GPU-based Parallel Computing for the Simulation of Complex Multibody Systems with Unilateral and Bilateral Constraints: An Overview

Alessandro Tasora[1]**, Dan Negrut**[2] **and Mihai Anitescu**[3]

[1] University of Parma, Parma, Italy, tasora@ied.unipr.it

[2] University of Wisconsin, Madison, WI-53706, USA, negrut@wisc.edu

[3] Argonne National Laboratory, Argonne, IL-60439, USA, anitescu@mcs.anl.gov

Abstract: This work reports on advances in large-scale multibody dynamics simulation facilitated by the use of the Graphics Processing Unit (GPU). A description of the GPU execution model along with its memory spaces is provided to illustrate its potential parallel scientific computing. The equations of motion associated with the dynamics of large system of rigid bodies are introduced and a solution method is presented. The solution method is designed to map well on the parallel hardware, which is demonstrated by an order of magnitude reductions in simulation time for large systems that concern the dynamics of granular material. One of the salient attributes of the solution method is its linear scaling with the dimension of the problem. This is due to efficient algorithms that handle in linear time both the collision detection and the solution of the nonlinear complementarity problem associated with the proposed approach. The current implementation supports the simulation of systems with more than one million bodies on commodity desktops. Efforts are under way to extend this number to hundreds of millions of bodies on small affordable clusters.

1. Introduction

Gauging through simulation the mobility of tracked and/or wheeled vehicles on granular terrain (sand and/or gravel) for commercial (construction equipment industry), military (off-road mobility), and deep space exploration (Rover mobility on Martian terrain) applications leads to very challenging multibody dynamics problems. In the past, when applicable, the only feasible approach to these and other granular dynamics dominated problems was to approximate the discrete nature of the material with a continuum representation. For the classes of problems of interest here, such as material mixing, vehicle mobility on sand, piling up of granular bulk material, the flow in pebble bed nuclear reactors, rate of flow in silos, stability of brick buildings to earthquakes, etc., a continuum representation of the problem is either inadequate or paints with too wide of a brush the dynamics of interest. Tackling head on the discrete problem, characterized by a large number of bodies that interact through frictional contact and might have vastly different mass/inertia attributes, has not been feasible in the past.

The computational multibody dynamics landscape has experienced recently changes fueled by both external and internal factors. In terms of the former, sequential computing appears to lose momentum at a time when the microprocessor industry ushers in commodity many-core hardware. In terms of internal factors, contributions made in understanding and handling frictional contact [1-11], have led to robust numerical algorithms that can tackle sizeable granular dynamics problems. This paper discusses how the interplay of these two factors will enable in the near future a discrete approach to investigating the dynamics of systems with hundreds of millions of rigid bodies.

The paper is organized as follows. Section 2 starts with a brief discussion of three roadblocks that adversely impact the potential of sequential computing and limit its future role in computational science in general, and computational multibody dynamics in particular. An argument is made that in large scale multibody dynamics emphasis should be placed on implementations that can leverage commodity high performance parallel computing. In this context, an overview is presented of NVIDIA's hardware architecture, which is adopted herein when tackling large scale multibody dynamics problems. The discussion focuses on a description of the parallel execution model, execution scheduling, and memory layout. Section 3 details how large scale frictional contact problems associated with granular dynamics are solved by a computational approach that maps well onto parallel execution hardware available on the GPU. The approach implemented has two compute intensive parts: the solution of a cone complementarity problem (CCP) and the resolution of a collision detection (CD) analysis. In both cases, the solution embraced draws on parallel computing and a discussion of the CCP algorithm adopted concludes section 3. Section 4 demonstrates the use of the solution approach implemented. First, the paper briefly reports on the largest granular dynamics problems solved using the methodology discussed in section 3. Next, a pebble bed nuclear reactor flow problem compares the efficiency of the parallel implementation on the GPU to that of the sequential implementation. The paper closes with concluding remarks and a discussion of future directions of research.

2. Review of computing on the Graphics Processing Unit

As pointed out in [12], three road blocks prevent traditional sequential computing from experiencing future major gains in flop rate: the memory block, the instruction level parallelism block, and the power dissipation block. The first one is a consequence of the fact that as the data processing power of a CPU core increases, the number of memory transactions in the time unit also goes up. From 1986 to 2000, CPU speed improved at an annual rate of 55% while memory access speed only improved at a 10% rate. One outcome of this trend was an increase in the likelihood of cache misses, which have been partially alleviated by employing hyper-threading technologies and considering ever increasing cache memories. Nonetheless, cache misses occur and they lead to the CPU waiting for chunks of data moved over a 32.5 GB/s connection that currently connects the CPU to the RAM. The second block stems from the exhaustion of the idea of speculative execution of future instructions to produce results ahead of time and make them available to the processor in case the actual computational path was correctly anticipated. However, this speculative execution strategy necessitates power and is plagued by a combinatorial increase in the number of possible computational paths. This translates into a short future execution horizon that can be sampled by these techniques. The attractive attribute of this strategy is that the programmer doesn't have to do anything to speed up the code. Instead, the CPU takes upon itself the task of employing this strategy. On the flip side, this avenue of speeding up execution has been thoroughly taken advantage of and its potential has been already fulfilled. Thirdly, the amount of power dissipated by a CPU per unit area has approached that of a nuclear plant [13]. Since the power dissipated is proportional to the square of the microprocessor clock frequency, it becomes apparent that significant microprocessor frequency increases, which were primarily responsible for past reductions in computational times in commodity scientific computing, are a thing of the past.

One bright spot in this bleak background against which the future of commodity hardware for scientific computing is projected comes from the consensus in the microprocessor industry that for at least one more decade Moore's law will hold. The law states that the number of transistors that can be placed inexpensively on an integrated circuit is doubling approximately every two years. Since this translates into a steady increase in the number of microprocessors that can be packed on the unit area, Moore's law indirectly defines the source of future increases in flop rate in scientific computing. Specifically, rather than hoping for frequency gains, one will have to count on an increase in number of cores as the means for speeding up simulation.

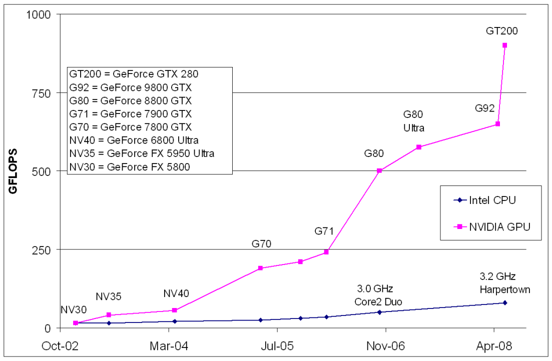

Figure 1. Evolution of Flop rate, comparison CPU vs. GPU**.**

Figure 1 confirms this trend by comparing top flop rates for the CPU and GPU. Since the plot compares double precision (DP) CPU flop rates with single precision (SP) rates for the GPU, the relevant point is not made by the absolute values. Rather, the trends are more important: the slope for the CPU is significantly smaller than that of the GPU. Table 1 partially explains the momentum behind parallel computing on the GPU. The last generation of NVIDIA cards packs 1.4 billion transistors, reaching 3 billion with the release of Fermi in early 2010, to produce a GPU with 240 scalar processors, or 512 on Fermi. Their clock frequency is lower, that is, 1.33GHz, thus partially alleviating the heat dissipation issue. Yet, the GPU compensates through a larger memory bandwidth (likely to increase to more than 200 GB/s on Fermi) and sheer number of scalar processors.

The idea of using the graphics card for scientific computing dates back more than one decade. Their use was motivated by the sheer amount of computational power available on the GPU. Fueled by a steady demand for a more realistic video game experience, the GPU experienced a continuous increase in flop rate to facilitate the rendering of more realistic visual effects at a rate of 20 frames per second or higher. The original graphics pipeline operated through graphics shaders and was meant to perform the same set of operations on multiple data sets. The data here is the information associated with a pixel; the operations were the set of instructions necessary to determine the state of each pixel of the screen. At high resolutions, this required a large number of threads to process in parallel the information that would make possible the output of one frame. This computational model, in which one set of instructions is applied for many instances of data, is called SIMD (single instruction multiple data). It is the paradigm behind processing data on the GPU and was leveraged before 2006 by drawing on existing graphics application programming interfaces (API) such as OpenGL and DirectX.

	Tesla C1060	**Intel I7 975 Extreme**
Cores	240	4
Memory	4 GB	32 KB L1 cache/core 256 KB L2 cache/core 8 MB L3 for all cores
Clock	1.33 GHz	3.20 GHz
Bandwidth	102 GB/s	32.0 GB/s
FLOPS	933 x 10^{9} (SP)	70 x 10^{9} (DP)

Table 1.CPU vs. GPU comparison. Flop rates reported are in single precision (SP) for the GPU and double precision (DP) for the CPU.

However, scientific computing through a graphics API was both cumbersome and rigid. It was cumbersome since any data processing task had to be cast into a shader operation. This either required a lot of imagination, or outright prevented one from using GPU computing for more complicated tasks. The approach was also rigid in that it only allowed a limited number of memory transaction operations (for instance one thread could only write to one memory location), it lacked certain arithmetic operations (such as integer and bit operations), and implementation of the IEEE754 standard for arithmetic operations was of secondary importance.

