ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Center of Mechanics
Prof. Dr. Ch. Glocker
ETH Zürich
CH- 8092 Zürich

IMES
Institute of Mechanical Systems

Master Thesis
Summer 2011

Computing Non-Smooth Dynamics on the GPU

Gabriel Nützi

Supervisors: Adrian Schweizer
Dr. Michael Möller
Prof. Dr. Ch. Glocker

August 2011

Selbstständigkeitserklärung

Ich erkläre mit meiner Unterschrift, das Merkblatt Plagiat zur Kenntnis genommen, die vorliegende Arbeit selbständig verfasst und die im betroffenen Fachgebiet üblichen Zitiervorschriften eingehalten zu haben.
Merkblatt Plagiat:
`http://www.ethz.ch/students/semester/plagiarism_s_de.pdf`

Zürich, 1. August 2011 

Ort, Datum Unterschrift

Acknowledgement

I would like to thank Adrian Schweizer for his vital encouragement and support. I am also heartily thankful to Dr. Michael Möller, who has made available his support in number of ways. I was very glad about his useful inputs during this thesis. To bear in mind too that without Michael Möller's engagement, many programming mistakes and numerical problems would have taken months to track down! Sincere thanks go also to Prof. Dr. Christoph Glocker for all the very interesting (sometimes very challenging) discussions about the art of mechanics. Last but not least, I would like to thank all other people at IMES who supported me in any way.

Abstract

Figure 0.1.: An example of a rigid body simulation with hundreds of small wood bricks.

The goal of this thesis is to develop and test numerical methods used in large rigid body simulations on the graphics processing unit (GPU). The graphics processing unit offers several hundred computing cores which can be used in parallel with the help of a high-level programming language such as CUDA or OpenCL. To determine the contact forces of thousands of contacts which arise during rigid body simulation, two popular numerical procedures can be used: The Jacobi Over-Relaxation (JOR) and the Over-Relaxed Gauss-Seidel (SOR) scheme in combination with a projection (Prox) onto a convex set. These two procedures have been parallelized and optimized for the NIVIDA GPU architecture CUDA. NVIDIA's GPU GTX 580 has been used as the major test platform. The JOR and SOR Prox have been tested in different aspects such as correctness and performance. This work should give an insight in the techniques and problems which arise by programming parallel algorithms for the GPU.

Contents

Notation

x	A scalar.
$\mathbf{x}$	A vector.
$\mathbf{x}_{S,(i)}$	The i-th element of a vector $\mathbf{x}_S$.
$\mathbf{x}_{S,(i:10-l)}$	The subvector starting at i and ending at $10-l$ of a vector $\mathbf{x}_S$.
${}_I\mathbf{x}$	A vector expressed in the basis I.
$\mathbf{x}_{AB}$	A vector from a point A to point B.
$\tilde{\mathbf{x}}$	A skew-symmetric matrix so that the cross product holds, $\mathbf{x} \times \mathbf{y} = \tilde{\mathbf{x}}\mathbf{y}$.
$\mathbf{A}$	A matrix.
$\mathbf{A}_{ij}$	A submatrix ij of matrix $\mathbf{A}$.
$\mathbf{A}_{ij,(k,l)}$	The scalar element (k,l) of matrix $\mathbf{A}_{ij}$.
$\mathbf{A}_{S,(l:l+4,k:k+10)}$	The submatrix starting at element (l,k) and ending at $(l+4, k+10)$ of a matrix $\mathbf{A}_S$.
$\mathbf{A}_{CB}$	A transformation matrix with dimensions $\mathbb{R}^{3\times3}$ from coordinate frame B to C.
${}_D\mathbf{R}_{CB}$	A rotation matrix with dimensions $\mathbb{R}^{3\times3}$ which rotates a vector ${}_D\mathbf{x}$ in coordinate frame D to a new vector ${}_D\mathbf{y}$ in coordinate frame D by a rotation defined from frame B to frame C.
$(I, \mathbf{e}_x^I, \mathbf{e}_y^I, \mathbf{e}_z^I)$	Defines a coordinate system with origin at the point I with the orthogonal vectors $\mathbf{e}_x^I, \mathbf{e}_y^I, \mathbf{e}_z^I$.
$(I, \mathbf{e}_x^I, \mathbf{e}_y^I)$	Defines a plane with origin at the point I by the two spanning vectors $\mathbf{e}_x^I, \mathbf{e}_y^I$.
$\mathbf{diag}(\dots, x_i, \dots)$	Diagonal matrix with scalar diagonal elements in brackets.
$\mathbf{diag}(\dots, \mathbf{x}_i, \dots)$	Diagonal matrix with vector elements in brackets.
$\mathbf{diag}(\mathbf{A})$	Diagonal matrix with diagonal elements of matrix $\mathbf{A}$.
$\dim(\mathbf{A})$	Returns a vector with the dimensions in all directions of matrix $\mathbf{A}$.
${}_D\mathring{\mathbf{a}}_{CB}$ or ${}_DA_{CB}$	A quaternion which corresponds to a rotation matrix ${}_D\mathbf{R}_{CB}$.
${}_I\boldsymbol{\omega}_{CB}$	The angular velocity vector in $\mathbb{R}^3$ from frame B to frame C resolved in frame I.
`int foo(int a)`	All source code related notations are displayed in this font.
GPU	All terms and abbreviations for which a glossary entry exists in the appendix are displayed in this font.

1. Introduction

Nowadays, simulating few rigid bodies on a desktop computer with unilateral contacts, spatial Coulomb friction and impacts is well known and can be implemented quickly. Numerically simulating rigid bodies with contacts and impacts involves two separate steps during one iteration of a time-stepping solver, which discretizes the equations of motion. All geometric collisions between bodies need to be found and a feasible set of contact points needs to be extracted in a first step. The second step determines the contact forces of all contacts found in the first step. The performance of a sequential implemented contact solver for larger rigid body simulations quickly becomes tremendously slow. Solving the contact problem is a major bottleneck in multibody simulation and a lot of effort has been made to alleviate this problem by parallelizing the numerics as much as possible. Parallel execution of numerical algorithms can be done in a number of ways. Graphics processing units (GPU) have become famous for their high parallel arithmetic throughput. Applications which use the GPU often benefit from orders of magnitude of performance improvement. The goal of this thesis is to outsource numerical algorithms which are used to solve these large-scale contact problems to the GPU. Two popular numerical methods for solving contact problems are the Projected Successive Over-Relaxed Jacobi iteration (JOR Prox) and the Projected Successive Over-Relaxed Gauss-Seidel itertion (SOR Prox). These two algorithms have been parallelized and implemented on the GPU. The SOR Prox, in contrary to the JOR Prox method, is not embarrassingly parallel which makes an efficient parallelization difficult. Embarrassingly parallel problems do not need lots of communication between parallel executed tasks and can be implemented efficiently on a parallel architecture such as the GPU.

This thesis is structured as follows: Chapter 2 gives a short overview of the history and the state of the art of parallel computing. Chapter 3 gives an introduction to GPU programming with CUDA. It covers the CUDA work-flow and gives also an insight into the hardware architectures, limitations and optimization techniques used for parallel programming a GPU. Chapter 4 covers the mechanical aspects of describing rigid bodies with non-smooth force laws. This chapter shows how the contact problem is defined mathematically and how it can be solved with either the JOR Prox or the SOR Prox algorithm. Chapter 5 discusses the main contribution of this work in details, namely the GPU implementations for the JOR Prox and SOR Prox method. Chapter 5 presents the results obtained from various GPU implementations which have been tested. At last, chapter 7 gives a conclusion and a short outlook for future

work.

2. Parallel Computing on the GPU

2.1. The History in a Nutshell

Parallel computing is the discipline in Computer Science that deals with the system architecture and programming issues related to the concurrent execution of applications and algorithms. In recent years, the computing industry has made a widespread shift towards parallel computing. This trend mainly evolved due to physical constraints preventing frequency scaling. Because of the various limitations in the fabrication of integrated circuits and the physical limit to transistor size, computer manufacturers and researchers have started to extend the CPUs with concurrent execution capabilities.

In high-performance computing, supercomputers have extracted massive leaps in performance by steadily increasing the number of processors so far. Multi-processor architectures have been used for decades in supercomputers. In 2005, the multi-core CPU for consumer computers came to the market and a new area in the field of parallel computing has been initiated. Nowadays, every personal computer is equipped with multicore central processors. From the early dual-core machines to 8- and 16-core workstation computers, parallel computing has become widely available, not only to researchers but also to end-users.

Parallel computing has started in the late 1950's where IBM introduced the first commercial machine with floating-point hardware named 704. It was capable of approximately 5000 floating point operations per second (FLOPS). In the late 80's, clusters have been developed. A cluster is a type of parallel computer built up of lots of single CPUs which are connected over a network. Today, clusters are the workhorse of high-performance parallel computing and are the novel dominant architecture which powers our modern information age.

The rise of GPU computing harkens back to the 1980's when Silicon Graphics provided 2D display accelerators for personal computers. In the early stages, GPUs were only used in a few market fields including government applications and scientific visualizations. However, the foundation for today's graphics cards was laid. In 1992, Silicon Graphics opened the programming interface to its hardware by releasing the OpenGL library. OpenGL was intended to be used as a standardized, platform-independent method for writing 3D graphics applications. At the same time, companies such as

NVIDIA, ATI and 3dfx Interactive have started to provide the market with graphics accelerators. Up to now, the two biggest companies, NVIDIA and AMD (former ATI Technologies) , have been competing for years in the graphics field, always striving for better and more powerful graphics cards. Also the API's, OpenGL (current at version 4.1) and DirectX (current at version 11.0), have been rapidly extended to fulfill the needs of game applications and visualization.

Aside from the graphics card revolution, in 1997, the programming interface OpenMP has been released and is up to now the standard which is used in combination with the Message Passing Interface MPI in shared-memory architectures such as clusters and supercomputers. OpenMP uses compiler directives in the source code which is then compiled into a parallel executing program.

Up to 2001, the graphics card was a black box and did not provide an interface which let programmers execute their own graphics code on the GPU. Since then, in February 2001, NVIDIA released the GeForce 3 series which was compliant with the DirectX 8.0 standard and provided for the first time both programmable vertex and programmable pixel shading stages (Shader) in the Graphics Pipeline. The capability of programmable shading stages in the graphics pipeline attracted many researchers to the possibility of misusing the GPU for more than simply a rendering task. Because OpenGL and DirectX were still the main way to interact with the GPU, researchers took the effort of restructuring their scientific problems into traditional rendering tasks which was then conform with the GPU and could be executed on it. What they mainly did was to write certain vertex, fragment or pixel shaders which computed then numerical results, far aside of any graphics context. In this way, the parallel architecture of the GPU could be misused for scientific computations. The caveat of these procedure was that the programming model was too restrictive as one had to reshape the original problem into a rendering task and also had to deal with limited resources on the GPU, such as memory. Furthermore, the researchers were forced to cope with a special graphics-only programming language, called *shading language*. The acceptance of this method, to exploit concurrent execution on the GPU, was not huge, mainly because of these disadvantages.

In November 2006, NVIDIA unveiled the GeForce 8800 GTX, which was the first GPU built with NVIDIA's CUDA architecture. The CUDA architecture contained several new components which were strictly designed for general-purpose parallel computing on the GPU. Programming on the GPU is made possible with the programming language CUDA C, which is C with some extensions. Nowadays, CUDA has been widely used in medical imaging, fluid dynamics and environmental science to speed up the computations. However, NVIDIA's CUDA architecture is not the only way for parallel computing on the GPU. Since 2008, the Open Computing Language (OpenCL) is being developed. OpenCL is an open-source standard for cross-platform parallel programming of modern processors. While CUDA is restricted to NVIDIA graphics

cards, OpenCL is a much more general standard and can be used for heterogeneous computing. So far, using OpenCL, it is possible to execute programs on different processors such as CPU's, GPU's and other processors while programs written in CUDA C are restricted to NVIDIA GPU's only. NVIDIA supports now also the OpenCL standard for their graphics cards, but CUDA is still their main focus.

OpenMP, CUDA and OpenCL are only a few concepts of todays methods for parallel computing. The next section will briefly summarize the 4 different categories of parallel hardware architectures.

Literature about the history of parallel computing can be found in [14, 29, 32]. More information and a comparison between OpenGL and DirectX can be found in [27].

2.2. Parallel Hardware Architectures

In 1966, Michael J. Flynn has proposed a classification of computer architectures, which is known as *Flynn's Taxonomy*. In the following, these 4 categories are explained and visualized in figure 2.1:

- **Single Instruction, Single Data (SISD):** A sequential computer which exploits no parallelism in either the instruction or data streams. This is a traditional uniprocessor machine like an old PC, (current manufactured PC's have multiple processors).
- **Single Instruction, Multiple Data (SIMD):** A computer which executes the same instructions on multiple data. Examples are the SIMD instructions which Intel has introduced in 1997. Many instruction extensions and improvements followed, the so called Streaming SIMD Extensions (SSE, SSE2, SSE3, SSE4).
- **Multiple Instruction, Single Data (MISD):** This is a rather theoretical concept and has not been applicable for the mass manufacturing of hardware.
- **Multiple Instruction, Multiple Data (MIMD):** Architectures with multiple processing units each working on different data. It is distinguished between *distributed* memory architectures such as clusters and *shared-memory* architectures such as multicore CPU's, where each processor has access to the same physical memory location. The latter is also named as *Symmetric Multiprocessing* and is the focus of OpenMP.

Nothing has been said so far about parallel computing on the GPU. The question arises, which category the GPU's matches best. GPU's are basically a mixture of a SIMD and a MIMD architecture. They are classified by NVIDIA into the category *Single Instruction, Multiple Threads* (SIMT). The difference lies in the execution

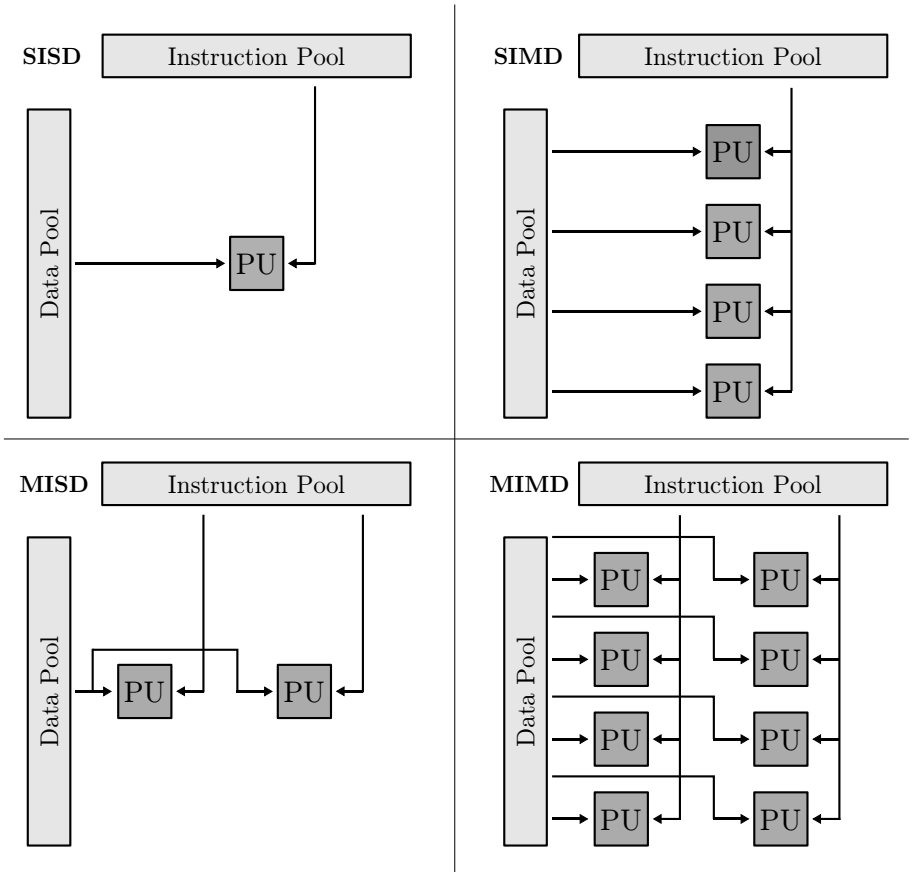


Figure 2.1.: The four hardware architecture SISD,SIMD,MISD and MIMD. PU denotes "'Processing Unit"'. Adpated from `http://en.wikipedia.org/wiki/Flynn's_taxonomy`

model, in a SIMD structure each core on the GPU should actually execute exactly the same instruction. Nevertheless, this is not exactly what happens on the GPU, because different SIMD cores on the GPU can execute different instructions. This behavior resembles the MIMD work-flow. However, there are some restrictions. Not all threads on the GPU can exactly execute a different instruction. Threads in group of 32, called Warp under NVIDIA's GPU architecture CUDA, execute one instruction at the same time. This is in contradiction to a pure MIMD architecture. More about this is said in the following chapters.

In the following, todays parallel programming languages and methods are briefly listed to give a brief overview about todays possibilities in parallel programming. This list is not complete and should only give a short overview of the available possibilities.

- **Threads:** These concepts are implemented in every consumer PC and allows to start multiple threads (processes) which all have the same shared memory.

It is used in multi-tasking applications, and multicore applications. Every operating system has nowadays multi-threading support. The standard depends on the operating system. Under UNIX like systems such as OS X and other Linux distributions, it is subsumed under the term POSIX Threads. Windows has its own implementation.

- **OpenMP:** This standard is used in high-performance computing. OpenMP exploits parallelism by creating several threads (see above) which run in parallel on several processors or cores. They access the same shared memory.
- **MPI:** This standard is used for distributed systems. It includes grid computing, clusters and massive parallel processing (MPP).
- **CUDA:** The architecture and programming model for general-purpose computing on a NVIDIA GPU.
- **OpenCL:** Used for heterogeneous computing, which allows to exploit parallelism among different processors such as CPU's, GPU's and others.

It should be noted as well, that in general, an application can combine different parallel layers (SIMD,MIMD, etc.) together. Furthermore, all methods presented above can generally work together in one program if this is supported by the computer architecture such as a cluster or personal computer. A program P can be executed in parallel with OpenMP on several processors of a shared-memory computer S. Then again, each processor of S can employ SIMD concepts such as SSE for example. It is also conceivable that on an outer organization layer, the program P can be a part of a distributed system, meaning that several more instances of the program P can be distributed and executed on a cluster. To achieve this, the Message Passing Interface (MPI) comes into action.

The next chapter gives a brief overview of the GPU programming architecture CUDA and all related literature, which is strongly recommended for the reader to successfully follow this work.

3. GPU Programming with CUDA

This chapter should give an insight into the programming model CUDA C and the architecture CUDA itself. The field in parallel computing on the GPU has grown so fast that this chapter is way beyond completeness. However, it should serve as an introduction into the CUDA architecture. Because covering most aspect of GPU programming is beyond the scope of this work, the reader is referred to additional, more complete references throughout this chapter.

3.1. OpenCL or CUDA?

The question probably arises, why CUDA has been chosen in this work for GPU computing. The answer is very simple given by the fact that CUDA and CUDA C is still the better supported programming language for GPU computing. Table 3.1 gives some pros and contras about these two standards. A more detailed comparison can be found in [12]. The fact that learning CUDA is much easier for beginners, has led to the conclusion to rather try CUDA for the first experience with GPU programming. The speed of OpenCL and CUDA is generally comparable. However, OpenCL is probably not yet tuned for speed, if no vendor specific features have been taken into account.

3.2. Prerequired Literature

In general, beginners in GPU computing experience, as the author of this thesis did as well, that the terminology and the huge amount of abbreviations and concepts can be seriously confusing. NVIDIA has in general a slight tendency to introduce new names and abbreviations for the reason of their marketing strategies. As a good starting point, it is warmly recommended to the reader to have a look at the literature presented in the following. With the presented literature, one should quickly get familiarized to CUDA and should be able to follow the work more accurately.

- **CUDA C Getting Started Guide:** This manual [23] should be downloaded for the specific platform. It helps to set up and install the CUDA

	CUDA	OpenCL
Pros:	CUDA has more mature tools such as a debugger, profiler, CUBLAS and CUFFT	Many vendors such as Apple, AMD, Intel, IBM and also NVIDIA support it.
	CUDA allows C++ constructs (templates etc.) in GPU code.	OpenCL code works on CPU's and GPUs, though save the need for a multicore CPU implementation.
	Has a steep learning curve.	The code is portable among different GPU devices.
Cons:	Only NVIDIA supports CUDA.	OpenCL is harder to learn for beginners.
	Only NVIDIA GPUs can be used	OpenCL code is based on C99.
	Works only on the GPU. A multicore CPU implementation needs to be written seperatly.	The code is probably not yet tuned for speed. Vendor specific implementations have to be made anyway.
		Harder to debug.

Table 3.1.: Pros and Contras of Cuda and OpenCL.

programming environment and drivers.

- **CUDA C Best Practices Guide:** It is in general a very good idea to read through the whole manual in [22]. To start with, the following chapters are suggested: Chapter 1 (Parallel Computing with CUDA), Chapter 2 (Performance Metrics) Chapter 3 (Memory Optimizations).
- **CUDA C Programming Guide:** The whole manual in [24] is not recommended, nevertheless is should be used as a reference while writing CUDA applications. The following chapters are important for the beginning: Chapter 2 (Programming Model), Chapter 3 (Programming Interface) especially 3.1 (Compilation with NVCC) and 3.2 (CUDA C Runtime).
- **CUDA By Example:** The book is a straight forward introduction into programming with CUDA C. It is easy to read but covers quite a lot of in-depth material. Despite that it gives no insight about the architecture and about how things work on the GPU, it gives good hints and code snippets how to write simple to intermediate parallel applications. Worth reading at the beginning are the chapters: 3 (Introduction to CUDA C), 4 (Parallel Programming in CUDA C), 5 (Thread Cooperation), 9 (Atomics).
- **Programming Massively Parallel Processors: A Hands-on Approach:** This book [11] is as well a very good introduction and gives profound information about a lot of CUDA related topics. Recommended chapters are: 3.2 (CUDA Program Structure), 3.4 (Device Memories and Data Transfer), 4

(Threads), 5 (CUDA Memories), 6 (Performance Considerations). Reading the whole book does not replace the need of the CUDA C Best Practices Guide. However, it is probably the best book so far, with a very high level of accuracy!

3.3. The CUDA Work-flow

CUDA is the hardware and software architecture which enables NVIDIA GPUs to execute programs written in CUDA C, Fortran, OpenCL, DirectCompute, and other languages. CUDA C is a minimal extension of the C and C++ programming languages. In the simplest case, and also done in this work, the programmer writes a program (in C or C++) that calls parallel GPU kernels (Kernel). This CUDA program is executed on the CPU, called Host. GPU kernels are launched from the host. The kernel code executes in parallel across a set of parallel threads on the GPU. The programmer organizes these threads into *thread blocks* which build a *kernel grid*. A thread block contains itself a set of concurrent threads that can cooperate among themselves through barrier synchronizations and shared access to a memory space private to the kernel block. Each thread in a kernel block executes its own instance of the kernel code and has a unique thread ID threadIdx. Each block has also a unique block ID blockIdx within its kernel grid. A kernel grid is an array (1D, 2D or even 3D) of thread blocks that execute instances of the same kernel, reads input from global memory, writes results to global memory, and can synchronize between dependent kernel calls. Figure 3.1 visualizes the CUDA programming model. Each thread has a per-thread private memory space used for register spills, function calls, and C automatic array variables. Each thread block has a per-block shared memory space used for inter-thread communication between thread in the thread block. Threads in a block can share data over shared memory.

This section and the following have been summarized and adapted mainly from [18, 26].

3.3.1. CUDA's Memory Model

Table 3.2 shows a list of all memory types which can be used in the CUDA architecture. The most common used memories are *global* and *shared* memory. Figure 3.2 visualizes the memory hierarchy. It shows a kernel grid spanning 2 thread blocks containing each 2 threads. Registers and *local* memory are private to each thread. *Shared* memory is visible to all threads in a block. *Global*, *constant* and *texture* memory are visible to all threads in the grid.

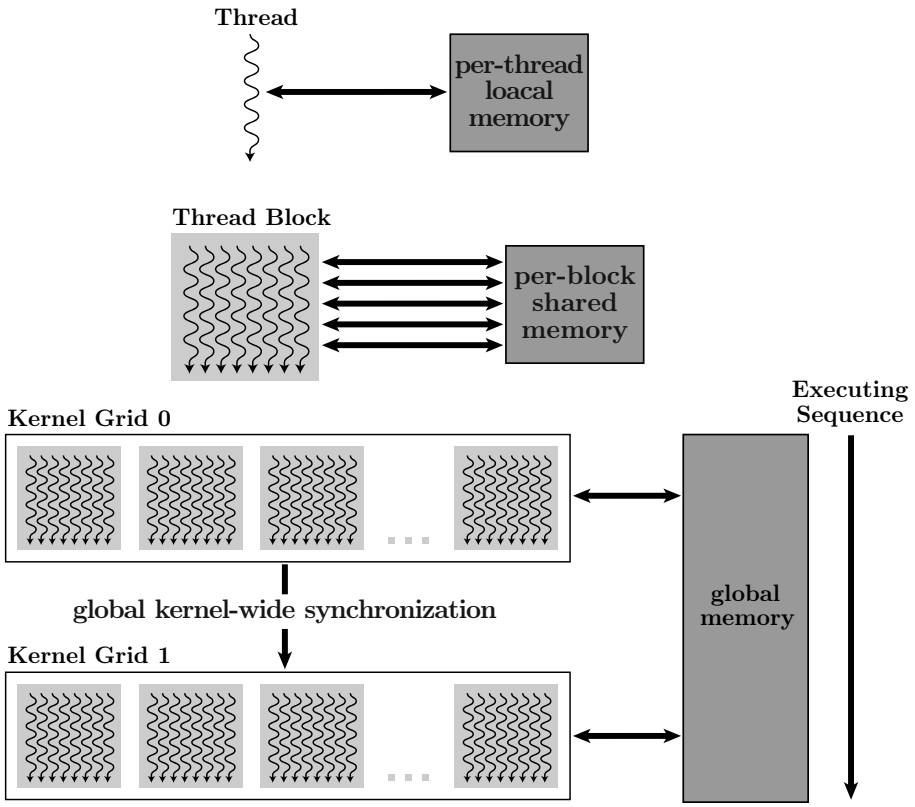


Figure 3.1.: The CUDA programming model. The hierarchy of threads, blocks, and grids, with corresponding per-thread private, per-block shared, and per-application global memory spaces.

Memory Type	Location	Scope of Access	Access Type	Cached
Registers	on-chip	1 thread	read-write	no
Local	off-chip	1 thread	read-write	(only for CC 2.x)
Shared	on-chip	all threads/block	read-write	no
Global	off-chip	all threads + host	read-write	(only for CC 2.x)
Constant	off-chip	all threads + host	read-only	yes
Texture	off-chip	all threads + host	read-only	yes (spatially)

Table 3.2.: The CUDA memory model.

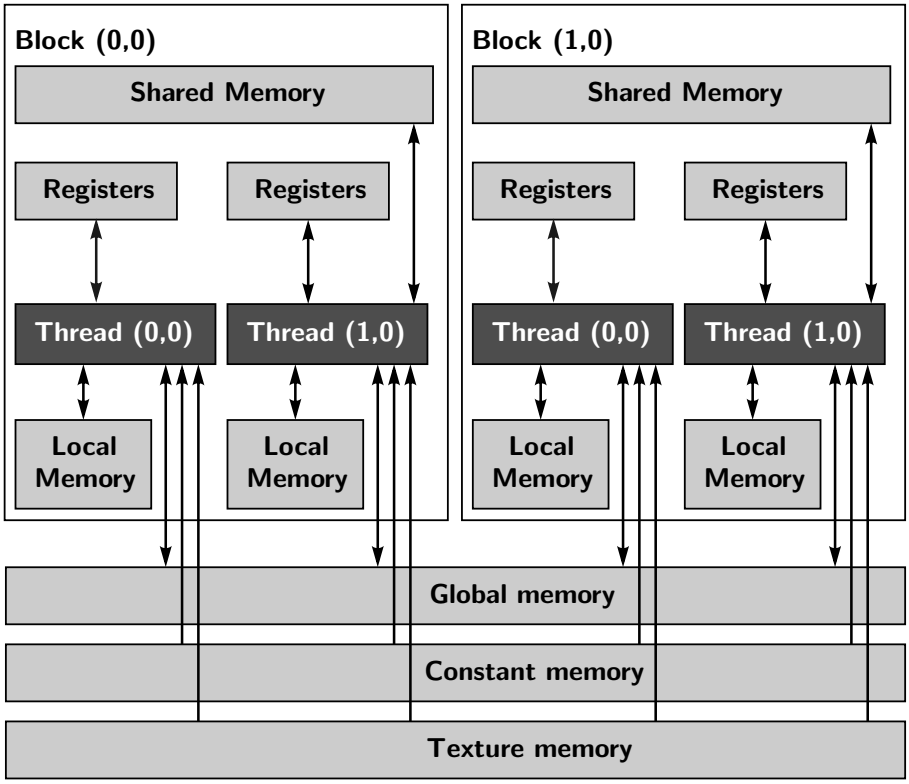


Figure 3.2.: The CUDA memory hierarchy with two thread blocks with indices (0,0) and (1,0). Each block contains two threads with indices (0,0) and (1,0).

3.3.2. Kernel Execution in CUDA C

To launch a kernel, the programmer first needs to think about the kernel grid size which is three-dimensional in general. The kernel grid is specified by the variable `dimGrid` and `dimBlock` which are 3 element vectors of type `dim3`. The vector `dimGrid` specifies how many blocks are inside the grid in each dimension x, y, z, e.g., a grid size of $(10, 3, 4)$ contains 10 blocks in x, 3 blocks in y and 4 blocks in z direction. This sums up to a total of $10 \cdot 3 \cdot 4 = 120$ blocks. The vector `dimBlock` specifies again how many threads are inside each block in each dimension x, y, z, e.g. a block size of $(2, 2, 3)$ has a total of $2 \cdot 2 \cdot 3$ threads in each block. The `__global__` declaration in front of a function declares that the procedure is a kernel entry point. A kernel named `calcKernel` can be launched (Kernel Execution) by the extended function-call syntax:

```
1    computeKernel<<<dimGrid, dimBlock>>>( parameter list );
```

The source code of a CUDA C kernel is a C function for one sequential thread. Thus, it is generally straight forward to write and the programmer mostly has to think about the tasks of one thread only. The parallelism is determined clearly by the

Listing 3.1: Kernel for a vector addition $\mathbf{c} = \mathbf{a} + \mathbf{b}$ in global memory.

```
1 __global__ void vectorAdd_kernel( double *C, double * A, ...
    double * B,  int N){
2
3   // GLOBAL MEMORY, VECTOR ADDITION
4
5   // Calculate indices
6   unsigned int index_x = threadIdx.x + blockIdx.x * ...
      blockDim.x;
7   unsigned int stride_x = gridDim.x * blockDim.x;
8
9   while(index_x < N){
10    // Do vector addition, each thread does one addition
11    C[index_x] = A[index_x] + B[index_x];
12    // ================================================
13    index_x += stride_x;
14  }
15 }
```

dimension of the kernel grid which is launched. The thread execution and scheduling on the GPU is handled "auto-magically" by the underlying hardware. An example of a simple kernel which computes a vector addition of $\mathbf{c} = \mathbf{a} + \mathbf{b}$ in parallel is given in listing 3.1. Vector $\mathbf{a}, \mathbf{b}, \mathbf{c}$ are of length N. The source code in listing 3.1 is executed for each thread in the kernel grid. A thread i should at the end calculate one addition $\mathbf{c}_{(i)} = \mathbf{a}_{(i)} + \mathbf{b}_{(i)}$. Therefore, in line 6 the thread specific index `index_x` is calculated. Line 11 calculates then the addition at the corresponding address for this specific thread. If the kernel is launched with a grid consisting of $k = N$ threads, each thread i linearly accesses the element $\mathbf{a}_{(i)}, \mathbf{b}_{(i)}$ and computes the sum $\mathbf{c}_{(i)} = \mathbf{a}_{(i)} + \mathbf{b}_{(i)}$. In only this way the while loop has no influence in line 9. If not enough threads are launched in the grid ($k < N$) then the while loop provides a commonly used stride pattern where each thread jumps k elements in the array and computes a new addition until the index falls out of the range which is N.

Threads in a block may also synchronize between each other. This can be achieved by the command `__syncthreads()`. This guarantees that all threads participating in the barrier can proceed until all participating threads reached the barrier. Threads within a thread block may access allocated per-block private shared memory. Shared memory can be declared inside a kernel with the key word `__shared__`. Thread blocks are scheduled independently and have no means of direct communication between them, although they can synchronize with special atomic functions on the global memory provided by the CUDA architecture. Global memory is visible to all threads and can therefore be used to share data between different blocks. In most cases *global* memory

is allocated and filled, by commands such as `cudaMalloc`, `cudaMemcpy` and `cudaFree`, from the host CPU before kernel execution. In simple applications, kernels and the specified kernel grids are launched sequentially from the host. More advanced launch configurations can be achieved by *streams* which enables overlapping of memory copy operations from the host with kernel executions and possibly also enables overlapping of different kernel launches. Sequential kernel launches share results in global memory after global kernel-wide synchronization (see figure 3.1).

3.4. CUDA Hardware Layout

NVIDIA released in November 2006 the GeForce 8800 which was built on a new GPU computing model, the G80 architecture. The information presented in the following has been adapted and summarized from [20, 21, 26]. The G80 architecture brought several key innovations to the parallel computing field listed in the following:

- Support for the programming language C, allowing programmers to write general-purpose computing programs.
- The separate vertex and pixel pipelines, back from the graphics era, has been replaced by a unified processor which additionally can execute computing programs.
- The SIMT execution model has been implemented, where multiple threads execute concurrently one single instruction.
- Shared memory and barrier synchronization has been introduced.

The G80 architecture is visualized in figure 3.3. The G80 consists of a Streaming Processor Array (SPA) with 8 Texture Processor Clusters (TPC). Each TPC, visualized in figure 3.4, consists of 2 Streaming Multiprocessors (SM). In total, the G80 consists of 16 SMs. Each SM consists of 8 Stream Processors (SP) and 2 Super Function Units (SFU). The name CUDA Core and Shader are equivalent to SP. The G80 architecture has 128 SPs in total. More in-depth information about the G80 architecture can be found in [21]. More about SM and SP is said in section 3.4.1.

In June 2008, NVIDIA revealed their revision to the G80 architecture, named GT200. Several improvements have been made from coalesced memory access to double precision floating point arithmetic The GT200 contains 10 TPCs with each 3 SMs which is visualized in 3.4. The SM contains 8 SPs like the G80. Then in 2010, NVIDIA released another improved architecture named GF100 with the codename Fermi. Some of the improvements were: Better double precision performance, more shared memory, faster context switching, faster atomic operations etc. In the next section the Fermi architecture is explained in detail and should help to understand the following hardware execution work-flow.

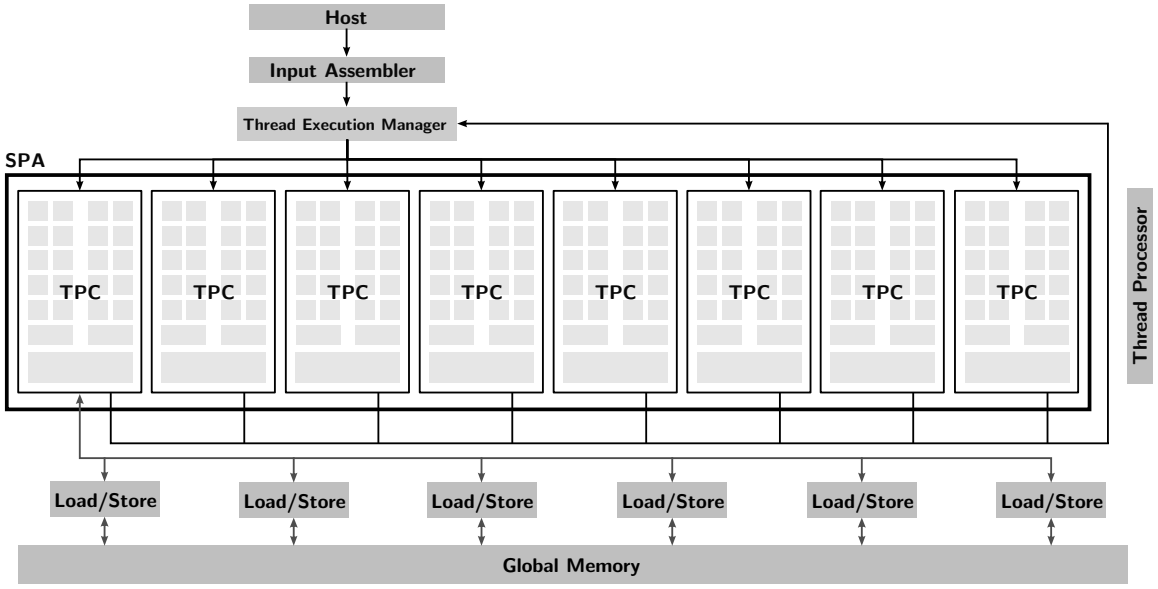


Figure 3.3.: NVIDIA's G80 (CC 1.0) architecture. The Streaming Processor Array (SPA) consists of 8 Texture Processor Clusters.

3.4.1. The Fermi Architecture

The Fermi architecture for the NVIDIA GeForce series are named with the prefix "GF" followed by an increasing number starting from 100. The GeForce GTX 580 was the fastest commercial graphics card available from NVIDIA in August 2011. It is based on the Fermi architecture called GF110. The NVIDIA Tesla Series, starting from the Tesla C870 with a G80 architecture to the newest Tesla S2070 with a 4 × Fermi architecture, comprise NVIDIA's sector in high-end computing GPUs. In the following, it is not distinguished between GF100 and the Fermi architecture because they are very similar to each other. From now on, the name Fermi is used throughout the text. Figure 3.5 shows the Fermi architecture layout (GF100). The Fermi architecture consists of 16 SMs each comprising of 32 SPs. This sums up to 512 CUDA Cores or SPs.

The Fermi Streaming Multiprocessor SM is shown in figure 3.6. It features 16 Load-/Store Units and 4 Super Function Units (SFU). A CUDA Core features one fully pipelined integer arithmetic logical unit (ALU) and one Floating Point Unit (FPU) with single and double precision (IEEE 754-2008 floating-point standard, [6]). The Load/Store units, which loads and stores data to cache or DRAM, can serve 16 threads per clock cycle. The SFU is responsible for the computation of transcendental functions such as sin, cosine, reciprocal, and square root. More technical specifications can be found in [26].

