.

To gain some final insight on the influence the reduction steps have on the overall performance, the following list provides an overview of the average run times of the CUDA kernels:

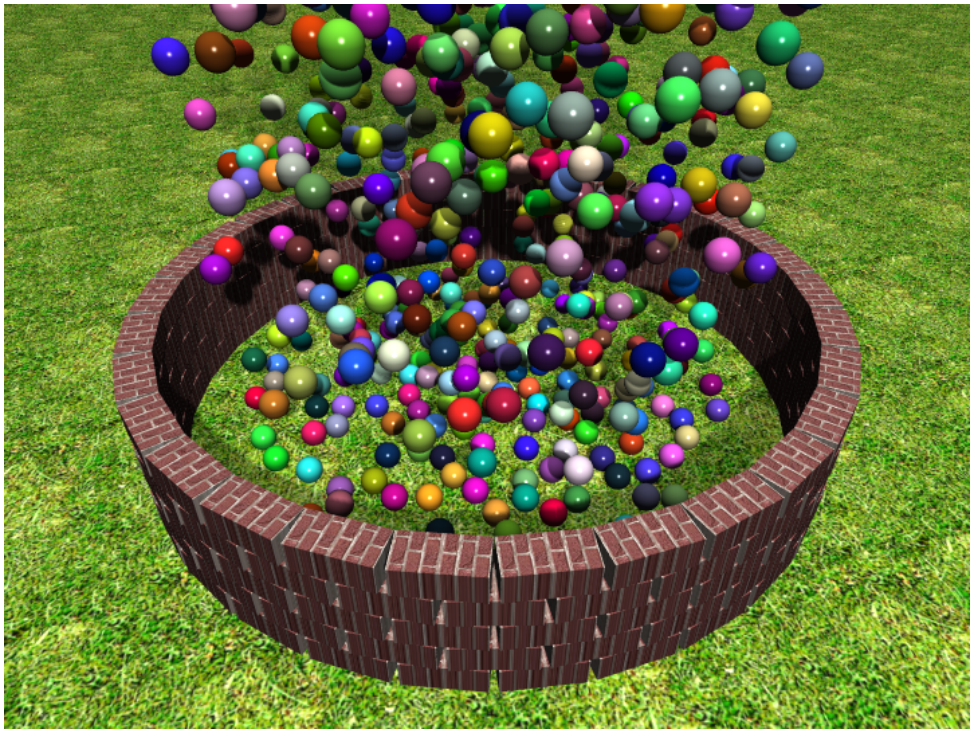
- CalcNextPhiKernel: $1.5e^{-4}s$
- ResolveCollisionsKernel: $4.0e^{-4}s$

- ReduceVelocitiesUpdatesKernel: $5.3e^{-5}s$
- ReduceResidualKernel: $1.1e^{-3}s$
- UpdateVelocitiesKernel: $3.5e^{-5}s$

If one assumes the number of reductions in each step is below 5 most of the time, then the reduction of the velocities and the update of the velocity entries in each body is still a little bit faster than the actual collision response. However for more than five reduction steps, this does not hold any more. So there is no single point in the solver where one could gain performance considerably. Instead the whole system of steps required to perform one iteration has to be optimized. But on the other hand, if one is able to reduce the memory throughput significantly for one kernel, the same should be applicable to all solvers.