The GPU computation landscape was revolutionized by the release in 2006 of the version 1.0 of the CUDA Software Development Kit (SDK) and library [14], which eliminated the vast majority of the barriers that prevented the use of the GPU for scientific computing. CUDA allows the user to write “C with extensions” code to directly tap into the computational resources of the GPU through a run-time API. The CPU, typically called the host, is linked to the GPU, called the device, through a Peripheral Component Interconnect Express 2.0 (PCIe 2.0 x 16) connection. This connection supports an 8.0 GB/s data transfer rate and represents the conduit for data exchange between the host and device.

The hardware layout of the latest generation of NVIDIA graphics cards for scientific computation called Tesla is schematically shown in Figure 2. The GPU is regarded as one big Stream Processor Array (SPA) that for the Tesla C1060 hosts a collection of 10 Texture Processor Clusters (TPC). Each TPC is made of a texture block (called TEX in Figure 2), and more importantly, of three Stream Multiprocessors (SM). The SM, sometimes also called the multiprocessor, is the quantum of scalability for the GPU hardware. Thus, entry level graphics cards might have four SMs, such as is the case for GPUs like NVIDIA’s 9700M GT

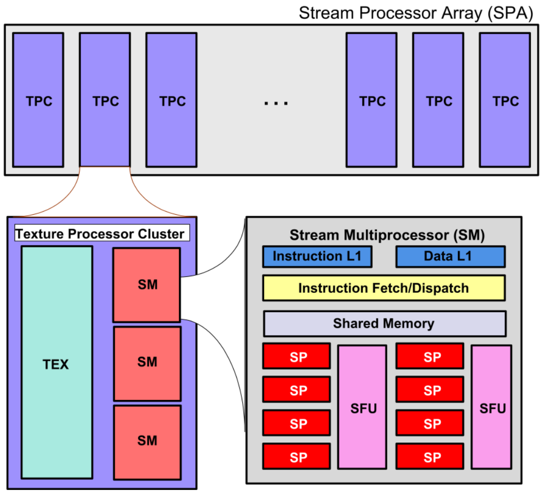
Stream Processor Array (SPA)
TPC
TPC
TPC
. . .
TPC
TPC
TPC
Texture Processor Cluster
TEX
SM
SM
SM
Stream Multiprocessor (SM)
Instruction L1
Data L1
Instruction Fetch/Dispatch
Shared Memory
SP
SP
SP
SP
SFU
SP
SP
SP
SP
SFU

Figure 2. Hardware layout for the Tesla C1060 card. The SPA has ten TPCs, each with three SMs, each of which has eight SPs for a total of 240 SPs.

which are used in computer laptops. High end GPUs, such as the NVIDIA GTX 280, have 30 SMs. The Tesla C1060 has also 30 SMs since the SPA has ten TPCs, each with three SMs. Finally, each SM has eight Scalar Processors (SP). It is these SPs that eventually execute the instructions associated with each function that is processed on the GPU. Specifically, the device acts as a co-processor for the host, which sends down to the device tasks for parallel execution. For this computational model to be effective, at least two requirements must be met. First, the ratio of arithmetic operations to data transfer should be high enough to cover the transfer overhead associated with the 8.0 GB/s data transfer from host to device for processing, and then back to the host for subsequent use. Second, the task sent for completion on the GPU, encapsulated in a C function called kernel, should have a high level of fine grain SIMD type parallelism.

For effective use of the available SMs, a kernel function must typically be executed by a number of threads in excess of 30,000. In fact, the more threads are launched, the larger the chance of full utilization of the GPU's resources. It should be pointed out that there is no contradiction in 240 SPs being expected to process hundreds of thousands or millions of parallel invocations of a kernel function. In practice, the largest number of times a kernel can be asked to be executed on Tesla C1060 is more than two trillion $(65,535 \times 65,535 \times 512)$ times.

When discussing about running kernels on the GPU, it is important to make a distinction between being able to execute a kernel function a large number times, and having these executions run in parallel. In practice, provisions should be made that there are enough instances of the kernel function that are lined up for execution so that the 240 SPs never become idle. This explains the speed-ups reported in conjunction with GPU computing when applications in image processing, quantum chemistry, and finance have run up to 150 times faster on the GPU although the peak flop rate is less than 10 times higher when compared to the CPU. For the latter, cache misses place the CPU in idle mode waiting for the completion of a RAM transaction. Conversely, when launching a job on the GPU that calls for a very large number of executions of a kernel function, chances are that the scheduler will always find warps, or collections of threads, that are ready for execution. In this context, the SM scheduler is able to identify and park with almost zero overhead the warps that wait for memory transaction completion and quickly feed the SM with warps that are ready for execution. The SM scheduler (which manages the "Instruction Fetch/Dispatch" block in Figure 2) can keep tabs on a pool of up to 32 warps of threads, where each warp is a collection of 32 threads that are effectively executed *in parallel*. Thus, for each SM, the scheduler jumps around with very little overhead in an attempt to find, out of the 32 active warps, the next warp ready for execution. This effectively hides memory access latency.

Note that the number of *threads* that are executed in parallel (32 of them), is typically orders of magnitude smaller than the number of times the kernel function will be executed by a user specified number of threads. The latter can be specified through a so called execution configuration, which is an argument passed along to the GPU with each kernel function call. The execution configuration is defined by specifying the number of blocks of threads that the user desires to launch. The maximum number of blocks is $65,535 \times 65,535$; i.e., one can specify a two dimensional grid of blocks. Additionally, one has to indicate the number of threads that each block will be made up of. There is an upper limit of threads in a block, which currently is set to 512. When invoking an execution configuration, that is a grid of m blocks each with n of threads, the kernel function that is invoked to be executed on the device will be executed a number of $m \times n$ times. In terms of scheduling, the m blocks are assigned to the available SMs and therefore a high end GPU comes ahead since the m blocks will end up assigned to four SMs on an entry level GPU, or to 30 SMs on a high end GPU. The assignment of blocks to SMs might lead to the simultaneous execution of more than one block per SM. Yet, this number cannot be larger than eight, which is more than sufficient since when they land on the same SM the eight blocks of threads are supposed to share resources. Indeed, due to the limited number of registers and amount of shared memory available on a SM, a sharing of resources between many threads ($n\times$ the number of blocks executed on the SM) makes very unlikely the scenario of having a large number of blocks simultaneously running on one SM.

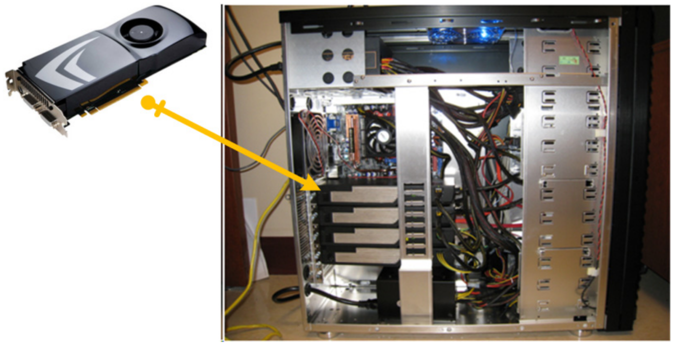

Figure 3. Image of GPU and desktop with a set of four cards that can be controlled by one CPU. There is no direct memory access between the four GPUs. The HW configuration in the figure is as follows. Processor: AMD Phenom II X4 940 Black Edition. Power supply 1:Silverstone OP1000-E 1000W. Power supply 2: Corsair CMPSU-750TX 750W. Memory: G.SKILL 16GB (4 x 4GB) 240-Pin DDR2. Case: LIAN LI PC-P80 ATX Full Tower. Motherboard: Foxconn Destroyer NVIDIA nForce 780a SLI. HDD: Western Digital Caviar Black 1TB 7200 RPM 3.0Gb/s. HSF: Stock AMD. Graphics: 4x NVIDIA Tesla C1060.

In terms of block scheduling, as one block of threads finishes the execution of the kernel function on a certain SM, another block of threads waiting to execute is assigned to the SM. Consequently, the device should be able to do scheduling at two levels. The first is associated with the assignment of a block to an SM that is ready to accept a new block for execution. What simplifies the scheduling here is the lack of time slicing associated with block execution: if a block is accepted for execution on an SM, no other block is accepted by that SM before it finishes the execution of a block that it is already dealing with. The second level of scheduling, which is more challenging, has to do with the scheduling for execution of one of the potentially 32 warps of threads that each SM can handle at any given time. Note that all the 32 threads in one warp execute the same instruction, even though this means, like in the case of if-then-else statements, serializing the code of the if-branches and running no-ops for certain threads in the warp (this thread divergence adversely impacts overall performance and should be avoided whenever possible). However, when switching between different warps, the SM typically executes different instructions when handling different warps; in other words, time slicing is present in thread execution.