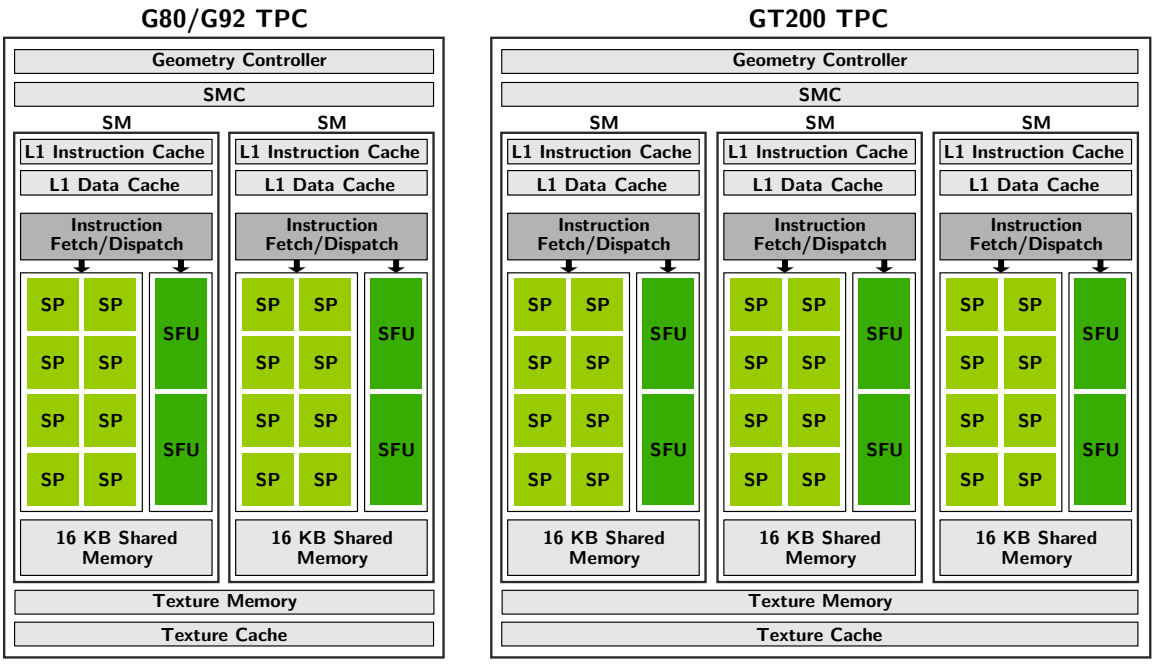


Figure 3.4.: NVIDIA's G80 (CC 1.0) and GT200 (CC 1.3) Texture Processing Cluster (TPC). The G80 contains 2 and the GT200 3 SMs. The G80 contains 8 such TPCs with in total 16 Streaming Multiprocessors (SM) each with 8 Stream Processors (SP, CUDA Core). The GT200 contains 10 such TPCs with in total 30 SMs each with 8 SPs.

3.4.2. Architecture Comparision

Table 3.3 compares the G80, GT200 and the Fermi architecture in different aspects.

In section 3.3, the work-flow of CUDA has been explained on a high-level basis. The next section deals with the execution on hardware level and explains what happens when a CUDA kernel is launched and how hardware related divisions such as SM and SP are associated to the kernel grid, thread blocks and threads.

3.5. Hardware Execution and Restrictions

Roughly speaking, a GPU executes one or more kernel grids, a Streaming Multiprocessor (SM) executes one or more thread blocks and the Streaming Processor (SP) or CUDA Core and other execution units in the SM execute threads. When a CUDA program on the host CPU invokes a kernel grid, the global compute work distribution (CWD) unit on the GPU enumerates the blocks of the grid and begins distributing

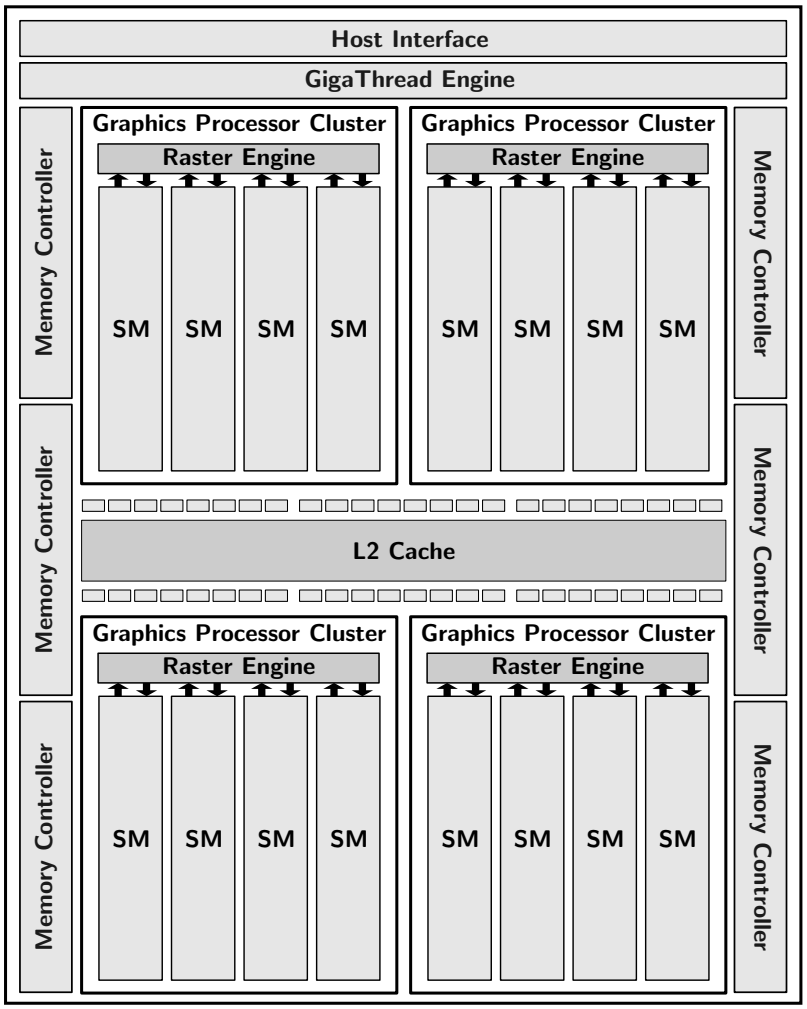


Figure 3.5.: NVIDIA's first Fermi architecture (GF100, CC 2.0). The 4 Graphics Processor Clusters (GPA) consist of 4 Streaming Multiprocessors (SM) summing up to a total of 16 SMs each with 32 Stream Processors (SP, CUDA Core). The Fermi architecture has a total of 512 CUDA Cores.

them to SMs with available execution capacity. As a thread block finishes, the CWD unit launches new blocks. One SM executes threads in groups of 32 threads called Warp.

The Fermi SM features two warp schedulers and two instruction dispatch units which allow to execute two warps concurrently (see figure 3.6). A very good explanation of the scheduling procedure has been adapted from a lecture in [28] and proceeds as follows. All threads in a kernel grid are enumerated linearly by their global thread index *thid* which calculates in column-major order as

$$thid = x + y \cdot B_x + z \cdot B_x \cdot B_y,$$

where the grid size is (B_x, B_y, B_z) and the thread in each block has the index (x, y, z). The same in CUDA C writes as

```
1   unsigned int thid = threadIdx.x + threadIdx.y * ...
      blockDim.x + threadIdx.z * blockDim.x * blockDim.y ,
```

where `threadIdx` and `blockDim` are 3 element vectors of type `dim3`. All threads are split into warps by their warp index *warpid* which is calculated linearly from the global thread ID by

$$warpid = \lfloor thid/warpSize \rfloor ,$$

where *warpSize* is 32 for the Fermi architecture.

Lets assume the SM in this example currently manages N warps, then the following sequence happens for each of the two warp schedulers and dispatch units shown in figure 3.6. Only the first warp scheduling is explained and visualized in figure 3.7. All comments in brackets refer to the example in the figure:

1. The warp scheduler chooses a warp, in this example the warp with $warpid = 2$ (❶).

2. Within the warp, one thread is chosen (bold black wave, ❷). The next instruction (`c = a + b`, ❸) determines the collective next instruction for all threads in the warp. In this case, some threads do not have the same next instruction (grey waves, ❹). Therefore, for each thread it is checked if the chosen instruction (`c = a + b`) matches or not. Only the threads for which this is true will participate in the execution of this instruction (24 threads, ❸), the other threads will not participate and will pause (8 threads, ❹).

3. The participating threads are now distributed in two clock cycles to the 16 SPs in the SM where the chosen instruction (`c = a + b`) is computed. In the example in the figure that means: In the first clock cycle, the first 16 threads in the warp are active and compute the instruction (❺). In the second clock cycle, the remaining 8 threads are distributed to the SPs (❻). In the second clock cycle, 8 SPs are vacant.

In this example, 8 threads are not participating in the computation due to the flow control instruction `if` in figure 3.7 which causes branching. This behaviour is called *thread divergence*. Thread divergence should be avoided by all means when programming GPU kernels. Any flow control instruction such as `if`, `switch`, `do`, `for`, `while` can significantly reduce the instruction throughput when threads of the same warp disagree on the execution path, that is if they branch into different execution paths. As shown in the example above, these different execution paths get serialized (8 vacant SP's in the second clock cycle). All 8 threads which execute the `else` case, `c =`

`a + d`, are scheduled in another 2 clock cylces after the first 2 shown in figure 3.7. Only if all different execution paths have been completed, the threads will converge back to the same execution path.

There are certain upper bounds on how many threads (warps) and how many thread blocks can be managed per SM. Additionally there are resource limits for registers and shared memory per SM. These restrictions are subsumed under the term Compute Capability. The limits specified by the Compute Capability is of great importance because it will restrict the SM in certain ways of how many blocks and threads it can schedule. The next section addresses this issue more detailed. Table 3.4 compares the different compute capabilities in different aspects.

GPU	**G80** (CC 1.0)	**GT200** (CC 1.3)	**Fermi** (CC 2.0)
Transistors	681 million	1.4 billion	3.0 billion
Total SPs or CUDA Cores	**128**	**240**	**512**
Total SMs	16 (8 TPCs)	30 (10 TPCs)	16
SPs or CUDA Cores per SM	8	8	32
Double Precision Floating Point Capability	None	30 FMA ops/-clock	256 FMA ops/-clock
Single Precision Floating Point Capability	128 MAD ops/-clock	240 MAD ops/-clock	512 FMA ops/-clock
Special Function Units / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	48 KB or 16 KB
L1 Cache (per SM)	None	None	16 KB or 48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels Load/Store Address Width	No 32-bit	No 32-bit	Up to 16 64-bit

Table 3.3.: Comparision of G80,GT200 and Fermi architecture. FMA: Fused multiply and add, MAD: Multiply and add.

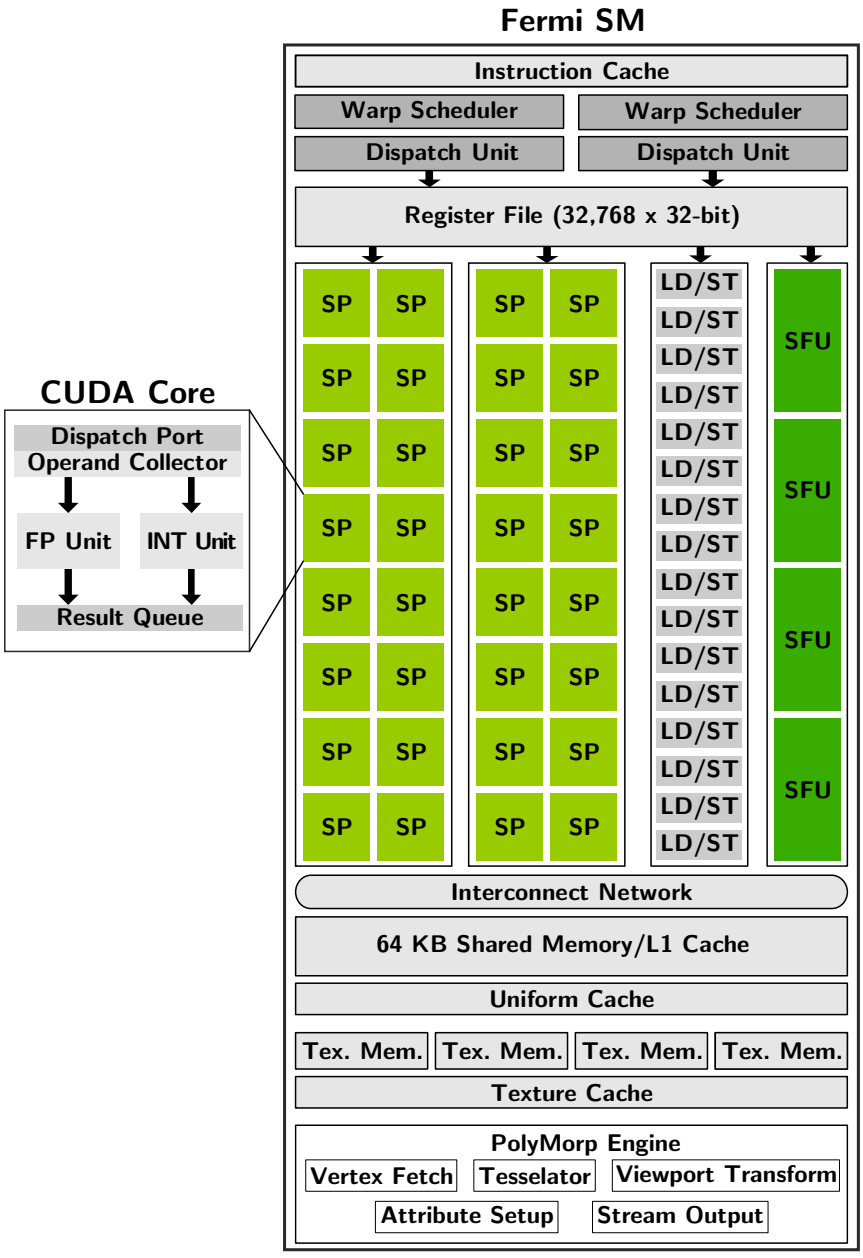


Figure 3.6.: The Fermi (CC 2.0) Streaming Multiprocessor (SM).

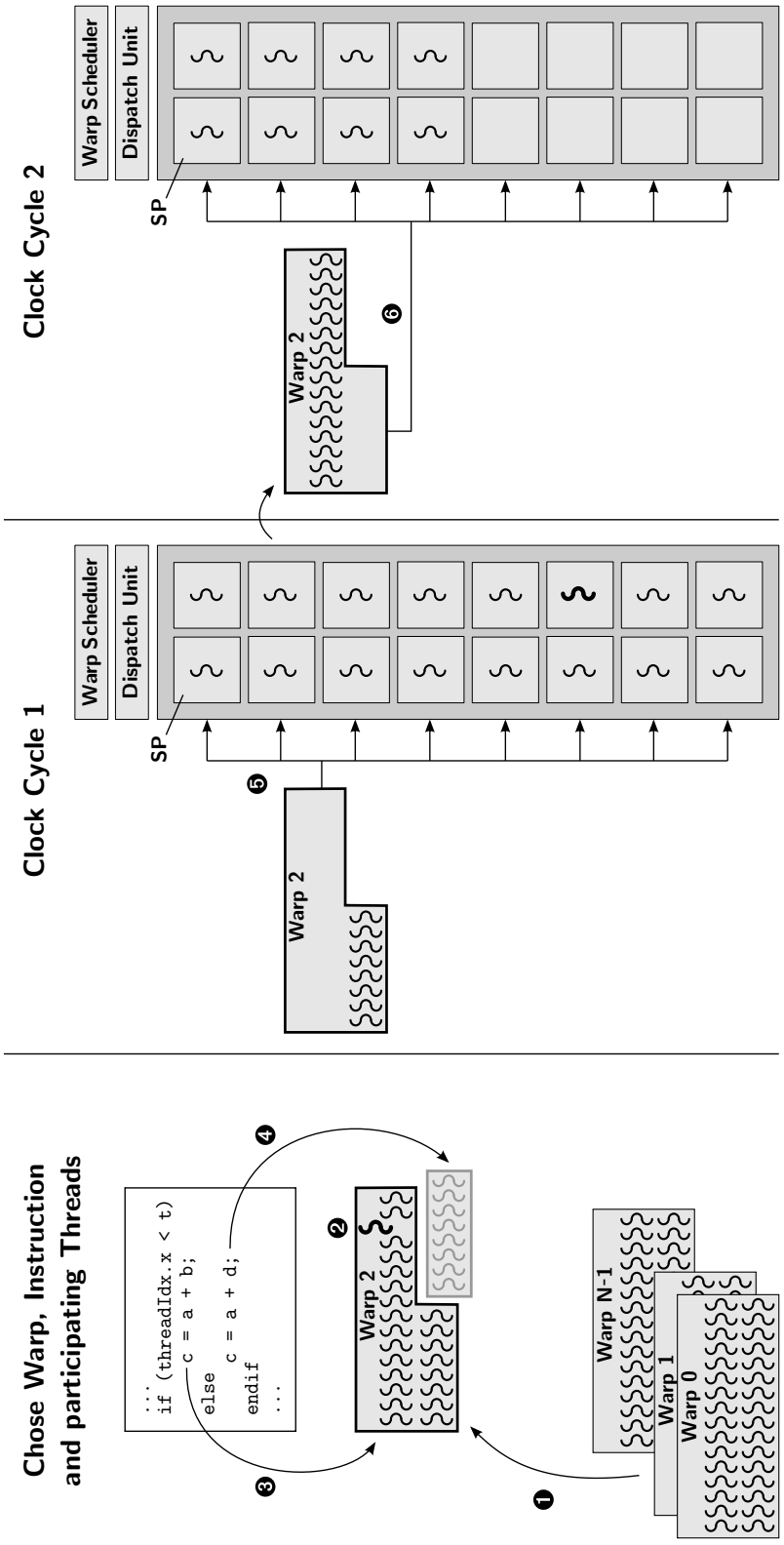


Figure 3.7.: The scheduling procedure on the Fermi architecture. Only one warp scheduler with 16 SPs are shown (half of the SM). The same thing happens for the other warp scheduler.

Compute Capability	**1.0**	**1.1**	**1.2**	**1.3**	**2.0**	**2.1**
SM Version	sm_10	sm_11	sm_12	sm_13	sm_20	sm_21
Threads/Warp	32	32	32	32	32	32
Warps/Multiprocessor	24	24	32	32	48	48
Threads/Multiprocessor	768	768	1024	1024	1536	1536
Thread Blocks/Multiprocessor	8	8	8	8	8	8
Max Shared Memory/Multiprocessor (bytes)	16384	16384	16384	16384	49152	49152
Register File Size	8192	8192	16384	16384	32768	32768
Register Allocation Unit Size	256	256	512	512	64	64
Allocation Granularity	block	block	block	block	warp	warp
Shared Memory Allocation Unit Size	512	512	512	512	128	128
Warp allocation granularity (for registers)	2	2	2	2		
Max Thread Block Size	512	512	512	512	1024	1024

Table 3.4.: Comparision table of the Compute Capability from version 1.0 to version 2.1.

3.6. Performance Metrics

This section briefly discusses some relevant aspects which are used when measuring the performance of certain parallel algorithms. This chapter focuses on the GPU architecture but a lot of aspects presented here are very similar to other parallel computing architectures such as multi-core CPUs.

3.6.1. Speedup Factor

The speedup factor S_n is a relative performance metric of a certain numerical algorithm A and is defined as

$$S_n = \frac{T_s}{T_n}, \tag{3.1}$$

where T_s is the running time for a sequential computation of A, meaning that it runs on 1 processor/computation unit/core. T_n is the running time of A if the algorithm has been executed on n processors. Hence, S_n tells how many times faster the algorithm is running on n processors compared to running sequentially. When comparing different parallel implementations $A_1, A_2, \ldots$ of algorithm A, the speedup factor is very much dependent on their sequential implementation of A and thus should be considered with caution. There are two concepts for speedup calculations, Amdahl's law and Gustafson's law explained in the following.

3.6.2. Efficiency

The efficiency of an algorithm A is defined as,

$$E_n = \frac{S_n}{n} = \frac{T_s}{nT_n}, \quad S_n \leq n \Rightarrow E_n \leq 1, \tag{3.2}$$

where n is the number of processors and S_n the speedup for n processors as defined above.

Amdahl's Law

Amdahl's law, named after Gene Amdahl, is a simple formula to find the maximum expected speedup of an algorithm. If an algorithm A has a portion p which can be parallelized and a portion $s = 1 - p$ which represents the sequential not parallelized part of A ($T_s = (s + p)T_s$), then the maximum speedup when using n processors for

the parallelized part is given as:

$$S_n = \frac{T_s}{T_n} = \frac{T_s}{(1-p)T_s + \frac{pT_s}{n}} = \frac{1}{(1-p) + \frac{p}{n}}. \tag{3.3}$$

The speedup has a limit if $N \to \infty$:

$$\lim_{n\to\infty} S_n = S_\infty = \frac{1}{(1-p)} = \frac{1}{s}. \tag{3.4}$$

For a 5 % percent parallelized part the speedup factor cannot exceed 20. The fraction of the sequential part directly determines the potential speedup. Amdahl's law assumes *strong scaling* which is defined as how the running time T_n varies with the number of processors n for a fixed *total* problem size. Therefore, the amount of work per processor decreases with increasing n.

Gustafson's Law

In contrary to Amdahl's law, Gustafson's law in [7] assumes *weak scaling* which is defined as how the running time varies for a fixed problem size *per processor*. Gustafson's law is a critique of Amdahl's law. Gustafson stated that it does not make sense to increase the number of processors while keeping the problem size fixed. The problem size should rather be fixed *per processor*. That is, if $T_n = (p+s)T_n$ is the time which is needed with n processors and as required the parallel execution time $T_p = pT_n$ is fixed, then the parallel part p will take n-times as long on 1 processor. This gives the following speedup:

$$S_n = \frac{T_s}{T_n} = \frac{(1-p)T_n + npT_n}{T_n} = (1-p) + np. \tag{3.5}$$

The relation to Amdahl's law can be seen if p in (3.3) is replaced by

$$p \to \frac{npT_n}{(1-p)T_n + npT_n} = \frac{np}{(1-p) + np}, \tag{3.6}$$

then this yields Gustafson's law in (3.5). The first part in (3.6) is exactly the fractional parallelized part p if A is run sequentially.

The efficiency can be evaluated to

$$E_n = \frac{1-p}{n} + p \tag{3.7}$$

which gives the fraction of time for which a processor is doing useful work.

GPU Type	Bandwidth in GB/sec
NVIDIA GeForce 256 (1999)	4.8
NVIDIA GeForce 8800 (2006) GTX	86.4
NVIDIA GeForce 580 GTX	192.4
NVIDIA Tesla C2070	144
AMD Rage 128 Pro (1999)	2.28
AMD FireStream 9370	147

Table 3.5.: Comparision of theoretical global memory bandwidth between several GPUs.

3.6.3. Memory Bandwidth

The memory bandwidth, also referred to as global memory bandwidth or memory throughput, relates to the rate at which memory can be transferred (read/write) and is mostly given in GB/sec. Memory bandwidth is the most important gating factor for performance. The memory bandwidth depends on the memory and architecture used, e.g. shared memory is much faster than global memory. For every computation on the GPU or CPU, data needs to be loaded into registers from memory for further instructions. Thus, the memory bandwidth can be the bottleneck to all further computations. Assuming that the CPU floating point unit is tremendously fast and the memory throughput is very slow, the memory bandwidth will determine the overall speed of the computation. More information can be found in section 2.2 “Bandwidth” in [22].

Theoretical Memory Bandwidth

Theoretical bandwidth MB_T is simply the peak performance of the *global* memory throughput of the device. It can be calculated from the memory bus clock speed and its interface width or can be simply looked up from the specification data sheet of the device. Table 3.5 gives a short comparison of the memory bandwidth of several GPU's.

Effective Memory Bandwidth

Effective bandwidth MB_E for kernel A is computed as follows:

$$MB_E = \frac{B_R + B_W}{10^9 \cdot T} \quad \left[\frac{\text{GB}}{\text{sec}}\right], \tag{3.8}$$

where B_R is the number of bytes read and B_W the number of bytes written by A. The execution time of A is T. The effective bandwidth can be computed by hand by counting the number of read/write operations for a kernel or by using a profiling tool such as cudaprof from NVIDIA. The memory bandwidth reported by the profiler $MB_{E,p}$ will in general be higher then the one calculated by hand $MB_{E,m}$. This is due to the fact that more memory is transferred. The minimum transaction size of memory results in unused transferred memory by the kernel. The calculation by hand however only includes the memory actually relevant to the algorithm. Thus, the bandwidth calculated by hand will be lower than the reported one by the profiler. Both numbers are very useful for optimizations in 2 aspects. First, the reported bandwidth by the profiler shows how close the effective bandwidth is to the theoretical bandwidth, the hardware limit. Secondly, comparing the profiler output to the manually calculated bandwidth shows how ineffective the memory transfer is. As an example, assume that the profiler reports $MB_{E,p} = 160$ GB/sec for the effective bandwidth and the hardware limit is supposed to be at $MB_T = 192$ GB/sec. However, the manually calculated bandwidth yields $MB_{E,m} = 150$ GB/sec. This shows that the memory transfers by A are almost giving ≈ 83 % of the peak performance. This is already very good. Comparing $MB_{E,m}$ to $MB_{E,p}$ shows additionally that not very much additional memory transfers have been made. Overall this kernel has good memory performance. But assuming $MB_{E,m} = 90$ GB/sec for the manually calculated bandwidth shows that kernel A makes actually much more memory transfers then necessary by the algorithm. This is a very good indication that the memory performance should be optimized if possible, for example by exploiting *coalesced* memory transfers. More about this method is said in section 3.7.3.

3.6.4. Compute Bandwidth

The compute bandwidth is mostly measured in floating point operations per seconds (FLOPS). Different instructions (addition, multiply, division) need different number of clock cycles on a processor. Thus, FLOPS is not direct proportional to the processor's clock speed. In order to use FLOPS as a performance measure of floating point performance, there must be a useful standard benchmark for the architecture. Commonly used on CPU's is the LINPACK benchmark. On the GPU, the compute bandwidth can be measured roughly by writing a simple kernel which only does addition or another instruction. This technique is know as micro-benchmarking. For the GT200 architecture there have been made some profound studies in [33].

For an algorithm A, the FLOPS can be roughly approximated by counting the number of floating point operations (flops) and dividing by the running time on the GPU. For calculating the number of flops, the reader is referred to [8], where various matrix-vector terms have been analyzed. Some matrix-vector terms are given in table 3.6.

3.7. Writing and Optimizing GPU Kernels

Writing a GPU kernel for an algorithm A assumes that the algorithm can be sufficiently parallelized. This is quite cumbersome and not an easy issue. After having found suitable subtasks $S_1, S_2 \dots S_n$ which make up the whole implementation of algorithm A, the GPU implementation should be tackled. Therefore, one should think about which memory spaces to use for each subtasks and how the work-flow of a CUDA kernel should look like. There are tons of possibilities how to structure these subtasks to fit on the GPU. One can think of implementing all subtasks S_i in one kernel only. Another possibility could be to launch all subtasks in a separate kernel due to global synchronization between the subtasks which might be important for the result of the algorithm. Because there are a vast number of possible parallel implementations of an algorithm, a good strategy is first to choose one parallel approach and try to implement a simple straight forward solution on the GPU, not yet thinking about all optimizations which can be made. The first concern should be the correctness of the parallel algorithm on the GPU and as a later step the optimization of the code to strive for better performance. Overall performance optimization can be gathered in these main steps:

- **Parallel Execution Optimization**: Restructure the subtasks $S_1, S_2 \dots S_n$ in a way that exposes as much data parallelism as possible.
- **Execution Optimization:** Try to find launch configurations for kernels that give good performance results. Parameters such as threads and threads per block influence the workload of the GPU. More is said in section 3.7.2.
- **Memory Optimization:** Optimizing memory access is the most important part in optimizing a kernel. This part is briefly reviewed in section 3.7.3.
- **Control Flow Optimization:** Ensuring that branching and diverging happens as few as possible in a kernel is another key factor to gain performance. Avoiding branching is not so difficult to achieve and is discussed further in section “Control Flow” in [22].
- **Instruction Optimization**: Using instructions with low throughput should be avoided. The reader is referred to chapter 5 “Instruction Optimizations” in [22].

3.7.1. Checking for Correctness

Because it is very hard to debug parallel executing code on the GPU, one should always cross check the parallel code to a sequential code on the host CPU. Sometimes cross checks for each parallel subtask S_i should be made as well. This requires a lot more work, but as a matter of fact it will pay off the additional effort as soon as the code grows and more and more optimizations are included. Cross checking is

a crucial point, because it will simplify debugging a lot and will also help to find numerical errors, inaccuracies, synchronization errors and much more. For numerical computations, the cross checking mostly involves checking an array of floating point numbers, either `double` or `float`, from the GPU to the exact same array on the CPU. Comparing numbers on an arithmetic architecture, CPU or GPU, is not easy at all due to the fact that each arithmetic unit has its limited precision. Comparing numbers by equality in programming is a common mistake and leads to instability of the code. One might think that instead of comparing by equality, comparing by an error bound ϵ is the solution to this anomaly. Comparing by an error bound has other drawbacks such as choosing the right epsilon. A better method is to use a *termination criterion* as it is used in numerical iterations. A common and often used termination criteria is given by

$$\left|x^{k+1} - x^k\right| \leq \left|x^k\right| T_{rel} + T_{abs}, \tag{3.9}$$

where $x^{k+1} \in \mathbb{R}$ is the scalar at iteration $k+1$ and $x^k \in \mathbb{R}$ at k respectively. T_{rel} is the relative tolerance and T_{abs} is the absolute tolerance in the corresponding unit of x. When this termination criterion becomes true, a numerical iteration can be aborted because x^{k+1} and x^k are sufficiently close together. Thus, the iteration has converged. This criterion can also be used to compare two arrays for cross checking. Just check if (3.9) holds for each element of the two arrays.

Another important method for comparing two arrays is by calculating the average and maximum ULP error. The floating point distance of two numbers gives a good clue if the two results from GPU and CPU are approximately the same or not. If for example the result from the GPU includes some mistakes, then a floating point distance, which will be very high, can already show that the error is due to a bug in the kernel. The floating point distance between CPU and GPU results will be very small if both implementations are identical, meaning that they approximately calculate the same and in the same order. For all GPU implementations presented in this work, there exists an equivalent version for the host CPU for cross checking. Cross checking has been done by looking at the ULP error and the result of eq. 3.9. More about ULP errors and floating point comparison in general can be found in [3, 6, 13].

3.7.2. Execution Optimizations

One of the ways to gain performance is to keep all SMs on the GPU as busy as possible. A kernel execution on a device which poorly balances the workload across the multiprocessors will deliver suboptimal performance. Hence, it is important for the programmer to design and launch a kernel with a configuration of blocks and threads which maximizes hardware utilization. Hardware utilization is commonly referred to as *occupancy*. This chapter is similar to section "Execution Configuration

Optimizations" in [22] which should be consulted for broader information.

Occupancy is defined from the CUDA viewpoint as:

$$Occupancy = \frac{\text{number of active warps per SM}}{\text{maximum possible active warps per SM}}$$

From this definition and by knowing the Compute Capability of the GPU it is actually possible to compute the occupancy from the given resource usage of a certain kernel, namely threads per block, registers used per thread and shared memory per block. NVIDIA provides an Excel spreadsheet named "CUDA_Occupancy_Calculator.xls" to compute the occupancy for these 4 inputs. This spreadsheet can be found in the CUDA Software Development Kit. The results in the next example can be extracted from this spreadsheet.

Example:
A kernel launches $T = 280$ threads per block and uses $R = 30$ registers per thread. For each block $S = 12000$ bytes of shared memory is allocated. If the Compute Capability 2.0 is assumed then with help of table 3.4 the number of warps per block w_B is computed as

$$w_B = \left\lceil \frac{T}{32} \right\rceil = 9,$$

where 32 is the number of threads per warp. Because the limit of warps per SM is $w_{SM,max} = 48$ for Compute Capability 2.0, the number of allocatable blocks per SM $b_{SM,w}$ is

$$b_{SM,w} = \left\lfloor \frac{w_{SM,max}}{w_B} \right\rfloor = \left\lfloor \frac{48}{9} \right\rfloor = 5.$$

This result, $b_{SM,w} = 5$, is not yet the final limiting number of allocatable blocks on the SM. The registers and the shared memory must be included in the calculation as well. The number of registers per block r_B is

$$r_B = R \cdot T = 10240.$$

Because a GPU with CC 2.0 has 32768 registers per SM, the maximum number of allocatable blocks per SM $b_{SM,r}$ is

$$b_{SM,r} = \left\lfloor \frac{32768}{r_B} \right\rfloor = \left\lfloor \frac{32768}{10240} \right\rfloor = 3.$$

The same again for the shared memory. Requested shared memory is $S = 12000$ bytes. The allocation unit size of shared memory is 128 bytes (see table 3.4). There-

fore the final allocatable shared memory size is $S_a = 12032$ bytes per block. The maximum number of shared memory per block is 49152 bytes. Therefore the number of allocatable blocks per SM $b_{SM,s}$ calculates to

$$b_{SM,s} = \left\lfloor \frac{49152}{S_a} \right\rfloor = \left\lfloor \frac{49152}{12032} \right\rfloor = 4.$$

The minimum

$$b_{SM} = \min(b_{SM,w}, b_{SM,r}, b_{SM,s}) = 3$$

determines the limiting factor of how many blocks per SM can be allocated. In this case, the maximum number of thread blocks per SM is limited by the registers per SM and is equal to 3 blocks/SM. The number of warps per SM sums up to $w_{SM} = b_{SM} \cdot w_B = 27$. The resulting *occupancy* yields

$$occupancy = \frac{w_{SM}}{w_{SM,max}} = \frac{27}{48} = 56\%.$$

This means that the used capacity of all multiprocessors is only 56 %. The number of registers for the compiled kernel can be outputted by passing the option `--ptxas-options=-v` to the CUDA compiler NVCC.

In the following, 3 important aspects (from [22]), which should be kept in mind, are quickly outlined. The complete explanations can be found in section 4.3 to 4.5 in [22].

- To hide latency arising from register dependencies, one should launch kernels with a sufficient numbers of threads per SM.
- Higher occupancy does not always equate to better performance. Typically once an occupancy of 50 % has been reached, additional increases in occupancy do not translate into improved performance. Some rules of thumb for the selection of block size and threads are:
  - Threads per block should be a multiple of the warp size to avoid wasting computation on under-populated warps and to facilitate coalescing.
  - Between 128 and 256 threads per block is a good initial range for experimentation with different block sizes.
  - Using several smaller thread blocks rather than one large thread block per multiprocessor is in particularly beneficial to kernels that frequently call `__syncthreads()`.

Expression	MUL	ADD	flops
$\alpha\mathbf{a}$	N		N
$\alpha\mathbf{A}$	MN		MN
$\mathbf{Ab}$	MN	$M(N-1)$	$2MN - M$ ($\mathcal{O}(n^2)$)
$\mathbf{Ab}$	MN	$M(N-1)$	$2MN - M$ ($\mathcal{O}(n^2)$)

Table 3.6.: Comparision of matrix-vector expressions and their flops count, number of additions (ADD) and number of products (MUL). $\mathbf{A} \in \mathbb{R}^{M\times N}, \mathbf{b} \in \mathbb{R}^N, \alpha \in \mathbb{R}$.

3.7.3. Optimizing Memory Access

To achieve better performance, good memory access is the main key factor. Memory bandwidth is basically the bottleneck of most computations.

Global Memory

Global memory can be accessed by all threads and most importantly by the host. It is used to transfer data used by the kernel to the GPU and back. Besides constant and texture memory, this is the only way to move data from and to the GPU. Accessing global memory should always be *coalesced*. Coalesced access means, that a half warp (CC 1.x) or a warp (CC 2.x) accesses global memory aligned and in a segment. The reader is referred to section 3.2.1 “Coalesced Access to Global Memory” in [22].

Shared Memory

Shared memory is on-chip and therefore roughly 100x faster than global memory, provided that there are no *bank conflicts* while accessing shared memory. Shared memory is divided into banks, 16 for CC 1.x and 32 for CC 2.x. In table 3.7, the layout is visualized for CC 2.x. Assumed CC 2.x, bank conflicts arise if two or more threads in a warp access different addresses in the same bank at the same instruction. For example, thread 1 accesses byte with address ’000’ and thread 2 accesses byte ’128’. Byte ’000’ and ’128’ are both in bank 0. When a bank conflict exists, the memory transactions are serialized, so in the example above this would result in 2 memory accesses instead of only one. If two or more threads access the same byte, then this results in a broadcast to all threads and no bank conflict arises.

Bank	**0**	**1**	**2**	...	**31**
Byte Address	000 001 002 003	004 005 006 007	008 009 010 011	...	124 125 126 127
Byte Address	128 129 130 131	132 133 134 135	136 137 138 139	...	252 253 254 255
...	⋮	⋮	⋮	⋮	⋮

Table 3.7.: Shared memory layout divided into 32 banks. 32 for CC 2.x and 16 for CC 1.x. The numbers represent the byte address. Each bank contains 4 bytes which is one `float` or one half of a `double`.

4. Large Scale Non-Smooth Dynamics

To simulate dynamic systems with friction and contacts a robust algorithm to solve the equation of motions together with the corresponding set-valued friction and contact laws is needed. In this section the background of Moreau's Time-Stepping algorithm in combination with a Prox-Iteration is summarized. This algorithm solves the dynamic system on velocity level[1]. Therefore, all contact and friction laws need to be formulated on velocity level. The contact and friction laws are formulated with normal cone inclusions which can be transformed into non-linear equations by proximal functions. These explicit non-linear functions can be solved with different iteration schemes within a time-step of Moreau's Time-Stepping algorithm. This chapter gives also insight into how large multibody systems, e.g. thousands of spheres in box, can be simulated by the help of a contact graph. This chapter is only a brief introduction and summary of works from other authors. The reader is referred to more profound literature about certain topics within the text.