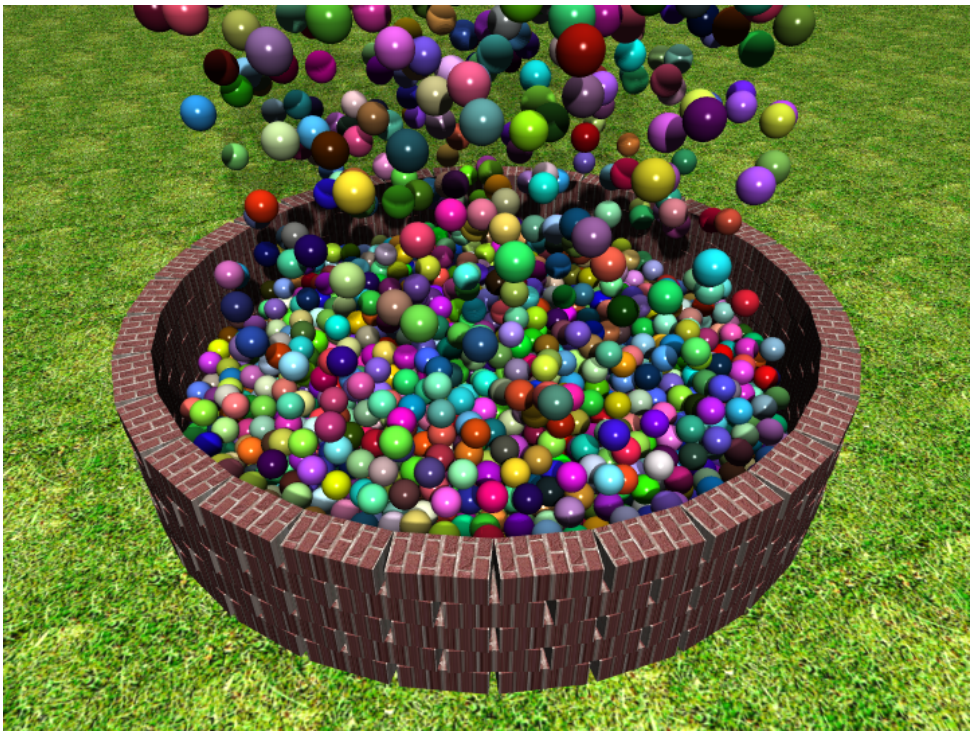
6.3 Images

Below some images rendered by *POV-ray*[6] using the output from the *well* simulation are shown. The visual results also verify the correct approximation of the solution computed by the Jacobi solver. The setup is the one described above, with the exactly same parameters.



(a) The first spheres have reached the ground (3 000 time steps)

[6] http://www.povray.org/



(b) The spheres are piling up in the well (10 000 time steps)

(c) All spheres have reached the overflowing well (20 000 time steps)

Figure 16: Some results from the *well* test case, rendered using *POV-ray*

7 Conclusion

In the scope of this thesis, the integration of a GPU accelerated rigid body dynamics simulator into an existing object-oriented physics engine was performed. By using some advanced C++ concepts it is possible to employ object orientation down to the level of GPU programming using an model like CUDA. This leads to clean interfaces and still performs very well. The application of the expression templates concept proved to be useful to reduce the resources each thread requires. The development of the *GPUsolve* library and the parts integrated into *pe* show that the time of GPU programming as a modification of graphics programming is definitely over. The main problem in this regard is probably the *NVIDIA* compiler suite and CUDA itself. As features are added and sometimes removed over time, it is constantly necessary to adjust the own code to perform optimal with the current version of the CUDA toolkit. Nevertheless it is possible to develop libraries not limited to one application. *GPUsolve* and the CUDA math module added to *pe* are by no means restricted to the usage presented here.

The complexity of rigid body dynamics, especially in the collision response phase remain. Some parts were easily ported like the time integration, mainly because of the independent calculations without any need for synchronizations and are perfectly suited for the execution using a parallel architecture. In contrast the collision response comes along with serious difficulties. The missing data locality due to the bodies being part of arbitrary contacts does not fit the architecture of current GPUs. Due to this the performance of this most time consuming step in the simulation loop is not as good as one would expect at first glance. The high memory bandwidth achievable with sequential memory accesses could not fully be used. Another factor is the necessary under-relaxation to prevent the system from diverging. This combined with the low convergence rate of Jacobi methods leads to a solver with rather bad convergence. Basically some additional iterations on the GPU are relatively cheap, as the expensive part is the setup and transfer of the body and contact data, but this is only valid to some point. Unfortunately the reduction of the velocity updates has to be performed after each solver iteration and as it uses multiple kernel calls poses some additional limiting factor.

However at least some of the addressed problems may be attenuated by further optimizations of the collision response kernels. One could try to modify the data structures so more of the bodies properties are included in the contacts and thus accessible by optimized memory transfers. This would require some additional copying and setup phases, but it may be worth the effort, if the solver performs many iterations.