In conclusion, one Tesla C1060, can be delegated with the execution of a kernel function up to approximately 2 trillion times. However, at each time, since there are 30 SMs available in this card, it will actively execute at most $30,720 = 30 \times 32$ warps $\times 32$ threads at any time. Moreover, as shown in Figure 3, existing motherboards can accommodate up to four Tesla C1060 cards, which

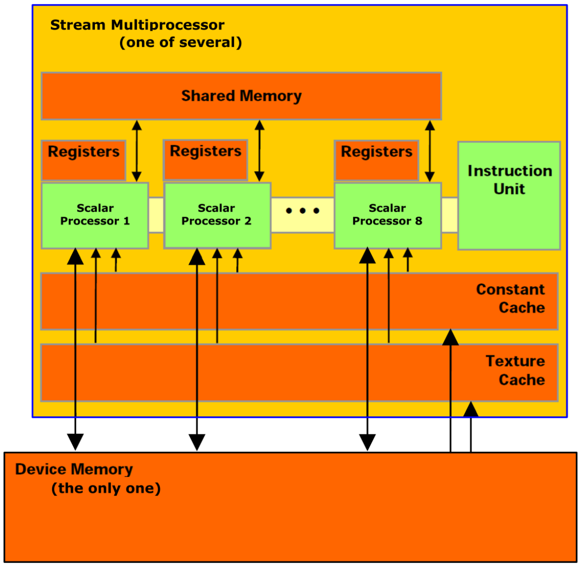

Figure 4. GPU memory layout. Device memory refers to the combination of global, texture, and constant memory spaces. Arrows indicate the way data can move between different memory spaces and SM. While the device memory is available to threads running on any SM, the registers, shared memory, and cached constant and texture data is specific to each SM.

effectively supports up to $122,880 = 4 \times 30,720$ threads being active at the same time. The single precision flop rate of this setup is approximately 3,600 billion operations per second.

It was alluded before that one of the factors that prevent an SM from actually running at full potential; i.e., managing simultaneously 32 warps of threads, is the exhaustion of shared memory and/or register resources. Each SM has 16 KB of shared memory in addition to 16,384 four byte registers. If the execution of the kernel function requires a large amount of either shared memory or registers, it is clear that the SM does not have enough memory available to host too many threads executing the considered kernel. Consequently, the ability of the SM to hide global memory access latencies with arithmetic instructions decreases since there are less warps that it can switch between.

In addition to shared memory and registers, each thread has access to global memory (4 GB of it on a Tesla C1060), constant memory (64 KB), and texture memory, the latter in an amount that is somewhat configurable but close to the amount of constant memory. Additionally, there is so called local memory used to store data that is not lucky enough to occupy a register and ends up in the global memory (register overflow). Effectively, local memory is virtual memory that is carved out of the global memory and, in spite of the word “local”, it is associated with high latency. In this context, accessing data in registers has practically no latency, shared memory transactions have less than four clock cycles of latency, as do cached constant and texture memory accesses. Global memory transactions are never cached and, just like un-cached constant and texture memory accesses or accesses to local memory, they incur latencies of the order of 600 clock cycles. Note that typically the device does not have direct access to host memory. There are ways to circumvent this by using mapped page-locked memory transactions, but this is an advanced feature not discussed here.

For the GPU to assist through co-processing a job run on the CPU, the host must first allocate memory and move data through the PCI connection into the device memory (global, texture, or constant memory spaces). Subsequently, a kernel function is launched on the GPU to process data that resides in the device memory. At that point, blocks of threads executing the kernel function access data stored in device memory. In unsophisticated kernels they can immediately process the data; alternatively, in more sophisticated kernel functions, they can use the shared memory and registers to store the data locally and thus avoid costly device memory accesses. If avoiding repeated data transfers between host and device is the first most important rule for effective GPU computing, avoiding repeated high-latency calls to device memory is the second most important rule to be observed in GPU computing. It should be pointed out that device memory access can be made even more costly when the access is not structured (uncoalesced). Using CUDA terminology, the device memory accesses result in multiple transactions if the data accessed by a warp of threads is scattered rather than nicely coalesced (contiguous) in memory. For more details, the interested reader is referred to [14].

One common strategy for avoiding race conditions in parallel computing is the synchronization of the execution at various points of the code. In CUDA, synchronization is possible but with a caveat. Specifically, threads that execute the kernel function yet belong to different blocks cannot be synchronized. This is a consequence of the earlier observation that there is no time slicing involved in block execution. When there are thousands of blocks that are lined up for execution waiting for their turn on one of the 30 SM of a Tesla C1060, it is clear that there can be no synchronization between a thread that belongs to the first block and one that belongs to the last block that might get executed much later and on a different SM. Overall synchronization can be obtained by breaking the algorithm in two kernel functions right at the point where synchronization is desired. Thus, after the execution of the first kernel the control is rendered back to the host, which upon the invocation of the subsequent kernel ensures that all threads start on equal footing. This approach is feasible since the device memory is persistent between subsequent kernel calls as long as they are made by the same host process. The strategy works albeit at a small computational cost as there is an overhead associated with each kernel call. Specifically, the overhead of launching a kernel for execution is on average between 90 microseconds (when no function arguments are present) and 120 microseconds (when arguments such as pointers to device memory are passed in the kernel argument list).

Looking ahead, the next generation of GPU hardware and CUDA software will make the heterogeneous computing model, where some tasks are executed by the host and other compute intensive parts of the code are delegated to the GPU, even more attractive. Slated to be released by March 2010, the Fermi family of GPUs will have 512 SPs in one SM and up to 1 TB of fast Graphics Double Data Rate, version 5 (GDDR5) memory. Moreover, the current weak double precision performance of the GPU (about eight times slower than single precision peak performance) will be improved to clock at half the value of the single precision peak performance. Finally, on the software side, the CUDA run-time API will provide (a) support for stream computing where expensive host-device data moving operations can be overlapped with kernel execution, and (b) a mechanism to simultaneously execute on the device different kernels that are data independent. It becomes apparent that if used for the right type of applications, that is, when the execution bottleneck fits the SIMD computational model, and if used right, GPU computing can lead to impressive reductions in computational time. Combined with its affordability attribute, GPU computing will allow scientific computing to tackle large problems that in the past fell outside the realm of tractable problems. The class of granular dynamics problems is one such example, where a discrete approach to equation formulation and solution was not feasible in most cases in the past.

3. Large scale Multibody Dynamics on the GPU

This section briefly introduces the theoretical background for mechanical systems made up of multiple rigid bodies whose time evolution is controlled by external forces, frictional contacts, bilateral constraints and motors.

3.1 The formulation of the Equations of Motion

The state of a mechanical system with n_b rigid bodies in three dimensional space can be represented by the generalized coordinates

$$\mathbf{q} = \left[\mathbf{r}_1^T, \epsilon_1^T, \ldots, \mathbf{r}_{n_b}^T, \epsilon_{n_b}^T\right]^T \in \mathbb{R}^{7n_b}$$

and their time derivatives

$$\dot{\mathbf{q}} = \left[\dot{\mathbf{r}}_1^T, \dot{\epsilon}_1^T, \ldots, \dot{\mathbf{r}}_{n_b}^T, \dot{\epsilon}_{n_b}^T\right]^T \in \mathbb{R}^{7n_b},$$

where $\mathbf{r}_j$ is the absolute position of the center of mass of the j-th body and the quaternion ϵ_j expresses its rotation. One can also introduce the generalized velocities $\mathbf{v} = \left[\dot{\mathbf{r}}_1^T, \bar{\omega}_1^T, \ldots, \dot{\mathbf{r}}_{n_b}^T, \bar{\omega}_{n_b}^T\right]^T \in \mathbb{R}^{6n_b}$, directly related to $\dot{\mathbf{q}}$ by means of the linear mapping $\dot{\mathbf{q}} = \mathbf{L}(\mathbf{q})\mathbf{v}$ that transforms each angular velocity $\bar{\omega}_i$ (expressed in the local coordinates of the body) into the corresponding quaternion derivative $\dot{\epsilon}_i$ by means of the linear algebra formula $\dot{\epsilon}_i = \frac{1}{2}G(\epsilon_j)\bar{\omega}_i$, with

$$G(\epsilon_j) = \begin{bmatrix} +\epsilon_1 & +\epsilon_0 & -\epsilon_3 & +\epsilon_2 \\ +\epsilon_2 & +\epsilon_3 & +\epsilon_0 & -\epsilon_1 \\ +\epsilon_3 & -\epsilon_2 & +\epsilon_1 & +\epsilon_0 \end{bmatrix}.$$

Mechanical constraints, such as revolute or prismatic joints, can exist between the parts: they translate into algebraic equations that constrain the relative position of pairs of bodies. Assuming a set $\mathcal{B}$ of constraints is present in the system, they lead to the scalar equations

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

To ensure that constraints are not violated in terms of velocities, one must also satisfy the first derivative of the constraint equations, that is

$$\nabla\Psi_i^T \mathbf{v} + \frac{\partial \Psi_i}{\partial t} = 0, \quad i \in \mathcal{B}$$

with the Jacobian matrix $\nabla_q \Psi_i = [\partial\Psi_i / \partial\mathbf{q}]^T$ and $\nabla\Psi_i^T = \nabla_q \Psi_i^T \mathbf{L}(\mathbf{q})$. Note that the term $\partial\Psi_i/\partial t$ is null for all scleronomic constraints, but it might be nonzero for constraints that impose some trajectory or motion law, such as in the case of motors and actuators.

If contacts between rigid bodies must be taken into consideration, colliding shapes must be defined for each body. A collision detection algorithm must be used to provide a set of pairs of contact points for bodies whose shapes are near enough, so that a set $\mathcal{A}$ of inequalities can be used to concisely express the non-penetration condition between the volumes of the shapes:

$$\Phi_i(\mathbf{q}) \geq 0, \quad i \in \mathcal{A}$$

Note that for curved convex shapes, such as spheres and ellipsoids, there is a unique pair of contact points, that is the pair of closest points on their surfaces, but in case of faceted or non-convex shapes there might be multiple pairs of contact points, whose definition is not always trivial and whose set may be discontinuous.