4.1. Equations of Motion

A mechanical system is described by its generalized coordinates $\mathbf{q}$ and its generalized velocities $\mathbf{u}$. For a general mechanical system the equations of motion can be expressed as

$$\begin{aligned} \mathbf{M}(\mathbf{q},t)\ \dot{\mathbf{u}} - \mathbf{h}(\mathbf{q},\mathbf{u},t) - \mathbf{W}(\mathbf{q},t)\boldsymbol{\lambda} &= \mathbf{0} \\ \dot{\mathbf{q}} = \mathbf{F}(\mathbf{q})\ \mathbf{u} + \boldsymbol{\beta}(\mathbf{q},t). & \end{aligned} \tag{4.1}$$

The symmetric and positive definite mass matrix is denoted by $\mathbf{M}$. Vector $\mathbf{h}$ denotes the non-linear terms including Coriolis terms, centripetal terms and impressed generalized forces. Matrix $\mathbf{F}(\mathbf{q})$ together with the non-linear term $\boldsymbol{\beta}(\mathbf{q},t)$ maps the generalized velocities to the derivative of the generalized coordinates. The generalized forces $\boldsymbol{\lambda}$ and their associated generalized force directions $\mathbf{W}(\mathbf{q},t)$ are explained in the following.

[1] A solver which acts on velocity level is also called *Index-2-Solver*.

In general, one can distinguish between *contacts*, collision of two bodies, and *joints* or *constraints* between two bodies:

A *contact* (subscript c) can be equipped with different combinations of force laws on velocity level or displacement level. Therefore, a contact force $\boldsymbol{\lambda}_{c,i}$ of a contact i may be comprised of several different force laws. The simplest is the unilateral contact, where the contact force is one-dimensional and points in normal direction, i.e., $\boldsymbol{\lambda}_{c,i} \in \mathbb{R}$. Another example is an unilateral contact with spatial Coulomb friction, where 2 tangential friction forces are added, i.e., $\boldsymbol{\lambda}_{c,i} \in \mathbb{R}^3$.

A *constraint* (subscript j) can be expressed on velocity or on displacement level in a mechanical system. Some examples are: a cylindrical joint, spherical joint or a prismatic joint between two bodies (displacement level) or an ideal rolling constraint (velocity level). It is conceivable to have friction inside these joints as well. Therefore, additional friction laws can be added. The constraint force $\boldsymbol{\lambda}_{j,i}$ of a constraint i can be multidimensional as well.

All forces $\boldsymbol{\lambda}_i$ and generalized force directions $\mathbf{W}_i$ in a mechanical system, either *constraint* or *contact* force, can be gathered in $\boldsymbol{\lambda}$ and $\mathbf{W}$. For structural reasons, it makes sense to group all *constraint* forces and generalized force directions $\boldsymbol{\lambda}_{j,i}, \mathbf{W}_{j,i}$ together. The same applies to all *contact* forces and generalized force directions $\boldsymbol{\lambda}_{c,i}, \mathbf{W}_{c,i}$. This yields the following structures:

$$\boldsymbol{\lambda}_c := \left(\dots, \boldsymbol{\lambda}_{c,i}^\top, \dots\right)^\top, \quad \boldsymbol{\lambda}_j := \left(\dots, \boldsymbol{\lambda}_{j,i}^\top, \dots\right)^\top, \tag{4.2}$$

$$\mathbf{W}_c := \left(\dots, \mathbf{W}_{c,i}, \dots\right), \quad \mathbf{W}_j := \left(\dots, \mathbf{W}_{j,i}, \dots\right), \tag{4.3}$$

$$\mathbf{W} := \left(\mathbf{W}_c, \mathbf{W}_j\right), \quad \boldsymbol{\lambda} := \left(\boldsymbol{\lambda}_c^\top, \boldsymbol{\lambda}_j^\top\right)^\top. \tag{4.4}$$

This work concentrates only on *contact* forces. *Constraint* forces are not considered, but can be added and solved in a similar manner. **In the further course of this work, the subscript i is used to denote that the term corresponds to a contact i.**

Section 4.2 gives a short overview about normal cones and proximal points which is needed in the next chapters, especially in section 4.3. Section 4.4 explains briefly how impacts are included in eq. (4.1). Section 4.5 will then give some insight into how large-scale mechanical systems are simulated numerically. Section 4.6 explains two methods of solving inclusion problems, which is used for the further chapters which deal with the GPU implementations of these methods. At last, chapter 4.7 gives some more detailed insight into the contact graph and other structures used for solving contact problems.

4.2. Normal Cones and Proximal Points

A set $\mathcal{C} \subseteq \mathbb{R}^n$ is convex if for all $\mathbf{x} \in \mathcal{C}$ and $\mathbf{y} \in \mathcal{C}$ the convex combination $\lambda\mathbf{x} + (1 - \lambda)\mathbf{y} \in \mathcal{C}$ holds for all $\lambda \in [0, 1]$.

The normal cone $\mathcal{N}_\mathcal{C}(\mathbf{x})$ to a convex set $\mathcal{C} \subseteq \mathbb{R}^n$ at the point $\mathbf{x} \in \mathcal{C}$ is the set of all vectors $\mathbf{y} \in \mathbb{R}^n$ which do not form an acute angle with any vector $\mathbf{x}^* - \mathbf{x}$ for all points $\mathbf{x}^* \in \mathcal{C}$. This yields the following definition:

$$\mathcal{N}_\mathcal{C}(\mathbf{x}) := \left\{ \mathbf{y} \mid \mathbf{y}^\top(\mathbf{x}^* - \mathbf{x}) \leq 0, \ \forall \mathbf{x}^* \in \mathcal{C} \right\}. \tag{4.5}$$

Other properties, such as the epigraph, the indicator function of a convex set and the subdifferential of a convex function are given in [15] in chapter 2.

The proximal point $\text{prox}_\mathcal{C}^R(\mathbf{x})$ of a point $\mathbf{x} \in \mathbb{R}^n$ to the convex set $\mathcal{C} \subseteq \mathbb{R}^n$ is the closest point in $\mathcal{C}$ to $\mathbf{x}$ with respect to the norm $\|\cdot\|_R$:

$$\text{prox}_\mathcal{C}^R(\mathbf{x}) = \underset{\mathbf{x}^* \in \ \mathcal{C}}{\text{argmin}} \, \|\mathbf{x} - \mathbf{x}^*\|_R \,, \tag{4.6}$$

where the norm $\|\mathbf{x}\|_R$ is defined as

$$\|\mathbf{x}\|_R = \sqrt{\mathbf{x}^\top \mathbf{R} \mathbf{x}}, \quad \mathbf{R} \in \mathbb{R}^{n \times n}, \ \mathbf{R} \text{ positive definite (PD)}. \tag{4.7}$$

It can be shown, for example in [15], that the normal cone relates to the proximal point in the following way:

$$\mathbf{y} \in \mathcal{N}_\mathcal{C}(\mathbf{x}) \quad \Leftrightarrow \quad \mathbf{x} = \text{prox}_\mathcal{C}^R(\mathbf{x} + \mathbf{R}^{-1}\mathbf{y}), \tag{4.8}$$

where the left side is the normal cone inclusion and the right side is the implicit proximal point equation.

In most cases, the euclidean norm with $\mathbf{R} = \mathbf{I}$ is used. $\mathbf{I}$ denotes the identity matrix. Because the proximal function is invariant to scaling of the matrix $\mathbf{R}$ of the associated norm $\|\cdot\|_R$ by a positive scalar α, i.e.,

$$\text{prox}_\mathcal{C}^R(\mathbf{x}) = \text{prox}_\mathcal{C}^{\alpha R}(\mathbf{x}), \quad \alpha \in \mathbb{R}^+, \tag{4.9}$$

setting $\mathbf{R} = \frac{1}{r}\mathbf{I}$ in (4.8) yields

$$\mathbf{y} \in \mathcal{N}_\mathcal{C}(\mathbf{x}) \quad \Leftrightarrow \quad \mathbf{x} = \text{prox}_\mathcal{C}^I(\mathbf{x} + r\mathbf{y}), \quad r \in \mathbb{R}^+. \tag{4.10}$$

The supercript I will be neglected in the following, because the euclidean norm is explicitly assumed. More advanced normal cone transformations are found in section 2.4 in [15]. There is shown how linear mappings $\mathbf{A}$ applied on a convex set $\mathcal{C}$

transforms the relation in eq. (4.8).

4.3. Set-Valued Force Laws

The system in (4.1) is not complete as long as the force laws for the generalized forces $\boldsymbol{\lambda}$ are not known. Force laws may be expressed as normal cone inclusions either on velocity or displacement level. A normal cone inclusion on displacement level has the form

$$\mathbf{g}(\mathbf{q}, \mathbf{u}, t) \in \mathcal{N}_{\mathcal{C}}(-\boldsymbol{\lambda}), \tag{4.11}$$

and on velocity level

$$\boldsymbol{\gamma}(\mathbf{q}, \mathbf{u}, t) \in \mathcal{N}_{\mathcal{C}}(-\boldsymbol{\lambda}). \tag{4.12}$$

The non-linear term $\mathbf{g}(\mathbf{q}, \mathbf{u}, t)$ is the relative displacement and $\boldsymbol{\gamma}(\mathbf{q}, \mathbf{u}, t)$ the relative velocity in the mechanical system and $\mathcal{C}$ is the convex set which describes the force law. The relative velocity can be expressed in general as

$$\boldsymbol{\gamma}(\mathbf{q}, \mathbf{u}, t) = \mathbf{W}(\mathbf{q}, t)^{\top}\mathbf{u} + \boldsymbol{\chi}(\mathbf{q}, t). \tag{4.13}$$

Matrix $\mathbf{W}$ contains the generalized force directions and $\boldsymbol{\chi}$ gathers all non-linear terms. In the further course of this work, the explicit dependencies on $\mathbf{q}$ and t in $\mathbf{W}$ and $\boldsymbol{\chi}$ are neglected in further notations. In the following, some commonly used set-valued force laws are explained very briefly. More information about other contact laws such as the Coulomb-Contensou friction model or the bilateral contact can be found in section 3.7 in [15].

Unilateral Contact

If two bodies collide with each other, a normal force λ_N acts on each body at the contact point. The normal force acts certainly at each body in opposite direction. The associated force law for this contact can be expressed on velocity level as

$$\begin{aligned} g_N > 0: &\quad \lambda_N = 0 \\ g_N = 0: &\quad \gamma_N \in \mathcal{N}_{\mathcal{C}_N}(-\lambda_N), \quad \mathcal{C}_N = \mathbb{R}_0^-, \end{aligned} \tag{4.14}$$

where g_N denotes the displacement in normal direction between the two bodies, and γ_N the relative velocity in normal direction. A visualization of the unilateral contact law can be found in figure 4.1(a).

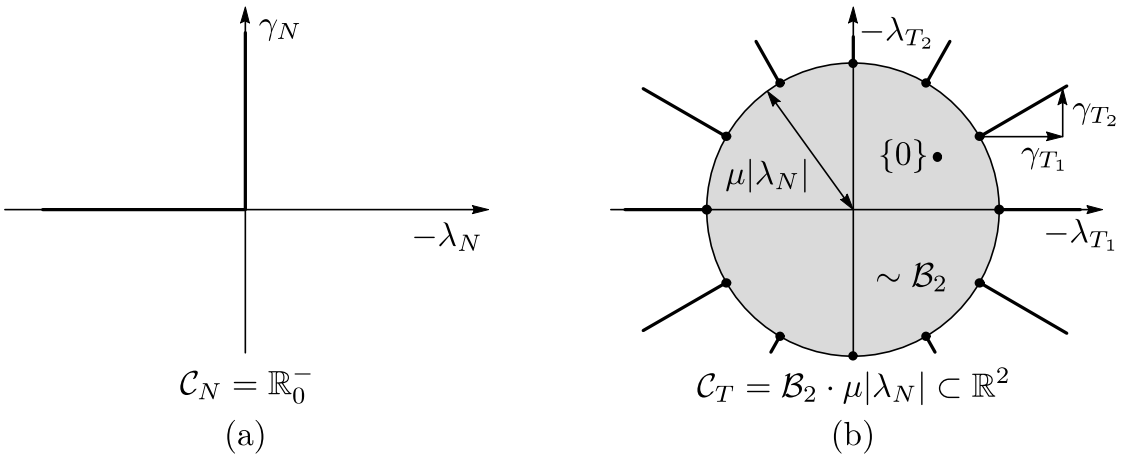


Figure 4.1.: Normal cone inclusions: (a) unilateral contact & (b) spatial Coulomb friction.

Spatial Coulomb Friction

Spatial Coulomb friction can be used to model a frictional contact between two bodies. The friction force $\boldsymbol{\lambda}_T = (\lambda_{T_1}, \lambda_{T_2})^\top$ acts in the tangential plane between the two bodies. The direction of the relative velocity $\boldsymbol{\gamma}_T = (\gamma_{T_1}, \gamma_{T_2})^\top$ between the two bodies points always in the negative direction to the friction force. The magnitude of the friction force is additionally proportional to the normal contact force λ_N with a friction parameter μ. The normal cone inclusion on velocity level is given as

$$\boldsymbol{\gamma}_T \in \mathcal{N}_{\mathcal{C}_T}(-\boldsymbol{\lambda}_T), \tag{4.15}$$

where the convex set $\mathcal{C}_T := \mathcal{C}_T(\mu\lambda_N) \in \mathbb{R}^2$ is defined as

$$\mathcal{C}_T(\mu\lambda_N) = \mathcal{B}_2 \cdot \mu\,|\lambda_N|\,, \text{ and } \mathcal{B}_2 := \{\boldsymbol{\lambda} \in \mathbb{R}^2 \mid \|\boldsymbol{\lambda}\| \leq 1\}. \tag{4.16}$$

The set $\mathcal{B}_2$ is the unit ball in $\mathbb{R}^2$. The spatial Coulomb friction law and its convex set is shown in 4.1(b).

4.4. Impacts

To include impact equations in the equations of motion in 4.1, the set-valued force laws have to be equipped with an impact law. When two bodies collide with each other and separate again, the relative velocity behaves discontinuous at impact time (jumps in the velocity-time profile). The generalized coordinates will be continuous since the

relative velocities stay finite. The impact problem consists of finding the post-impact velocity $\mathbf{u}^+$ if the pre-impact velocity $\mathbf{u}^-$ is known and under the assumption that the generalized coordinates stay the same during an impact, i.e., $\mathbf{q}^- = \mathbf{q}^+$. It is out of the scope of this work to explain all the sophisticated subtleties which are involved in the derivation of the impact equations, the reader is referred to [4, 5, 15]. In the following only some facts which are used later are stated. If Newton's impact law is used for a force law k which states that

$$\boldsymbol{\gamma}_k^+ = \epsilon_k \boldsymbol{\gamma}_k^-, \quad \epsilon_k \in [0,1] \tag{4.17}$$

the associated impact inclusion to equation (4.12) can be formulated as

$$\boldsymbol{\gamma}_k^+ + \epsilon_k \boldsymbol{\gamma}_k^- = \boldsymbol{\xi}_k \in \mathcal{N}_{\mathcal{D}_k}(-\boldsymbol{\Lambda}_k).$$

The scalar ϵ_k is the restitution factor and the superscript $\pm$ denotes the pre- and post-impact values. The impact-impulse $\boldsymbol{\Lambda}_k$ is the integrated contact force $\boldsymbol{\lambda}_k$ over the impact time. For $\epsilon_k = 1$, one obtains a complete elastic impact and for $\epsilon_k = 0$ a complete inelastic impact. In this way, each normal cone inclusion can be equipped with an impact inclusion. Strictly speaking, the set $\mathcal{D}_k$ is the integrated convex set $\mathcal{C}_k$ over the impact time. However, for unilateral contacts with spatial Coulomb friction this can be neglected because the convex sets remain the same, i.e.:

$$\mathcal{D}_N = \mathcal{C}_N = \mathbb{R}_0^-, \quad \mathcal{D}_T = \mathcal{C}_T(\mu \Lambda_N). \tag{4.18}$$

4.5. Moreau's Time-Stepping Algorithm

This algorithm was firstly introduced in [17] and is a difference scheme to evaluate the complete set of the set-valued contact laws which includes both the impact and impact free motion. The Moreau Time-Stepping is an explicit time-stepper and is basically a midpoint discretization on displacement level and an Euler backward method on velocity level. It is assumed that the equations of motion are equipped with set valued force laws and have the form:

$$\begin{gathered} \mathbf{M}(\mathbf{q},t)\ \dot{\mathbf{u}} - \mathbf{h}(\mathbf{q},\mathbf{u},t) - \mathbf{W}\boldsymbol{\lambda} = \mathbf{0}, \quad \dot{\mathbf{q}} = \mathbf{F}(\mathbf{q})\ \mathbf{u} + \boldsymbol{\beta}(\mathbf{q},t) \\ \left.\begin{gathered} \boldsymbol{\gamma}_i = \mathbf{W}_i^\top \mathbf{u} + \boldsymbol{\chi}_i \\ (\boldsymbol{\gamma}_i, \boldsymbol{\lambda}_i) \in S_i \end{gathered}\right\} \quad \forall i \in \mathcal{I}(\mathbf{q},t). \end{gathered} \tag{4.19}$$

It is assumed that $\mathbf{q}$ and $\mathbf{u}$ of the mechanical system is assembled of all rigid body minimal coordinates $\mathbf{q}_K$ and velocities $\mathbf{u}_K$, i.e.:

$$\mathbf{q} := (\dots, \mathbf{u}_K^\top, \dots)^\top, \quad \mathbf{u} = (\dots, \mathbf{u}_K^\top, \dots)^\top. \tag{4.20}$$

The set $\mathcal{I}$ contains all contacts i which are closed on displacement level and is defined as

$$\mathcal{I}(\mathbf{q},t) := \{i \mid \mathbf{g}_i(\mathbf{q},t) \leqslant 0\}. \tag{4.21}$$

The set S_i in (4.19) is just for convenience and consists of all normal cone inclusions k for the corresponding contact i, i.e.:

$$\begin{aligned} S_i &:= \{(\boldsymbol{\gamma},\boldsymbol{\lambda}) \mid \boldsymbol{\gamma}_k \in \mathcal{N}_{\mathcal{C}_k}(-\boldsymbol{\lambda}_k) \quad \forall k\}, \\ \text{with: } &\begin{cases} \boldsymbol{\gamma} := (\dots, \boldsymbol{\gamma}_k^\top, \dots)^\top \\ \boldsymbol{\lambda} := (\dots, \boldsymbol{\lambda}_k^\top, \dots)^\top. \end{cases} \end{aligned} \tag{4.22}$$

Thus, for an unilateral contact i (see section 4.3) with only one normal cone inclusion this yields

$$\begin{aligned} S_i &:= \left\{(\boldsymbol{\gamma},\boldsymbol{\lambda}) \mid \gamma_N \in \mathcal{N}_{\mathbb{R}_0^-}(-\lambda_N)\right\} \\ \text{with: } &\boldsymbol{\lambda} := (\lambda_N), \ \boldsymbol{\gamma} := (\gamma_N) \in \mathbb{R}. \end{aligned} \tag{4.23}$$

The set S_i for an unilateral contact i (see section 4.3) with spatial Coulomb friction yields

$$\begin{aligned} S_i &= \left\{(\boldsymbol{\gamma},\boldsymbol{\lambda}) \mid \gamma_N \in \mathcal{N}_{\mathbb{R}_0^-}(-\lambda_N), \ \boldsymbol{\gamma}_T \in \mathcal{N}_{\mathcal{C}_{T_i}}(-\boldsymbol{\lambda}_T)\right\}, \\ \text{with: } &\boldsymbol{\lambda} := \left(\lambda_N, \boldsymbol{\lambda}_T^\top\right)^\top, \ \boldsymbol{\gamma} := \left(\gamma_N, \boldsymbol{\gamma}_T^\top\right)^\top \in \mathbb{R}^3, \quad \mathcal{C}_{T_i} := \mathcal{C}_T(\mu_i \lambda_N). \end{aligned} \tag{4.24}$$

The terms $\mathbf{W}$ and $\boldsymbol{\lambda}$ contain only the contact force laws which are closed on displacement level, i.e.:

$$\boldsymbol{\lambda} := \left(\dots, \boldsymbol{\lambda}_i^\top, \dots\right)^\top, \quad \mathbf{W} := (\dots, \mathbf{W}_i, \dots) \quad \forall i \in \mathcal{I}(\mathbf{q},t). \tag{4.25}$$

The equation of motion in (4.19) are completed by adding the impact inclusions in the following form:

$$\left.\begin{aligned} \mathbf{M}(\mathbf{q},t)\ (\mathbf{u}^+ - \mathbf{u}^-) - \mathbf{W}\boldsymbol{\Lambda} &= \mathbf{0} \\ \boldsymbol{\gamma}_i^\pm = \mathbf{W}_i^\top \mathbf{u}^\pm + \boldsymbol{\chi}_i \\ \boldsymbol{\xi}_i = \boldsymbol{\gamma}_i^+ + \boldsymbol{\epsilon}_i \boldsymbol{\gamma}_i^- \\ (\boldsymbol{\xi}_i, \boldsymbol{\Lambda}_i) \in R_i \end{aligned}\right\} \quad \forall i \in \mathcal{I}(\mathbf{q},t). \tag{4.26}$$

The set R_i sums again all impact normal cone inclusions for the corresponding contact i, (analog to S_i). Thus, for an unilateral contact i with spatial Coulomb friction, the

set R_i is given as

$$R_i = \left\{ (\boldsymbol{\xi}, \boldsymbol{\Lambda}) \mid \xi_N \in \mathcal{N}_{\mathbb{R}_0^-}(-\Lambda_N),\ \boldsymbol{\xi}_T \in \mathcal{N}_{\mathcal{D}_{T_i}}(-\boldsymbol{\Lambda}_T) \right\}, \tag{4.27}$$
$$\text{with: } \boldsymbol{\Lambda} := \left(\Lambda_N, \boldsymbol{\Lambda}_T^\top\right)^\top,\ \boldsymbol{\xi} := \left(\xi_N, \boldsymbol{\xi}_T^\top\right)^\top \in \mathbb{R}^3, \quad \mathcal{D}_{T_i} := \mathcal{C}_T(\mu_i \Lambda_N)$$

and the diagonal matrix of the restitution coefficients $\boldsymbol{\epsilon}_i$ is defined as

$$\boldsymbol{\epsilon}_i := \mathbf{diag}(\epsilon_{N_i}, \epsilon_{T_i}, \epsilon_{T_i}) \in \mathbb{R}^{3\times 3}. \tag{4.28}$$

In the following, all vectors $\boldsymbol{\xi}_i, \boldsymbol{\Lambda}_i, \boldsymbol{\chi}_i$ and matrices $\boldsymbol{\epsilon}_i$ are gathered to the terms

$$\begin{aligned} \boldsymbol{\Lambda} &:= \left(\dots, \boldsymbol{\Lambda}_i^\top, \dots\right)^\top, \quad \boldsymbol{\xi} := \left(\dots, \boldsymbol{\xi}_i^\top, \dots\right)^\top, \\ \boldsymbol{\chi} &:= \left(\dots, \boldsymbol{\chi}_i^\top, \dots\right)^\top, \quad \boldsymbol{\epsilon} := \mathbf{diag}(\dots, \boldsymbol{\epsilon}_i, \dots) \quad \forall i \in \mathcal{I}(\mathbf{q}, t). \end{aligned} \tag{4.29}$$

Moreau's midpoint rule uses one difference scheme to approximate both the equations of motion in inclusion form (4.19) and the impact inclusions (4.26). The algorithm directly uses the impact inclusions (4.26) to obtain the discretized inclusion over a time step $\Delta t = [t_S, t_E]$, i.e.:

$$\left.\begin{gathered} \mathbf{M}\,(\mathbf{u}^E - \mathbf{u}^S) - \mathbf{h} - \mathbf{W}\boldsymbol{\Lambda} = \mathbf{0} \\ \boldsymbol{\gamma}_i^S = \mathbf{W}_i^\top \mathbf{u}^S + \boldsymbol{\chi}_i, \quad \boldsymbol{\gamma}_i^E = \mathbf{W}_i^\top \mathbf{u}^E + \boldsymbol{\chi}_i, \\ \boldsymbol{\xi}_i = \boldsymbol{\gamma}_i^E + \boldsymbol{\epsilon}_i \boldsymbol{\gamma}_i^S, \\ (\boldsymbol{\xi}_i, \boldsymbol{\Lambda}_i) \in R_i \end{gathered}\right\} \quad \forall i \in \mathcal{I}(\mathbf{q}, t). \tag{4.30}$$

With the help of the definitions in (4.25,4.29), eq. (4.30) can be rewritten as an inclusion in the form:

$$\begin{gathered} \boldsymbol{\xi} = \mathbf{G}\boldsymbol{\Lambda} + \mathbf{c} \\ (\boldsymbol{\xi}_i, \boldsymbol{\Lambda}_i) \in R_i, \quad \forall i \in \mathcal{I}(\mathbf{q}, t) \\ \text{with: } \left\{ \begin{array}{rcl} \mathbf{G} & := & \mathbf{W}^\top \mathbf{M}^{-1} \mathbf{W} \\ \mathbf{c} & := & \mathbf{W}^\top \mathbf{M}^{-1} \mathbf{h} \Delta t + (\mathbf{I} + \boldsymbol{\epsilon})(\mathbf{W}^\top \mathbf{u}^S + \boldsymbol{\chi}). \end{array} \right. \end{gathered} \tag{4.31}$$

More information can be found in [15]. Moreau's time-stepping algorithm is presented in table 4.2.

4.6. Solving Normal Cone Inclusions

This section considers solving the inclusion in eq. (4.31). As shown in eq. (4.8, 4.10), all normal cone inclusions in a mechanical system can be formulated as an implicit

Moreau's Time-Stepping Algorithm

For a given start time t^S and known displacements $\mathbf{q}^S := \mathbf{q}(t^S)$ and velocities $\mathbf{u}^S := \mathbf{u}(t^S)$ one computes an approximated solution for $\mathbf{q}^E := \mathbf{q}(t^E)$ and $\mathbf{u}^E := \mathbf{u}(t^E)$ at the end time t^E of the time interval $\Delta t := [t^S, t^E]$.

Step 1: Calculate for a chosen time-step Δt the midpoint $t^M = t^S + \frac{1}{2}\Delta t$ and the end time $t^E = t^S + \Delta t$ of the time interval $[t^S, t^E]$

Step 2: Calculate the displacements at time t^M by:
$\mathbf{q}^M = \mathbf{q}^S + \frac{1}{2}\Delta t\ \mathbf{F}(\mathbf{q}^S)\ \mathbf{u}^S$.

Step 3: Calculate the mass matrix $\mathbf{M} := \mathbf{M}(\mathbf{q}^M, t^M)$ and the vector $\mathbf{h} := \mathbf{h}(\mathbf{q}^M, \mathbf{u}^S, t^M)$. Set up the index set $\mathcal{I} := \{i \mid \mathbf{g}_i(\mathbf{q}^M, t^M) \leqslant 0\}$. For all numerically closed contacts $i \in \mathcal{I}$ collect all $\mathbf{W}_i$ evaluated at $(\mathbf{q}^M, t^M)$ in matrix $\mathbf{W}$. Collect also the non-linear terms $\boldsymbol{\chi}_i$ evaluated at $(\mathbf{q}^M, t^M)$ in vector $\boldsymbol{\chi}$.

Step 4: Calculate the velocity $\mathbf{u}^E$ by solving the inclusion problem:

if the set $\mathcal{I}$ is not empty **then**

Calculate the terms $\mathbf{G}$, $\mathbf{c}$:

$$\mathbf{G} = \mathbf{W}^\top \mathbf{M}^{-1} \mathbf{W}$$

$$\mathbf{c} = \mathbf{W}^\top \mathbf{M}^{-1}(\mathbf{h}^M \Delta t) + (\mathbf{I} + \boldsymbol{\epsilon})(\mathbf{W}^\top \mathbf{u}^S + \boldsymbol{\chi}).$$

Solve the set of normal cone inclusions R_i to obtain $\boldsymbol{\Lambda}$:

$$(\boldsymbol{\xi}_i, \boldsymbol{\Lambda}_i) \in R_i, \quad \forall i \in \mathcal{I}(\mathbf{q}, t) \text{ and } \boldsymbol{\xi} = \mathbf{G}\boldsymbol{\Lambda} + \mathbf{c}.$$

Compute the velocities $\mathbf{u}^E$:

$$\mathbf{u}^E = \mathbf{u}^S + \mathbf{M}^{-1}(\mathbf{h}\Delta t + \mathbf{W}\boldsymbol{\Lambda}).$$

else

Compute the velocities $\mathbf{u}^E$:

$$\mathbf{u}^E = \mathbf{u}^S + \mathbf{M}^{-1}(\mathbf{h}\Delta t).$$

end if

Step 6: Calculate the end displacements $\mathbf{q}^E$ by a half time-step:
$\mathbf{q}^E = \mathbf{q}^M + \frac{1}{2}\Delta t\ \mathbf{F}(\mathbf{q}^M)\ \mathbf{u}^E$.

Step 7: Set $\mathbf{q}^S = \mathbf{q}^E, \mathbf{u}^S = \mathbf{u}^E$ and go to **Step 1** for a next time-step.

Table 4.2.: Moreau's Time-Stepping algorithm.

proximal function. Thus, eq. (4.31) can be rewritten in the form:

$$\begin{gathered} \boldsymbol{\xi} = \mathbf{G}\boldsymbol{\Lambda} + \mathbf{c}, \\ (\boldsymbol{\xi}_i, \boldsymbol{\Lambda}_i) \in P_i, \quad \forall i \in \mathcal{I}(\mathbf{q}, t), \\ \text{with: } \left\{ \begin{array}{rcl} \mathbf{G} & := & \mathbf{W}^\top \mathbf{M}^{-1} \mathbf{W} \\ \mathbf{c} & := & \mathbf{W}^\top \mathbf{M}^{-1} (\mathbf{h}\Delta t) + (\mathbf{I} + \boldsymbol{\epsilon})(\mathbf{W}^\top \mathbf{u}^S + \boldsymbol{\chi}). \end{array} \right. \end{gathered} \tag{4.32}$$

Analog to the sets defined in eq. (4.23,4.24), the set P_i of an unilateral contact i with spatial Coulomb friction can be defined as

$$P_i = \left\{ (\boldsymbol{\xi}, \boldsymbol{\Lambda}) \;\middle|\; \begin{array}{l} \Lambda_N = -\text{prox}_{\mathbb{R}_0^-}(-\Lambda_N + r_{N_i}\xi_N) \\ \boldsymbol{\Lambda}_T = -\text{prox}_{\mathcal{D}_{T_i}}(-\boldsymbol{\Lambda}_T + r_{T_i}\boldsymbol{\xi}_T) \end{array} \right\}, \tag{4.33}$$

$$\text{with: } \left\{ \begin{array}{l} \boldsymbol{\Lambda} := (\Lambda_N, \boldsymbol{\Lambda}_T^\top)^\top \quad \boldsymbol{\xi} := (\xi_N, \boldsymbol{\xi}_T^\top)^\top, \quad \mathcal{D}_{T_i} = \mathcal{C}_T(\mu_i \Lambda_N) \\ \mathbf{R}_i^{-1} := \mathbf{diag}(r_{N,i}, r_{T,i}, r_{T,i}) \in \mathbb{R}^3, \quad \boldsymbol{\mu}_i := (\mu_i) \in \mathbb{R}^+, \end{array} \right.$$

where the euclidean norm has been used as shown in (4.10).

In the following, eq. (4.32) is rewritten in the form:

$$\boldsymbol{\Lambda} = \mathbf{prox}(\boldsymbol{\Lambda} - \mathbf{R}^{-1}(\mathbf{G}\boldsymbol{\Lambda} + \mathbf{c})) \tag{4.34}$$

$$\text{with: } \left\{ \begin{array}{rcl} \mathbf{G} & := & \mathbf{W}^\top \mathbf{M}^{-1} \mathbf{W} \\ \mathbf{c} & := & \mathbf{W}^\top \mathbf{M}^{-1} (\mathbf{h}\Delta t) + (\mathbf{I} + \boldsymbol{\epsilon})(\mathbf{W}^\top \mathbf{u}^S + \boldsymbol{\chi}) \\ \mathbf{R}^{-1} & := & \mathbf{diag}(\dots, \mathbf{R}_i^{-1}, \dots). \end{array} \right.$$

The term **prox** will denote the vectorial collection of all proximal functions $\text{prox}_{\mathcal{C}_k}$ over the subvectors $\mathbf{x}_k$ of its argument $\mathbf{x}$, i.e.:

$$\mathbf{prox}(\mathbf{x}) := \begin{pmatrix} \vdots \\ -\text{prox}_{\mathcal{C}_k}(-\mathbf{x}_k) \\ \vdots \end{pmatrix}, \quad \forall k. \tag{4.35}$$

The term **prox** contains all different proximal functions for the different contacts i. The reader should also note that all minus signs are now inside the function **prox**.

Equation (4.34) can be solved in different ways. The next section demonstrates briefly two very commonly used iteration schemes following the numerical procedures for solving a linear system in the form $\mathbf{G}\boldsymbol{\Lambda} + \mathbf{c} = \mathbf{0}$.

4.6.1. The JOR Prox Iteration

The over-relaxed Jacobi iteration (JOR) solves the linear system $\mathbf{G}\boldsymbol{\Lambda} + \mathbf{c} = \mathbf{0}$ for $\boldsymbol{\Lambda}$. The normal Jacobi iteration can be easily derived by splitting $\mathbf{G}$ into a strictly lower triangular matrix $\mathbf{L}$, a strictly upper triangular matrix $\mathbf{U}$ and a diagonal matrix $\mathbf{D}$, i.e., $\mathbf{G} = \mathbf{L} + \mathbf{D} + \mathbf{U}$. The following two equivalent iteration schemes for the Jacobi iteration can be derived as

$$\begin{aligned}(\mathbf{L}+\mathbf{D}+\mathbf{U})\boldsymbol{\Lambda} + \mathbf{c} = \mathbf{0} \quad &\Rightarrow \quad \mathbf{D}\boldsymbol{\Lambda} = -(\mathbf{L}+\mathbf{U})\boldsymbol{\Lambda} - \mathbf{c} \\ &\Rightarrow \boldsymbol{\Lambda}^{k+1} = -\mathbf{D}^{-1}((\mathbf{L}+\mathbf{U})\boldsymbol{\Lambda}^{k} + \mathbf{c}) \qquad (4.36) \\ &\Leftrightarrow \boldsymbol{\Lambda}^{k+1} = \boldsymbol{\Lambda}^{k} - \mathbf{D}^{-1}(\mathbf{G}\boldsymbol{\Lambda}^{k} + \mathbf{c}). \qquad (4.37)\end{aligned}$$

The latter can be derived by adding and subtracting $\mathbf{D}\boldsymbol{\Lambda}^k$ on the right-hand side in (4.36). The over-relaxed Jacobi iteration (JOR) can be derived from (4.36) by a linear combination between old values and new values as

$$\begin{aligned}\boldsymbol{\Lambda}^{k+1} &= -\alpha\mathbf{D}^{-1}((\mathbf{L}+\mathbf{U})\boldsymbol{\Lambda}^{k} + \mathbf{c}) + (1-\alpha)\boldsymbol{\Lambda}^{k} \qquad (4.38) \\ \Leftrightarrow \boldsymbol{\Lambda}^{k+1} &= \boldsymbol{\Lambda}^{k} - \alpha\mathbf{D}^{-1}(\mathbf{G}\boldsymbol{\Lambda}^{k} + \mathbf{c}), \qquad (4.39)\end{aligned}$$

where α is the relaxation parameter. Wrapping now a proximal function around the right-hand side, one gets the projected over-relaxed Jacobi scheme (JOR Prox) for the inclusion problem in (4.34):

$$\begin{aligned}\boldsymbol{\Lambda}^{k+1} &= \mathbf{prox}\left(\boldsymbol{\Lambda}^{k} - \mathbf{R}^{-1}(\mathbf{G}\boldsymbol{\Lambda}^{k} + \mathbf{c})\right) \\ &= \mathbf{prox}\left((\mathbf{I} - \mathbf{R}^{-1}\mathbf{G})\boldsymbol{\Lambda}^{k} - \mathbf{R}^{-1}\mathbf{c})\right), \\ &= \mathbf{prox}\left(\mathbf{T}\boldsymbol{\Lambda}^{k} + \mathbf{d}\right), \qquad (4.40)\end{aligned}$$

where the definitions $\mathbf{T} = (\mathbf{I} - \mathbf{R}^{-1}\mathbf{G})$, $\mathbf{d} = -\mathbf{R}^{-1}\mathbf{c}$ and $\mathbf{R}^{-1} = \alpha\mathbf{D}^{-1}$ are used. The structure in eq. (4.40) is helpful for the further development of a GPU implementation. It is not so obvious to see that the iteration with an additional projection converges to the solution of the inclusion problem (4.34). If the iteration scheme in (4.40) does converge, the fix point fulfills at least eq. (4.34). This can be seen by setting $\boldsymbol{\Lambda}^{k+1} = \boldsymbol{\Lambda}^{k}$ in (4.40). More in-depth information about convergence can be found in [15, 30]. Ideally, the matrix $\mathbf{R}^{-1}$ should be chosen as $\mathbf{R}^{-1} = \alpha\mathbf{D}^{-1}$ to make (4.40) as equal to the JOR scheme as possible. This cannot be achieved, because the scalar diagonal values $\mathbf{R}^{-1}_{i,(j,j)}$ for a contact i cannot in general be chosen independently because of the multi-dimensional nature of the proximal function (see eq. (4.33)). Therefore, the maximum is chosen. For an unilateral contact i with spatial

Coulomb friction and assuming the euclidean norm, this yields

$$\mathbf{R}_i^{-1} = \mathbf{diag}(r_{N,i}, r_{T,i}, r_{T,i}), \tag{4.41}$$

$$r_{N,i} = \frac{\alpha}{\mathbf{G}_{ii,(1,1)}}, \tag{4.42}$$

$$r_{T,i} = \frac{\alpha}{\max\left(\mathbf{G}_{ii,(2,2)}, \mathbf{G}_{ii,(3,3)}\right)}. \tag{4.43}$$

4.6.2. The SOR Prox Iteration

The projected successive over-relaxation (SOR Prox or projected over-relaxed Gauss-Seidel) follows directly from the associated successive over-relaxation (SOR, over-relaxed Gauss-Seidel) for the same linear system as in the JOR scheme. The normal Gauss-Seidel iteration for a linear system $\mathbf{G}\boldsymbol{\Lambda} + \mathbf{c} = \mathbf{0}$ with $\mathbf{G} = \mathbf{L} + \mathbf{D} + \mathbf{U}$ is given as

$$\begin{aligned}
(\mathbf{L} + \mathbf{D} + \mathbf{U})\boldsymbol{\Lambda} + \mathbf{c} = \mathbf{0} \quad &\Rightarrow \quad \mathbf{D}\boldsymbol{\Lambda} = -(\mathbf{L} + \mathbf{U})\boldsymbol{\Lambda} - \mathbf{c} \\
&\rightarrow \mathbf{D}\boldsymbol{\Lambda}^{k+1} = -\mathbf{L}\boldsymbol{\Lambda}^{k+1} - \mathbf{U}\boldsymbol{\Lambda}^{k} - \mathbf{c} \\
&\Rightarrow \boldsymbol{\Lambda}^{k+1} = -\mathbf{D}^{-1}(\mathbf{L}\boldsymbol{\Lambda}^{k+1} + \mathbf{U}\boldsymbol{\Lambda}^{k} + \mathbf{c}) \qquad (4.44) \\
&\Leftrightarrow \boldsymbol{\Lambda}^{k+1} = \boldsymbol{\Lambda}^{k} - \mathbf{D}^{-1}(\mathbf{L}\boldsymbol{\Lambda}^{k+1} + (\mathbf{U} + \mathbf{D})\boldsymbol{\Lambda}^{k} + \mathbf{c}), \qquad (4.45)
\end{aligned}$$

where the latter can be derived by adding and subtracting $\mathbf{D}\boldsymbol{\Lambda}^k$ on the right-hand side in (4.44). Introducing a relaxation parameter α in the same way as in the previous section yields the over-relaxed Gauss-Seidel (SOR) scheme:

$$\begin{aligned}
\boldsymbol{\Lambda}^{k+1} &= -\alpha\mathbf{D}^{-1}(\mathbf{L}\boldsymbol{\Lambda}^{k+1} + \mathbf{U}\boldsymbol{\Lambda}^{k} + \mathbf{c}) + (1 - \alpha)\boldsymbol{\Lambda}^{k} \qquad (4.46) \\
\Rightarrow \boldsymbol{\Lambda}^{k+1} &= \boldsymbol{\Lambda}^{k} - \alpha\mathbf{D}^{-1}(\mathbf{L}\boldsymbol{\Lambda}^{k+1} + (\mathbf{U} + \mathbf{D})\boldsymbol{\Lambda}^{k} + \mathbf{c}). \qquad (4.47)
\end{aligned}$$

The projected version which solves (4.34) is then given as

$$\begin{aligned}
\boldsymbol{\Lambda}^{k+1} &= \mathbf{prox}\left(\boldsymbol{\Lambda}^{k} - \alpha\mathbf{D}^{-1}(\mathbf{L}\boldsymbol{\Lambda}^{k+1} + (\mathbf{U} + \mathbf{D})\boldsymbol{\Lambda}^{k} + \mathbf{c})\right) \\
&= \mathbf{prox}\left(\boldsymbol{\Lambda}^{k} - \mathbf{R}^{-1}(\mathbf{L}\boldsymbol{\Lambda}^{k+1} + (\mathbf{U} + \mathbf{D})\boldsymbol{\Lambda}^{k} + \mathbf{c})\right) \\
&= \mathbf{prox}\left(-\mathbf{R}^{-1}\mathbf{L}\boldsymbol{\Lambda}^{k+1} + (\mathbf{I} - \mathbf{R}^{-1}(\mathbf{U} + \mathbf{D}))\boldsymbol{\Lambda}^{k} + \mathbf{d}\right) \\
&= \mathbf{prox}\left(\tilde{\mathbf{L}}\boldsymbol{\Lambda}^{k+1} + \tilde{\mathbf{U}}\boldsymbol{\Lambda}^{k} + \mathbf{d}\right) \qquad (4.48)
\end{aligned}$$

The matrix $\tilde{\mathbf{L}}$ is a strictly lower triangular matrix and $\tilde{\mathbf{U}}$ is an upper triangular matrix of $\mathbf{T}$. For choosing $\mathbf{R}$, the same rules apply as for the JOR Prox scheme.