On a different note it would be very interesting to examine some other LCP solvers. It is very well possible that there is a solver or maybe a combination of solvers which a both fitted for the GPU architecture and provide a accurate solution and better convergence. The Jacobi solver should be seen as a first step. A significant part of the invested work was centered on the data structures, which are not specific for one solver. Also the part of the physics engine currently remaining on the CPU, the collision detection, may be an interesting field for future work. There exist some very interesting concepts like spatial subdivision [see 8, chap. 32] which seem to work well on GPUs. If all parts of the engine were ported to the GPU, some of the synchronization overhead could be avoided. A further concept would be the support for multiple GPUs in one system or for several systems equipped with CUDA-enabled GPUs in conjunction with some message passing system like MPI. So a lot of exiting work remains in the domain of rigid body dynamics and GPGPU.

References

[1] Mihai Anitescu and Alessandro Tasora. An iterative approach for cone complementarity problems for nonsmooth dynamics. *Computational Optimization and Applications*, 2008. doi: http://dx.doi.org/10.1007/s10589-008-9223-4.

[2] David Baraff. Siggraph 1997 Online Course - An Introduction to Physically Based Modeling: Rigid Body Simulation II – Nonpenetration Constraints . Course Notes, 1997. http://www.cs.cmu.edu/~baraff/sigcourse/.

[3] Jochen Härdtlein. *Moderne Expression Templates Programmierung - Weiterentwickelte Techniken und deren Einsatz zur Lösung partieller Differentialgleichungen*. PhD thesis, Friedrich-Alexander University Erlangen-Nuremberg, 2007.

[4] Mark Harris. Optimizing Parallel Reduction in CUDA. CUDA SDK, 2007. http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/reduction/doc/reduction.pdf.

[5] E. J. Haug. *Computer aided kinematics and dynamics of mechanical systems. Vol. 1: basic methods*. Allyn & Bacon, Inc., Needham Heights, MA, USA, 1989. ISBN 0205116698.

[6] David Kanter. NVIDIA's GT200: Inside a Parallel Processor. Online article, September 2008. http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242.

[7] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, 2008. ISSN 0272-1732.

[8] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley Professional, 2007. ISBN 9780321515261.

[9] NVIDIA Corporation. NVIDIA GeForce GTX 200 GPU Architectural Overview. Technical report, May 2008.

[10] NVIDIA Corporation. *NVIDIA CUDA C Programming - Best Practices Guide*, 2.3 edition, July 2009.

[11] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2.3 edition, July 2009.

[12] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture - Reference Manual*, 2.3 edition, July 2009.

[13] NVIDIA Corporation. *NVIDIA Compute - PTX: Parallel Thread Execution*, 1.4 edition, June 2009.

[14] John Owens. Siggraph 2007 GPGPU Course - GPU Architecture Overview. Course Notes, August 2007. http://www.gpgpu.org/s2007/.

[15] Matt Pharr. *GPU Gems 2*. Addison-Wesley, Boston, 2005. ISBN 0321335597.

[16] T. Preclik. Frictional Rigid Body Dynamics. Studienarbeit, Friedrich-Alexander University Erlangen-Nuremberg, Aug 2007. http://www10.informatik.uni-erlangen.de/Publications/Theses/2007/Preclik_SA_07.pdf.

[17] T. Preclik. Iterative Rigid Body Dynamics. Diplomarbeit, Friedrich-Alexander University Erlangen-Nuremberg, Nov 2008. http://www10.informatik.uni-erlangen.de/Publications/Theses/2008/Preclik_DA08.pdf.

[18] Jörg Sauer and Elmar Schömer. A constraint-based approach to rigid body dynamics for virtual reality applications. In *VRST '98: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 153–162, New York, NY, USA, 1998. ACM. ISBN 1581130198. doi: http://doi.acm.org/10.1145/293701.293721.

[19] Florian Scheck. *Theoretische Physik 1: Mechanik. Von den Newtonschen Gesetzen zum deterministischen Chaos*. Springer, 7. Aufl. edition, 2002. ISBN 9783540435464.

[20] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. ISBN 9781595936257.

[21] S. Strobl. Efficient Implementation of Finite Element Operators on GPUs. Studienarbeit, Friedrich-Alexander University Erlangen-Nuremberg, Sept 2008. http://www10.informatik.uni-erlangen.de/Publications/Theses/2008/Strobl_SA08.pdf.

[22] Alessandro Tasora and Dan Negrut. A parallel algorithm for solving complex multibody problems with stream processors. *International Journal for Computational Vision and Biomechanics*, 2009. ISSN 0973-6778.

[23] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates*. Addison-Wesley, Boston, 2003. ISBN 0201734842.