Given two bodies in contact $A, B \in \{1, 2, \ldots, n_b\}$ let $\mathbf{n}_i$ be the normal at the contact pointing toward the exterior of body A, and let $\mathbf{u}_i$ and $\mathbf{w}_i$ be two vectors in the contact plane such that $\mathbf{n}_i, \mathbf{u}_i, \mathbf{w}_i \in \mathbb{R}^3$ are mutually orthogonal vectors. When a contact i is active, that is, for $\Phi_i(\mathbf{q}) = 0$, the frictional contact force acts on the system by means of multipliers $\widehat{\gamma}_{i,n} \geq 0$, $\widehat{\gamma}_{i,u}$, and $\widehat{\gamma}_{i,w}$. Specifically, the normal component of the contact force acting on body B is $\mathbf{F}_{i,N} = \widehat{\gamma}_{i,n}\mathbf{n}_i$ and the tangential component is $\mathbf{F}_{i,T} = \widehat{\gamma}_{i,u}\mathbf{u}_i + \widehat{\gamma}_{i,w}\mathbf{w}_i$ (for body A these forces have the opposite sign).

Also, according to the Coulomb friction model, in case of nonzero relative tangential speed, $\mathbf{v}_{i,T}$, the direction of the tangential contact force is aligned to $\mathbf{v}_{i,T}$ and it is proportional to the normal force as $||\mathbf{F}_{i,T}|| = \mu_{i,d}||\mathbf{F}_{i,N}||$ by means of the dynamic friction coefficient $\mu_{i,d} \in \mathbb{R}^+$. However, in case of null tangential speed, the strength of the tangential force is limited by the inequality $||\mathbf{F}_{i,T}|| \leq \mu_{i,s}||\mathbf{F}_{i,N}||$ using a static friction coefficient $\mu_{i,s} \in \mathbb{R}^+$, and its direction is one of the infinite tangents to the surface. In our model we assume that $\mu_{i,d}$ and $\mu_{i,s}$ have the same value that we will write μ_i for simplicity, so the abovementioned Coulomb model can be stated succinctly as follows:

$$\begin{array}{rcl} \widehat{\gamma}_{i,n} & \geq & 0, \quad \Phi_i(\mathbf{q}) \geq 0, \quad \Phi_i(\mathbf{q})\widehat{\gamma}_{i,n} = 0, \\ \mu_i\widehat{\gamma}_{i,n} & \geq & \sqrt{\widehat{\gamma}_{i,u}^2 + \widehat{\gamma}_{i,w}^2} \\ & & \langle \mathbf{F}_{i,T}, \mathbf{v}_{i,T} \rangle = - \left\|\mathbf{F}_{i,T}\right\| \; \left\|\mathbf{v}_{i,T}\right\| \\ & & \left\|\mathbf{v}_{i,T}\right\| \left(\mu_i\widehat{\gamma}_{i,n} - \sqrt{\widehat{\gamma}_{i,u}^2 + \widehat{\gamma}_{i,w}^2} \right) = 0 \end{array}$$

Note that the condition $\widehat{\gamma}_{i,n} \geq 0, \Phi_i(\mathbf{q}) \geq 0, \Phi_i(\mathbf{q})\widehat{\gamma}_{i,n} = 0$ can also be written as a complementarity constraint: $\widehat{\gamma}_{i,n} \geq 0 \perp \Phi_i(\mathbf{q}) \geq 0$, see [15]. This model can also be interpreted as the Karush-Kuhn-Tucker first order conditions of the following equivalent maximum dissipation principle [16-17]:

$$(\widehat{\gamma}_{i,u}, \widehat{\gamma}_{i,w}) = \underset{\sqrt{\widehat{\gamma}_{i,u}^2 + \widehat{\gamma}_{i,w}^2} \leq \mu_i\widehat{\gamma}_{i,n}}{\operatorname{argmin}} \mathbf{v}_{i,T}^T \left(\widehat{\gamma}_{i,u}\mathbf{u}_i + \widehat{\gamma}_{i,w}\mathbf{w}_i \right). \tag{1}$$

Finally, one should also consider the effect of external forces with the vector of generalized forces $\mathbf{f}(t, \mathbf{q}, \mathbf{v}) \in \mathbb{R}^{6n_b}$, that might contain gyroscopic terms, gravitational effects, forces exerted by springs or dampers, and torques applied by motors; i.e. all forces except joint reaction and frictional contact forces.

Considering the effects of both the set $\mathcal{A}$ of frictional contacts and the set $\mathcal{B}$ of bilateral constraints, the system cannot be reduced to either a set ordinary differential equations (ODEs) of the type $\dot{\mathbf{v}} = f(\mathbf{q}, \mathbf{v}, t)$, or to a set of differential-algebraic equation (DAEs). This is because the inequalities and the complementarity constraints turn the system into a differential inclusion of the type $\dot{\mathbf{v}} \in \mathcal{F}(\mathbf{q}, \mathbf{v}, t)$, where $\mathcal{F}(\cdot)$ is a set-valued multifunction [18]. In fact, the time evolution of the dynamical system is governed by the following differential variational inequality (DVI):

$$
\begin{array}{rcl}
\dot{\mathbf{q}} & = & \mathbf{L}(\mathbf{q})\mathbf{v} \\
\mathbf{M}\dot{\mathbf{v}} & = & \mathbf{f}\left(t,\mathbf{q},\mathbf{v}\right) + \sum\limits_{i\in\mathcal{B}} \widehat{\gamma}_{i,b}\nabla\Psi_i + \\
 & & + \sum\limits_{i\in\mathcal{A}} \left(\widehat{\gamma}_{i,n}\,\mathbf{D}_{i,n} + \widehat{\gamma}_{i,u}\,\mathbf{D}_{i,u} + \widehat{\gamma}_{i,w}\,\mathbf{D}_{i,w}\right) \\
i\in\mathcal{B} & : & \Psi_i(\mathbf{q},t) = 0 \\
i\in\mathcal{A} & : & \widehat{\gamma}_{i,n} \geq 0 \;\perp\; \Phi_i(\mathbf{q}) \geq 0, \qquad \text{and} \\
\left(\widehat{\gamma}_{i,u},\widehat{\gamma}_{i,w}\right) & = & \underset{\mu_i\widehat{\gamma}_{i,n}\geq\sqrt{\widehat{\gamma}_{i,u}^2+\widehat{\gamma}_{i,w}^2}}{\operatorname{argmin}} \mathbf{v}^T\left(\widehat{\gamma}_{i,u}\,\mathbf{D}_{i,u} + \widehat{\gamma}_{i,w}\,\mathbf{D}_{i,w}\right)
\end{array}
\qquad (2)
$$

Here, to express the contact forces in generalized coordinates, we used the tangent space generators $D_i = [\,\mathbf{D}_{i,n},\,\mathbf{D}_{i,u},\,\mathbf{D}_{i,w}] \in \mathbb{R}^{6n_b\times 3}$ that are sparse and are defined given a pair of contacting bodies A and B as:

$$
D_i^T = \begin{array}{cccccc}
[\mathbf{0} & \ldots & -A_{i,p}^T & A_{i,p}^T A_A \tilde{\bar{\mathbf{s}}}_{i,A} & \mathbf{0} & \ldots \\
\mathbf{0} & \ldots & A_{i,p}^T & -A_{i,p}^T A_B \tilde{\bar{\mathbf{s}}}_{i,B} & \mathbf{0} & \ldots]
\end{array}
\qquad (3)
$$

Here $A_{i,p} = [\mathbf{n}_i, \mathbf{u}_i, \mathbf{w}_i]$ is the $\mathbb{R}^{3\times 3}$ matrix of the local coordinates of the i-th contact, and the vectors $\bar{\mathbf{s}}_{i,A}$ and $\bar{\mathbf{s}}_{i,B}$ to represent the positions of the contact points expressed in body coordinates. The skew matrices $\tilde{\bar{\mathbf{s}}}_{i,A}$ and $\tilde{\bar{\mathbf{s}}}_{i,B}$ are defined as

$$
\tilde{\bar{\mathbf{s}}}_{i,A} = \begin{bmatrix} 0 & -s_{i,A_z} & +s_{i,A_y} \\ +s_{i,A_z} & 0 & -s_{i,A_x} \\ -s_{i,A_y} & +s_{i,A_x} & 0 \end{bmatrix}, \tilde{\bar{\mathbf{s}}}_{i,B} = \begin{bmatrix} 0 & -s_{i,B_z} & +s_{i,B_y} \\ +s_{i,B_z} & 0 & -s_{i,B_x} \\ -s_{i,B_y} & +s_{i,B_x} & 0 \end{bmatrix}
$$

The DVI in (2) can be solved by time-stepping methods. The discretization requires the solution of a complementarity problem at each time step, and it has been demonstrated that it converges to the solution to the original differential inclusion for $h \rightarrow 0$ [15, 19]. Moreover, the differential inclusion can be solved in terms of vector measures: forces can be impulsive and velocities can have discontinuities, thus supporting also the case of impacts and giving a weak solution to otherwise unsolvable situations like in the Painlevé paradox [20].