4.6.3. Termination Criteria

The termination criterion for the JOR Prox and SOR Prox iteration in (4.40, 4.48) can be chosen for each element in $\mathbf{\Lambda}$ in the form:

$$|\mathbf{\Lambda}_{(i)}^{k+1} - \mathbf{\Lambda}_{(i)}^{k}| \leq |\mathbf{\Lambda}_{(i)}^{k}| \; T_{rel} + T_{abs} \quad \forall i, \tag{4.49}$$

or in a less strong form:

$$\left\|\mathbf{\Lambda}^{k+1} - \mathbf{\Lambda}^{k}\right\|_2 \leq \left\|\mathbf{\Lambda}^{k}\right\|_2 T_{rel} + T_{abs}. \tag{4.50}$$

T_{rel} and T_{abs} are the relative and absolute tolerance. $\|\cdot\|_2$ denotes the euclidean norm. This termination criterion controls the absolute error if $\left\|\mathbf{\Lambda}^{k}\right\|_2 \ll 1$, (resp. $|\mathbf{\Lambda}_{(i)}^{k}| \ll 1$), and the relative error otherwise.

4.6.4. Convergence of JOR and SOR Prox

This section is out of the scope of this work. How to choose the relaxation parameter α such that the inclusion problem still converges to a feasible solution is the topic of the following works [15, 16, 30].

4.6.5. Summary

Both iteration schemes have been summarized in table 4.3.

4.7. Structure of G and Contact Graph

In this work, we are interested in simulating a vast number of rigid bodies to test the JOR Prox and SOR Prox implementations. To give a more programmatic insight about the simulation of large rigid multibody systems, the following section starts off with some simple calculations for two rigid bodies and then shows how matrix $\mathbf{G}$ is related to the *contact graph*. From now on, it is assumed that the contacts between corresponding bodies are equipped with an unilateral force law with spatial Coulomb friction as defined in section 4.3.

Linear Equation	Inclusion Problem
JOR Iteration	
$\begin{aligned}\boldsymbol{\Lambda}^{k+1} &= \boldsymbol{\Lambda}^{k} - \alpha\mathbf{D}^{-1}(\mathbf{L}\boldsymbol{\Lambda}^{k+1} + (\mathbf{U}+\mathbf{D})\boldsymbol{\Lambda}^{k} + \mathbf{c}) \\ &= (\mathbf{I} - \alpha\mathbf{D}^{-1}\mathbf{G})\boldsymbol{\Lambda}^{k} - \alpha\mathbf{D}^{-1}\mathbf{c} \\ \left\lfloor \boldsymbol{\Lambda}^{k+1} \right\rfloor &= \square \cdot \left\lfloor \boldsymbol{\Lambda}^{k} \right\rfloor + \left\lfloor \right\rfloor \end{aligned}$	$\begin{aligned}\boldsymbol{\Lambda}^{k+1} &= \mathbf{prox}\left((\mathbf{I} - \mathbf{R}^{-1}\mathbf{G})\boldsymbol{\Lambda}^{k} - \mathbf{R}^{-1}\mathbf{c}\right) \\ &= \mathbf{prox}\left(\mathbf{T}\boldsymbol{\Lambda}^{k} + \mathbf{d}\right) \\ \left\lfloor \boldsymbol{\Lambda}^{k+1} \right\rfloor &= \mathbf{prox}\left(\square \cdot \left\lfloor \boldsymbol{\Lambda}^{k} \right\rfloor + \left\lfloor \right\rfloor\right) \end{aligned}$
SOR Iteration	
$\begin{aligned}\boldsymbol{\Lambda}^{k+1} &= \boldsymbol{\Lambda}^{k} - \alpha\mathbf{D}^{-1}(\mathbf{L}\boldsymbol{\Lambda}^{k+1} + (\mathbf{U}+\mathbf{D})\boldsymbol{\Lambda}^{k} + \mathbf{c}) \\ &= -\alpha\mathbf{D}^{-1}\mathbf{L}\boldsymbol{\Lambda}^{k+1} + (\mathbf{I} - \alpha\mathbf{D}^{-1}(\mathbf{U}+\mathbf{D}))\boldsymbol{\Lambda}^{k} - \alpha\mathbf{D}^{-1}\mathbf{c} \\ \left\lfloor \boldsymbol{\Lambda}^{k+1} \right\rfloor &= \boxtimes \cdot \left\lfloor \boldsymbol{\Lambda}^{k+1} \right\rfloor + \boxslash \cdot \left\lfloor \boldsymbol{\Lambda}^{k} \right\rfloor + \left\lfloor \right\rfloor \end{aligned}$	$\begin{aligned}\boldsymbol{\Lambda}^{k+1} &= \mathbf{prox}\left(-\mathbf{R}^{-1}\mathbf{L}\boldsymbol{\Lambda}^{k+1} + (\mathbf{I} - \mathbf{R}^{-1}(\mathbf{U}+\mathbf{D}))\boldsymbol{\Lambda}^{k} + \mathbf{d}\right) \\ &= \mathbf{prox}\left(\tilde{\mathbf{L}}\boldsymbol{\Lambda}^{k+1} + \tilde{\mathbf{U}}\boldsymbol{\Lambda}^{k} + \mathbf{d}\right) \\ \left\lfloor \boldsymbol{\Lambda}^{k+1} \right\rfloor &= \mathbf{prox}\left(\boxtimes \cdot \left\lfloor \boldsymbol{\Lambda}^{k+1} \right\rfloor + \boxslash \cdot \left\lfloor \boldsymbol{\Lambda}^{k} \right\rfloor + \left\lfloor \right\rfloor\right) \end{aligned}$
Termination Criteria	
$\left\lvert \boldsymbol{\Lambda}_{(i)}^{k+1} - \boldsymbol{\Lambda}_{(i)}^{k} \right\rvert \leq \left\lvert \boldsymbol{\Lambda}_{(i)}^{k} \right\rvert T_{rel} + T_{abs} \quad \forall i$	
$\left\lVert \boldsymbol{\Lambda}^{k+1} - \boldsymbol{\Lambda}^{k} \right\rVert_2 \leq \left\lVert \boldsymbol{\Lambda}^{k} \right\rVert_2 T_{rel} + T_{abs}$	

Table 4.3.: The JOR and SOR iteration for the linear system $\mathbf{G}\boldsymbol{\Lambda} + \mathbf{c} = \mathbf{0}$ on the left side and the associated inclusion problem on the right side. Matrix $\mathbf{G} = \mathbf{L} + \mathbf{D} + \mathbf{U}$ is split into a strictly lower triangular matrix $\mathbf{L}$, a strictly upper triangular matrix $\mathbf{U}$ and a diagonal matrix $\mathbf{D}$.

4.7.1. Calculation of Generalized Force Directions

Each rigid body K which takes part in a mechanical simulation is equipped with generalized coordinates $\mathbf{q}_K$ and generalized velocities $\mathbf{u}_K$. The generalized minimal coordinates and velocities for a rigid body K with body frame K in the center of gravity S_K are given as

$$\mathbf{q}_K = \begin{pmatrix} {}_I\mathbf{r}_{S_K} \\ {}_I\mathring{\mathbf{a}}_{KI} \end{pmatrix} \in \mathbb{R}^7, \quad \mathbf{u}_K = \begin{pmatrix} {}_I\mathbf{v}_{S_K} \\ {}_K\boldsymbol{\omega}_{IK} \end{pmatrix} \in \mathbb{R}^6, \tag{4.51}$$

where ${}_I\mathbf{r}_{S_K} \in \mathbb{R}^3$ and ${}_I\mathbf{v}_{S_K} \in \mathbb{R}^3$ are the position and velocity of S_K. The angular velocity of K is denoted by ${}_K\boldsymbol{\omega}_{IK} \in \mathbb{R}^3$. The unit quaternion ${}_I\mathring{\mathbf{a}}_{KI} = (a_0, \mathbf{a})^\top \in \mathbb{R}^4$ relates to the rotation matrix ${}_I\mathbf{R}_{KI}$ and the transformation matrix $\mathbf{A}_{IK}$ by

$${}_I\mathbf{R}_{KI} = \mathbf{I} + 2a_0\tilde{\mathbf{a}} + 2\tilde{\mathbf{a}}^2 = \mathbf{A}_{IK}. \tag{4.52}$$

Matrix $\tilde{\mathbf{a}}$ is the skew-symmetric matrix of $\mathbf{a}$ so that $\mathbf{a} \times \mathbf{b} = \tilde{\mathbf{a}}\mathbf{b}$.

Assume two bodies A and B have a contact in a point C, where C_A, C_B refers to the contact point on body A and B. An orthogonal contact frame $(C, \mathbf{e}_x^C, \mathbf{e}_y^C, \mathbf{e}_z^C) = (C, \mathbf{n}_{C,A}, \mathbf{t}_{1,C,A}, \mathbf{t}_{2,C,A})$ is attached in the contact point C such that the x-axis points in normal direction $\mathbf{n}$ to the second body B. The other complementary frame which belongs mutually to body B is obviously given as $(C, \mathbf{n}_{C,B}, \mathbf{t}_{1,C,B}, \mathbf{t}_{2,C,B}) = (C, -\mathbf{n}_{C,A}, -\mathbf{t}_{1,C,A}, -\mathbf{t}_{2,C,A})$. The inertial frame is denoted by I and the transformation matrix from frame I to C is $\mathbf{A}_{IC} = ({}_I\mathbf{e}_x^C, {}_I\mathbf{e}_y^C, {}_I\mathbf{e}_z^C) \in \mathbb{R}^3$.

Given the velocity $\mathbf{v}_{C_A}$ and $\mathbf{v}_{C_B}$ of the contact points C_A and C_B, the relative velocity ${}_C\boldsymbol{\gamma}$ between body A and B expressed in the contact frame C can be computed as

$$\begin{aligned}
{}_C\boldsymbol{\gamma} &= \begin{pmatrix} \gamma_N & \gamma_{T1} & \gamma_{T2} \end{pmatrix}^\top = \mathbf{A}_{IC}^\top({}_I\mathbf{v}_{C_B} - {}_I\mathbf{v}_{C_A}) && (4.53)\\
&= \mathbf{A}_{IC}^\top({}_I\mathbf{v}_{S_B} - {}_I\tilde{\mathbf{r}}_{S_BC_B}\ \mathbf{A}_{IB}\ {}_B\boldsymbol{\omega}_{IB}) - \mathbf{A}_{IC}^\top({}_I\mathbf{v}_{S_A} - {}_I\tilde{\mathbf{r}}_{S_AC_A}\ \mathbf{A}_{IA}\ {}_A\boldsymbol{\omega}_{IA}) && (4.54)\\
&= \mathbf{A}_{IC}^\top \underbrace{\begin{pmatrix} \mathbf{I} & , -{}_I\tilde{\mathbf{r}}_{S_BC_B}\ \mathbf{A}_{IB} \end{pmatrix}}_{{}_I\mathbf{J}_{C_B}} \begin{pmatrix} {}_I\mathbf{v}_{S_B} \\ {}_B\boldsymbol{\omega}_{IB} \end{pmatrix} - \mathbf{A}_{IC}^\top \underbrace{\begin{pmatrix} \mathbf{I} & , -{}_I\tilde{\mathbf{r}}_{S_AC_A}\ \mathbf{A}_{IA} \end{pmatrix}}_{{}_I\mathbf{J}_{C_A}} \begin{pmatrix} {}_I\mathbf{v}_{S_A} \\ {}_A\boldsymbol{\omega}_{IA} \end{pmatrix} && (4.55)\\
&= \mathbf{A}_{IC}^\top {}_I\mathbf{J}_{C_B}\mathbf{u}_B - \mathbf{A}_{IC}^\top {}_I\mathbf{J}_{C_A}\mathbf{u}_A && (4.56)\\
&= \mathbf{W}_{C,B}^\top\ \mathbf{u}_B - \mathbf{W}_{C,A}^\top\ \mathbf{u}_A. && (4.57)
\end{aligned}$$

A term such as $\mathbf{W}_{C,A}$ for example can be expanded to

$$
\begin{aligned}
\mathbf{W}_{C,A} &= \begin{pmatrix} \mathbf{w}_{C,A,N} & \mathbf{w}_{C,A,T_1} & \mathbf{w}_{C,A,T_2} \end{pmatrix} \in \mathbb{R}^{6\times 3}, \\
\mathbf{w}_{C,A,N} &= \begin{pmatrix} \mathbf{I}_3 \\ \mathbf{A}_{AI}\ {}_I\tilde{\mathbf{r}}_{S_AC_A} \end{pmatrix} {}_I\mathbf{n}_{C,A} \in \mathbb{R}^6, \\
\mathbf{w}_{C,A,T_1} &= \begin{pmatrix} \mathbf{I}_3 \\ \mathbf{A}_{AI}\ {}_I\tilde{\mathbf{r}}_{S_AC_A} \end{pmatrix} {}_I\mathbf{t}_{1,C,A} \in \mathbb{R}^6, \\
\mathbf{w}_{C,A,T_2} &= \begin{pmatrix} \mathbf{I}_3 \\ \mathbf{A}_{AI}\ {}_I\tilde{\mathbf{r}}_{S_AC_A} \end{pmatrix} {}_I\mathbf{t}_{2,C,A} \in \mathbb{R}^6.
\end{aligned} \tag{4.58}
$$

When simulating a vast number of bodies, setting up all matrices $\mathbf{W}$ and eventually $\mathbf{G}$ turns out to be quite cumbersome. The next section provides the reader with an example of three bodies which should help to understand how the terms such as $\mathbf{G}$ are computed.

4.7.2. Example with 3 Simulated Bodies:

The system in 4.2 is considered. Three bodies A, B, C are simulated ($nBodies = 3$). The floor is denoted by D. Each body has the minimal coordinates given in (4.51), hence

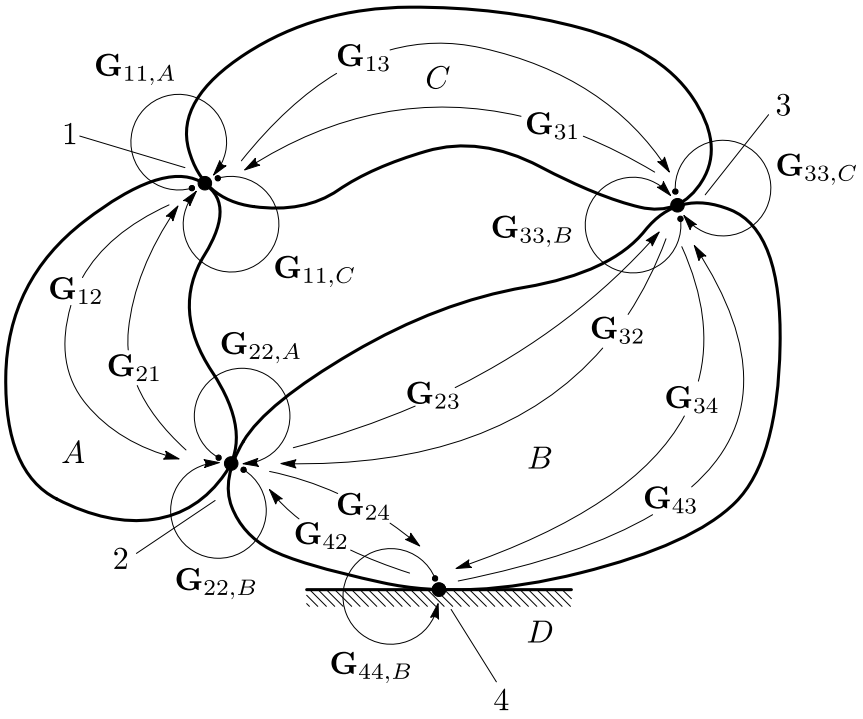
$$
\mathbf{q} = (\mathbf{q}_A^\top, \mathbf{q}_B^\top, \mathbf{q}_C^\top)^\top \in \mathbb{R}^{7\cdot nBodies}, \quad \mathbf{u} = (\mathbf{u}_A^\top, \mathbf{u}_B^\top, \mathbf{u}_C^\top)^\top \in \mathbb{R}^{6\cdot nBodies}. \tag{4.59}
$$

Each body K has a mass matrix $\mathbf{M}_K$ and a nonlinear term $\mathbf{h}_K$. In the configuration shown in figure 4.2 four contacts are shown ($nContacts = 4$), denoted by $1, 2, 3, 4$. Each contact $k = \{1, 2, 3, 4\}$ is equipped with an unilateral force law with spatial Coulomb friction.

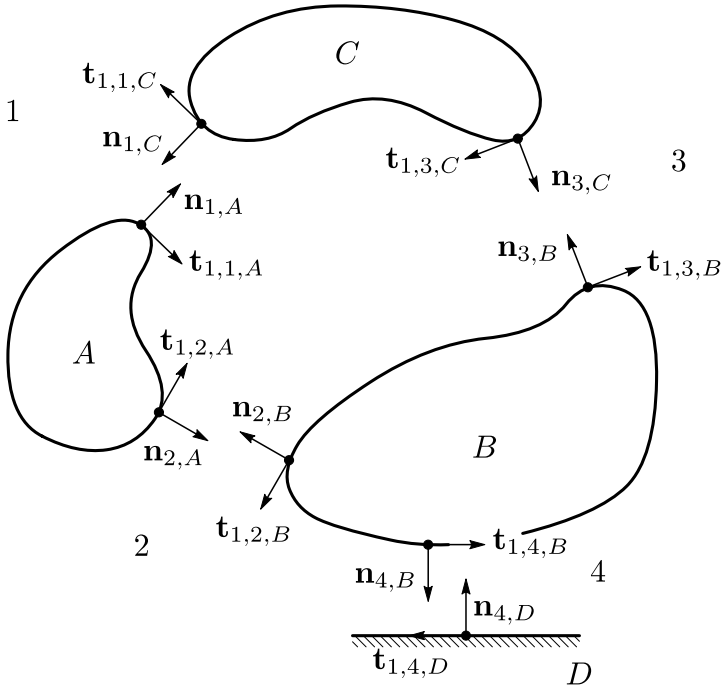
Assembling all contact forces ($ContactDim = 3$) yields

$$
\begin{aligned}
\boldsymbol{\lambda} &= \left(\boldsymbol{\lambda}_1^\top, \boldsymbol{\lambda}_2^\top, \boldsymbol{\lambda}_3^\top, \boldsymbol{\lambda}_4^\top\right)^\top \in \mathbb{R}^{nContacts\cdot ContactDim} \\
\boldsymbol{\lambda}_k &= (\lambda_{1,N}, \lambda_{1,T_1}, \lambda_{1,T_2})^\top \in \mathbb{R}^{ContactDim}.
\end{aligned}
$$

As shown in eq. (4.58), the generalized force direction $\mathbf{W}$ can be assembled from all



(a)



(b)

Figure 4.2.: Three body example: (a) Shows 3 Bodies A, B, C with a floor D and 4 contacts (1,2,3,4). The arrows between the contacts represent the contact graph and shows how all contacts are related to each other. (b) Shows the 3 bodies disconnected, with their contact frames. Only one tangential direction ($\mathbf{t}_1$) is shown.

generalized directions $\mathbf{W}_i$ of each contact i as

$$\begin{aligned}\mathbf{W} = \begin{pmatrix} \mathbf{W}_1 & \mathbf{W}_2 & \mathbf{W}_3 & \mathbf{W}_4 \end{pmatrix} &= \begin{pmatrix} \mathbf{W}_{1,A} & \mathbf{W}_{2,A} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{W}_{2,B} & \mathbf{W}_{3,B} & \mathbf{W}_{4,B} \\ \mathbf{W}_{1,C} & \mathbf{0} & \mathbf{W}_{3,C} & \mathbf{0} \end{pmatrix} \\ &= \left(\begin{array}{ccc|ccc|c} \mathbf{w}_{1,A,N} & \mathbf{w}_{1,A,T_1} & \mathbf{w}_{1,A,T_2} & \mathbf{w}_{2,A,N} & \mathbf{w}_{2,A,T_1} & \mathbf{w}_{2,A,T_2} & \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{w}_{2,B,N} & \mathbf{w}_{2,B,T_1} & \mathbf{w}_{2,B,T_2} & \dots \\ \mathbf{w}_{1,C,N} & \mathbf{w}_{1,C,T_1} & \mathbf{w}_{1,C,T_2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \\ \hline \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \\ \mathbf{w}_{3,B,N} & \mathbf{w}_{3,B,T_1} & \mathbf{w}_{3,B,T_2} & \mathbf{w}_{4,B,N} & \mathbf{w}_{4,B,T_1} & \mathbf{w}_{4,B,T_2} & \\ \mathbf{w}_{3,C,N} & \mathbf{w}_{3,C,T_1} & \mathbf{w}_{3,C,T_2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \end{array}\right).\end{aligned} \tag{4.60}$$

The generalized force $\mathbf{f}$ is then given as

$$\mathbf{f} = \mathbf{W}\boldsymbol{\lambda}, \quad \mathbf{W} \in \mathbb{R}^{(6 \cdot nBodies) \times (ContactDim \cdot nContacts)}.$$

The mass matrix $\mathbf{M}$ and its inverse can be assembled as

$$\begin{gathered}\mathbf{M} = \mathbf{diag}(\mathbf{M}_A, \mathbf{M}_B, \mathbf{M}_C), \quad \mathbf{M}^{-1} = \mathbf{diag}(\mathbf{M}_A^{-1}, \mathbf{M}_B^{-1}, \mathbf{M}_C^{-1}) \\ \mathbf{M}, \mathbf{M}^{-1} \in \mathbb{R}^{6 \cdot nBodies \times 6 \cdot nBodies}.\end{gathered} \tag{4.61}$$

Finally, matrix $\mathbf{G}$ in (4.31) can be computed. If each term is multiplied out, the following structure is obtained:

$$\begin{aligned}&\mathbf{G} = \mathbf{W}^\top \mathbf{M}^{-1} \mathbf{W} = \\ &\begin{pmatrix} \mathbf{W}_{1,\mathrm{A}}^\top \mathbf{M}_A^{-1} \mathbf{W}_{1,A} + \mathbf{W}_{1,\mathrm{C}}^\top \mathbf{M}_C^{-1} \mathbf{W}_{1,C} & \mathbf{W}_{1,\mathrm{A}}^\top \mathbf{M}_A^{-1} \mathbf{W}_{2,A} \\ \mathbf{W}_{2,\mathrm{A}}^\top \mathbf{M}_A^{-1} \mathbf{W}_{1,A} & \mathbf{W}_{2,\mathrm{B}}^\top \mathbf{M}_B^{-1} \mathbf{W}_{2,B} + \mathbf{W}_{2,\mathrm{A}}^\top \mathbf{M}_A^{-1} \mathbf{W}_{2,A} \\ \mathbf{W}_{3,\mathrm{C}}^\top \mathbf{M}_C^{-1} \mathbf{W}_{1,C} & \mathbf{W}_{3,\mathrm{B}}^\top \mathbf{M}_B^{-1} \mathbf{W}_{2,B} \\ \mathbf{0}_3 & \mathbf{W}_{4,\mathrm{B}}^\top \mathbf{M}_B^{-1} \mathbf{W}_{2,B} \end{pmatrix} \\ &\begin{pmatrix} \mathbf{W}_{1,\mathrm{C}}^\top \mathbf{M}_C^{-1} \mathbf{W}_{3,C} & \mathbf{0}_3 \\ \mathbf{W}_{2,\mathrm{B}}^\top \mathbf{M}_B^{-1} \mathbf{W}_{3,B} & \mathbf{W}_{2,\mathrm{B}}^\top \mathbf{M}_B^{-1} \mathbf{W}_{4,B} \\ \mathbf{W}_{3,\mathrm{C}}^\top \mathbf{M}_C^{-1} \mathbf{W}_{3,C} + \mathbf{W}_{3,\mathrm{B}}^\top \mathbf{M}_B^{-1} \mathbf{W}_{3,B} & \mathbf{W}_{3,\mathrm{B}}^\top \mathbf{M}_B^{-1} \mathbf{W}_{4,B} \\ \mathbf{W}_{4,\mathrm{B}}^\top \mathbf{M}_B^{-1} \mathbf{W}_{3,B} & \mathbf{W}_{4,\mathrm{B}}^\top \mathbf{M}_B^{-1} \mathbf{W}_{4,B} \end{pmatrix}.\end{aligned} \tag{4.62}$$

Renaming and grouping the terms, the above yields the final matrix $\mathbf{G}$, i.e.:

$$\begin{aligned}&\mathbf{G} = \mathbf{W}^\top \mathbf{M}^{-1} \mathbf{W} = \\ &\begin{pmatrix} \mathbf{G}_{11,A} + \mathbf{G}_{11,C} & \mathbf{G}_{12,A} & \mathbf{G}_{13,C} & \mathbf{0}_3 \\ \mathbf{G}_{21,A} & \mathbf{G}_{22,B} + \mathbf{G}_{22,A} & \mathbf{G}_{23,B} & \mathbf{G}_{24,B} \\ \mathbf{G}_{31,C} & \mathbf{G}_{32,B} & \mathbf{G}_{33,C} + \mathbf{G}_{33,B} & \mathbf{G}_{34,B} \\ \mathbf{0}_3 & \mathbf{G}_{42,B} & \mathbf{G}_{43,B} & \mathbf{G}_{44,B} \end{pmatrix}.\end{aligned} \tag{4.63}$$

Each part $\mathbf{G}_{ij,K}$ in $\mathbf{G}$ is calculated by the following rule:

$$\mathbf{G}_{ij,K} = \mathbf{W}_{i,\mathrm{K}}^{\top}\mathbf{M}_{K}^{-1}\mathbf{W}_{j,K} \in \mathbb{R}^{ContactDim \times ContactDim} \quad \forall i,j. \tag{4.64}$$

The size of the diagonal submatrices $\mathbf{G}_{ij}$ are dependent on the dimensions of the underlying contact laws. In this example they have the dimension $\mathbb{R}^{3\times 3}$. The reader should note that there is no term $\mathbf{G}_{44,D}$ added to $\mathbf{G}_{44,B}$ in the last diagonal submatrix of $\mathbf{G}$. This is due to the fact that the floor D is not simulated in this example or equivalently its mass is infinite and therefore $\mathbf{G}_{44,D}$ vanishs. Matrix $\mathbf{G}$ is always symmetric and is positive definite (PD) for linear independent generalized directions in $\mathbf{W}$ and positive semi definite (PSD) otherwise. Matrix $\mathbf{G}$ can now be visualized by connections from contact i to all contacts j. The result is the *contact graph* which is visualized in figure 4.2(a) (the last subscript, denoting the body, is not shown for off-diagonal terms in $\mathbf{G}$). The submatrices of $\mathbf{G}$ have a very defined structure. This can be exploited when programming and implementing this graph on a computer. Also the vector $\mathbf{c}$ can be calculated in a structural manner. The terms $\mathbf{h}, \mathbf{c}$ are defined as

$$\begin{aligned} \mathbf{h} &= \begin{pmatrix} \mathbf{h}_A^{\top} & \mathbf{h}_B^{\top} & \mathbf{h}_C^{\top} \end{pmatrix}^{\top} \in \mathbb{R}^{6\cdot nBodies}, \\ \mathbf{c} &= \begin{pmatrix} \mathbf{c}_1^{\top} & \mathbf{c}_2^{\top} & \mathbf{c}_3^{\top} & \mathbf{c}_4^{\top} \end{pmatrix}^{\top} \in \mathbb{R}^{nContacts\cdot ContactDim}. \end{aligned} \tag{4.65}$$

Evaluating $\mathbf{c} = \mathbf{W}^{\top}\mathbf{M}^{-1}(\mathbf{h}^{M}\Delta t) + (\mathbf{I}+\boldsymbol{\epsilon})(\mathbf{W}^{\top}\mathbf{u}^{S} + \boldsymbol{\chi})$ yields:

$$\begin{aligned} \mathbf{c}_1 &= \mathbf{W}_{1,A}^{\top}\mathbf{M}_{\mathrm{A}}^{-1}\mathbf{h}_{\mathrm{A}}\Delta t + \mathbf{W}_{1,C}^{\top}\mathbf{M}_{\mathrm{C}}^{-1}\mathbf{h}_{\mathrm{C}}\Delta t + (\mathbf{I}+\boldsymbol{\epsilon}_1)(\mathbf{W}_{1,A}\mathbf{u}_{\mathrm{A}}^{S} + \mathbf{W}_{1,C}\mathbf{u}_{\mathrm{C}}^{S} + \boldsymbol{\chi}_1) \\ \mathbf{c}_2 &= \mathbf{W}_{2,A}^{\top}\mathbf{M}_{\mathrm{A}}^{-1}\mathbf{h}_{\mathrm{A}}\Delta t + \mathbf{W}_{2,B}^{\top}\mathbf{M}_{\mathrm{B}}^{-1}\mathbf{h}_{\mathrm{B}}\Delta t + (\mathbf{I}+\boldsymbol{\epsilon}_2)(\mathbf{W}_{2,A}\mathbf{u}_{\mathrm{A}}^{S} + \mathbf{W}_{2,B}\mathbf{u}_{\mathrm{B}}^{S} + \boldsymbol{\chi}_2) \\ \mathbf{c}_3 &= \mathbf{W}_{3,B}^{\top}\mathbf{M}_{\mathrm{B}}^{-1}\mathbf{h}_{\mathrm{B}}\Delta t + \mathbf{W}_{3,C}^{\top}\mathbf{M}_{\mathrm{C}}^{-1}\mathbf{h}_{\mathrm{C}}\Delta t + (\mathbf{I}+\boldsymbol{\epsilon}_3)(\mathbf{W}_{3,B}\mathbf{u}_{\mathrm{B}}^{S} + \mathbf{W}_{3,C}\mathbf{u}_{\mathrm{C}}^{S} + \boldsymbol{\chi}_3) \\ \mathbf{c}_4 &= \mathbf{W}_{4,B}^{\top}\mathbf{M}_{\mathrm{B}}^{-1}\mathbf{h}_{\mathrm{B}}\Delta t + (\mathbf{I}+\boldsymbol{\epsilon}_4)(\mathbf{W}_{4,B}\mathbf{u}_{\mathrm{B}}^{S} + \boldsymbol{\chi}_4). \end{aligned}$$

The terms involved in the subterms $\mathbf{c}_i$ have again a very logic structure which is usefull for fast assembling on the computer.

5. GPU Implementations

The main contribution of this thesis, namely the integration of certain numerical methods for the simulation of large mechanical systems, is presented in this chapter. The chapter starts off with an introduction and some analysis about parallel implementations in the context of simulating mechanical systems on the computer. Certain GPU implementations of the JOR and SOR Prox method, which have been introduced in section 5.2 and 5.3, are then explained in section 4.6. A selection of several performance tests are summarized in chapter 6. This chapter presents source code which makes extensive use of the programming languages C/C++ and CUDA C. It is left to the reader to acquire this knowledge in order to successfully follow this work.

In this chapter, all parallel implementations are discussed only for contacts which comprise all of an unilateral force law together with spatial Coulomb friction (see (4.24). All parallel implementations are generic and can be easily extended for other contact laws such as using the Coulomb-Contensou friction model, instead of the Coulomb friction model for example.

5.1. Parallel Computing for Multibody Simulations

Several comprehensive software frameworks have been developed to simulate rigid bodies. The most important achievement is the rigid body engine *pe*, initiated in 2006 at the university of Erlangen (FAU). This framework is able to cope with hundred thousands even millions of rigid bodies. It uses MPI for the communication on distributed systems. The possibilities of this framework, as can be seen in various simulations, are incredible. Unfortunately, hardly any information could be found about the computational efforts of these massive rigid body simulations. The *pe* engine uses MPI parallelization and is able to solve millions of contacts with different solvers such as Lemke's LCP algorithm, Projected Gauss-Seidel algorithm or the Conjugate Projected Gradient algorithm. More information can be found in [9, 10].

5.2. The JOR Prox Iteration

The JOR Prox iteration is given as:

$$\boldsymbol{\Lambda}^{k+1} = \mathbf{prox}\left((\mathbf{I} - \mathbf{R}^{-1}\mathbf{G})\boldsymbol{\Lambda}^{k} - \mathbf{R}^{-1}\mathbf{c}\right) \tag{5.1}$$

$$= \mathbf{prox}\left(\mathbf{T}\boldsymbol{\Lambda}^{k} + \mathbf{d}\right) \tag{5.2}$$

$$\left|^{k+1}\right. = \mathbf{prox}\left(\square \cdot \left|^{k}\right. + \left|\right.\right). \tag{5.3}$$

It can be seen that in each iteration one matrix multiplication with $\mathbf{T} = (\mathbf{I} - \mathbf{R}^{-1}\mathbf{G})$ and an addition of a vector $\mathbf{d} = -\mathbf{R}^{-1}\mathbf{c}$ is needed. It is assumed that $\mathbf{T}, \mathbf{d}$ are computed before the iteration starts. During the iteration, 2 arrays (C language) of data $\boldsymbol{\Lambda}^{old}, \boldsymbol{\Lambda}^{new}$ are needed; one for $\boldsymbol{\Lambda}^{k}$ and one for $\boldsymbol{\Lambda}^{k+1}$. Each array points to one of the memory spaces D_1, D_2. The iteration is initialized with $\boldsymbol{\Lambda}^{old}$ pointing to memory space D_1 and $\boldsymbol{\Lambda}^{new}$ pointing to the memory space D_2. Memory space D_1, respectively $\boldsymbol{\Lambda}^{old}$, is filled with initial values $\boldsymbol{\Lambda}^{0}$. The pseudo procedure on the host CPU is now presented in algorithm 5.2.1. The variable $nIter_{max}$ defines how many iteration loops are made in total.

Algorithm 5.2.1 Jor Prox Iteration

1: **Initialization:**
   - Compute: $\mathbf{T} \leftarrow (\mathbf{I} - \mathbf{R}^{-1}\mathbf{G})$ and $\mathbf{d} \leftarrow -\mathbf{R}^{-1}\mathbf{c}$.
   - Setup pointers: $\boldsymbol{\Lambda}^{old} \rightarrow D_1$, $\boldsymbol{\Lambda}^{new} \rightarrow D_2$.
   - Fill initial values into D_1: $\boldsymbol{\Lambda}^{old} \leftarrow \boldsymbol{\Lambda}^{0}$.
2: - $(\boldsymbol{\Lambda}^{old}, \boldsymbol{\Lambda}^{new}) \leftarrow$ SwapPointers($\boldsymbol{\Lambda}^{old}$,$\boldsymbol{\Lambda}^{new}$)
3: **for** $nIter \leftarrow 0, nIter_{max}$ **do**
4: - $(\boldsymbol{\Lambda}^{old}, \boldsymbol{\Lambda}^{new}) \leftarrow$ SwapPointers($\boldsymbol{\Lambda}^{old}$,$\boldsymbol{\Lambda}^{new}$)
5: $\rightrightarrows \boldsymbol{\Lambda}^{new} \leftarrow \mathbf{prox}\left(\mathbf{T}\boldsymbol{\Lambda}^{old} + \mathbf{d}\right)$
6: $\rightrightarrows$ *abort* $\leftarrow$ CancelCriterion($\boldsymbol{\Lambda}^{old}$,$\boldsymbol{\Lambda}^{new}$)
7: **if** abort == true **then**
8: - **break**
9: **end if**
10: **end for**

Line 5 and 6 in algorithm 5.2.1 have the potential to be parallelized on the GPU. During this work, two really different parallelizations for these two lines have been compiled. The two methods are summarized below:

- **Method 1, Split:** For the matrix multiplication and addition $\mathbf{T}\boldsymbol{\Lambda}^{old} + \mathbf{d}$, the CUBLAS library call `cublas<S,D>gemv` has been re-engineered to get an own modular matrix-vector multiplication kernel (abbreviated by `matrixVectorMultGPU`). The second kernel call is an own written simple kernel which com-

putes the $\mathbf{prox}(\cdot)$ in parallel (abbreviated by `proxGPU`). The third simple kernel call is the termination criterion (denoted by `cancelCriterionGPU`).