3.2 The time stepping solver

Within the aforementioned measure differential inclusion approach, the unknowns are not the reaction forces and the accelerations $\dot{\mathbf{v}}$ as in usual ODEs or DAEs. Instead, given a position $\mathbf{q}^{(l)}$ and velocity $\mathbf{v}^{(l)}$ at the time step $t^{(l)}$, the unknowns are the impulses γ_s, for $s = n, u, w, b$ (that, for smooth constraints, can be interpreted as $\hat{\gamma}_n = h\gamma_n$, $\hat{\gamma}_u = h\gamma_u$, $\hat{\gamma}_w = h\gamma_w$, $\hat{\gamma}_b = h\gamma_b$) and the speeds $\mathbf{v}^{(l+1)}$ at the new time step $t^{(l+1)} = t^{(l)} + h$. These unknowns are obtained by solving the following optimization problem with equilibrium constraints [2]:

$$\begin{aligned}
\mathbf{M}(\mathbf{v}^{(l+1)} & -\mathbf{v}^{(l)}) = h\mathbf{f}(t^{(l)}, \mathbf{q}^{(l)}, \mathbf{v}^{(l)}) + \sum_{i \in \mathcal{B}} \gamma_{i,b} \nabla \Psi_i + \\
& + \textstyle\sum_{i \in \mathcal{A}} \left(\gamma_{i,n}\, \mathbf{D}_{i,n} + \gamma_{i,u}\, \mathbf{D}_{i,u} + \gamma_{i,w}\, \mathbf{D}_{i,w} \right), \\
i \in \mathcal{B} : & \quad \tfrac{1}{h} \Psi_i(\mathbf{q}^{(l)}, t) + \nabla \Psi_i^T \mathbf{v}^{(l+1)} + \tfrac{\partial \Psi_i}{\partial t} = 0 \\
i \in \mathcal{A} : & \quad 0 \le \tfrac{1}{h} \Phi_i(\mathbf{q}^{(l)}) + \mathbf{D}_{i,n}^T \mathbf{v}^{(l+1)} \perp \gamma_n^i \ge 0, \\
(\gamma_{i,u}, \gamma_{i,w}) & = \underset{\mu_i \gamma_{i,n} \ge \sqrt{\gamma_{i,u}^2 + \gamma_{i,w}^2}}{\operatorname{argmin}} \quad \mathbf{v}^T \left(\gamma_{i,u}\, \mathbf{D}_{i,u} + \gamma_{i,w}\, \mathbf{D}_{i,w} \right) \\
\mathbf{q}^{(l+1)} = & \quad \mathbf{q}^{(l)} + h\mathbf{L}(\mathbf{q}^{(l)}) \mathbf{v}^{(l+1)}. \qquad (4)
\end{aligned}$$

The $\frac{1}{h}\Phi_i(\mathbf{q}^{(l)})$ term is introduced to ensure contact stabilization, and its effect is discussed in [3]. Similarly, the term $\frac{1}{h}\Psi_i(\mathbf{q}^{(l)})$ achieves stabilization for bilateral constraints.

Several numerical methods can be used to solve (4). For instance, one can approximate the Coulomb friction cones in 3D as faceted pyramids, thus leading to a LCP whose solution is possible by using off-the-shelf pivoting methods. However, these methods usually require a large computational overhead and can be used only for a limited number of variables.

Therefore, in a previous work [21] we demonstrated that the problem can be cast as a monotone optimization problem by introducing a relaxation over the complementarity constraints, replacing $0 \le \frac{1}{h}\Phi_i(\mathbf{q}^{(l)}) + \mathbf{D}_{i,n}^T \mathbf{v}^{(l+1)} \perp \gamma_n^i \ge 0$ with $0 \le \frac{1}{h}\Phi_i(\mathbf{q}^{(l)}) + \mathbf{D}_{i,n}^T \mathbf{v}^{(l+1)} - \mu_i \sqrt{(\mathbf{v}^T \mathbf{D}_{i,u})^2 + (\mathbf{v}^T \mathbf{D}_{i,w})^2} \perp \gamma_n^i \ge 0$. The solution of the modified time stepping scheme approaches the solution of the original differential inclusion for $h \to 0$ just as the original scheme [3]. Most importantly, the modified scheme becomes a Cone Complementarity Problem (CCP), which can be solved efficiently by an iterative numerical method that relies on projected contractive maps. Omitting for brevity some of the details discussed in [22], the algorithm makes use of the following vectors and matrices:

$$\begin{aligned}
\gamma_{i,a} & \equiv \{\gamma_{i,n}, \gamma_{i,u}, \gamma_{i,w}\}^T, \quad i \in \mathcal{A}, \\
\mathbf{b}_i & \equiv \left\{ \tfrac{1}{h}\Phi_i(\mathbf{q}^{(l)}), 0, 0 \right\}^T, \quad i \in \mathcal{A}, \\
b_i & \equiv \tfrac{1}{h}\Psi_i(\mathbf{q}^{(l)}, t) + \tfrac{\partial \Psi_i}{\partial t}, \quad i \in \mathcal{B} \qquad (5)
\end{aligned}$$

The solution of the CCP is obtained by iterating the following expressions on r until convergence, or until r exceeds a maximum amount of iterations, starting from $\mathbf{v}^0 = \mathbf{v}^{(l)}$:

$$\forall i \in \mathcal{A} : \quad \gamma_{i,a}^{r+1} = \Pi_{\Upsilon_i} \left[\gamma_{i,a}^r - \omega \eta_i \left(D_i^T \mathbf{v}^r + \mathbf{b}_i \right) \right] \qquad (6)$$

$$\forall i \in \mathcal{B} : \quad \gamma_{i,b}^{r+1} = \quad \gamma_{i,b}^r - \omega \eta_i \left(\nabla \Psi_i^T \mathbf{v}^r + b_i \right) \qquad (7)$$

$$\mathbf{v}^{r+1} = \mathbf{v}^r + M^{-1} \left(\sum_{z \in \mathcal{A}} D_z \gamma_{z,a}^{r+1} + \sum_{z \in \mathcal{B}} \nabla \Psi_z \gamma_{z,b}^{r+1} + h\, \mathbf{f}(t^{(l)}, \mathbf{q}^{(l)}, \mathbf{v}^{(l)}) \right) \qquad (8)$$

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

The iterative process uses the projector $\Pi_{\Upsilon_i}(\cdot)$, which is a non-expansive metric map $\Pi_{\Upsilon_i} : \mathbb{R}^3 \to \mathbb{R}^3$ acting on the triplet of multipliers associated with the i-th contact [21]. In detail, if the multipliers fall into the friction cone

$$\Upsilon_i = \{\gamma_{i,a} \in \mathbb{R}^3 \;:\; (\gamma_{i,u}^2 + \gamma_{i,w}^2)^{1/2} \leq \mu_i \gamma_{i,n}\}$$

they are not modified; if they are in the polar cone

$$\Upsilon_i^{\circ} = \{\mathbf{x}_i \in \mathbb{R}^3 \;:\; \langle \mathbf{x}_i, \gamma_{i,a} \rangle \leq 0, \; \forall \gamma_{i,a} \in \Upsilon_i\}$$

they are set to zero; in the remaining cases they are projected orthogonally onto the surface of the friction cone. The over-relaxation factor ω and η_i parameters are adjusted to control the convergence. Interested readers are referred to [22] for a proof of the convergence of this method.

For improved performance, the summation of Eq. (8) can be computed only once at the beginning of the CCP iteration, while the following updates can be performed using an incremental version that avoids adding the $\mathbf{f}(t^{(l)}, \mathbf{q}^{(l)}, \mathbf{v}^{(l)})$ term all the time; in case there is no initial guess for the multipliers and $\gamma_{i,b}^0 = 0$, $\gamma_{i,a}^0 = 0$, Eq. (8) turns into:

$$\mathbf{v}^0 = \mathbf{v}^{(l)} + M^{-1} h \, \mathbf{f}(t^{(l)}, \mathbf{q}^{(l)}, \mathbf{v}^{(l)}) \tag{9}$$

$$\mathbf{v}^{r+1} = \mathbf{v}^r + \sum \Delta \mathbf{v}_i \tag{10}$$

where

$$i \in \mathcal{A}: \quad \Delta \mathbf{v}_i = \sum_{i \in \mathcal{A}} M^{-1} D_i \Delta \gamma_{i,a}^{r+1}$$
$$i \in \mathcal{B}: \quad \Delta \mathbf{v}_i = \sum_{i \in \mathcal{B}} M^{-1} \nabla \Psi_i \Delta \gamma_{i,b}^{r+1}$$

In the case that only bilateral constraints are used, this method behaves like the typical fixed-point Jacobi iteration for the solution of linear problems. If one interleaves the update (8) after each time that a single *i*-th multiplier is computed in (6) or (7), the resulting scheme behaves like a Gauss-Seidel method. This variant can benefit from the use of Eq. (10) instead of Eq. (8) because it can increment only the $\Delta \mathbf{v}_i$ term corresponding to the constraint that has been just computed. Also, this immediate update of the speed vector provides better properties of convergence (especially in case of redundant constraints) but it does not fit well in a parallel computing environment because of its inherently sequential nature.

3.3 The GPU formulation of the CCP solver

Since the CCP iteration is a computational bottleneck of the numerical solution proposed, a great benefit will follow from an implementation that can take advantage of the parallel computing resources available on GPU boards.

In the proposed approach, the data structures on the GPU are implemented as large arrays (*buffers*) to match the execution model associated with NVIDIA's CUDA. Specifically, threads are grouped in rectangular thread blocks, and thread blocks are arranged in rectangular grids. Four main buffers are used: the contacts buffer, the constraints buffer, the reduction buffer, and the bodies buffer. Since repeated transfers of large data structures can adversely impact the performance of the entire algorithm, an attempt was made to organize the data structures in a way that minimized the number of fetch and store operations and maximized the arithmetic intensity of the kernel code. This ensures that the latency of the global memory can be hidden by the hardware multithread scheduler if the GPU code interleaves the memory access with enough arithmetic instructions.

Figure 5 shows the data structure for contacts, which contains two pointers B_A and B_B to the two touching bodies. There is no need to store the entire D_i matrix for the i-th contact because it has zero entries everywhere except for the two 12x3 blocks corresponding to the coordinates of the two bodies in contact. In detail, we store only the following 3x3 matrices:

$$\begin{array}{rclrcl} D^T_{i,v_A} & = & -A^T_{i,p}, & D^T_{i,\omega_A} & = & A^T_{i,p} A_A \tilde{\bar{\mathbf{s}}}_{i,A} \\ D^T_{i,v_B} & = & A^T_{i,p}, & D^T_{i,\omega_B} & = & -A^T_{i,p} A_B \tilde{\bar{\mathbf{s}}}_{i,B} \end{array}$$

Once the velocities of the two bodies $\dot{\mathbf{r}}_{A_i}$, $\bar{\omega}_{A_i}$, $\dot{\mathbf{r}}_{B_i}$ and $\bar{\omega}_{B_i}$ have been fetched, the product $D_i^T \mathbf{v}^r$ in Eq.(6) can be performed as

$$D_i^T \mathbf{v}^r = D^T_{i,v_A} \dot{\mathbf{r}}_{A_i} + D^T_{i,\omega_A} \bar{\omega}_{A_i} + D^T_{i,v_B} \dot{\mathbf{r}}_{B_i} + D^T_{i,\omega_B} \bar{\omega}_{B_i} \tag{11}$$

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

Also, the velocity update vector $\Delta \mathbf{v}_i$, needed for the sum in Eq. (10) is sparse: it can be decomposed in small 3x1 vectors. Specifically, given the masses and the inertia tensors of the two bodies m_{A_i}, m_{B_i}, J_{A_i} and J_{B_i}, the term $\Delta \mathbf{v}_i$ will be computed and stored in four parts as follows:

$$\begin{array}{ll} \Delta \dot{\mathbf{r}}_{A_i} = m^{-1}_{A_i} D_{i,v_A} \Delta \gamma^{r+1}_{i,a}, & \Delta \bar{\omega}_{A_i} = J^{-1}_{A_i} D_{i,\omega_A} \Delta \gamma^{r+1}_{i,a} \\ \Delta \dot{\mathbf{r}}_{B_i} = m^{-1}_{B_i} D_{i,v_B} \Delta \gamma^{r+1}_{i,a}, & \Delta \bar{\omega}_{B_i} = J^{-1}_{B_i} D_{i,\omega_B} \Delta \gamma^{r+1}_{i,a} \end{array} \tag{12}$$

Note that those four parts of the $\Delta \mathbf{v}_i$ terms are not stored in the i-th contact or data structures of the two referenced bodies (because multiple contacts may refer the same body, hence they would overwrite the same memory position). These velocity updates are instead stored in the reduction buffer, which will be used to efficiently perform the summation in Eq. (10). This will be discussed shortly.

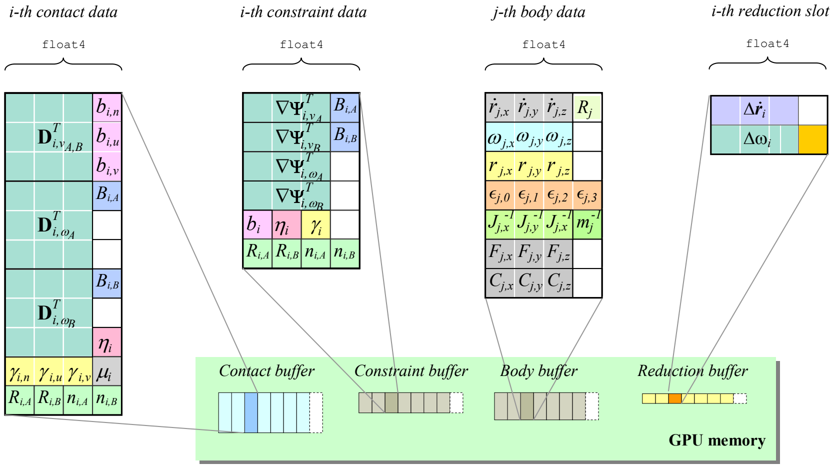
i-th contact data
i-th constraint data
j-th body data
i-th reduction slot
float4
float4
float4
float4
$\mathbf{D}^T_{i,v_{A,B}}$
$b_{i,n}$
$b_{i,u}$
$b_{i,v}$
$\mathbf{D}^T_{i,\omega_A}$
$B_{i,A}$
$\mathbf{D}^T_{i,\omega_B}$
$B_{i,B}$
η_i
$\gamma_{i,n}$ $\gamma_{i,u}$ $\gamma_{i,v}$ μ_i
$R_{i,A}$ $R_{i,B}$ $n_{i,A}$ $n_{i,B}$
$\nabla\Psi^T_{i,v_A}$ $B_{i,A}$
$\nabla\Psi^T_{i,v_B}$ $B_{i,B}$
$\nabla\Psi^T_{i,\omega_A}$
$\nabla\Psi^T_{i,\omega_B}$
b_i η_i γ_i
$R_{i,A}$ $R_{i,B}$ $n_{i,A}$ $n_{i,B}$
$\dot{r}_{j,x}$ $\dot{r}_{j,y}$ $\dot{r}_{j,z}$ R_j
$\omega_{j,x}$ $\omega_{j,y}$ $\omega_{j,z}$
$r_{j,x}$ $r_{j,y}$ $r_{j,z}$
$\epsilon_{j,0}$ $\epsilon_{j,1}$ $\epsilon_{j,2}$ $\epsilon_{j,3}$
$J^{-1}_{j,x}$ $J^{-1}_{j,y}$ $J^{-1}_{j,x}$ m^{-1}_j
$F_{j,x}$ $F_{j,y}$ $F_{j,z}$
$C_{j,x}$ $C_{j,y}$ $C_{j,z}$
$\Delta\dot{r}_i$
$\Delta\omega_i$
Contact buffer
Constraint buffer
Body buffer
Reduction buffer
GPU memory

Figure 5. Data structures in GPU global memory

The constraints buffer, shown in Figure 5, is based on a similar concept. Jacobians $\nabla\Psi_i$ of all scalar constraints are stored in a sparse format, each corresponding to four rows $\nabla\Psi_{i,v_A}$, $\nabla\Psi_{i,\omega_A}$, $\nabla\Psi_{i,v_B}$, $\nabla\Psi_{i,\omega_B}$. Therefore the product $\nabla\Psi_i^T\mathbf{v}^r$ in Eq.(7) can be performed as the scalar value:

$$\nabla\Psi_i^T\mathbf{v}^r = \nabla\Psi_{i,v_A}^T\dot{\mathbf{r}}_{A_i} + \nabla\Psi_{i,\omega_A}^T\omega_{A_i} + \nabla\Psi_{i,v_B}^T\dot{\mathbf{r}}_{B_i} + \nabla\Psi_{i,\omega_B}^T\omega_{B_i} \qquad (13)$$

Also, the four parts of the sparse vector $\Delta\mathbf{v}_i$ can be computed and stored as

$$\begin{aligned} \Delta\dot{\mathbf{r}}_{A_i} &= m_{A_i}^{-1}\nabla\Psi_{i,v_A}\Delta\gamma_{i,b}^{r+1}, \quad & \Delta\bar{\omega}_{A_i} &= J_{A_i}^{-1}\nabla\Psi_{i,\omega_A}\Delta\gamma_{i,b}^{r+1} \\ \Delta\dot{\mathbf{r}}_{B_i} &= m_{B_i}^{-1}\nabla\Psi_{i,v_B}\Delta\gamma_{i,b}^{r+1}, \quad & \Delta\bar{\omega}_{B_i} &= J_{B_i}^{-1}\nabla\Psi_{i,\omega_B}\Delta\gamma_{i,b}^{r+1} \end{aligned} \qquad (14)$$

Figure 5 shows that each body is represented by a data structure containing the state (velocity and position), the mass moments of inertia and mass values, and the external applied force $\mathbf{F}_j$ and torque $\mathbf{C}_j$. Those data are needed to compute the CCP iteration and solve for unknowns.