1. `matrixVectorMultGPU`: $\mathbf{\Lambda}^{new} \leftarrow \mathbf{T}\mathbf{\Lambda}^{old} + \mathbf{d}$.
2. `proxGPU`: $\mathbf{\Lambda}^{new} \leftarrow \mathbf{prox}(\mathbf{\Lambda}^{new})$.
3. `cancelCriterionGPU`: $abort \leftarrow$ CancelCriterion$(\mathbf{\Lambda}^{old}, \mathbf{\Lambda}^{new})$.

- **Method 2, Combined:** One own written kernel call (denoted by `jorProxGPU`) is used to compute $\mathbf{prox}\left(\mathbf{T}\mathbf{\Lambda}^{old} + \mathbf{d}\right)$ in one call only. Afterwards the kernel `cancelCriterionGPU` is called.

  1. `jorProxGPU`: $\mathbf{\Lambda}^{old} \leftarrow \mathbf{prox}(\mathbf{T}\mathbf{\Lambda}^{old} + \mathbf{d})$.
  2. `cancelCriterionGPU`: $abort \leftarrow$ CancelCriterion$(\mathbf{\Lambda}^{old}, \mathbf{\Lambda}^{new})$.

In general, each of the above methods has its own parameters which define the individual kernel calls. The effects of these different settings are discussed in chapter 6.

The most time-critical part in both methods is the matrix-vector multiplication which is of order $\mathcal{O}(n^2)$ because it needs $2N^2 - N$ (see table 3.6) flops, where $N = nContacts \cdot ContactForceDim$. Therefore, most care in the context of performance is taken for the matrix-vector kernel `matrixVectorMultGPU`.

The next sections will explain the kernels `matrixVectorMultGPU`, `proxGPU`, `jorProxGPU` in detail.

5.2.1. Method Details: Split

Kernel Details: `matrixVectorMultGPU`

This kernel has been re-engineered from the CUBLAS kernel `cublas<S,D>gemv`. The question arises why this kernel needed to be rewritten. The main reason was that the CUBLAS kernel `cublas<S,D>gemv` in [25] computes the following structure:

$$\mathbf{y} = \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y} \tag{5.4}$$

This is not really useful in our context, because it is not desirable to write the result of the matrix-vector multiplication from $\mathbf{d}$ ($\mathbf{y}$ in (5.4) into $\mathbf{d}$ again. Our matrix-vector multiplication should rather have the following structure:

$$\mathbf{y} = \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{b}, \tag{5.5}$$

The kernel has been programmed very generic for any size of $\mathbf{A} \in \mathbb{R}^{m_A \times n_A}$, $\mathbf{x} \in \mathbb{R}^{n_A}$, $\mathbf{b}, \mathbf{y} \in \mathbb{R}^{m_A}$. It could have been further optimized to an only quadratic version, but

this has not been done due to the fact that the generic version can so be used in more cases such as in the SOR Prox implementation. Of course, this matrix-vector kernel in (5.5) is launched during algorithm 5.2.1 with the parameters:

$$\alpha = 1, \quad \beta = 1, \quad \mathbf{y} := \mathbf{\Lambda}^{new}, \quad \mathbf{x} := \mathbf{\Lambda}^{old}.$$

Another reason for an own implementation is that this kernel is also a part of kernel `jorProxGPU`. It is favoured to have an implementation which is not a black-box like the CUBLAS kernel call, where the programmer has no insight into the code and optimizations at all. Last but not least, understanding the source code of the CUBLAS 1.0 kernel call `cublas<S,D>gemv` helped the most in adapting the source code to our wishes.

The next explanations for kernel `matrixVectorMultGPU` and all variables are visualized all in figure 5.1. The kernel assumes that $\mathbf{A} \in \mathbb{R}^{m_A \times n_A}$ is stored in column-major order in global memory on the GPU. Vector $\mathbf{y}$ is virtually overlaid by an one-dimensional kernel grid spanning several blocks. Each thread block computes $blockDim$ values of $\mathbf{y}$ and each consists of tPB threads ($blockDim \leq tPB$). In figure 5.1, $\mathbf{y}$ is split into four blocks with each three threads ($blockDim = tPB = 3$). Each thread is responsible for the calculation of one element in $\mathbf{y}$ if $blockDim = tPB$, otherwise, there are idle threads which do not contribute to elements in $\mathbf{y}$. Thus, at the end, each thread needs to compute a scalar product of the corresponding row in $\mathbf{A}$ with $\mathbf{x}$. All threads start at the left vertical border of $\mathbf{A}$ to compute the scalar product. Each thread computes its scalar product in junks of $dotJunkSize = tPB \cdot xElementsPerThread$. In the example in the figure 5.1 this means: $xElementsPerThread = 2$. The second thread in the second block for example computes first a scalar result for $\mathbf{a}_{2,1} \cdot \mathbf{x}_1$ and then jumps to the second junk to compute a result for $\mathbf{a}_{2,2} \cdot \mathbf{x}_2$ and so on till the whole dot product has been computed. $\mathbf{a}_{i,j}$ denotes the j-th partial row vector in row i (relative to the block). Before each thread in a block starts its scalar product, all corresponding elements in $\mathbf{x}$ are written to a per-block shared memory space $\mathbf{x}_{sh}$ which consists of exactly $dotJunkSize$ elements. In the second block in the example, each thread out of three writes two elements from $\mathbf{x}_1$ (in global memory) to the shared memory vector $\mathbf{x}_{sh}$. After vector $\mathbf{x}_{sh}$ has been filled, all threads in one thread block are synchronized, to make sure that the global to shared memory transaction has properly finished. Afterwards, each thread in the block starts its scalar product. This is basically the whole procedure which happens in CUBLAS "automagically". The algorithm has been summarized in a pseudo kernel algorithm 5.2.2, which is very close to the actual CUDA C implementation in A.1. The numbers in brackets correspond to the line numbers in A.1.

It is important to mention some additional concepts used for this implementation because they are hard to see at first sight.

Algorithm 5.2.2 Matrix-Vector Multiplication, Summarized Kernel

1: **do** Shift kernel grid vertically over $\mathbf{y}$ to compute all row dot products. ▷ [29]
2: ▪ $dot = 0$.
3: **do** Shift thread blocks horizontally over $\mathbf{A}$ to compute all junks of one dot product. ▷ [37]
4: ▪ Sync all threads in a block!
5: ▪ Copy the sub-vector $\mathbf{x}_j$ into shared memory $\mathbf{x}_{sh}$. ▷ [46-81]
6: ▪ Sync all threads in a block again!
7: ▪ Each thread i computes one junk of one dot product: $dot = dot + \mathbf{a}_{i,j} \cdot \mathbf{x}_j$. ▷ [89-124]
8: **end** ▷ [126]
9: ▪ Each thread i computes $y_i = dot + b_i$. ▷ [129-136]
10: **end** ▷ [138]

- **Coalesced Access:** When all threads in the kernel grid compute junks of dot products, access to elements in $\mathbf{A}$ are coalesced. This can be seen in figure 5.1, thread 1 and 2 in block 2 compute both $a \times b$ at the same time. Both elements b in $\mathbf{a}_{1,1}$ and in $\mathbf{a}_{1,2}$ are located next to each other in memory, because $\mathbf{A}$ is stored in column major order. Therefore, during this instruction, all threads in the grid linearly access a memory segment in $\mathbf{A}$. Illustrated in figure 5.1, 12 threads would access linearly a part of the first column. This memory access is therefore coalesced, which is extremely beneficial. If matrix $\mathbf{A}$ is stored in row major order, this would not be the case anymore. The global memory transactions, reading vector $\mathbf{b}$ and writing the results back to vector $\mathbf{y}$, is coalesced as well.
- **Broadcast:** There is always a broadcast read transaction to $\mathbf{x}_{sh}$ during the junk dot products. In figure 5.1, thread 1 and 2 in block 2 both access the identical element in $\mathbf{x}_{sh}$, i.e., element $(a, b, \dots)$. Only one read access to the shared memory is made and the value is broadcasted to all threads. This has not been the case yet in earlier versions of the Compute Capability. This kernel exploits also no bank conflicts.
- **Register Blocking with Loop Unrolling:** In line 98 of the implementation in A.1, a technique called register blocking has been used. Register blocking is a technique to improve register reuse. This is achieved here by looping over `DOT_PROD_SEGMENT` elements of one junk dot product. It is done by first loading all stuff into registers `regA,regx` and then computing the dot product. The compiler directive `#pragma unroll` in line 101 expands the `for` loop `DOT_PROD_SEGMENT` times before it is compiled. Unrolling (to a certain amount) helps to decrease loop incrementing instructions (loop overhead). The compiler can pre-calculate offsets into the arrays and embed this into machine code.

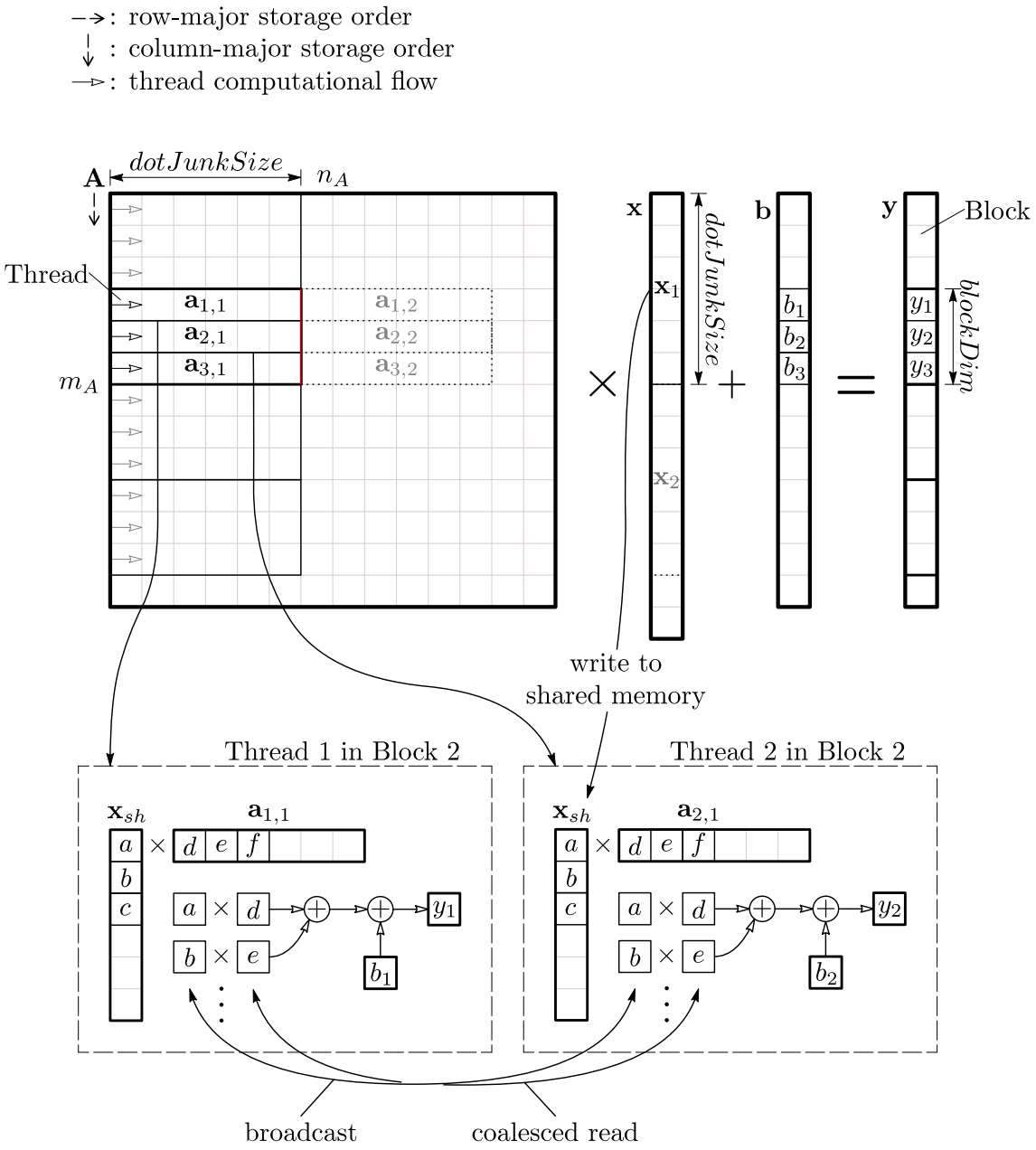


Figure 5.1.: The parallel concept for the matrix-vector multiplication. Matrix **A** is stored in column major order.

In general, the kernel is launched with a grid that fits the whole **y** vector. If this is not the case, the whole kernel grid has to be shifted again over **y** which can impair the coalesced read/write pattern. The number of threads per block has been chosen to 128. This gave the best performance results.

Kernel Details: `proxGPU`

This kernel computes the following:

$$\mathbf{y} = \mathbf{prox}(\mathbf{x}). \tag{5.6}$$

As our mechanical definitions in section 4.5 allows different contact laws for each contact, the question arises how to write the projection: $\mathbf{prox}(\cdot)$. The two feasible solutions are: Write a kernel which is totally generic and agrees with the definitions in 4.5 or write a specialized kernel which assumes a certain structure for all contacts. The performance of a generic kernel would degenerate completely and would not be highly efficient compared to a specialized one, mainly because a general approach comes hand-in-hand with more thread divergence problems and suboptimal memory access. If vector $\mathbf{x}$ would consist of contacts with different contact laws, which all need another scheme for the proximal function, an additional structure which specifies somehow the type of each contact is needed.

Another question is how to assign threads for a set S_i. Should a single thread compute the projected values for all normal cone inclusion in S_i or can some parallelism be exploited within a set S_i? As it turned out, it does not make much sense to assign several threads to a set of normal cone inclusions S_i for one contact. Assume 3 threads are assigned to a contact i with a normal inclusion set S_i as defined in 4.24 and $\mathcal{C}_T = \mathcal{C}_T(\mu\Lambda_N^{new})$, then the second and third thread eventually need to wait till the first thread has finished its normal projection. Afterwards, the second and third thread can do certain tasks in parallel for the projection on the disk with radius $\mu\Lambda_N^{new}$. In this case, there is need of a synchronization between all 3 threads.

Even if 2 threads are assigned to a set S_i, one for the normal direction and the second for the tangential direction, does not remove the need for synchronization between threads if $\mathcal{C}_T = \mathcal{C}_T(\mu\Lambda_N^{new})$. If the convex set $\mathcal{C}_T$ is independent of the new value Λ_N^{new} of the normal projection, $\mathcal{C}_T = \mathcal{C}_T(\mu\Lambda_N^{old})$, no synchronization is needed between the 2 threads. However, both threads execute completely different instructions and thread divergence happens inevitably if both threads are in the same warp. In this case, the instructions of both threads are executed sequentially. One might be tempted to assign all threads for one set S_i such that each one is located in a different warp. This, on the other hand, leads quickly to inconvenient thread-distribution patterns which worsens memory transactions.

In consideration of all this reasons, the first approach was to simply assign 1 thread to a set S_i which handles all inclusions. Anyway, the programmer should make sure that one thread has enough instructions to work on, otherwise there would be a lot of overhead, mainly only memory read and write which will decrease the performance. The implementation for the kernel `proxGPU` is a specialized one, assuming all contacts are defined as in 4.24. Of course it can be extended easily to other inclusion sets S_i.

Because all threads execute the same instructions at each clock cycle, there will not be any thread divergence. However, assume that vector $\mathbf{x}$ looks as follows:

$$\mathbf{x} = (n_1, \mathbf{t}_1^\top, n_2, \mathbf{t}_2^\top, \dots)^\top. \tag{5.7}$$

This structure is of course not desirable for memory access, because each thread k loads first the value in normal direction n_k, then the two values in tangential direction $\mathbf{t}_k$. The resulting memory access cannot be coalesced due to the gaps, i.e., n_1 and n_2 will not be located next to each other in memory anymore. The solution to this would be to group the terms according to their directions:

$$\mathbf{x} = (n_1, n_2, \dots \mid \mathbf{t}_{1,(1)}, \mathbf{t}_{2,(1)}, \dots \mid \mathbf{t}_{1,(2)}, \mathbf{t}_{2,(2)}, \dots)^\top. \tag{5.8}$$

Now, the memory access can be coalesced and is better then in (5.7), but this layout has its caveats as well. The first reason is that for coalesced access, the memory needs to be aligned (e.g. aligned to 128 bytes, see [24]). The border in (5.7), where the first and second tangential directions (at $\mathbf{t}_{1,(1)}$ and $\mathbf{t}_{1,(2)}$) start, needs to be aligned in memory. This calls for some additional padding rows *pad* in $\mathbf{x}$ which then structures as

$$\mathbf{x} = (n_1, n_2, \dots, \leftarrow pad \rightarrow \mid \mathbf{t}_{1,(1)}, \mathbf{t}_{2,(1)}, \dots, \leftarrow pad \rightarrow \mid \mathbf{t}_{1,(2)}, \mathbf{t}_{2,(2)}, \dots)^\top \tag{5.9}$$

On the other hand, this structure leads to padding columns in matrix $\mathbf{G}$. There are a lot of possibilities for ordering the vector $\mathbf{\Lambda}$. In this work, the structure (5.7) is used, despite the fact that no coalesced access is exploited. This does not matter because the kernel `proxGPU`, as later results will show, will not be at all the limiting factor for the iteration.

The kernel presented in listing 5.1 does the following: In line 12, all indexes into $\mathbf{x}$ are calculated once per thread. Then in line 17, the normal direction is projected onto $\mathbb{R}_0^+$. In line 22, all values are first loaded into registers, the radius of the friction disk is calculated and the values are then projected onto this disc. Line 29 makes use of the fast transcendental function `rsqrt` which calculates $1/\sqrt{(\cdot)}$. This function is much faster then `sqrt` on NVIDIA GPU architectures (see [33]).

The whole `while` loop in line 10 around the actual kernel is only used to shift the kernel grid over the vector $\mathbf{x}$. Anyway the kernel is always launched with a grid size so that every contact has one thread assigned. Therefore $\dim(\mathbf{x})/3$ threads are launched. The grid is divided into blocks of 192 or 256 threads, which both perform almost equally.

Listing 5.1: The complete CUDA C source code for the `proxGPU` multiplication in the form $\mathbf{y} = \mathbf{prox}(\mathbf{x})$. This code can be found in `KernelsProx.cuh`.

```
1 template<typename PREC, typename TConvexSet>
2 __global__ void proxContactOrdered_1threads_kernel(Matrix<PREC> mu_dev, ...
      Matrix<PREC> proxTerm_dev){
```

```
3
4   STATIC_ASSERT2( (IsSame<TConvexSet,ConvexSets::RPlusAndDisk>::result) , ...
        ONLY_RPLUS_AND_DISK_CONVEX_SET_IMPLEMENTED_SO_FAR );
5
6   if(IsSame<TConvexSet,ConvexSets::RPlusAndDisk>::result){
7     int index_x = threadIdx.x + blockIdx.x * blockDim.x;
8     int stride_x = blockDim.x * gridDim.x;
9
10    while((index_x+1)*3 <= proxTerm_dev.M){
11
12      int idx_n = index_x*3;
13      int idx_t1 = index_x*3 + 1;
14      int idx_t2 = index_x*3 + 2;
15
16      //Normal direction
17      if(proxTerm_dev.data[idx_n] < 0){
18         proxTerm_dev.data[idx_n] = 0;
19      }
20
21      // Tangential direction
22      PREC lambda_T1 =  proxTerm_dev.data[idx_t1];
23      PREC lambda_T2 =  proxTerm_dev.data[idx_t2];
24      PREC radius    =  mu_dev.data[index_x]* proxTerm_dev.data[idx_n];
25      PREC absvalue  = lambda_T1*lambda_T1 + lambda_T2*lambda_T2;
26
27      if(absvalue > radius*radius){
28         if(IsSame<PREC,double>::result){
29           absvalue = radius * rsqrt(absvalue);
30         }
31         else{
32           absvalue = radius * rsqrtf(absvalue);
33         }
34         proxTerm_dev.data[idx_t1] = lambda_T1 *  absvalue;
35         proxTerm_dev.data[idx_t2] = lambda_T2 *  absvalue;
36      }
37
38      // Shift index
39      index_x += stride_x;
40    }
41  }
42 }
```

Kernel Details: `cancelCriterionGPU`

The kernel for the termination criterion in (4.49) is very simple and cannot really be optimized further then as given in listing 5.2. A kernel grid with as many threads as elements in vector $\mathbf{x}$ is launched. Each thread checks one row for convergence. If one element has not converged, the variable `convergedFlag_dev` is simply set to 0. The variable `convergedFlag_dev` is set to 1 before kernel launch. When the kernel has finished, `convergedFlag_dev` is copied to the host CPU and checked if it is still 1 for convergence or 0 if not converged.

Listing 5.2: The complete CUDA C source code for the `cancelCriterionGPU`. This code can be found in `KernelsProx.cuh`.

```
1 template<typename PREC>
2 __global__ void convergedEach_kernel(Matrix<PREC> x_new_dev, Matrix<PREC> ...
      x_olb_dev, bool * convergedFlag_dev, PREC absTOL, PREC relTOL){
3
4   // Calculate  indexes
5   int index_x = threadIdx.x + blockIdx.x * blockDim.x;
```

```
6   int stride_x = gridDim.x * blockDim.x;
7
8   while(index_x < x_olb_dev.M){
9     if(!checkConverged(x_new_dev.data[index_x], x_olb_dev.data[index_x] , ...
          absTOL, relTOL)){
10      *convergedFlag_dev = 0;
11    }
12    index_x+= stride_x;
13  }
14 }
15
16 __forceinline__ __device__ bool checkConverged(double x_new ,double x_old ...
      , double absTOL, double relTOL){
17   if(fabs(x_new - x_old) > (absTOL  +relTOL * fabs(x_old)) ){
18     return false;
19   }
20   return true;
21 }
```

5.2.2. Method Details: All-in-One

Kernel Details: `jorProxGPU`

The intention to combine `matrixVectorMultGPU` and `proxGPU` into one kernel `jorProxGPU`, was mainly to reduce the global synchronization overhead. This kernel is not shown here because there are no new tricks involved in it. As it turned out, there are several different difficulties which make this all-in-one kernel not more powerful then the separated ones. For a combination, the kernel grid dimensions need to be reconsidered first. To successfully project the values in the way described in section 5.2.1 after the matrix-vector multiplication, one needs a kernel block size with a multiple of the dimension of the contact law. If the set S_i as in 4.24 is assumed, the block size *blockDim* (see figure 5.1) for the combined kernel needs to be a multiple of 3. As an example, the kernel grid is split up in thread blocks of 126 threads each. Therefore in each block one has $126/3 = 42$ contacts which need to be projected. Additionally, the kernel block size should be a multiple of the warp size. The set K which fulfills both these requirements is given as the least common multiple lcm of 32 and 3:

$$\begin{aligned} K = \text{lcm}(32,3) &= \text{lcm}(2^5,3) \\ &= \{k \mid k = 2^l 3^j, \quad \forall l \geq 5, j \geq 1\} = \{96, 192, 384, 768, \dots\} \end{aligned} \tag{5.10}$$

Two options have been tested, a kernel grid with 128 threads and a grid with 192 threads per block. The combined kernel can use the same parameters as for `matrixVectorMultGPU`. Table 5.1 summarizes again the used parameters and the influence for each subroutine. Note that both *blockDim* and tPB are only for the second option both in set K. It is not really obvious at first sight, why variant 1 performs so much worse than option 2. This will be explained in chapter 6.

Option 1:	$blockDim = 126$ and $tPB = 128$ **matrix-vector multiplication:** 128 active threads for shared memory copy, 126 of 128 active threads for dot product. **prox computation:** $126/3 = 42$ of 128 active threads.
Option 2:	$blockDim = 192$ and $tPB = 192$ **matrix-vector multiplication:** 192 active threads for shared memory copy, 192 for dot product. **prox computation:** $192/3 = 64$ of 192 active threads.

Table 5.1.: The two options for `jorProxGPU` and the influence on subroutine: matrix-vector multiplication and prox function.

5.3. The SOR Prox Iteration

This implementation tries to find a certain parallelism of the equation:

$$\begin{aligned} \boldsymbol{\Lambda}^{k+1} &= \mathbf{prox}\left(-\mathbf{R}^{-1}\mathbf{L}\boldsymbol{\Lambda}^{k+1} + (\mathbf{I} - \mathbf{R}^{-1}(\mathbf{U}+\mathbf{D}))\boldsymbol{\Lambda}^{k} + \mathbf{d}\right) \\ &= \mathbf{prox}\left(\tilde{\mathbf{L}}\boldsymbol{\Lambda}^{k+1} + \tilde{\mathbf{U}}\boldsymbol{\Lambda}^{k} + \mathbf{d}\right) \\ \begin{vmatrix} \\ \\ \end{vmatrix}^{k+1} &= \mathbf{prox}\left(\begin{vmatrix} \diagdown & \\ & \diagdown \end{vmatrix} \cdot \begin{vmatrix} \\ \\ \end{vmatrix}^{k+1} + \begin{vmatrix} & \diagdown \\ & \\ \end{vmatrix} \cdot \begin{vmatrix} \\ \\ \end{vmatrix}^{k} + \begin{vmatrix} \\ \\ \end{vmatrix}\right). \end{aligned} \tag{5.11}$$

As can be seen by the symbolic representation, the SOR algorithm uses successively the new computed values in $\boldsymbol{\Lambda}^{k+1}$ on the right hand side. One way to solve this equation is to compute row by row, and always reuse the old values. This procedure is in fact implemented on a sequential CPU.

Exploiting certain parallelism from this structure is a real burden and as one can already guess, the possibilities are quite limited. However, there exist some sophisticated methods to get the most out of a SOR implementation. Our SOR implementation has been based on the work in [2]. The reader should take notice that matrix $\tilde{\mathbf{L}} + \tilde{\mathbf{U}}$ collected together yields again matrix $\mathbf{T}$. This means that any implementation can operate only on the elements in $\mathbf{T}$.

The workflow of the SOR Prox iterations on the host CPU is presented in algorithm 5.3.1. The variable definitions are the same as for the JOR Prox method in algorithm 5.2.1. The vector $\mathbf{t}$ in 5.3.1 will be explained in the next section. The main interest is certainly to parallelize line 5 in algorithm 5.3.1.

The work in [2] describes 3 main parallel strategies: a row-based, column-based and a block-based strategy. A block-based strategy works in blocks over matrix $\mathbf{T}$ till one iteration is computed. The procedure is demonstrated in figure 5.2(b).

Algorithm 5.3.1 Sor Prox Iteration

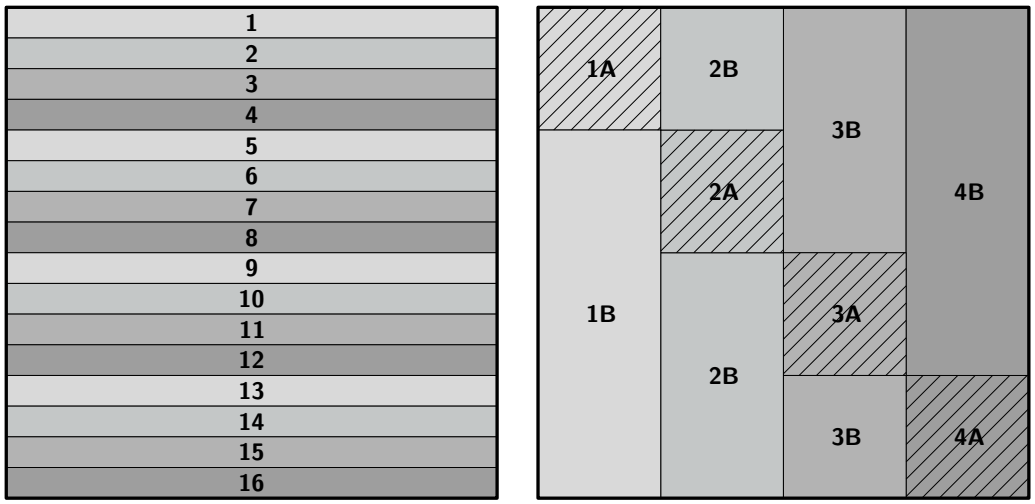
1: **Initialization:**
   - Compute: $\mathbf{T} \leftarrow (\mathbf{I} - \mathbf{R}^{-1}\mathbf{G})$ and $\mathbf{d} \leftarrow -\mathbf{R}^{-1}\mathbf{c}$.
   - Setup pointers: $\mathbf{\Lambda}^{old} \rightarrow D_1$, $\mathbf{\Lambda}^{new} \rightarrow D_2$.
   - Fill initial values into D_1: $\mathbf{\Lambda}^{old} \leftarrow \mathbf{\Lambda}^0$.
   - Initialize vector $\mathbf{t}$.
2: - $(\mathbf{\Lambda}^{old}, \mathbf{\Lambda}^{new}) \leftarrow$ SwapPointers($\mathbf{\Lambda}^{old}$,$\mathbf{\Lambda}^{new}$)
3: **for** $nIter \leftarrow 0, nIter_{max}$ **do**
4: - $(\mathbf{\Lambda}^{old}, \mathbf{\Lambda}^{new}) \leftarrow$ SwapPointers($\mathbf{\Lambda}^{old}$,$\mathbf{\Lambda}^{new}$)
5: $\rightrightarrows \mathbf{\Lambda}^{k+1} \leftarrow \mathbf{prox}\left(\tilde{\mathbf{L}}\mathbf{\Lambda}^{k+1} + \tilde{\mathbf{U}}\mathbf{\Lambda}^{k} + \mathbf{d}\right)$
6: $\rightrightarrows abort \leftarrow$ CancelCriterion($\mathbf{\Lambda}^{old}$,$\mathbf{\Lambda}^{new}$)
7: **if** abort == true **then**
8: - **break**
9: **end if**
10: **end for**

In this work, two methods of a block-based approach have been implemented, a full dependency SOR Prox (Full SOR Prox) and a SOR Prox with relaxed dependencies (Relaxed SOR Prox). In the following, these two methods, which parallelize line 6 in algorithm 5.3.1, are briefly summarized and explained in a further step.

- **Method 1, Full SOR Prox:** This iteration method is a correct parallel implementation of eq. (5.11). One iteration needs several calls of a tuple of two kernels A (`sorProxStepA`) and B (`sorProxStepB`) (see figure 5.2(b)). Kernel A redeems all dependencies for the block-diagonal part in $\mathbf{T}$. It includes a part where it computes the function $\mathbf{prox}(\cdot)$ for all values and a part where it propagates the new values in $\mathbf{\Lambda}^{k+1}$. Kernel B does only propagate new values $\mathbf{\Lambda}^{k+1}$ on the off-diagonal terms of $\mathbf{T}$. Kernel A and B together completely redeem all dependencies which are needed to comply with eq. (5.11).

- **Method 2, Relaxed SOR Prox:** This method works the same as the Full SOR Prox, except that kernel A (`sorProxRelaxedStepA`) does not redeem completely all its dependencies, thus its only a pure projection kernel for $\mathbf{prox}(\cdot)$. Kernel B (`sorProxRelaxedStepB`) stays the same and will propagate the new values. Thus, kernel A and B, after several calls for one iteration, will not have redeemed all dependencies correctly. This corresponds to the scenario, if one would evaluate d rows of eq. (5.11) at the same time and then use the new d values for the next d rows. Therefore, this is a mixture between a normal JOR and a SOR scheme.

The reason to split up into a small kernel A and a bigger kernel B is desirable and has two main reasons:



(a) Row-based (b) Block-based

Figure 5.2.: Two SOR strategies: Shown is a matrix $\mathbf{T} \in \mathbb{R}^{16\times16}$. The numbers indicate which part of the matrix is processed first. The row-based method is mainly implemented on a sequential CPU but can also be implemented in parallel by recursive reduction on the GPU. The block-based method consists of two kernel calls denoted by A and B. In this example, the block based method calls 4 times the kernels A and B but always on a different part of the matrix $\mathbf{T}$. Kernel A only operates on the diagonal, where kernel B fills the rest of the matrix.

- Because some sort of synchronization is needed to redeem the dependencies, it is desirable to use only synchronizations between threads in one block. Thread synchronization between several blocks is adverse because the need for atomic counters is inherently given, which gives bad performance. Therefore, moving all synchronizations into one kernel A with only one kernel block results in no need for atomic counters. A small kernel block is efficient. Launching a too big kernel block with a lot of synchronizations will lead to a performance loss due to the synchronization overhead. The size of this kernel block is determined experimentally.

- The second kernel B can therefore be used only to propagate the new values and does not need any synchronization between threads. Kernel B does a matrix-vector multiplication, which can be efficiently adapted from the kernel `matrixVectorMultGPU`.

5.3.1. Method Details: Full SOR Prox

Both kernels A and B are explained with the help of figure 5.3. The matrix $\mathbf{T}$, i.e.,

$$\mathbf{T} \in \mathbb{R}^{n \times n}, \quad n = nContacts \cdot ContactDim, \tag{5.12}$$

is split up in block diagonal matrices, on which only kernel A operates (Block A). The square block for kernel A contains $nContactsA$ contacts (vertically). Kernel B operates on the remaining $nContactDim \cdot (nContacts - nConctactsA)$ contacts (Block B). Block A and B are then shifted sequentially over $\mathbf{T}$. In figure 5.3(a), $\mathbf{T}$ consists of 10 contacts. Kernel A contains 4 contacts each. For each contact a set S_i, as in 4.24, is assumed. For one iteration, three A and B kernels are launched sequentially. The third A and B kernel will only calculate the remaining part of matrix $\mathbf{T}$.

To get both kernels working properly, an additional vector $\mathbf{t}$ is needed which has the same size as $\mathbf{\Lambda}^k$. This vector is used to continuously accumulate for each row the multiplication of new and old values with elements in $\mathbf{T}$. Supposed in a sequential SOR algorithm (without any projection, see figure 5.2(a)), computing the l-th row needs all $l-1$ new values and $n-l$ old values multiplied with the corresponding values in $\mathbf{T}$ which then gives the new value for $\mathbf{\Lambda}_{(l)}^{k+1}$ as

$$\mathbf{\Lambda}_{(l)}^{k+1} = \underbrace{\mathbf{T}_{(l,1:l-1)} \cdot \mathbf{\Lambda}_{(1:l-1)}^{k+1} + \mathbf{T}_{(l,l:n)} \cdot \mathbf{\Lambda}_{(l:n)}^{k}}_{\mathbf{t}_{(l)}} + \mathbf{d}_{(l)} \tag{5.13}$$

$$= \mathbf{t}_{(l)} + \mathbf{d}_{(l)}. \tag{5.14}$$

Vector $\mathbf{t}$ stores exactly the values as in e.q. (5.13) for each row l.

Kernel Details: `sorProxStepA`

The kernel grid for kernel A contains one kernel block, which contains itself one thread for each row in block A. The reason for only one kernel block is due to the need of several barrier synchronizations to maintain the dependencies. Thread synchronization can be done easily within a kernel block. In figure 5.3(a), a matrix $\mathbf{T} \in \mathbb{R}^{30 \times 30}$ is shown. This corresponds to 10 unilateral contacts with spatial Coulomb friction. For this example, the kernel block for kernel A would contain exactly 12 threads for each row in block A.

As shown in figure 5.3(b), each thread moves through the elements of 1 row in block A. If a thread is on an off-diagonal element l in $\mathbf{T}$, the l-th element of vector $\mathbf{t}$ is accumulated. If a thread l moves onto a diagonal element in $\mathbf{T}$, the new value $\mathbf{\Lambda}_{(l)}^{k+1}$ is computed similar to eq. (5.13), but with a projection included.