When it comes to the implementation of the CCP solver on the GPU, using kernels that operate on the abovementioned data buffers, the task is not trivial because the iteration cannot be performed with a single kernel. In fact, considering the iteration over Eqs. (6), (7), and (10), one can see that Eqs. (6) and (7) fit into parallel kernels that operate, respectively, one thread per contact and one thread per bilateral constraint. Moreover, the summation in Eq. (10) cannot be easily parallelized in the same way because it may happen that two or more contacts need to add their velocity updates $\Delta\mathbf{v}_i$ to the same rigid body: this would cause a race condition where multiple threads might need to update the same memory value, something that can cause errors or indefinite/nondeterministic behaviors on the

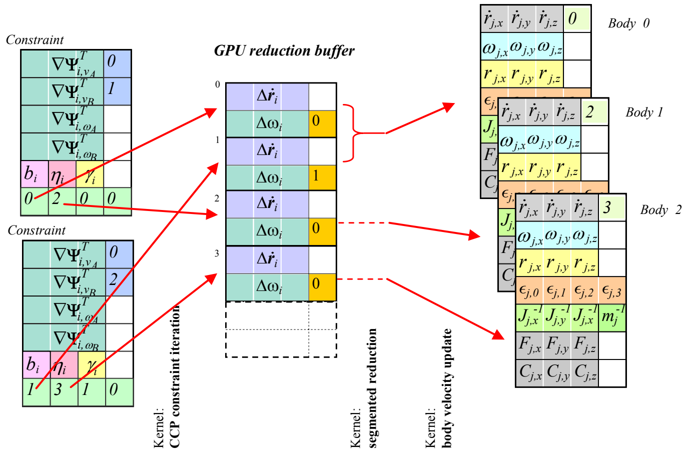

Figure 6. Example of reduction buffer for summing up body velocities.

GPU hardware. Therefore, in order to parallelize Eq. (10), a parallel segmented scan algorithm [23] was adopted that operates on an intermediate reduction buffer (see Figure 6); this method sums the values in the buffer using a binary-tree approach that keeps the computational load well balanced among the many processors. In the example of Figure 6, the first constraint refers to bodies 0 and 1, the second to bodies 0 and 2; multiple updates to body 0 are then accumulated with parallel a segmented reduction.

Note that several other auxiliary kernels that have minimal impact on the computation time are used to prepare pre-process data before the CCP starts, for example to compute Eq. (9). Also, to speed up the computation, matrices D^T_{i,v_A}, D^T_{i,ω_A} and D^T_{i,ω_B} are not provided by the host; instead they are computed on the GPU using the data coming from the collision detection code, that is, $\bar{\mathbf{s}}_{i,A}$, $\bar{\mathbf{s}}_{i,B}$ and $\mathbf{n}_i$.

The following pseudocode shows the sequence of the main computational stages at each time step, which for the most part are executed as parallel kernels on the GPU.

Table 2. Pseudocode for the CCP solver

Stage	Context	Operations / kernels
1	HOST *Serial*	**Copy memory CPU→ GPU** Copy contact and body data structures from host memory to GPU buffers. Copy constraint data (residuals b_i and Jacobians) into the constraint buffer.

2	GPU *Parallel on* ***bodies***	**Force kernel** For each body, compute forces $\mathbf{f}(t^{(l)}, \mathbf{q}^{(l)}, \mathbf{v}^{(l)})$, if any. Store these forces and torques into $\mathbf{F}_j$ and $\mathbf{C}_j$.
3	GPU *Parallel on* ***contacts***	**Contact preprocessing kernel** For each contact, given contact normal and position, compute in place the matrices D^T_{i,v_A}, D^T_{i,ω_A} and D^T_{i,ω_B}, then compute η_i and the contact residual $\mathbf{b}_i = \{\frac{1}{h}\Phi_i(\mathbf{q}), 0, 0\}^T$.
4	GPU *Parallel on* ***bodies***	**CCP force kernel** For each body j, initialize body velocities: $\dot{\mathbf{r}}_j^{(l+1)} = h\, m_j^{-1}\mathbf{F}_j$ and $\bar{\omega}_j^{(l+1)} = h\, J_j^{-1}\mathbf{C}_j$.
5	GPU *Parallel on* ***contacts***	**CCP contact iteration kernel** For each contact i, do $\gamma^{r+1}_{i,a} = \Pi_{\Upsilon_i}[\gamma^r_{i,a} - \omega\eta_i \left(D_i^T \mathbf{v}^r + \mathbf{b}_i \right)]$. Note that $D_i^T \mathbf{v}^r$ is evaluated with sparse data, using Eq.(11). Store $\Delta\gamma^{r+1}_{i,a} = \gamma^{r+1}_{i,a} - \gamma^r_{i,a}$ in contact buffer. Use Eq.(12) to compute sparse updates $\Delta\dot{\mathbf{r}}$ and $\Delta\bar{\omega}$ to the velocities of the two connected bodies A and B, and store them in the $R_{i,A}$ and $R_{i,B}$ slots of the reduction buffer.
6	GPU *Parallel on* ***constraints***	**CCP constraint iteration kernel**. For each constraint i, do $\gamma^{r+1}_{i,b} = \gamma^r_{i,b} - \omega\eta_i \left(\nabla\Psi_i^T \mathbf{v}^r + b_i \right)$. Note that $\nabla\Psi_i^T \mathbf{v}^r$ is evaluated with sparse data, using Eq.(11). Store $\Delta\gamma^{r+1}_{i,b} = \gamma^{r+1}_{i,b} - \gamma^r_{i,b}$ in contact buffer. Use Eq.(14) to compute sparse updates $\Delta\dot{\mathbf{r}}$ and $\Delta\bar{\omega}$ to the velocities of the two connected bodies A and B, and store them in the $R_{i,A}$ and $R_{i,B}$ slots of the reduction buffer.
7	GPU *Parallel on* ***reduction slots***	**Segmented reduction kernel**. Sum all the $\Delta\dot{\mathbf{r}}$ and $\Delta\bar{\omega}$ terms belonging to the same body, in the reduction buffer. This may require a sequence of short kernels.
8	GPU *Parallel on* ***bodies***	**Body velocity updates kernel**. For each j body, add the cumulative velocity updates which can be fetched from the reduction buffer, using the index R_j.
9	HOST *Serial*	*Check convergence and repeat from stage 5 if convergence tolerance is not reached*

10	HOST *Serial*	**Copy memory GPU→ CPU** Copy contact multipliers from GPU buffers to host memory, if interested in reaction forces. Copy constraints multipliers from GPU buffers to host memory, if interested in reaction forces. Copy rigid body velocities from GPU buffers to host memory.

Stages 1 and 10 can be avoided if one manages to keep all the data on the GPU, by letting the collision detection engine communicate with the CCP solver directly. Even if those memory transfers are executed only at the beginning and at the end of the CCP solution process, their impact on the overall simulation time might be significant.

4. Numerical Experiments

The largest simulation run to date using the CCP-based GPU solver discussed herein contained approximately 1.1 million bodies that interacted through frictional contact as illustrated in Figure 7. This problem has a large number of small spheres made up of a material with high density. There is one large ball of low density mixed up with the rest of the spheres. The collection of balls is inside a three dimensional rectangular box that experiences a left-to-right harmonic motion. Because the large ball has lower density, it will eventually "float" on the spheres of high density as illustrated in the figure. This test, along with other simulations focused on tracked vehicle mobility on granular terrain are discussed in detail in [24-25]. An animation is available at [26].

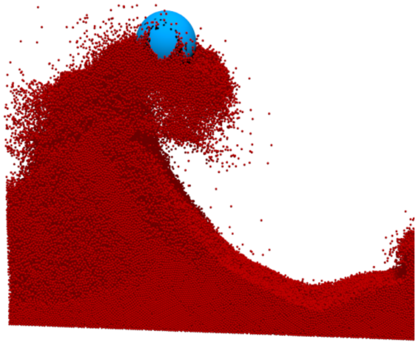

Figure 7. Largest problem simulated to date, the system has about 1.1 million bodies that are shaken in a moving box

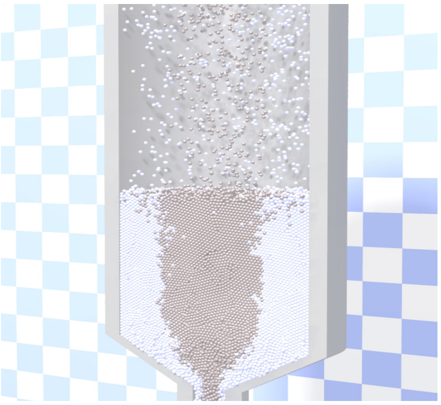

Figure 8. Pebble bed nuclear reactor simulation

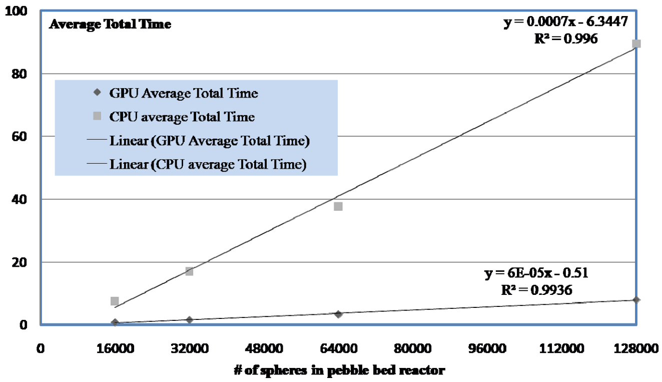

Figure 9. Average duration in seconds for taking one integration time step (0.01 seconds) when taking 150 iterations in the CCP solver

In what follows the emphasis is on a comparison between the GPU-based solution and a sequential approach used to solve a benchmark problem; i.e., the flow of a pebble bed nuclear reactor (Figure 8). The fuel is encased in tennis-ball-size graphite spheres, each filled with nuclear fuel, specifically, coated UO_2, with submillimeter diameter [27]. The approximately 400,000 pebbles are continuously recirculated or refreshed at a rate of about 2 per minute [28]. They are densely packed, at volume fractions approaching 0.6, and thus constitute a dense granular flow [29]. The center pebbles, represented with a different color, are moderator pebbles with comparable weight to the fuel pebbles, even if they do not contain particles of coated UO_2. The reactor is cooled with a fast helium flow blown top-down that has negligible drag effects on the spheres when compared to gravitational forces [29]. Predicting the dynamics of the fuel pebbles in the pebble-bed reactor is important for its safety and gauging its performance [30].