It is important to know that, if one wants to start the Full SOR Prox algorithm as in

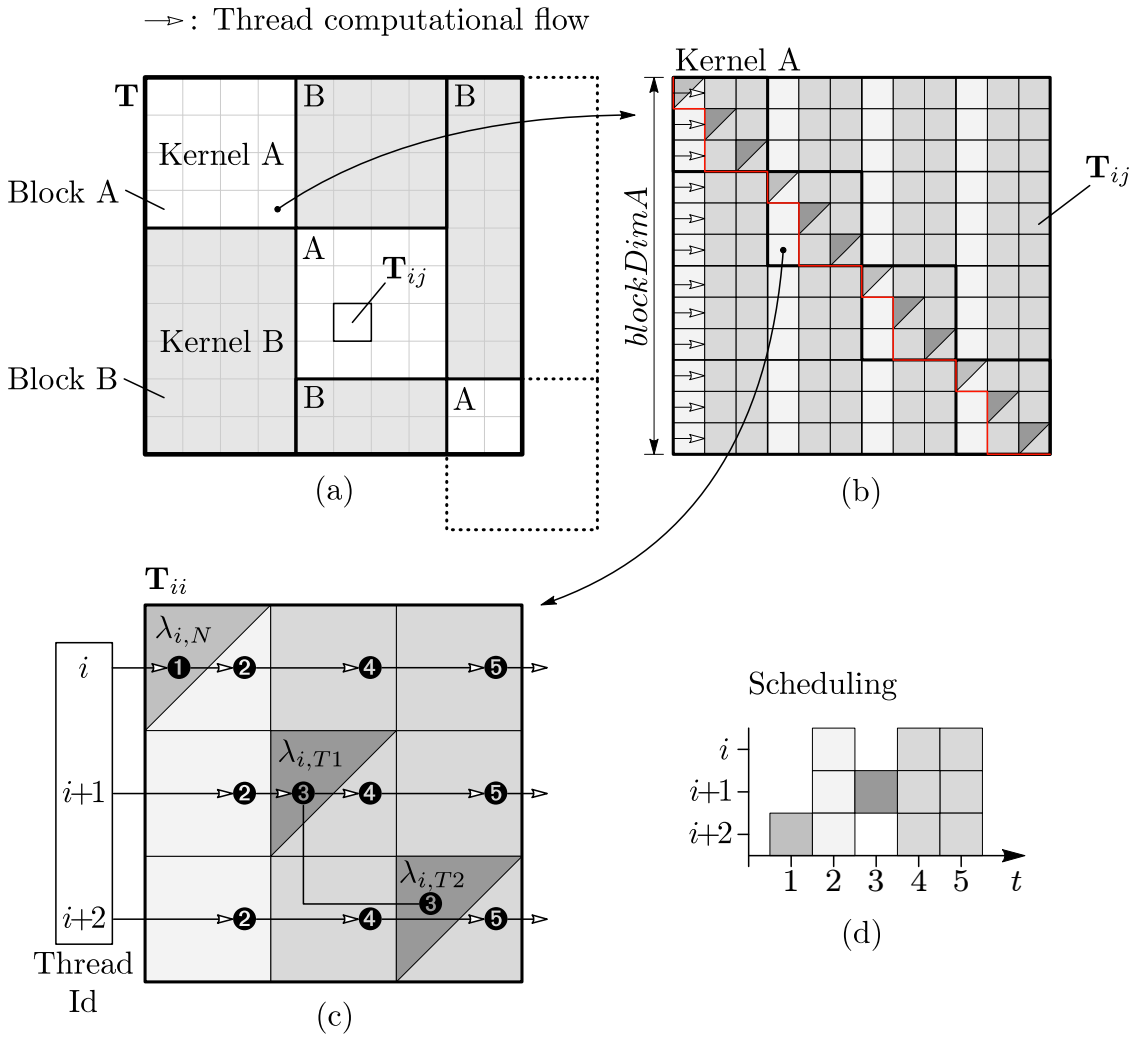


Figure 5.3.: Workflow for kernel `sorProxStepA`. Matrix $\mathbf{T}$ is overlaid by diagonal blocks for kernel A and off-diagonal blocks for kernel B. These blocks do not refer to a kernel grid (CUDA blocks and threads). Figure (a) only shows on which part each kernel operates. Furthermore, a conctact set S_i as in 4.24 is assumed. Figure (b) shows as an example the first block on which kernel A operates. Figure (c) and (d) visualize the scheduling of the threads, when they operate over one diagonal block $\mathbf{T}_{ii}$.

eq. (5.11) with a given initial $\mathbf{\Lambda}^0$, one needs to initialize (see line 1 in algorithm 5.3.1) the accumulation vector $\mathbf{t}$ by multiplication of $\mathbf{\Lambda}^0$ with a special upper triangular block matrix $\overline{\mathbf{U}}$ of $\mathbf{T}$:

$$\mathbf{t} = \overline{\mathbf{U}} \cdot \mathbf{\Lambda}^0. \tag{5.15}$$

A diagonal part of the pattern for this special matrix $\overline{\mathbf{U}}$ is shown as a red line in figure 5.3(b). How this line can be drawn through matrix $\mathbf{T}$ eventually depends on the used contact sets S_i. Of course, if one sets $\mathbf{\Lambda}^0 = \mathbf{0}$, then $\mathbf{t} = \mathbf{0}$*too*.

The workflow of this kernel is demonstrated with an example in figure 5.3(b),(c). Assuming that each thread on each row is now at the start of the 4-th column of $\mathbf{T}$, meaning that each thread has already passed $nContactDim = 3$ values (1 contact). The fourth thread ($i = 4$) computes now the projected value in the normal direction as:

$$\mathbf{\Lambda}_{(4)}^{k+1} = -\text{prox}_{\mathbb{R}^-}(-(\mathbf{t}_{(4)} + \mathbf{d}_{(4)})), \tag{5.16}$$

assuming that $\mathbf{t}_{(4)}$ contains the values as in eq. (5.13). After this, the 4-th thread needs to reset its value in $\mathbf{t}$, $\mathbf{t}_{(4)} = 0$. Each thread on a diagonal element always resets their t values to zero, to start the accumulation again for the next iteration. At first sight this behavior does not seem to produce a correct SOR scheme. The subtlety lies in the initialization of $\mathbf{t}$ before any iteration is done! If one initializes $\mathbf{t}$ as described in eq. (5.15), the elements in $\mathbf{t}$ will have the right accumulation in the first iteration if one thread moves onto a diagonal element in $\mathbf{T}$. Continuing in this fashion will always make sure that $\mathbf{t}$ contains the right accumulation. All the other 9 threads wait till the 4-th thread has finished the normal direction. This is step ❶ in figure 5.3(c).

In step ❷, all threads update their t values. Thus, for all threads $l = \{1, 2, 3, \dots 12\}$:

$$\mathbf{t}_{(l)} = \mathbf{t}_{(l)} + \mathbf{T}_{(l,4)} \cdot \mathbf{\Lambda}_{(4)}^{k+1}. \tag{5.17}$$

Step ❸ is special, because the 5-th thread ($i + 1 = 5$) now computes the projection onto the friction cone. For this, the next two values $\mathbf{t}_{(5)} + \mathbf{d}_{(5)}$, $\mathbf{t}_{(6)} + \mathbf{d}_{(6)}$ as well as the already computed normal direction $\mathbf{\Lambda}_{(4)}^{k+1}$ are needed. All other threads wait for the 5-th thread to complete. After the new values $\mathbf{\Lambda}_{(5)}^{k+1}$ and $\mathbf{\Lambda}_{(6)}^{k+1}$ have been computed, $\mathbf{t}_{(5)}$ and $\mathbf{t}_{(6)}$ are reset to zero as before.

Step ❹ and ❺ are update steps on the columns 5 and 6. Thus for all threads $l = \{1, 2, 3, \dots 12\}$:

$$\mathbf{t}_{(l)} = \mathbf{t}_{(l)} + \mathbf{T}_{(l,5)} \cdot \mathbf{\Lambda}_{(5)}^{k+1} \tag{5.18}$$

$$\mathbf{t}_{(l)} = \mathbf{t}_{(l)} + \mathbf{T}_{(l,6)} \cdot \mathbf{\Lambda}_{(6)}^{k+1}. \tag{5.19}$$

After step ❺, all threads continue their work with a new contact.

The summary for kernel A is given in algorithm 5.3.2. All numbers in brackets will refer to the source code attached in appendix A.2.

Option 1:	$blockDimA = threadsPerBlockA = 92 \rightarrow nContactsA = 32$
Option 2:	$blockDimA = threadsPerBlockA = 196 \rightarrow nContactsA = 64$

Table 5.2.: The two options for kernel `sorProxKernelA`.

As already mentioned, the kernel grid contains one kernel block. The number of threads is equal to the dimension of the square block A ($blockDimA$). The number of threads should be a multiple of the warp size and $blockDimA$ should be a multiple of $nContactDim$. With a set S_i as in 4.24, the number of threads for this one kernel block results in a set K as in 5.10. Table 5.2 shows the two tested options.

Kernel Details: `sorProxStepB`

This kernel does not need any further explanations. It simply does a propagation of the new values calculated by kernel A. It multiplies the block B (see figure 5.3) with all the new values from kernel A in $\mathbf{\Lambda}^{k+1}$ to update the accumulation vector $\mathbf{t}$ correctly. It is an adapted version of kernel `matrixVectorMultGPU`. Adapted in the sense that the kernel needs to know where to jump the diagonal block A to properly continue the multiplication.

5.3.2. Method Details: Relaxed SOR Prox

The relaxed version of the Full SOR Prox implementation works almost the same as shown in figure 5.3. Figure 5.4 shows the procedure of the two kernels `sorProxRelaxedStepA` and `sorProxRelaxedStepB`. The difference is that the whole propagation mechanism in kernel `sorProxStepA` of the Full SOR Prox method is now moved into the kernel `sorProxRelaxedStepB`. This results in a block B which fills the whole vertical dimension in $\mathbf{T}$. Kernel A reduces to a simple projection kernel similar to `proxGPU`. Additionally, it sets the part of the accumulation vector $\mathbf{t}$ to zero. Figure 5.4 shows a kernel A which projects 4 contacts ($blockDimA = 12$) at the same time. Kernel B will afterwards propagate these 12 new values by a matrix-vector multiplication. The red polygon in figure 5.4, denotes the special diagonal matrix $\overline{\mathbf{U}}$ to initialize vector $\mathbf{t}$ correctly.

5.4. Sparse Matrix Considerations

Considering a sparse matrix implementation for a SOR or JOR Prox method might be beneficial, because matrix $\mathbf{T}$ is in most cases highly sparse. Under the vast amount of possible implementations for a matrix-vector multiplication and several different

Algorithm 5.3.2 Sor Prox Step A, Summarized Kernel

1: ▪ Compute the thread Id, the actual size of this diagonal block, allocate shared memory $\mathbf{t}_{sh}$ for the part in $\mathbf{t}$ and also for the partial results $\mathbf{\Lambda}_{sh}$ in $\mathbf{\Lambda}^{k+1}$. ▷ [10-13]
2: ▪ Write the part of vector $\mathbf{t}$ into shared memory. ▷ [20-22]
3: ▪ Write the value from $\mathbf{d}$, which is used by this thread into a register. ▷ [24-30]
4: **do** Shift threads over the columns in block A. ▷ [37]
Step size: *nContactDim*.
5: ▪ **Diagonal Finish (Normal):** Select thread on the diagonal element, and project the normal direction and write the result back to shared memory $\mathbf{\Lambda}_{sh}$. Reset the value in $\mathbf{t}_{sh}$. ▷ [42-52]
6: ▪ **Synchronization Barrier:** All other threads fall into this barrier and wait till the normal direction is finished. ▷ [55]
7: ▪ **Propagate:** All threads update vector $\mathbf{t}_{sh}$, with the new value. ▷ [58-60]
8: ▪ **Synchronization Barrier:** The next thread on the diagonal may not compute the prox till the update of $\mathbf{t}_{sh}$ is finished for all threads. ▷ [62]
9: ▪ Move to next column. ▷ [66]
10: ▪ **Diagonal Finish (Tangential):** Select thread on the diagonal element, and compute the projection onto the friction cone with the 2 tangential directions and write back the result to $\mathbf{\Lambda}_{sh}$. Reset the two t values in $\mathbf{t}_{sh}$. ▷ [67-93]
11: ▪ **Synchronization Barrier:** All other threads fall into this barrier and wait till the tangential directions are finished. ▷ [95]
12: ▪ **Propagate:** All threads update vector $\mathbf{t}_{sh}$ with the 2 new values. ▷ [98-103]
13: ▪ Move to next column. ▷ [107-109]
14: **end** ▷ [110]
15: ▪ Write the partial vector $\mathbf{t}_{sh}$ back to vector $\mathbf{t}$ in global memory. Also write back the partial results $\mathbf{\Lambda}_{sh}$ to vector $\mathbf{\Lambda}^{k+1}$ in global memory. ▷ [113-116]

sparse matrix representations, the so far easiest and most promising approaches are briefly outlined in the following.

5.4.1. Ideas for a Sparse Relaxed SOR Prox

A Sparse Relaxed SOR Prox algorithm can be realized with a sparse matrix $\mathbf{T}$ in a Compressed Sparse Row format (CSR). Information about different sparse matrix formats can be found in [1, 31]. An example is visualized in figure 5.5. Matrix $\mathbf{T}$ is stored in a block CSR format which means that all $\mathbf{T}_{ij}$ are stored in row-major order.

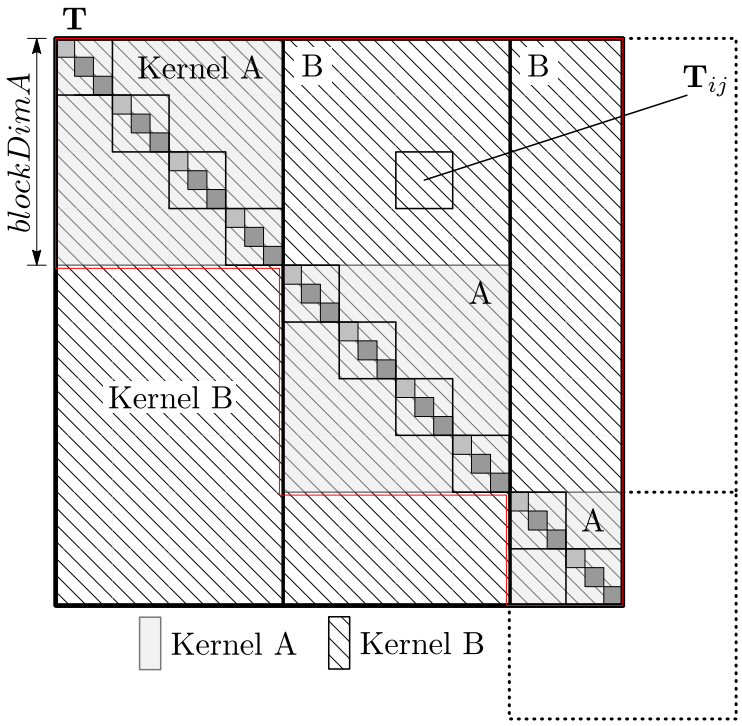


Figure 5.4.: Visualization of kernel `sorProxRelaxedStepA` and `sorProxRelaxedStepB`. Compared to figure 5.3, kernel B computes now the propagation of the new values of kernel A. Therefore Kernel A collapses to a simple vector kernel which only computes the projected values using parts of $\mathbf{t}$ and $\mathbf{d}$ and afterwards resets the part of the accumulation vector $\mathbf{t}$ to zero.

Corresponding to the numbers of matrices $\mathbf{T}_{ij}$ in figure 5.5, all blocks $\mathbf{T}_{ij}$ would be stored continuously after each other in a memory array `csrT`. Additionally, an array of column indices `csrColIndT` for each block and a compressed array `csrRowPtrT`, where the indices point at the start of a new block-row in $\mathbf{T}$, is needed, i.e.:

$$\begin{aligned}
\texttt{csrT} &= [\ \mathbf{0},\mathbf{1},\mathbf{2},\mathbf{3} \mid \mathbf{4},\mathbf{5} \mid \mathbf{6},\mathbf{7} \mid \mathbf{8},\mathbf{9} \mid \mathbf{10},\mathbf{11},\mathbf{12} \mid \mathbf{13},\mathbf{14} \mid \ldots \\
&\qquad \mathbf{15},\mathbf{16},\mathbf{17} \mid \mathbf{18},\mathbf{19},\mathbf{20} \mid \mathbf{21},\mathbf{22} \mid \mathbf{23},\mathbf{24},\mathbf{25}\ ] \\
\texttt{csrColIndT} &= [\ 0,3,7,9 \mid 1,5 \mid 2,6 \mid 0,3 \mid 4,7,9 \mid 1,5 \mid \ldots \\
&\qquad 2,6,8 \mid 0,4,7 \mid 6,8 \mid 0,4,8\ ] \\
\texttt{csrRowPtrT} &= [\ 0,4,6,8,10,13,15,18,21,23,26\ ].
\end{aligned} \tag{5.20}$$

For the following explanations each array is zero-indexed. The blocks $\mathbf{T}_{ij}$ $(\mathbf{0}-\mathbf{25})$ would then again be stored in either row-major or column-major order. A Sparse Relaxed SOR Prox algorithm consists now of an sparse matrix-vector multiplication for the kernel B. Kernel A stays the same. It is necessary to store for each multiplication of kernel B a different compressed row array `csrRowPtrT`. Corresponding to figure

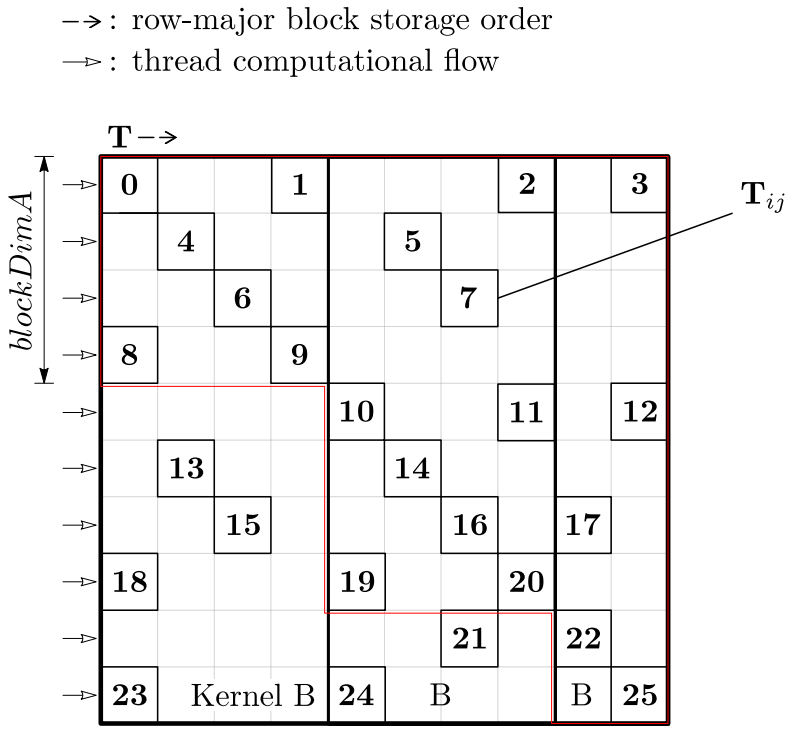


Figure 5.5.: The sparse matrix-vector multiplication for kernel B of the Sparse SOR Prox algorithm.

5.5, this sums up to three row pointer arrays as

$$\begin{aligned} \texttt{csrRowPtrT1} &= [\ 0, 4, 6, 8, 13, 13, 15, 18, 23, 23, 26\ ], \\ \texttt{csrRowPtrT2} &= [\ 2, 5, 7, 10, 10, 14, 16, 19, 21, 24, 26\ ], \\ \texttt{csrRowPtrT3} &= [\ 3, 12, 12, 12, 12, 17, 17, 22, 22, 25, 26\ ]. \end{aligned} \tag{5.21}$$

One thread, or possibly a collection of threads, ideally not in the same warp, are linearly assigned to each block-row in 5.5. To simplify explanations, only one thread is assumed per block-row. The 3 arrays in (5.21) are needed to start the small matrix-vector multiplication of the sub-blocks for each thread at the right block $\mathbf{T}_{ij}$. How many sub-blocks a thread needs to calculate is given by the indices in `csrRowPtrT`. As an example, lets look at the first kernel launch of kernel B: The thread which handles the k-th block-row in figure 5.5, computes in the best case $b = (\texttt{csrRowPtrT1}(k) - \texttt{csrRowPtrT1}(k-1))$ sub-blocks starting from the block $\texttt{csrT(csrRowPtrT1}(k-1))$. The k-th thread continues multiplying the sub-blocks (maximum b blocks) till its column index is out of range, e.g. for the first kernel call: the column index is smaller than $blockDimA$.

5.4.2. Ideas for a Sparse Full SOR Prox

Another idea would be to take representation in (5.20) and simply sweep all threads from left to right over the sparse matrix. Each thread (or a collection of threads) does the matrix-vector multiplication of one block-row. Together with an atomic counter which acts globally on the GPU, each thread checks before the matrix-vector multiplication of each off-diagonal sub-block $\mathbf{T}_{ij}$ if the diagonal $\mathbf{T}_{ii}$ block has already been processed. This can be achieved by checking an atomic counter which acts globally and counts the number of processed diagonal blocks. If a thread jumps onto a diagonal block, it computes the projected values similar to figure 5.3(b) and increments the atomic counter by one.

6. Performance Tests and Results

6.1. Raw Performance Tests

The most interesting part of course is how all these methods described in chapter 5 perform against each other. Performance tests have been built around algorithms in 5.2.1 and 5.3.1 by adding timing functions and various data collection code for the different methods.

Performance measures for the two methods of the JOR implementation, the Full SOR Prox as well as the Relaxed SOR Prox have been made. A framework in C++ has been written only to provide a generic test bench to run several performance tests on each method. Listing 6.1 for example shows the C++ source code of how to run a JOR Prox performance test. This performance test measures all performance metrics for a problem size in the range from $2000 - 4000$ contacts, where the step size is 20 contacts per run. Such a test outputs a text file with measurements per number of contacts such as:

- number of FLOPs,
- GFLOPS,
- memory bandwith [bytes/s],
- average time to copy memory to and from the GPU [s],
- the average time for one iteration [s],
- number of iterations for trade-off between GPU and CPU,
- the speedup factor,
- tolerances (relative and absolute tolerance, ULP error) of the cross check and infinity and 'NaN' checks.

More information can be found in the source code directly. The number of iterations N for trade-off between GPU and CPU can be computed as:

$$N > \frac{((T_{mallocGPU}) + T_{cpyToGPU} + T_{cpyFromGPU} + (T_{freeGPU}))}{T_{iterCPU} - T_{iterGPU}}, \tag{6.1}$$

where $T_{cpyToGPU}$, resp. $T_{cpyFromGPU}$ is the time to copy all data to and from the GPU memory and $T_{iterCPU}$, resp. $T_{iterGPU}$ is the time for one iteration on the CPU,

resp. GPU. The allocation and deallocation time $T_{mallocGPU}, T_{freeGPU}$ for the used memory on the GPU have not been included so far in the formula above. The reason for this is that allocation of all data can be done efficiently by using for example a memory pool. The memory pool should be allocated once before the time-stepping starts. This way, in each time-step, it is not necessary anymore to allocate memory directly using `malloc`. Free memory can be retrieved directly by the memory pool. The memory pool can be initialized for example with the maximum problem size which can still be computed on the GPU. The maximum problem size for the GTX 580 with 1.5 GB of global memory lies at roughly 4300-4500 contacts for the (dense matrix, double precision) JOR and SOR Prox methods. A sparse implementation would increase this border dramatically to millions of contacts, depending on the sparsity pattern of $\mathbf{G}$. A memory pool is the scope of future work and has not been implemented so far.

To compare only the pure performance of one iteration of each variant, matrix $\mathbf{G}$ and vector $\mathbf{c}$ have been filled up with random values. For only testing the speed of each method, it was not necessary to build $\mathbf{G}$ with real data from a mechanical simulation. Anyway, $\mathbf{G}$ is set up randomly with a diagonal dominant structure. All performance tests can be run in either double precision (`double`) or single precision (`float`).

The next two sections will present the main results of various performance tests which have been run during this work. The next section shows various plots which have been obtained by using NVIDIA's GPU model GTX 580. All other plots for NVIDIA's GPU model Tesla C2050 can be found in appendix B.

6.1.1. Results: JOR Prox

Figure 6.1(a) - 6.3(a) show a partial set of the five most important variants which have been tested. Many more variants can be tested in the test framework similar to listing 6.1. Only the results for double precision are shown. The single precision results turned out similarly, more about this is said later in this chapter. Table 6.1 summarizes these five variants and shows also the settings used for each kernel which are directly linked to the source code. All descriptions in brackets ([…]) in table 6.1 correspond to one kernel, ordered from left to the right on the time line. For example: [cuBlas Multiplication][Addition][Prox] means that three kernels have been used. A CUBLAS kernel for the multiplication in the first kernel launch, a self-developed kernel for addition for the second kernel launch and a Prox kernel as described in section 5.2.1 for the third. Variant 3 and 4 in table 6.1 correspond to option 1 and 2 in table 5.1 of the all-in-one method in section 5.2.2. Variant 5 uses the best options for the kernels `matrixVectorMultGPU` and `proxGPU` of the split method in section 5.2.1.

Listing 6.1: C++ example for running a JOR Prox performance test.

```
typedef KernelTestMethod<
  KernelTestMethodSettings<false,true> , // Log to file = ... false, CrossCheck = Yes
  ProxTestVariant< // Kernel Test Variant: ProxTestVariant
   ProxSettings<
   double,    // The precision: double
   SeedGenerator::Fixed<5>, // How the test problem is ... built, Fixed seed with 5
   false,     // Wirte Matlab file? No
   10,        // How many prox iterations should be done: 10
   false,     // Should the cancelCriteria be used: No
   10,        // Check cancelCriteria each 10 iterations.
   true,      // Match the crossCheck exactly with the GPU ... implementation: Yes
   ProxPerformanceTestSettings<2000,20,4000,5>, ...
                    // Run test from 2000-4000 contacts in ... step of 20, for each 5 random runs!
   JorProxGPUVariantSettings<  4,
                               ConvexSets::RPlusAndDisk,
                               true> // Use the JOR Prox: ... Variant 4 with ConvexSet: ... RPlusAndDisk, align matrix T!
   >
  >
> MyJorTestSettings;

PerformanceTest<MyJorTestSettings> jorTest("test1");
jorTest.run();
```

One of the important main results is that all JOR Prox (and also SOR Prox) variants are *memory bound* or also called memory bandwidth limited. This means that the speed of a kernel is mainly limited by the effective memory bandwidth. The contrary would be a *compute bound* kernel which is limited by its effective compute bandwidth. Most commonly programmed kernels are likely to be memory bound. In this case, all further optimizations should address the exploitation of a much higher memory throughput by improving for example coalescing and minimizing bank conflicts in shared memory. The performance will not pay off if only the instruction throughput is increased. The limiting factor is the memory throughput and if it cannot be increased, the instruction throughput will be limited exactly by how fast the data is available for further floating point instructions. This is the case with our JOR and SOR variants. The matrix-vector multiplication is a memory bound kernel. This can be seen firstly

by simply comparing double precision to single precision, which is shown only for variant 5 in figure 6.1(a). There is a factor 2 between the iteration time for single and double precision. For NVIDIA's GTX 580, this leads to a first conclusion that this factor can only arise due to the memory access which is 2 times higher for double precision. Single precision floating point instructions are 8 times faster than double precision on the GTX 580. Thus, a factor of 8 would result for a purely compute bound kernel.

The question arises how to conclude that a kernel is memory or compute bound in general. This is not a trivial issue but can be worked out by the help of a profiler or by separately running the same kernel in 3 different versions: with only memory access, with only computational workload (math) and the full kernel. The latter method is quite hard to achieve because it needs serious compiler trickery to maintain the same compiled kernel code without losing too much by the compiler's optimization routine. A good tutorial was given at the International Supercomputing Conference from NVIDIA in [19]. Another way is to look at the instruction per byte ratio which is computed by the Compute Visual Profiler from NVIDIA. The ideal instruction per byte ratio is given by the hardware. For the Fermi architecture, it is roughly 4.5 instructions/byte. That means that a balanced kernel (neither compute bound, nor memory bound) needs approximately 4.5 instructions per one global memory byte access. The `matrixVectorMultGPU` kernel achieves $\approx$ 170 GB/s of the peak memory bandwidth of 192 GB/s and has an instruction/byte ratio of 0.88. This implies that the matrix-vector multiplication has a very good memory throughput and is memory bound because $0.88 < 4.5$. The contrary is a compute bound kernel with an instruction/byte ratio higher than 4.5. Another fact which strengthens the conclusion that the JOR and SOR methods are memory bound is that the performance of NVIDIA's GPU GTX 580 compared to NVIDIA's GPU Tesla C2050 scales almost exactly with a factor 1.3. This factor is not surprising because the GTX 580 has a 1.3 times greater memory bandwidth (192 GB/s) compared to the Tesla C2050 (144 GB/s). Figure 6.1 shows only the iteration time for JOR Prox variant 5 and Full SOR Prox variant 1 which was obtained with the Tesla C2050. All other plots for the Tesla C2050 can be found in appendix B.

In figure 6.1(a), almost all variants have the same performance, except variant 3 and 5 (not aligned). It is not possible to give clear account of the large jumps of the iteration time profile of variant 3. One plausible explanation might be that they arise due to the bad memory access pattern. This applies to the matrix-vector multiplication due to a block dimension of 126 elements in $\mathbf{T}$ with 128 threads per each block. In successive kernel blocks, the memory in $\mathbf{T}$ always starts at multiples of 126 elements of $\mathbf{T}$. The number 126 is neither a multiple of 8 or 16 nor 32, which would prevent bad memory access in global memory.

Another important observation was that when matrix $\mathbf{T}$ is uploaded to global memory

on the GPU in not-aligned form, the performance as in variant 5 (gray dotted) in figure 6.1 occurred. Note the big wriggles of variant 5 (not aligned, gray dotted) compared to variant 5 (blue). Not-aligned, for a matrix $\mathbf{T}$ in column-major order, means that $\mathbf{T}$ has no padding rows attached at the end of each column so that each successive starting element of a column is nicely aligned in global memory on the GPU to meet the requirements for coalescing. Of course, the starting address of $\mathbf{T}$ is always aligned in memory by the CUDA call `cudaMalloc`. CUDA provides a call `cudaMallocPitch` which can be used to ensure that each starting address of a column (column-major storage) or row (row-major storage) is aligned such that the requirements for coalescing are met.

It should be noted too, that our matrix multiplication has the same performance as variant 1 and 2 which use a CUBLAS 2.0 kernel. Variant 4 is only different in the location where the jumps occur and has similar performance.

Variant 5 is the most efficient implementation and achieves a speedup factor of $\approx$ 10-16 compared to a sequential implementation. The sequential implementation has been built with EIGEN 3.0 which uses SSE instructions. Therefore, our sequential algorithm is considered to be quite optimized. It does not exploit any parallelism by multi-threading. The question arises how a multi-threaded parallel CPU implementation would perform against our base sequential implementation. For the JOR Prox, a 6-threaded parallel BLAS routine (GOTOBlas) for the multiplication has been implemented (JOR Prox variant 7 in figure 6.2(a)). It only resulted in a constant performance boost of factor 1.5. The CPU BLAS multiplication will certainly be memory bound as well, so that this constant speedup factor can be expected.

The trade-off for all JOR variants is around 200 contacts. From this point, less then < 50 iterations pay off to use the GPU.

6.1.2. Results: SOR Prox

Figure 6.1(b) - 6.3(b) show the SOR Prox methods compared to the best JOR Prox variant, i.e., variant 5. The Full SOR Prox variant 1 and 2 correspond to the settings in table 5.2. The Relaxed SOR Prox is shown with 4 different relaxing sizes. For Relaxed SOR Prox variant 1-4: 2, 4, 16, 128 contacts are collected together.

The average iteration time for the Full SOR Prox variant 1 and 2 are mainly the same and are twice as slow as the JOR Prox variant 5. This is quite significant and does not yet lead to the conclusion that a JOR Prox variant is much better in a real mechanical simulation too. The Full SOR Prox algorithm converges in general faster than the JOR Prox. If the convergence behavior of the Full SOR Prox can outperform JOR Prox variant 5 is questionable and can only be tested with some real mechanical benchmarks in section 6.2. If it does, the Full SOR Prox needs to converge at least 2

times faster than the JOR Prox. The iteration time for the most relaxed SOR Prox variant 4 comes already quite close to JOR Prox variant 5. The Relaxed SOR Prox variant 4 is considered to be one of the most promising implementations because it provides almost the same speed as JOR Prox variant 5 and should be even better in its convergence behaviour as it is a mixture between the JOR Prox and Full SOR Prox method. Also notable is that as stricter the Relaxed SOR Prox becomes, the closer is the performance to the Full SOR Prox. The Relaxed SOR Prox variant 2, which collects 4 contacts together, yields the same performance as the Full SOR variant 1 and 2. Surprisingly, the Relaxed SOR Prox (fair blue) variant 1 collects 2 contacts together and produces so much overhead that it is already less efficient than the Full SOR Prox method. The results reveal that for a real mechanical benchmark only the JOR Prox variant 5, the Relaxed SOR Prox and the Full SOR Prox need to be tested against each other.

Figure 6.4 shows variant 1 split into its parts, step A (colored) and step B (gray). Step A has further been split into several parts (colored). All these timings for step A have been computed from the procedure of uncommenting parts in kernel A in the following sequence:

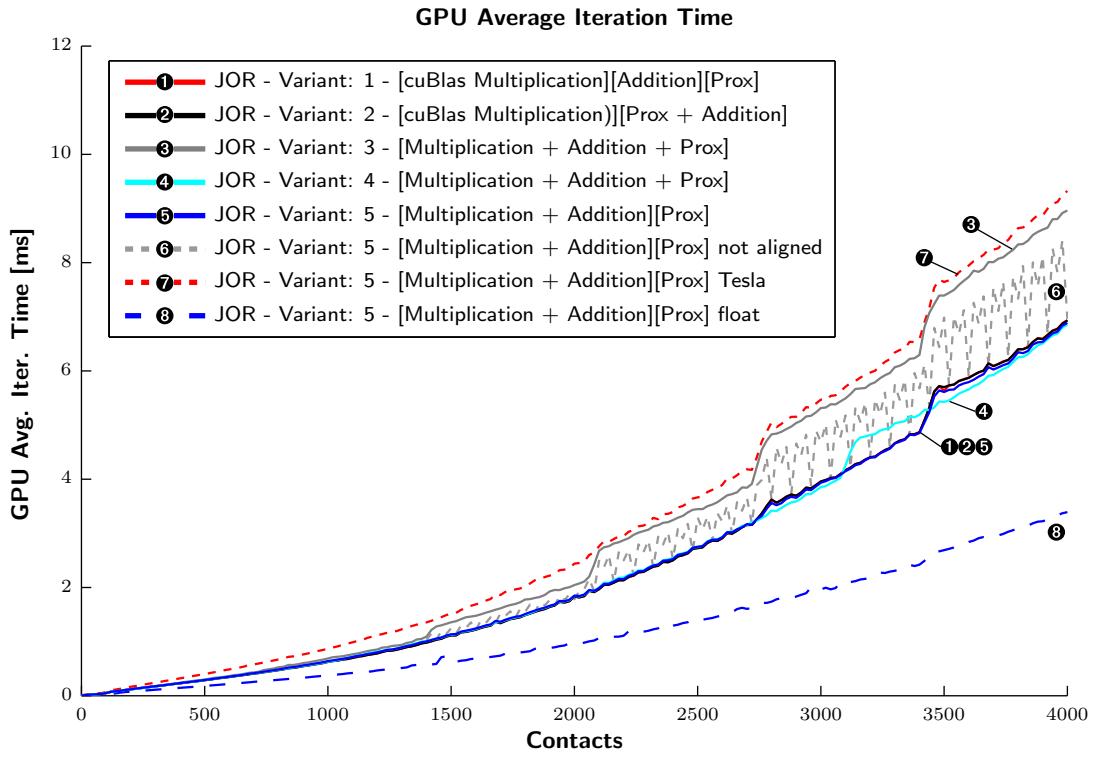
1. All propagations of new values, see ▷[58-60] and ▷[98-103] (blue).
2. The projection in normal direction, see ▷[44-48] (red).
3. The projection in tangential direction, see ▷[70-85] (fair blue).
4. All diagonal writes of new values and reset of elements in $\mathbf{t}$, see ▷[50-51] and ▷[88-91] (green).

Note that for the parts in A, no synchronization barriers have been uncommented because this would yield a biased result as the algorithm does not respect the dependencies anymore. All colored parts in A do indeed match up to the iteration time of kernel A, which is not shown in the figure. The results show obviously that step B is of quadratic order in the problem size. This makes sense as the size of block B of $\mathbf{T}$ varies quadratically as a function of the number of contacts. Step A and its parts scale linearly. The normal projection has almost no influence on the total time of part A as can be seen in figure 6.4(b). The tangential projection already takes 10-20% of the time. This is due to transcendental functions like `sqrt` and `rsqrt`. Over 25% is spent in the propagation in step A. The synchronization time (black line) in figure 6.4(b) shows the percentage of the iteration time which is needed to synchronize between threads. This has been measured by simply uncommenting only the synchronization barriers from step A. Therefore, the black line denotes that up to 4000 contacts approximately 20% of the total iteration time is needed for synchronization between threads.

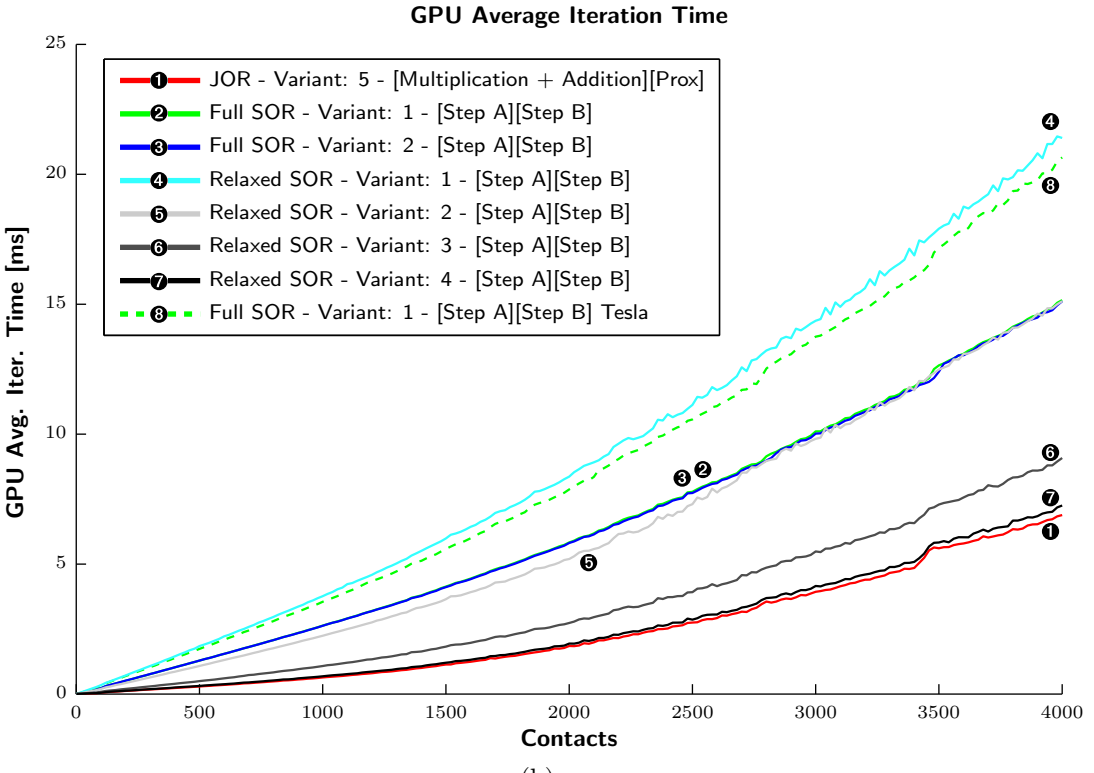
The speedup factors have been measured again versus a sequential equivalent CPU implementation for the Full SOR and Relaxed SOR Prox algorithms. The reader should note that in figure 6.2(b) the Relaxed SOR Prox variant 2 has not the same CPU implementation as the Full SOR variant 1 and 2. This justifies that they have not the same speedup, what figure 6.1(b) would actually suggest, because the iteration times lay close to each other.

JOR Prox Variants: Kernel Settings	
JOR Variant: 1	[gemv routine][ThreadsPerBlock : 192][ThreadsPerBlock : 128], Matrix T aligned: on
JOR Variant: 2	[gemv routine][ThreadsPerBlock : 256], Matrix T aligned: on
JOR Variant: 3	[ThreadsPerBlock: 128, BlockDim: 126, XElementsPerThread: 4, ProxPackageSize: 3, ProxPackages: 42, UnrollBlockDotProduct: 6], Matrix T aligned: on
JOR Variant: 4	[ThreadsPerBlock: 192, BlockDim: 192, XElementsPerThread: 4, ProxPackageSize: 3, ProxPackages: 64, UnrollBlockDotProduct: 6], Matrix T aligned: on
JOR Variant: 5	[ThreadsPerBlock : 128, BlockDim : 128, XElementsPerThread : 4, UnrollBlockDotProduct : 6][ThreadsPerBlock : 128], Matrix T aligned: on
SOR Prox Variants: Kernel Settings	
Full SOR Variant: 1	[ThdsPerBl A: 96, BlDim A: 96, ProxPkgSize A: 3, ProxPkgs A: 32][ThdsPerBl B: 128, BlDim B: 128, XElemsPerThd B: 1, UnrollBlDotProd B: 6], Matrix T aligned: on
Full SOR Variant: 2	[ThdsPerBl A: 192, BlDim A: 192, ProxPkgSize A: 3, ProxPkgs A: 64][ThdsPerBl B: 128, BlDim B: 128, XElemsPerThd B: 2, UnrollBlDotProd B: 6], Matrix T aligned: on
Relaxed SOR Variant: 1	[ThdsPerBl A: 128, BlDim A: 6, ProxPkgSize A: 3, ProxPkgs A: 2][ThdsPerBl B: 128, BlDim B: 128, XElemsPerThd B: 4, UnrollBlDotProd B: 6], Matrix T aligned: on
Relaxed SOR Variant: 2	[ThdsPerBl A: 128, BlDim A: 12, ProxPkgSize A: 3, ProxPkgs A: 4][ThdsPerBl B: 128, BlDim B: 128, XElemsPerThd B: 4, UnrollBlDotProd B: 6], Matrix T aligned: on
Relaxed SOR Variant: 3	[ThdsPerBl A: 128, BlDim A: 48, ProxPkgSize A: 3, ProxPkgs A: 16][ThdsPerBl B: 128, BlDim B: 128, XElemsPerThd B: 4, UnrollBlDotProd B: 6], Matrix T aligned: on
Relaxed SOR Variant: 4	[ThdsPerBl A: 128, BlDim A: 384, ProxPkgSize A: 3, ProxPkgs A: 128][ThdsPerBl B: 128, BlDim B: 128, XElemsPerThd B: 4, UnrollBlDotProd B: 6], Matrix T aligned: on

Table 6.1.: The kernel settings of the five JOR Prox, the two Full SOR Prox and the four Relaxed SOR Prox variants in figure 6.1 - 6.3. GPU model used: NVIDIA GTX 580.

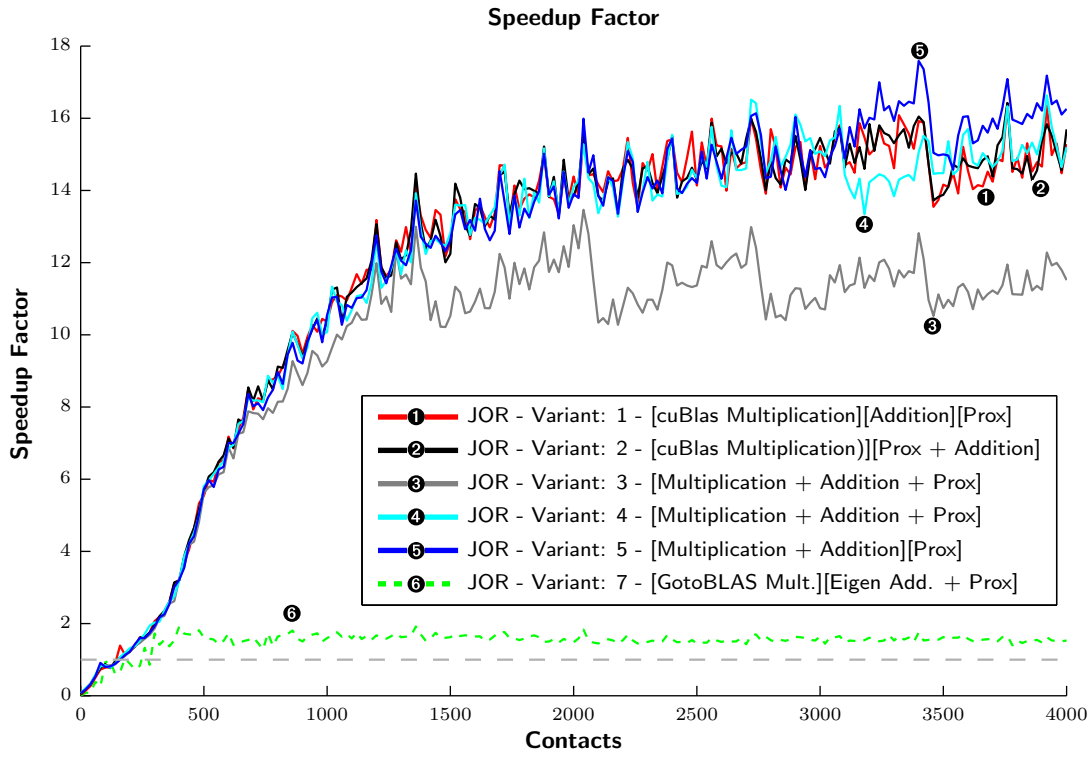


(a)

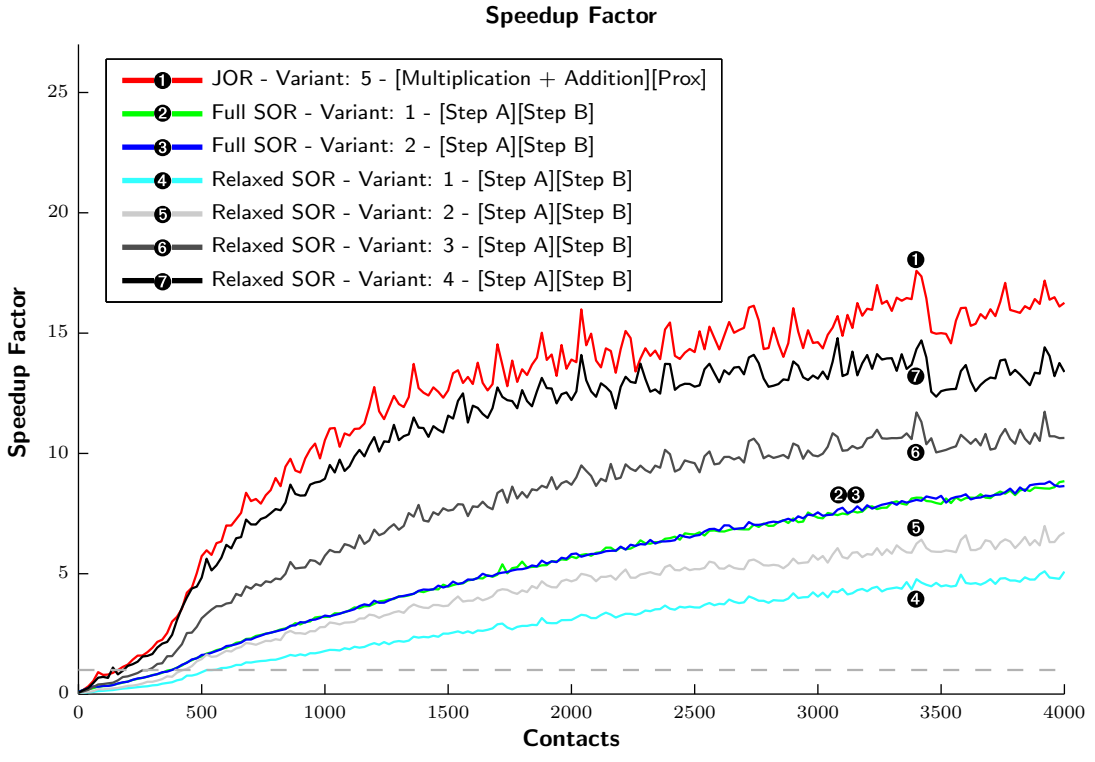


(b)

Figure 6.1.: The iteration time in [ms] of the JOR and SOR Prox variants. GPU model used: NVIDIA GTX 580.

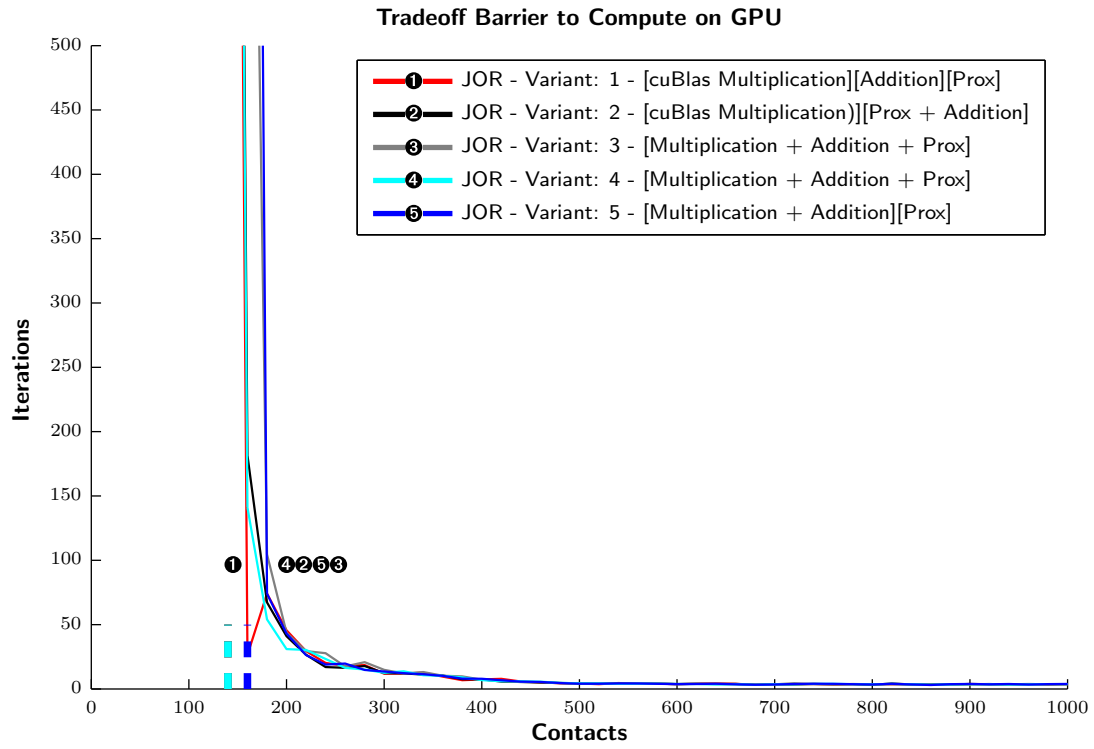


(a)

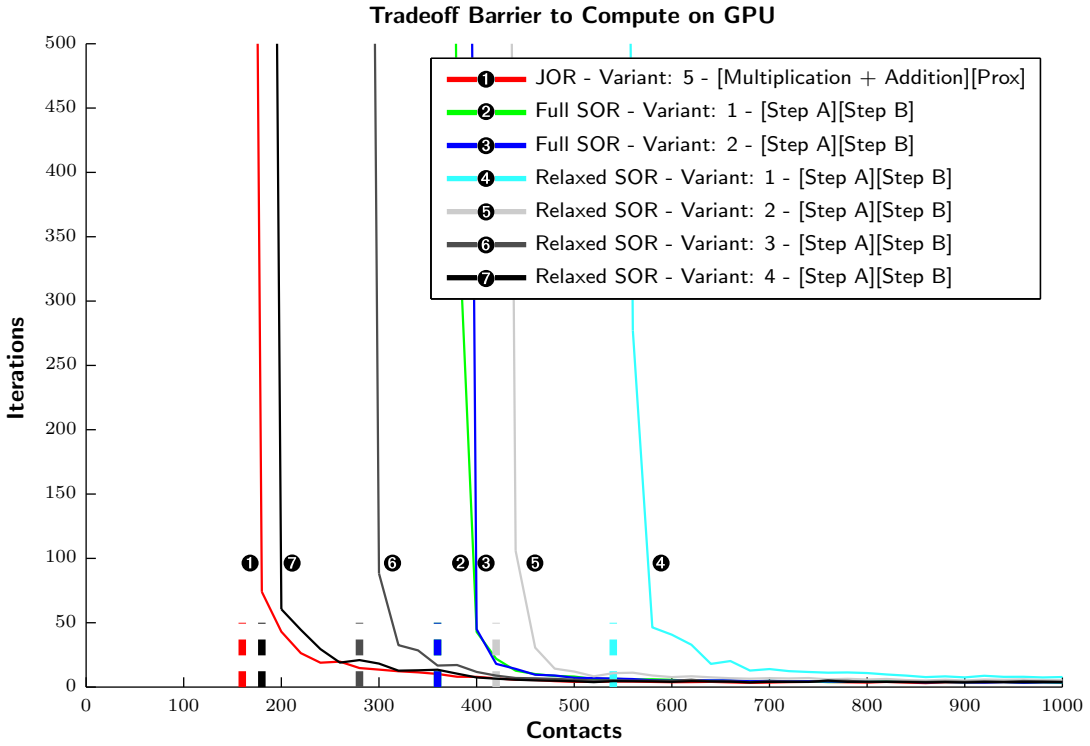


(b)

Figure 6.2.: The speedup factor of the JOR and SOR variants. GPU model used: NVIDIA GTX 580.

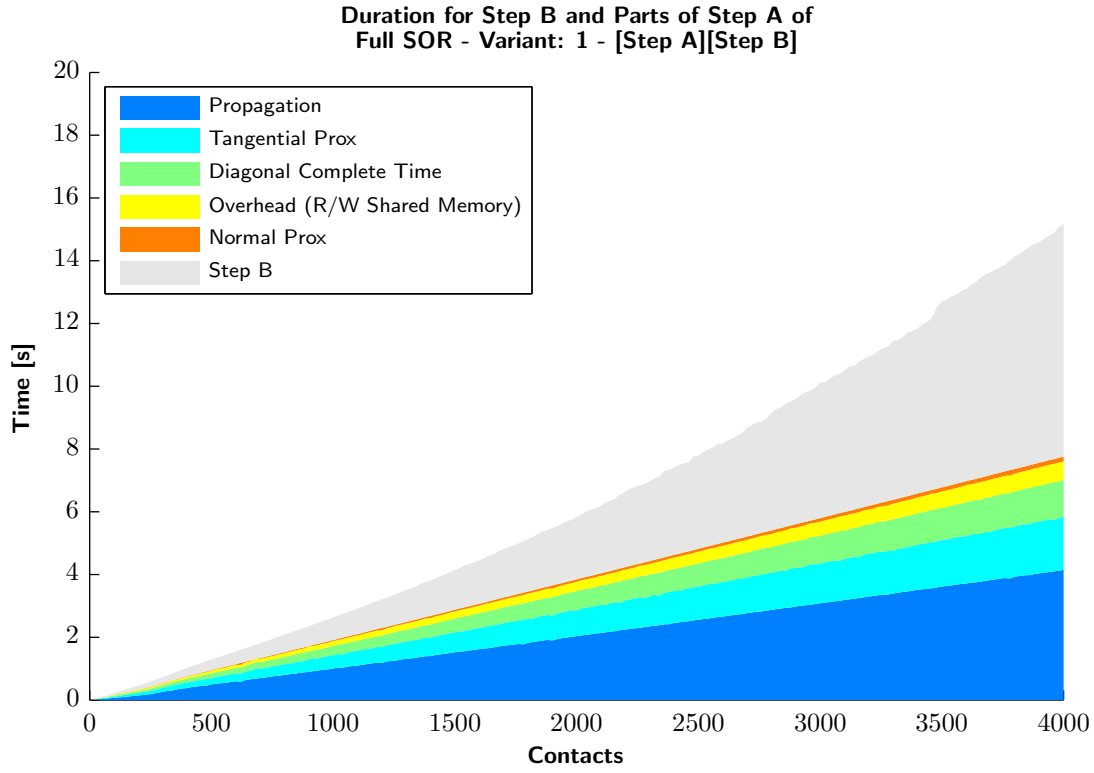


(a)

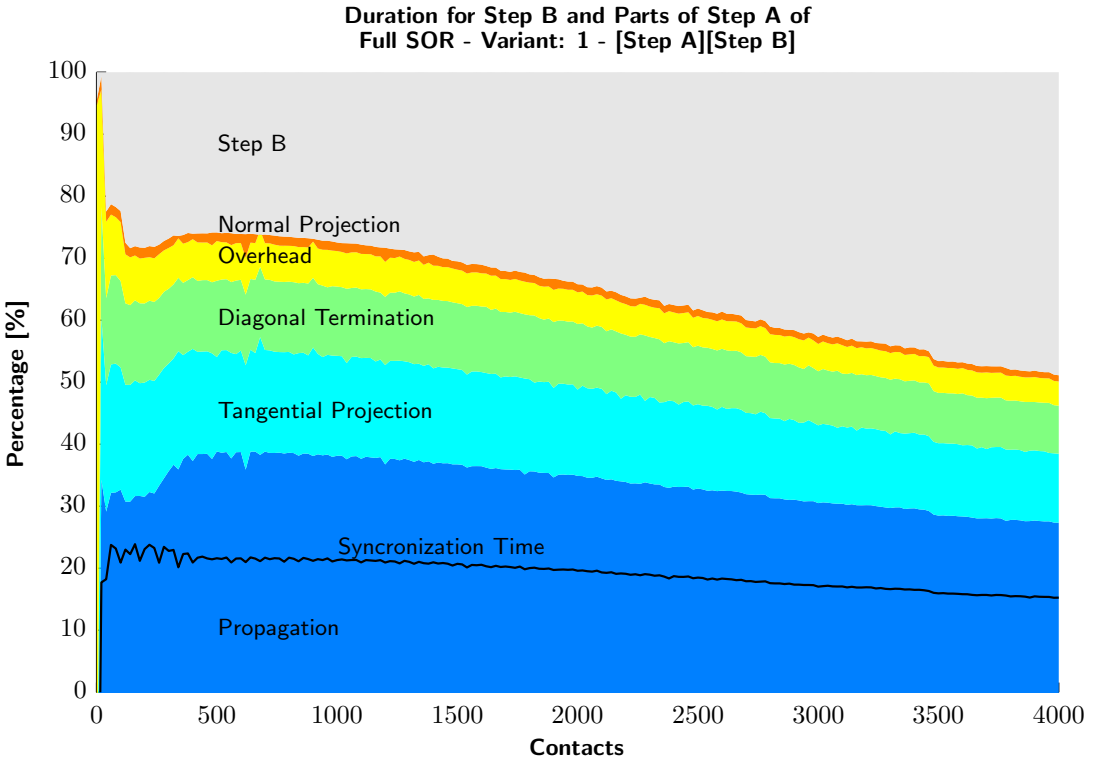


(b)

Figure 6.3.: The trade-off curve of the JOR and SOR Prox variants. GPU model used: NVIDIA GTX 580.



(a)



(b)

Figure 6.4.: The relation of Step A and Step B. Different parts in Step A have been measured. GPU model used: NVIDIA GTX 580.

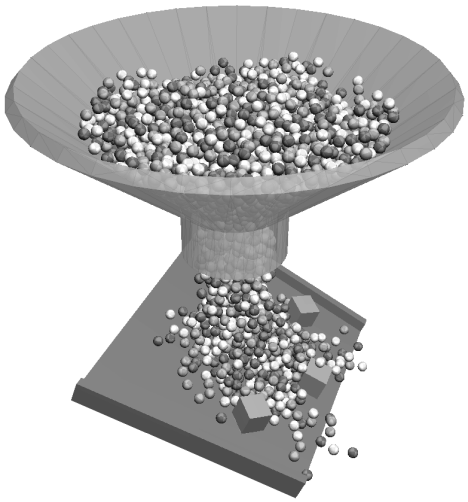
6.2. Dynamics Simulation Tests

To get a deeper insight about how these implementations perform in a real simulation, a mechanical system consisting of 2000-3000 spheres has been simulated to be able to extract a range of contact problems for all further tests. The goal was to produce as many contacts as possible in a short time and over a large range. The reference has been simulated with a time-step of $\Delta t = 0.001$ seconds and with the Full SOR Prox algorithm with a relaxation parameter $\alpha = 1.0$. For the reference simulation, the SOR Prox iteration has been aborted if more than 5000 iterations have been made. A visualization, together with the sampled contact problems of such a simulation, can be seen in figure 6.5. The best JOR Prox, Full SOR Prox and finally one variant of the Relaxed SOR Prox method have been used for the simulation of these sampled approximately 40 contact problems. Each method iterates till the termination criterion in (4.50) with an absolute and relative tolerance of 10^{-7} has been fulfilled. All methods have always used the GPU implementation for each sampled problem. The results in figure 6.6 are of course highly dependent on the specific, simulated mechanical problem. Nevertheless, the following observations can be briefly outlined:

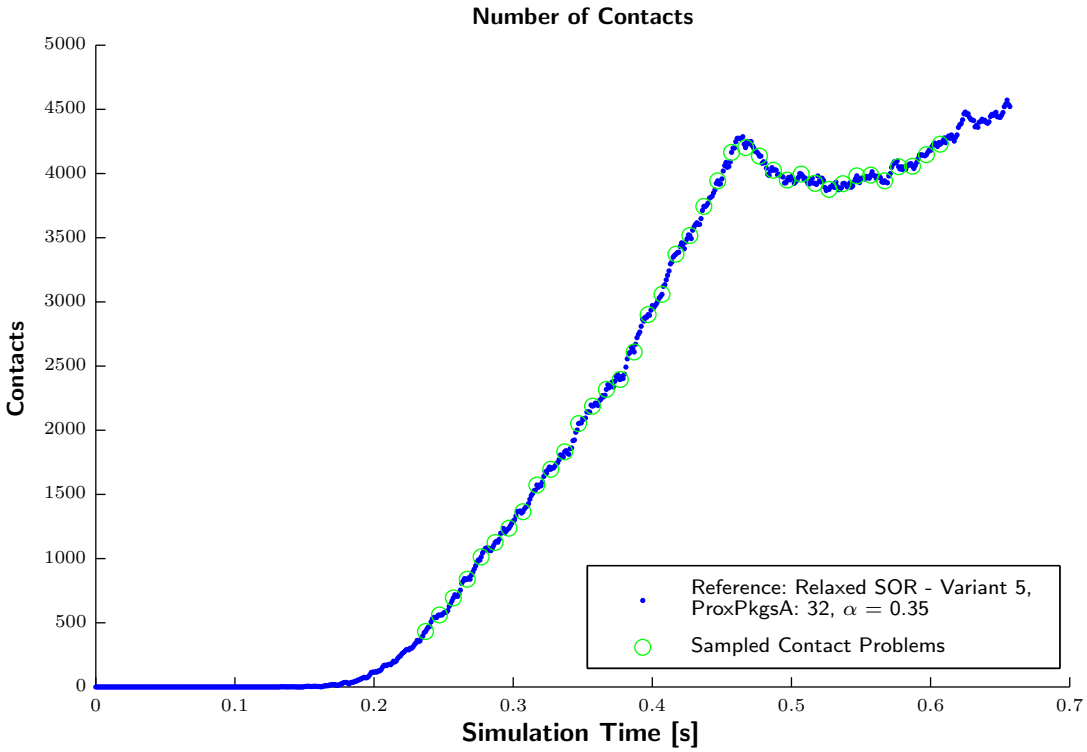
- The approximately 2 times faster iteration time of JOR Prox variant 5, in comparison to the Full SOR Prox variant 1, does not pay off because of much more (twice and more) iterations.
- The JOR Prox method is highly unstable for large problems where the matrix $\mathbf{G}$ becomes very singular, due to lots of linear dependent generalized force directions in matrix $\mathbf{W}$. For example, the value $\alpha = 0.35$ for this specific simulation is very low and cannot be increased due to the diverging behaviour.
- The Full SOR Prox is much more stable and needs much less iterations to achieve the same accuracy.
- The Relaxed SOR Prox algorithm may give not a better performance than the JOR Prox. Simulations showed that it performs like a JOR Prox method if the number of gathered contacts is higher than the maximum number of coupled contacts for a given contact problem. In this simulation, the maximum number of coupled contacts was $\approx$ 2-5 in one time-step compared to 32 gathered contacts of the Relaxed SOR Prox. This leads to a performance comparable to the JOR Prox method. However, it is likely that the Relaxed SOR Prox can give better results if the contacts are highly coupled. In this case, the JOR Prox is likely to have a very bad convergence rate.

The above observations lead to the unsurprising conclusion that for simulating a mechanical system, the Full SOR Prox method should be prefered. The relaxation parameter can be set in a first step to $\alpha = 1$. During different simulations with

spheres, the Full SOR Prox algorithm was always faster than the JOR Prox, mainly because the Full SOR Prox keeps a much lower profile on the number of iterations compared to the JOR Prox. This is illustrated in figure 6.6.

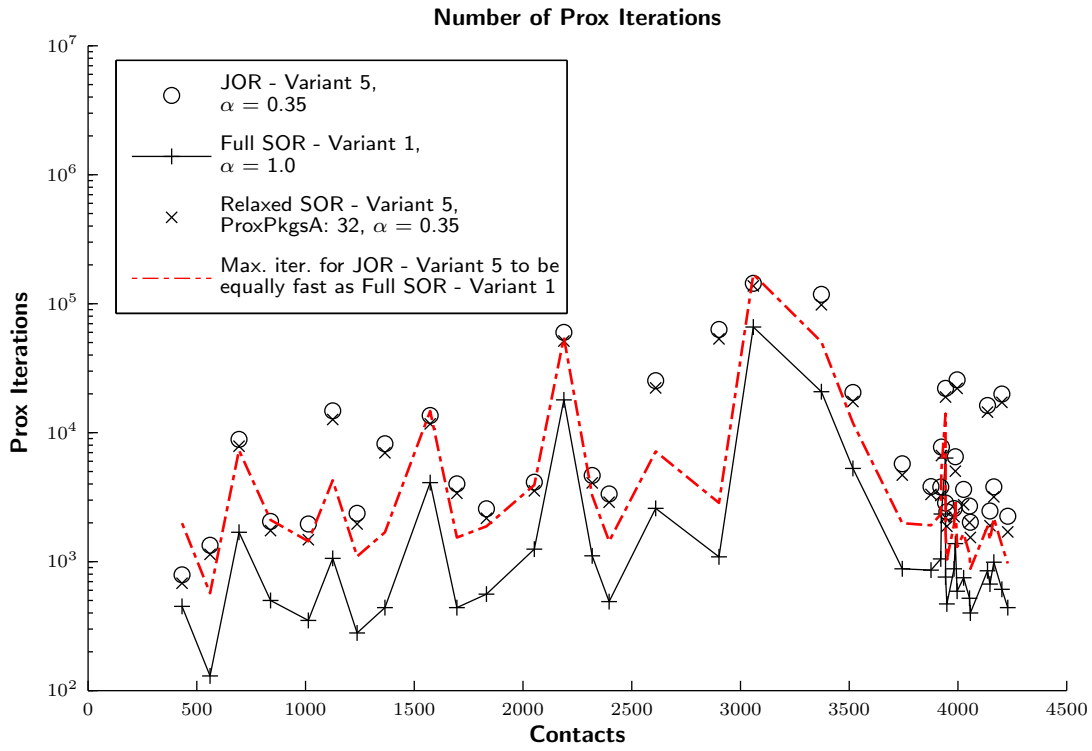


(a) Visualization between $t = 0.5$ and 0.6 seconds.

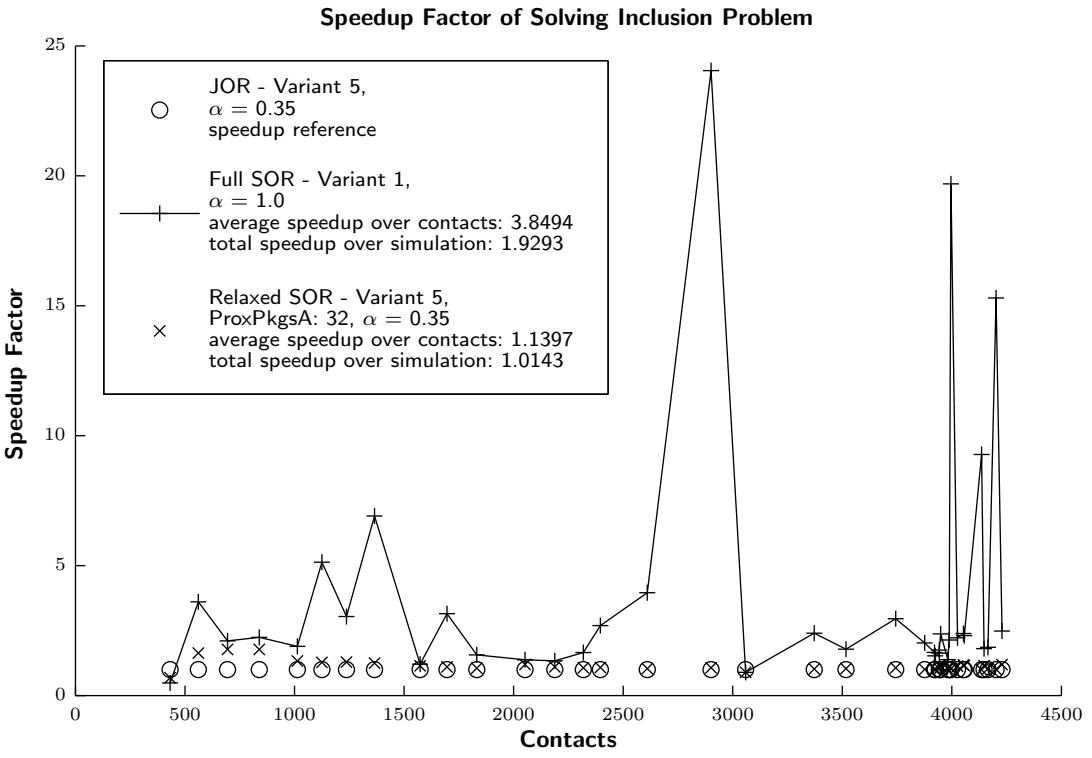


(b)

Figure 6.5.: (a) A visualization of the reference simulation between $t = 0.5$ and $t = 0.6$ seconds. The sampled data of a reference simulation of 3000 spheres.



(a)



(b)

Figure 6.6.: (a) The number of iterations of the three tested GPU methods. (b) The total time for solving the inclusion problem with the GPU compared to the slowest method JOR Prox variant 5.

7. Conclusion and Outlook

It can be concluded that the use of a GPU for large-scale mechanical simulations is very beneficial. In this work, numerical methods such as the JOR Prox and SOR Prox algorithm have been outsourced to the GPU to exploit its high parallelism. NVIDIA's programming architecture CUDA has been mainly used in this work. It comprises of both easy, fast applicable and also sophisticated, more extensive concepts at the same time in one programming model. Several kernel have been written and thoroughly tested for their performance. The starting point was a simple JOR Prox algorithm which turned out to be much more complex than expected at first. This was the case especially for the matrix-vector multiplication kernel. The SOR Prox has been implemented in a Full and Relaxed variant. The most difficult issues have been the dependencies and the question how they can be included into a parallel approach. The results in chapter 6 have shown that the most beneficial algorithm for large multibody simulations is a Full SOR Prox method. The 2 times faster iteration time of the JOR Prox algorithm does not pay off in comparision to the Full SOR Prox method because of the higher number of iterations during real rigid body simulations.

As can be seen from figure 6.3, it is therefore always recommended to use the GPU above a problem size of roughly 200-300 contacts for each variant. The limitation of the dense JOR and SOR Prox algorithm is mainly the available global memory on the GPU. The limitation of global memory becomes really burdensome in real mechanical simulations where one exploits quickly far more then 4000-4500 contacts. Hence, the only way to cope with this problem properly is to build a sparse implementation of a SOR Prox algorithm. Ideas for this task have been quickly addressed in 5.4. Some other ideas for further work are summarized in the following:

- One extremely important main goal for further work is a successful implementation of a sparse SOR Prox method.
- It is likely that if $\mathbf{G}$ would be structured in a better way, the convergence behaviour would be faster. Some graph based algorithms could be applied to obtain a more compact structure in $\mathbf{G}$. Some ideas are the separation of the contact graph into contact islands, graph coloring to separate highly coupled contacts.
- The use of multiple GPUs is another task which should be adressed in the future as well as to use a Message Passing Interface (MPI) to compute the inclusion problem on a cluster, see [9, 10].

- More studies should also be devoted to a Conjugate Gradient Method in combination with a projection. If successful results could be obtained, a parallel approach could be beneficial as well.

A. Source Code

A.1. Kernel `matrixVectorMultGPU`

Listing A.1: The complete CUDA C source code for a matrix-vector multiplication in the form $\mathbf{y} = \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{b}$. This code can be found in `KernelsMatrixVectorMult.cuh`

```
1
2 #define X_ELEMS_PER_THREAD (4)
3 #define THREADS_PER_BLOCK (128)  // How many threads we have assigned to ...
      each block ( 128 is Blas configuration!)
4
5 #define BLOCK_DIM (128)          // How many elements one Block has,  ...
      BLOCK_DIM < THREADS_PER_BLOCK, In each Block, BLOCK_DIM dot products ...
      are computed, THREADS_PER_BLOCK-BLOCK_DIM threads are idle!
6
7 #define UNROLL_BLOCK_DOTPROD (6)
8 #define DOT_PROD_SEGMENT (UNROLL_BLOCK_DOTPROD)
9
10 #define XXINCR (THREADS_PER_BLOCK)
11 #define JINCR (X_ELEMS_PER_THREAD * THREADS_PER_BLOCK)
12 #define IDXX(row) ((row)*incr_x)
13 #define IDXB(row) ((row)*incr_b)
14 #define IDXY(row) ((row)*incr_y)
15
16 template<typename PREC>
17 __global__ void matrixVectorMultiply_kernel( utilCuda::Matrix<PREC> y_dev, ...
      int incr_y, utilCuda::Matrix<PREC> A_dev,utilCuda::Matrix<PREC> x_dev, ...
      int incr_x, PREC beta, utilCuda::Matrix<PREC> b_dev,int incr_b){
18    __shared__ PREC XX[JINCR]; // Shared values for the x_dev;
19
20    int thid = threadIdx.x;
21
22    int i,j, ii, jj, row,  jjLimit, idx, incr;
23    // i: first level index in row direction to the start of the current ...
          grid (grid level),
24    // ii: second level index in row direction to the start of the current ...
          block (block level),
25
26    PREC dotp; // dot product which is accumulated by 1 thread.
27
28    // Shift the whole cuda grid over the rows!
29    for(i = 0 ; i < A_dev.M ; i += gridDim.x * BLOCK_DIM ){
30
31       ii = i + blockIdx.x * BLOCK_DIM;   // Represents the row index to ...
             the current block start
32       if( ii >= A_dev.M) break;          // The blocks which lie outside ...
             the matrix can be rejected already here
33
34       dotp = 0;   // Set the dot product to zero!
35
36       /** Shift the block with dim[ BLOCK_DIM , ...
             (THREADS_PER_BLOCK*X_ELEMS_PER_THREAD)] horizontally over A_dev, ...
```

```
            each thread calculates the sub dot product of ...
            THREADS_PER_BLOCK*X_ELEMS_PER_THREAD elements. Index j points to ...
            the beginning of the block and is shifted with JINCR = ...
            THREADS_PER_BLOCK*X_ELEMS_PER_THREAD */
37        for(j = 0 ; j < A_dev.N; j += JINCR){
38
39            jj = j + thid;                              // Represents here the ...
                  start index into x_dev for each thread!
40            jjLimit = min (j + JINCR, A_dev.N);         // Represents here the ...
                  maximal index for jj for this shifted block!
41
42            /** Syncronize before loading the ...
                  (THREADS_PER_BLOCK*X_ELEMS_PER_THREAD) shared values of ...
                  x_dev. Each thread loads X_ELEMS_PER_THREAD elements. */
43            __syncthreads();
44
45            /** Load loop (all THREADS_PER_BLOCK threads participate!). Each ...
                  of this THREADS_PER_BLOCK threads goes into another if ...
                  statement. */
46            incr = incr_x * XXINCR;
47            idx = IDXX(jj);
48  #if (X_ELEMS_PER_THREAD == 4)
49            if ((jj + 3 * XXINCR)< jjLimit ) { // See if we can read one plus ...
                   3 shifted values?
50               XX[thid+ 0*XXINCR] = alpha * x_dev.data[idx + 0 * incr];
51               XX[thid+ 1*XXINCR] = alpha * x_dev.data[idx + 1 * incr];
52               XX[thid+ 2*XXINCR] = alpha * x_dev.data[idx + 2 * incr];
53               XX[thid+ 3*XXINCR] = alpha * x_dev.data[idx + 3 * incr];
54            }
55            else if ((jj + 2 * XXINCR) < jjLimit ) {// See if we can read one ...
                  plus 2 shifted values?
56               XX[thid+ 0*XXINCR] = alpha * x_dev.data[idx + 0 * incr];
57               XX[thid+ 1*XXINCR] = alpha * x_dev.data[idx + 1 * incr];
58               XX[thid+ 2*XXINCR] = alpha * x_dev.data[idx + 2 * incr];
59            }
60            else if ((jj + 1 * XXINCR) < jjLimit) {// See if we can read one ...
                  plus 1 shifted values?
61               XX[thid+ 0*XXINCR] = alpha * x_dev.data[idx + 0 * incr];
62               XX[thid+ 1*XXINCR] = alpha * x_dev.data[idx + 1 * incr];
63            }
64            else if (jj < jjLimit) {// See if we can read one plus 0 shifted ...
                  values?
65               XX[thid+0*XXINCR] = alpha * x_dev.data[idx + 0 * incr];
66            }
67  #elif (X_ELEMS_PER_THREAD == 2)
68            if (jj + 1 * XXINCR < jjLimit) {// See if we can read one plus 1 ...
                  shifted values?
69               XX[thid+ 0*XXINCR] = alpha * x_dev.data[idx + 0 * incr];
70               XX[thid+ 1*XXINCR] = alpha * x_dev.data[idx + 1 * incr];
71            }
72            else if (jj < jjLimit) {// See if we can read one plus 0 shifted ...
                  values?
73               XX[thid+0*XXINCR] = alpha * x_dev.data[idx + 0 * incr];
74            }
75  #elif (X_ELEMS_PER_THREAD == 1)
76            if (jj < jjLimit) {// See if we can read one plus 0 shifted values?
77               XX[thid+0*XXINCR] = alpha * x_dev.data[idx + 0 * incr];
78            }
79  #else
80  #error Current code cannot handle X_ELEMS_PER_THREAD != 4
81  #endif
82
83            // Syncronize again
84            __syncthreads();
85
86            row = ii + thid; // Global index into A_dev. The row which is ...
                  processed by this thread!
87
```

```
88          // Accumulate the dot product (only BLOCK_DIM Threads participate!)
89          if(row < A_dev.M && thid < BLOCK_DIM){
                // If this row is active, ...
90             PREC * A_ptr = PtrElem_ColM(A_dev,row,j); // Start into A_dev ...
                   for this block
91
92             jjLimit = jjLimit - j; // Represents the length of the dot ...
                   product!
93             jj=0;
94
95             /** Do block dot product in segments of DOT_PROD_SEGMENT -> ...
                   Register blocking: first load all stuff, then do dot ...
                   product */
96             PREC regA[DOT_PROD_SEGMENT]; PREC regx[DOT_PROD_SEGMENT];
97             incr = A_dev.outerStrideBytes;
98             while ( (jj + (DOT_PROD_SEGMENT-1)) < jjLimit) {
                      // See if we can do one and DOT_PROD_SEGMENT-1 more values
99
100               // Load stuff into registers
101            #pragma unroll
102               for(int k = 0 ; k < DOT_PROD_SEGMENT ; k++){
103                  regA[k] = *(A_ptr);
104                  regx[k] = XX[jj + k];
105                  A_ptr = PtrColOffset_ColM(A_ptr,1,incr);
106               }
107
108               // Compute dot product
109            #pragma unroll
110               for(int k = 0 ; k < DOT_PROD_SEGMENT ; k++){
111                  dotp += regA[k] * regx[k] ;
112               }
113
114               jj += DOT_PROD_SEGMENT; // Jump DOT_PROD_SEGMENT rows in x_dev
115            }
116
117            // If no more DOT_PROD_SEGMENT segments are available do the rest
118            while (jj < jjLimit) {
119               dotp += (*A_ptr) * XX[jj + 0];
120               jj   += 1; // jump 1 rows in XX
121               A_ptr = PtrColOffset_ColM(A_ptr, 1, incr); // jump 1 col in ...
                      A_dev
122            }
123
124         }   // Accumulate dot product
125
126      }   // Shift blocks horizontally to obtain overall dot product!
127
128
129      if( row < A_dev.M && thid < BLOCK_DIM){
130         idx = IDXB(row);
131         if (beta != 0.0) {
132            dotp += beta * b_dev.data[idx];
133         }
134         idx = IDXY(row);
135         y_dev.data[idx] = dotp;
136      }
137
138   }   // Shift grid
139 }
```

A.2. Kernel sorProxStepA

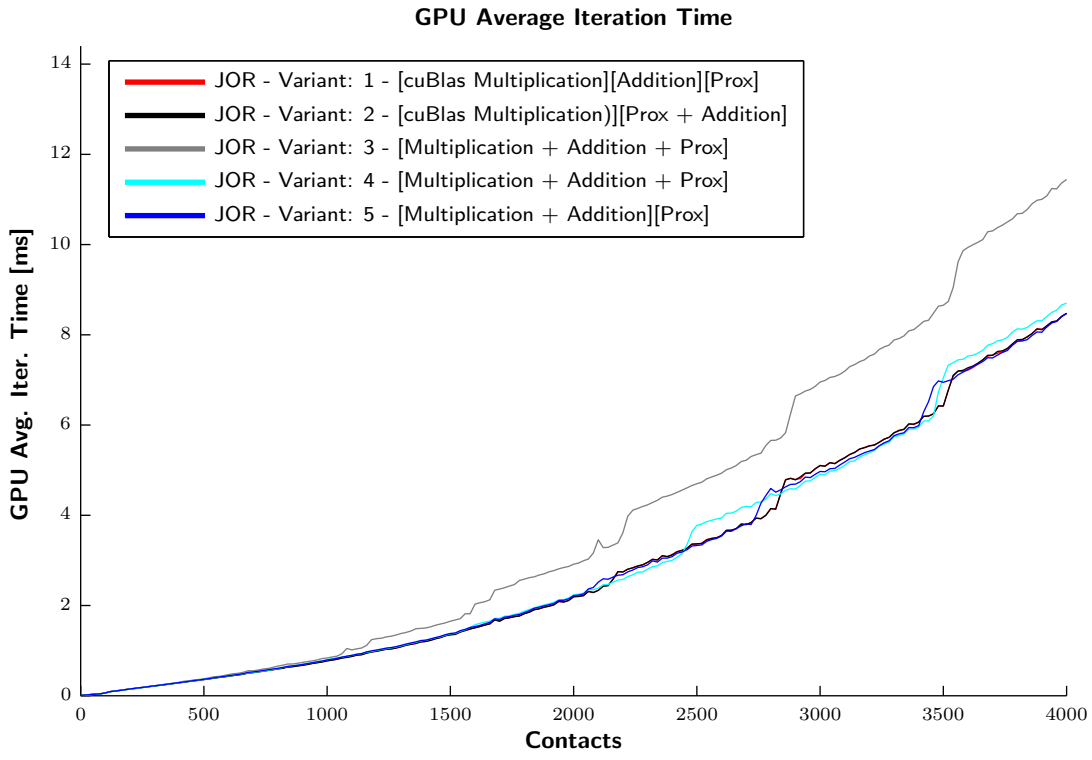
Listing A.2: The complete CUDA C source code for the kernel `sorProxStepA` of the Full SOR Prox algorithm, which is described in section 5.3.1. This code can be found in KernelsProx.cuh

```
1  template<typename PREC, int THREADS_PER_BLOCK, int BLOCK_DIM, int ...
       PROX_PACKAGES, typename TConvexSet>
2  __global__ void sorProxContactOrdered_1threads_StepA_kernel( Matrix<PREC> ...
       mu_dev, Matrix<PREC> x_new_dev, Matrix<PREC> T_dev, Matrix<PREC> ...
       d_dev, Matrix<PREC> t_dev, int kernelAIdx,int maxNContacts, bool * ...
       convergedFlag_dev, PREC _absTOL, PREC _relTOL){
3
4     STATIC_ASSERT( (IsSame<TConvexSet,ConvexSets::RPlusAndDisk>::result));
5
6      __shared__ PREC xx[BLOCK_DIM]; // Each thread writes one element, if ...
           its in the limit!!
7      __shared__ PREC tt[BLOCK_DIM];
8
9      // Assumend 1 Block, with THREADS_PER_BLOCK Threads and Column Major ...
           Matrix T_dev
10     int thid = threadIdx.x;
11     int m = min(maxNContacts*PROX_PACKAGE_SIZE, BLOCK_DIM); // This is the ...
           actual size of the diagonal block!
12     int i = kernelAIdx * BLOCK_DIM;
13     int ii = i + thid;
14
15     // References to often used values in shared mem.
16     PREC & xx_thid = xx[thid];
17     PREC & tt_thid = tt[thid];
18
19     //First copy t_dev in shared
20     if(thid < m){
21       tt_thid = t_dev.data[ii];
22     }
23
24     PREC d_value1, d_value2;
25     if(thid<m){
26       d_value1 = d_dev.data[ii];
27     }
28     if(thid<m-1){
29       d_value2 = d_dev.data[ii+1];
30     }
31
32     int jj;
33
34     PREC * T_dev_ptr  = PtrElem_ColM(T_dev,i,i);
35     PREC * mu_dev_ptr = &mu_dev.data[PROX_PACKAGES*kernelAIdx];
36
37     for(int j_t = 0; j_t < m ; j_t+=PROX_PACKAGE_SIZE){
38       // In this loop we process one [m x PROX_PACKAGE_SIZE] Block of T
39
40       // Project Normal Direction ==================
41       jj =  i  +  j_t;
42       if( ii == jj ){   // Select thread on the diagonal ...
43
44          PREC lambda_N = (d_value1 + tt_thid);
45          //Prox Normal!
46          if(lambda_N <= 0.0){
47            lambda_N = 0.0;
48          }
49          // Write the new value back
50          xx_thid = lambda_N;
51          tt_thid = 0.0;
52       }
53       // All threads not on the diagonal fall into this sync!
54       // They need to wait, because the next tangential prox will need the ...
             new value!
55       __syncthreads();
56
```

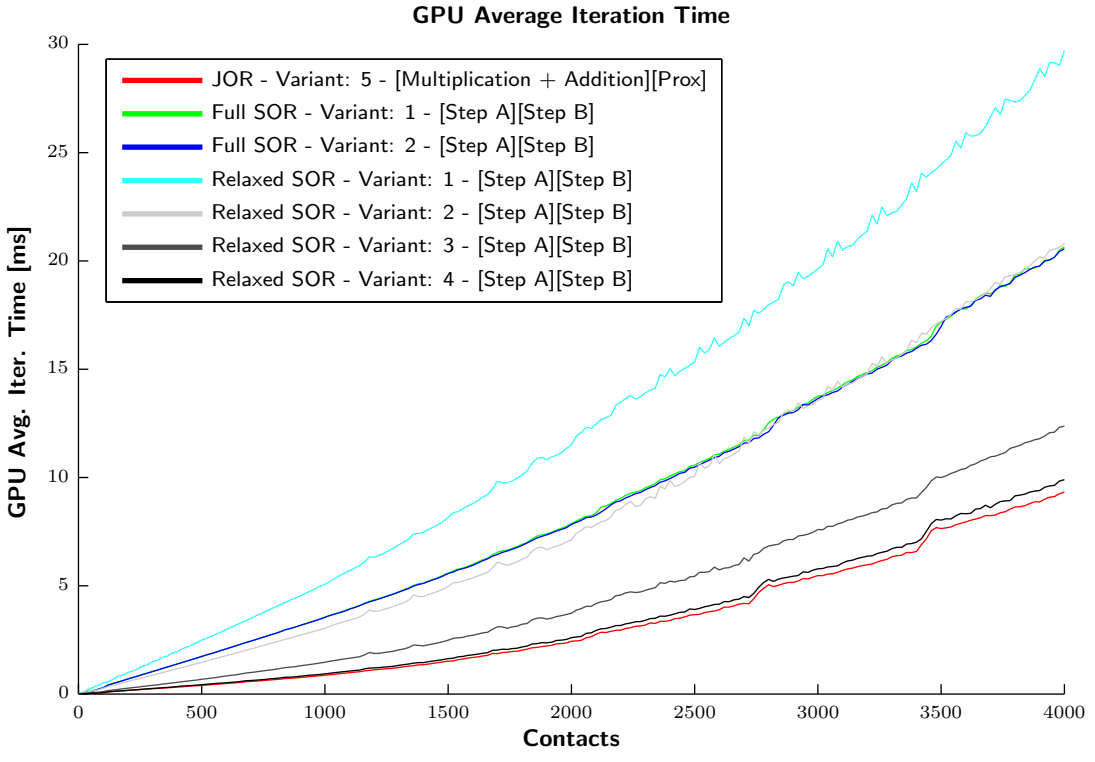
```
57        // Select only m threads for the update of vector t!
58        if(thid < m){
59           tt_thid += T_dev_ptr[thid] * xx[j_t];
60        }
61        // Syncronize here because one threads finished lambda_t2 with ...
              shared mem tt, which is updated from another thread!
62        __syncthreads();
63        // ========================================
64
65         // Project Tangential Direction =============
66        jj++;
67        if( ii == jj ){   // select thread on diagonal, one thread finishs ...
              T1 and T2 directions.
68
69           // Prox tangential
70           PREC lambda_T1 =  (d_value1 + tt_thid);
71           PREC lambda_T2 =  (d_value2 + tt[thid+1]);
72
73           PREC radius = (*mu_dev_ptr) * xx[thid-1];
74           PREC absvalue = lambda_T1*lambda_T1 + lambda_T2*lambda_T2;
75
76          if(absvalue > radius * radius){
77            if(IsSame<PREC,double>::result){
78             absvalue = radius * rsqrt(absvalue);
79            }
80            else{
81             absvalue = radius * rsqrtf(absvalue);
82            }
83             lambda_T1 *= absvalue;
84             lambda_T2 *= absvalue;
85          }
86
87           // Write the two new values back!
88           xx_thid =  lambda_T1;
89           tt_thid = 0.0;
90           xx[thid+1] = lambda_T2;
91           tt[thid+1] = 0.0;
92
93        }
94        // all threads not on the diagonal fall into this sync!
95        __syncthreads();
96
97        // Update the vector t, with the two new values
98        if(thid < m){
99           T_dev_ptr = PtrColOffset_ColM(T_dev_ptr,1,T_dev.outerStrideBytes);
100          tt_thid += T_dev_ptr[thid] * xx[j_t+1];
101          T_dev_ptr = PtrColOffset_ColM(T_dev_ptr,1,T_dev.outerStrideBytes);
102          tt_thid += T_dev_ptr[thid] * xx[j_t+2];
103       }
104       // ========================================
105
106       // Move T_dev_ptr 1 column
107       T_dev_ptr = PtrColOffset_ColM(T_dev_ptr,1,T_dev.outerStrideBytes);
108       // Move mu_ptr to next contact
109       mu_dev_ptr += 1;
110    }
111
112    // Write back the results, no need for syncronization!
113    if(thid < m){
114       x_new_dev.data[ii] = xx_thid;
115       t_dev.data[ii] = tt_thid;
116    }
117 }
```

B. Performance Tests with NVIDIA Tesla C2050

All plots for the same variants as described in section 6.1 with NVIDIA's GPU model Tesla C2050 instead of the GTX 580.

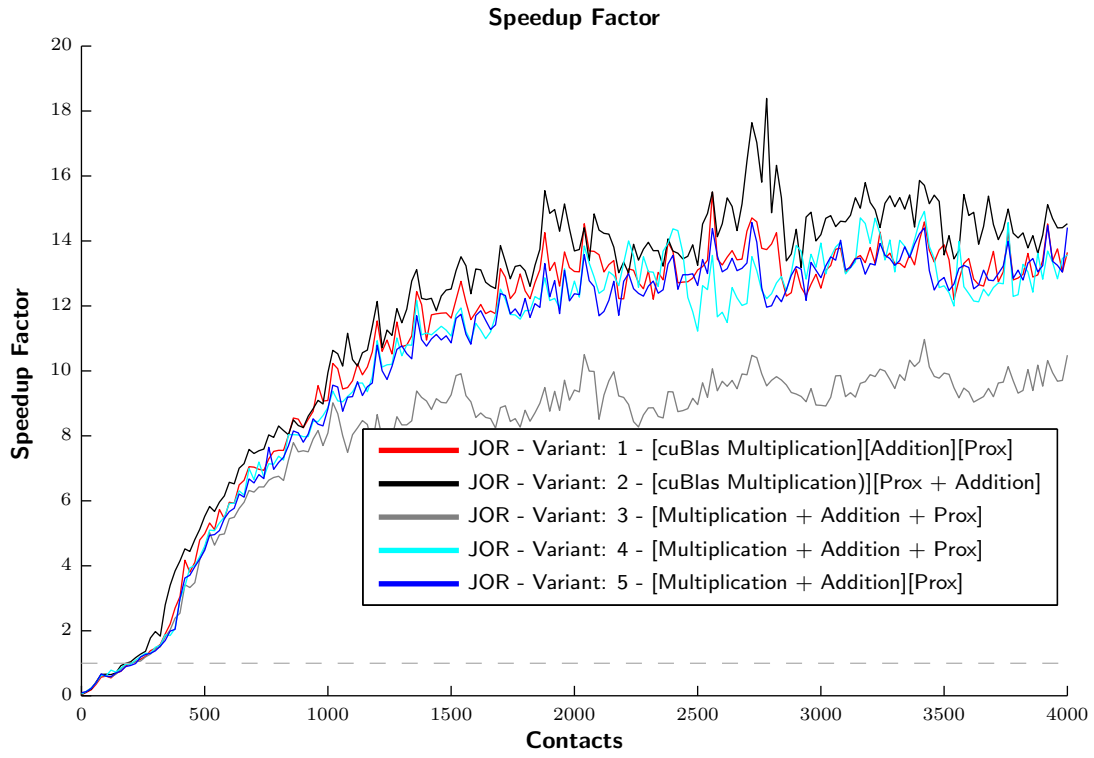


(a)

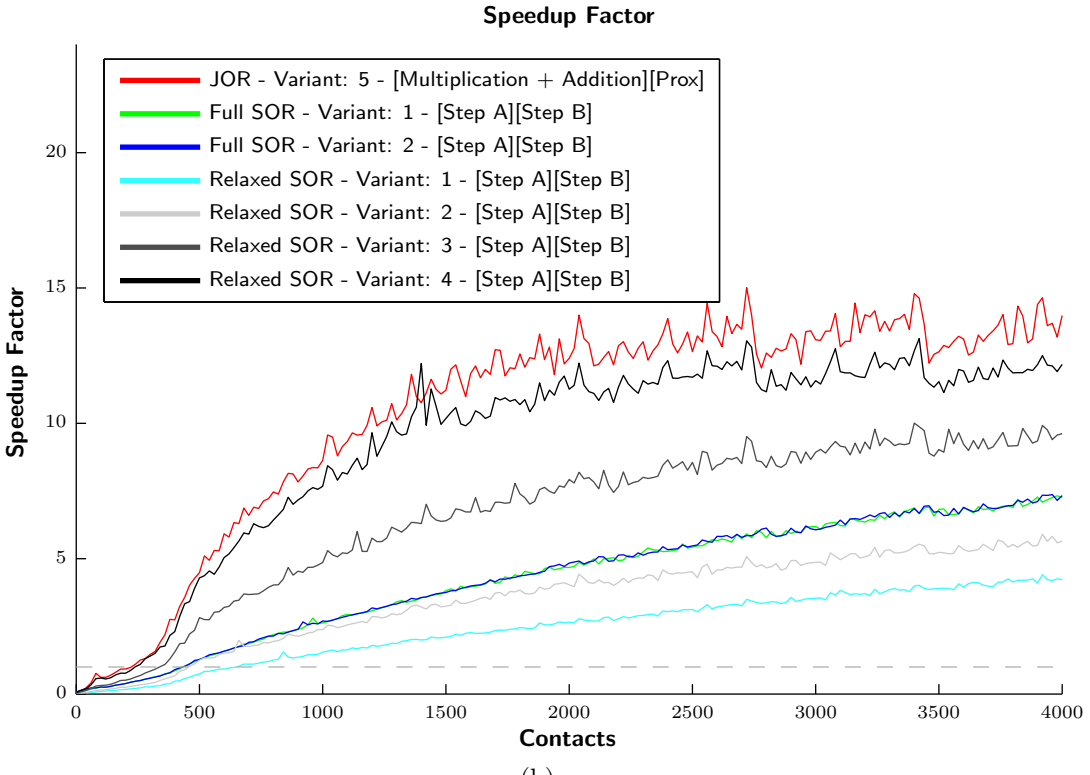


(b)

Figure B.1.: The iteration time in [ms] of the JOR and SOR Prox variants. GPU model used: NVIDIA Tesla C2050.

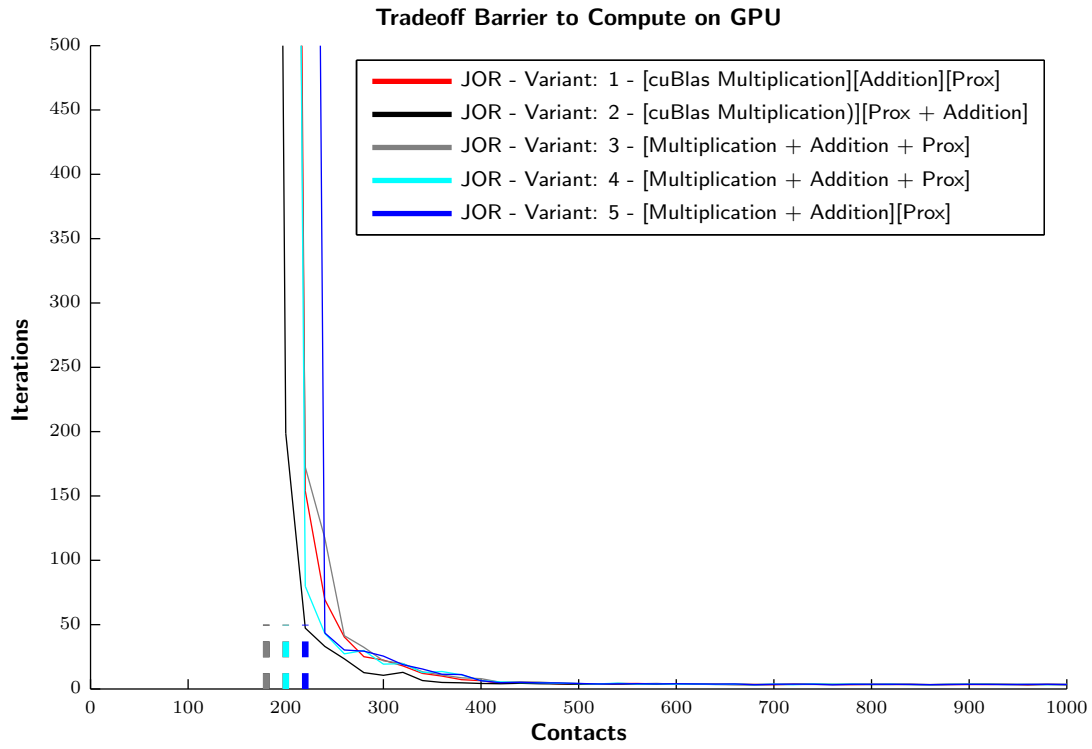


(a)

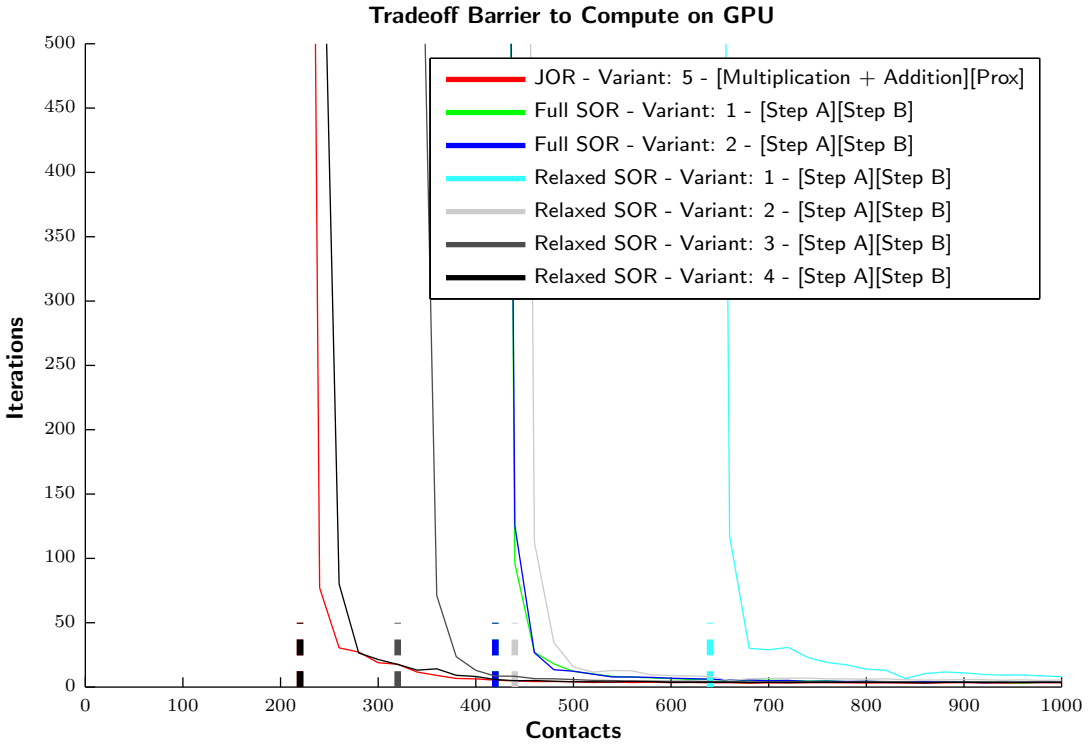


(b)

Figure B.2.: The speedup factor of the JOR and SOR variants. GPU model used: NVIDIA Tesla C2050.



(a)



(b)

Figure B.3.: The trade-off curve of the JOR and SOR Prox variants. GPU model used: Tesla C2050.

Glossary

__global__ This is a CUDA C keyword used before a function definition to specify that this function should be compiled by the NVIDIA CUDA compiler `nvcc` and that it is a function which runs on a GPU device.

__shared__ This is the CUDA C keyword used before a variable definition to specify a variable assigned in the shared memory buffer of each block. All dynamically allocated shared memory variables in a kernel function are all together allocated in one strip. Each defined shared variable needs to be aligned to a block of 4 bytes.

API Application Programming Interface. API is basically a library which can be used by programmers to extend the functionality of their program. On Windows platforms, a library often has the name suffix: "'.lib / .dll"' ; on UNIX platforms: "'.a / .so"' and on Mac platforms: "'.a / .dylib"'. The former is the suffix for a static library and the latter the suffix for a dynamic linked library..

blockIdx This CUDA C variable is used on the device to get the index of the block in a kernel grid..

CC See Compute Capability .

Compute Capability NVIDIA refers to the different features of a GPU as its compute capability (CC). NVIDIA has an updated list of the compute capability of their GPU's. Atomic operations are supported by compute capability 1.1 or higher. Atomic operations on shared memory requires compute capability 1.2 or higher. The compiler `nvcc` can be informed about the compute capability it requires by setting the compiler flag `-arch=sm_xx` where `xx` is the two digit number of the required version and higher. First CUDA capable hardware like the GeForce 8800 GTX have a compute capability of 1.0 and recent GeForce like the GTX 580 have a Compute Capability of 2.1..

CPU Central Processing Unit is the core unit of computer. It is the main computing unit which is capable of executing a program..

CUBLAS This GPU library provides linear algebra routines and implements certain algorithm from BLAS.

CUDA Abbreviation for Compute Unified Device Architecture. Cuda is a parallel computing architecture developed by NVIDIA. CUDA is the computing engine in NVIDIA graphics processing units (GPUs) that is accessible to software developers through variants of industry standard programming languages. Programmers use 'C for CUDA' (C with NVIDIA extensions and certain restrictions), compiled through a PathScale Open64 C compiler, to code algorithms for execution on the GPU. CUDA architecture shares a range of computational interfaces with two competitors - the Khronos Group's Open Computing Language and Microsoft's DirectCompute. Third party wrappers are also available for Python, Perl, Fortran, Java, Ruby, Lua, MATLAB and IDL, and native support exists in Mathematica.

CUDA Core See SP .

cudaFree This function is used to free the assigned memory on the GPU.

cudaMalloc This is the equivalence to the C `malloc` function, which reserves memory in the RAM. `cudaMalloc` does the same but in global memory of the GPU. Do not reference any pointer which has been allocated with `cudaMalloc` on the host.

cudaMemcpy This is the equivalence to the C function `memcpy`. This function copies memory from the device to the host and vice versa. One can specify by `cudaMemcpyDeviceToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToDevice` which operation should be performed. The CUDA Driver uses Direct Memory Access (DMA) to transfer the buffer to the GPU. Therefor the copy happens twice, first from a pageable system buffer to a page-locked "staging" buffer and then from the page-locked system buffer to the GPU. This call is synchronous.

cudaprof NVIDIA's profiling tool for GPU kernels .

CUFFT A Fast Fourier Transform GPU library provided by NVIDIA.

DirectX See OpenGL. See [`http://msdn.microsoft.com/de-de/directx/`] .

FLOPS Floating Point Operations Per Seconds. FLOPS is a performance metric for measuring the performance of algorithms which are computed on a processor. FLOPS counts the number of floating point operations which can be executed by a processor in a second. Typically a floating point operation (flop) is referred to an addition, subtraction, multiplication or division of two floating point numbers. Although FLOP and FLOPS is used in the field of computer science

it is not an SI unit.

GPU A graphics processing unit or GPU is a specialized microprocessor that offloads and accelerates graphics rendering from the central (micro-)processor CPU. It is used in embedded systems, mobile phones, personal computers, workstations, and game consoles. Modern GPUs are very efficient at manipulating computer graphics, and their highly parallel structure makes them more effective than general-purpose CPUs for a range of complex algorithms. They are nowadays used for general purpose parallel programming.

Graphics Pipeline See [`http://en.wikipedia.org/wiki/Graphics_pipeline`] .

Host The host controls the devices, here GPU's, which are attached. One distinguishes between host code and device code. Host code is executed on the CPU and not on the GPU.

Kernel In the CUDA and OpenCL context, a kernel is a program, mostly written in C, with some extensions depending on the programming language, which can be executed on a GPU or the CPU (OpenCL).

Kernel Execution A kernel execution is launched by specifying the grid dimension `gridDim` and the block dimension `blockDim`. In CUDA this is written as follows `kernel<<<10,16>>>(…)`, where 10 is the 1D grid dimension (10 blocks) and 16 is a 1D block dimension (16 threads per block). A very important property of a kernel execution is that it is asynchronous. See Asynchronous Execution.

MPI MPI is a language independent communication protocol used to program parallel computers. It allows to pass messages among processes in a distributed system, e.g. cluster. It has become the de facto standard for communication among processes in distributed high-performance computing systems. See [`http://www.mpi-forum.org/`] .

OpenCL Open Computing Language is an open-source programming language which allows heterogeneous computing on any platform. This means that a program can be executed in parallel on different OpenCL capable devices such as CPU's, GPU's and other devices. See [`http://www.khronos.org/opencl/`] .

OpenGL Open Graphics Library is a specification for a platform independent , programming language independent interface for developing 2D and 3D graphics applications. The Microsoft pendant is DirectX. See [`http://www.opengl.org/`] .

OpenMP Open Multi-Processing is an API that supports platform independend shared-memory multiprocessing programming in C,C++ and Fortran. See [`http://openmp.org/wp/`] .

SFU Super Function Unit. It executes transcendental instructions such as sin, cosine, reciprocal, and square root. Each SFU executes one instruction per thread, per clock .

Shader A shader is a set of software instructions that is used primarily to calculate rendering effects on graphics hardware with a high degree of flexibility. Shaders are used to program the graphics processing unit (GPU) programmable rendering pipeline, which has mostly superseded the fixed-function pipeline that allowed only common geometry transformation and pixel-shading functions; with shaders, customized effects can be used [`http://en.wikipedia.org/wiki/Shader`]. Shader is a deprecated name for SP .

SM Streaming Multiprocessor. Contains several Streaming Processors and Super Function Units (SFU). Thread blocks are distributed serially to all SM's. As soon a SM receives a thread block, it splits the block into warps and launches them. As soon all warps in a thread block have finished, the control unit of the SM gets another thread block. The SM assigns and maintains thread ID's. The SM uses a scheduling algorithm to manage and schedule the thread execution. The scheduling units are warps. All threads in a warp execute the same instruction.

SP Streaming Processor.

SPA Streaming Processor Array. A grid is launched on the SPA.

threadIdx This CUDA C variable is used on the device to get the index of the thread in the block. A block has a limited number of threads which can be called. See the `cudaDeviceProp`.

TPC Tecture Processing Cluster. See the GF100 architecture for more information .

ULP Units in the last place or unit of least precision is the spacing between floating point numbers on a computer .

Warp For those familiar with Star Trek, this has nothing do to with the speed of travel through space. In the CUDA Architecture, a warp refers to a collection of 32 threads that are "woven together" and in general all threads in a warp are executing the same instructions at the same time .

Bibliography

[1] Bell, N., and Garland, M. Efficient Sparse Matrix-Vector Multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[2] Courtecuisse, H., and Allard, J. Parallel dense gauss-seidel algorithm on many-core processors. In *Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 139–147.

[3] Dawson, B. Comparing floating point numbers. `http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm`. [Online; accessed 02.08.2011].

[4] Glocker, Ch. *Dynamik von Starrkörpersystemen mit Reibung und Stößen*, vol. 18, no. 182 of *Fortschr.-Ber. VDI*. VDI Verlag, Düsseldorf, 1995.

[5] Glocker, Ch. Simulation von harten Kontakten mit Reibung. *Schwingungen in Antrieben 2006, VDI-Berichte 1968* (2006), 19–44.

[6] Goldberg, D. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys 23*, 1 (1991), 5–48. `http://download.oracle.com/docs/cd/E19957-01/806-3568/ncg_goldberg.html`.

[7] Gustafson, J. L. Reevaluating amdahl's law. *Commun. ACM 31* (May 1988), 532–533.

[8] Hunger, R. Floating Point Operations in Matrix-Vector Calculus, Technical Report. `http://www.msv.ei.tum.de/MSV/people/rahu/Floating_Point_Handbook_v13.pdf` at Technische Universit t M nchen Associate Institute for Signal Processing, 2007. [Online; accessed 02.08.2011].

[9] Iglberger, K. The *pe* Physics Engine. `http://www10.informatik.uni-erlangen.de/Research/Projects/pe/`, 2011. [Online; accessed 02.08.2011].

[10] Iglberger, K., and Rüde, U. Large-scale rigid body simulations. *Multibody System Dynamics 25* (2011), 81–95. 10.1007/s11044-010-9212-0.

[11] Kirk, D. B., and Hwu, W.-m. W. *Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series)*, 1 ed. Morgan Kaufmann, Feb. 2010.

[12] Kloeckner, A. CUDA vs OpenCL: Which should I use? `http://wiki.tiker.net/CudaVsOpenCL`, 2011. [Online; accessed 02.08.2011].

[13] Li, Y.-M. Comparison of Floating-point Values. `http://www.infogoaround.org/JBook/FloatingComparison.pdf`, 2009. [Online; accessed 02.08.2011].

[14] Microsoft. THE MANYCORE SHIFT: Microsoft Parallel Computing Initiative Ushers Computing into the Next Era, Parallel Computing: Background. `http://www.intel.com/pressroom/kits/upcrc/ParallelComputing_backgrounder.pdf`. [Online; accessed 02.08.2011].

[15] M ller, M. *Consistent Integrators for Non-Smooth Dynamical System.* PhD thesis, ETH Zürich, 2011. Diss. ETH No. 19715.

[16] Möller, M. Lösung der Time-Stepping Gleichungen mit der Augmented Lagrangian Methode, Semester Thesis.

[17] Moreau, J. *Unilateral contact and dry friction in finite freedom dynamics. In: J.J. Moreau and P.D. Panagiotopoulos*, vol. 302 of International Centre for Mechanical Sciences, Courses and Lecture. Springer, Verlag, 1988.

[18] Nickolls, J., Buck, I., Garland, M., and Skadron, K. Scalable parallel programming with cuda. *Queue 6* (March 2008), 40–53.

[19] NVIDIA. Analysis-driven optimization. `http://people.maths.ox.ac.uk/gilesm/cuda/lecs/ISC11_Analysis_Driven_Optimization.pdf` and `http://www.nvidia.com/content/PDF/sc_2010/CUDA_Tutorial/SC10_Analysis_Driven_Optimization.pdf`.

[20] NVIDIA. GeForce GTX 400 Graphics Architecture, White Paper. `http://www.nvidia.com/object/IO_89569.html`, `http://www.nvidia.com/object/GTX_400_architecture.html`, 2010. [Online; accessed 02.08.2011].

[21] NVIDIA. Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview. `http://www.nvidia.com/page/8800_tech_briefs.html`, 2010. [Online; accessed 02.08.2011].

[22] NVIDIA. CUDA C Best Practices Guide. `http://developer.nvidia.com/nvidia-gpu-computing-documentation`, 2011. [Online; accessed 02.08.2011].

[23] NVIDIA. CUDA C Getting Started Guide. `http://developer.nvidia.com/nvidia-gpu-computing-documentation`, 2011. [Online; accessed 02.08.2011].

[24] NVIDIA. CUDA C Programming Guide. `http://developer.nvidia.com/nvidia-gpu-computing-documentation`, 2011. [Online; accessed 02.08.2011].

[25] NVIDIA. CUDA Toolkit 4.0 CUBLAS Library. `http://developer.nvidia.com/nvidia-gpu-computing-documentation`, 2011. [Online; accessed 02.08.2011].

[26] NVIDIA. Fermi Compute Architecture Whitepaper. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2011. [Online; accessed 02.08.2011].

[27] Patrick Schmid, C. P. DirectX vs. OpenGL. `http://www.piquadrat.com/Projekte/ComGra/Dokumentation.pdf`, University of Applied Science Isny, 88316 Germany, 2005. [Online; accessed 02.08.2011].

[28] Prof. S. Gorlatch, M. S. Multicore und GPU: Parallele Programmierung. Lecture: Gruppe PVS (Parallele und Verteilte Systeme), Institut für Informatik, Westfälische Wilhelms-Universität Münster, `http://pvs.uni-muenster.de/pvs/lehre/SS11/mgpp/`, 2011. [Online; accessed 02.08.2011].

[29] Sanders, J., and Kandrot, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1 ed. Addison-Wesley Professional, July 2010.

[30] Studer, C., and Glocker, C. Representation of normal cone inclusion problems in dynamics via non-linear equations. *Archive of Applied Mechanics 76* (2006), 327–348. 10.1007/s00419-006-0031-y.

[31] Sun, Y., and Tong, Y. Cuda based fast implementation of very large matrix computation. In *Proceedings of the 2010 International Conference on Parallel and Distributed Computing, Applications and Technologies* (Washington, DC, USA, 2010), PDCAT '10, IEEE Computer Society, pp. 487–491.

[32] Wilson, G. V. The history of the development of parallel computing. `http://ei.cs.vt.edu/~history/Parallel.html`, 2004. [Online; accessed 02.08.2011].

[33] Wong, H., Papadopoulou, M.-M., Sadooghi-Alvandi, M., and Moshovos, A. Demystifying GPU Microarchitecture through Microbenchmarking. `http://www.stuffedcow.net/files/gpuarch-ispass2010.pdf`, `http://www.stuffedcow.net/research/cudabmk`, 2010. [Online; accessed 02.08.2011].