To better understand the potential of parallel computing when employed to solve the problem at hand, both the sequential and parallel implementations draw on the same solution procedure detailed in section 3. The only difference is that in one case the collision detection and the solution of the cone-complementarity problem are carried out sequentially, on the CPU, while for the parallel implementation these two stages, along with several other less computationally intensive steps of the solution methodology, are executed on the GPU. The benchmark problem was run for a set of 16, 32, 64, and 128 thousand particles. The sequential simulation was run on a single threaded Quad Core Intel Xeon E5430 2.66 GHz computer. For the parallel version, the collision detection was implemented on a NVIDIA 8800 GT card, while the cone complementarity problem was solved on a Tesla C870. The integration time step considered for this problem was 0.01 seconds. A number of 150 iterations was considered in the solution of the CCP problem.

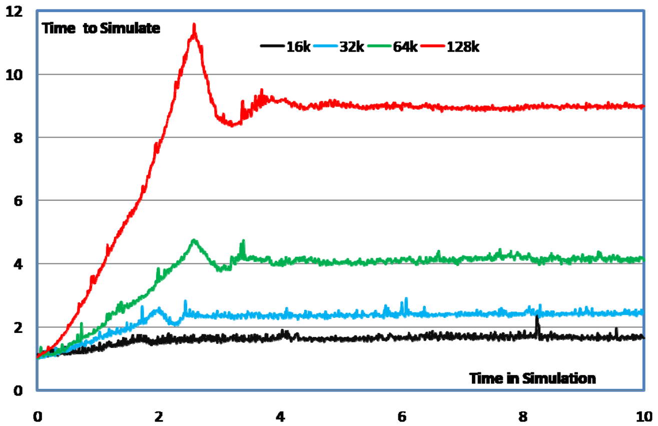

Figure 10. For the GPU solver, the plot shows how much time it took the solver to advance the simulation by one time step. On the horizontal axis is shown the simulation time. After approximately 3 seconds, when the flow reaches a steady state, each time step takes about the same amount of time, that is, approximately 9 seconds. These times were used in Figure 9 to generate the lower curve (one with smaller slope).

The dynamics of the pebble flow is as follows. First, the silo is closed and the balls are dropped from the top until the desired number of spheres is reached. The silo is subsequently opened, at which time the pebble flow commences. Shortly thereafter the flow reaches a steady state. At this time, the amount of time it takes to advance the simulation by one time step is measured. An average of this value obtained over several simulations is reported in Figure 9. This process is carried out for both the GPU and CPU implementations for each of the four scenarios (16 through 128 thousand bodies). The plot reveals that (a) both the CPU and GPU implementations scale linearly with the number of bodies in the problem, and (b) the slope of the plot associated with the GPU implementation is smaller than that associated with the CPU solver. In fact, for 128,000 particles, the GPU solver is about 10 times faster than the CPU solver. As the interest is in multi-million body problems, this slope difference will result in significant reduction in simulation times.

The plot in Figure 10 provides the history for the amount of time it took the GPU solver to perform one integration step. In the beginning, when the balls are filling up the silo, there are few contacts and one integration time step is cheap. As the number of spheres in contact increases due to piling up of the bodies at the bottom of the silo, the time it takes to complete one time step increases. This is due to the gradual increase in the dimension of the CCP problem that needs to be solved. An artifact of the fact that only 150 iterations were considered in the CCP problem is the spurious increase (the peak) that is more pronounced for the 128,000 body case. This was intentionally kept in order to demonstrate what happens if the CCP problem is not solved accurately. Specifically, if the number of iterations is not enough to lead to the convergence of the CCP solver, the amount of penetration between bodies will increase leading to a larger number of contacts and therefore a larger CCP problem. However, as the bottom of the silo opens up, the bodies start falling and this regime is less challenging for the solver since the number of contacts suddenly decreases until reaching a steady state shortly after 3 seconds. At that point the amount of time required to advance the simulation by one time step stabilizes. Note that the average of this value over several simulations was used in Figure 9 to generate the plot associated with the GPU-based solution.

5. Conclusions and directions of future work

Two observations justify today a fresh look at the solution of frictional contact problems in large scale multibody dynamics simulation. First, existing graphics cards provide tremendous flop rate at very low costs. Second, there is a wide spectrum of real-life applications that lead to large frictional contact dominated multibody dynamics problems that couldn’t be solved in the past. The contribution of this paper is in presenting an approach for parallel computational multibody dynamics that can be used to tackle many of these applications. The frictional contact problem was formulated in a form that is suitable to be numerically solved in parallel and could take advantage of commodity parallel computing support available on the GPU. The collision detection and solution of the cone complementarity problem turned out to be the main computational bottlenecks of the simulation. Both these stages of the solution have been parallelized thus enabling the implementation of an approach that can tackle problems with more than one million bodies.

Ongoing projects are aimed at: (a) increasing the size of the problem that can be solved by the proposed approach, (b) improving the speed of convergence of the CCP solver, (c) establishing the hardware infrastructure that can support the simulation of multibody dynamics problems with tens to hundreds of millions of bodies, and (d) performing an experimental validation of the simulation approach proposed. In terms of (a), current numerical experiments revealed that the 4 GB memory on the Tesla C1060 cards is exhausted for simulations that exceed 1.1 million bodies. A domain decomposition approach is anticipated to further increase this number by distributing a large problem to multiple GPUs using a spatial domain decomposition idea. The net outcome of this approach will be a pooling together of the memory resources of multiple cards. In terms of (b), it is anticipated that algebraic multi-grid methods will enable a reduction of the number of iterations required for convergence. Unless this issue gets addressed, problems with tens of millions of bodies might require prohibitively long convergence times that render the approach infeasible. In terms of (c), a 21 SP Teraflop cluster is currently assembled at the University of Wisconsin to support the domain decomposition approach described. The cluster will have one head node and six compute nodes, each of the latter with four Tesla C1060 NVIDIA GPUs. Finally, experimental validation is currently carried out both at macroscale, using 5 mm plastic particles, and microscale, using 100 and 500 micron glass spheres, respectively. In both cases, the experiments measure flow rates in silo replicas and a small hopper to validate the correctness of the simulation results. In addition to these four initiatives, there is a multitude of small projects that remain to be completed, the most important of which being the integration of the collision detection and CCP data structures. Currently, data is moved back and forth between the device and host right after the collision detection and before performing the CCP. This adds a significant overhead that once eliminated is anticipated to further improve the performance of the GPU solver.

References

1. Pfeiffer, F. and C. Glocker, *Multibody dynamics with unilateral contacts*. 1996: Wiley-Interscience.
2. Anitescu, M., *Optimization-based simulation of nonsmooth dynamics.* Mathematical Programming, 2006. **105**(1): p. 113-143.
3. Anitescu, M. and G.D. Hart, *A constraint-stabilized time-stepping approach for rigid multibody dynamics with joints, contact and friction.* International Journal for Numerical Methods in Engineering, 2004. **60**(14): p. 2335-2371.
4. Anitescu, M., F.A. Potra, and D.E. Stewart, *Time-stepping for three-dimensional rigid body dynamics.* Computer Methods in Applied Mechanics and Engineering, 1999. **177**(3-4): p. 183-197.
5. Lotstedt, P., *Mechanical Systems of Rigid Bodies Subject to Unilateral Constraints.* SIAM Journal on Applied Mathematics, 1982. **42**(2): p. 281-296.
6. Marques, M., *Differential inclusions in nonsmooth mechanical problems: shocks and dry friction*. 1993: Birkhäuser.
7. Moreau, J.J., *Standard inelastic shocks and the dynamics of unilateral constraints: CISM Courses and Lectures*, in *Unilateral Problems in Structural Analysis*, G.D. Piero and F. Macieri, Editors. 1983: New York. p. 173-221.

8. Pang, J.S., V. Kumar, and P. Song, *Convergence of Time-Stepping Method for Initial and Boundary-Value Frictional Compliant Contact Problems.* SIAM Journal on Numerical Analysis, 2005. **43**: p. 2200.

9. Pang, J.S. and J.C. Trinkle, *Complementarity formulations and existence of solutions of dynamic multi-rigid-body contact problems with Coulomb friction.* Mathematical Programming, 1996. **73**(2): p. 199-226.

10. Song, P., et al., *Analysis of Rigid-Body Dynamic Models for Simulation of Systems With Frictional Contacts.* Journal of Applied Mechanics, 2001. **68**(1): p. 118-128.

11. Glocker, C. and F. Pfeiffer, *Multiple impacts with friction in rigid multibody systems.* Nonlinear Dynamics, 1995. **7**(4): p. 471-497.

12. Manferdelli, J.L., *The Many-Core Inflection Point for Mass Market Computer Systems.* CTWatch Quarterly, 2007. **3**(1).

13. Negrut, D., *High Performance Computing for Engineering Applications, Course Notes ME964 (September 9 Lecture): http://sbel.wisc.edu /Courses/ME964/2008/index.htm*. 2008, University of Wisconsin.

14. NVIDIA. *Compute Unified Device Architecture Programming Guide 2.3: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit /docs/NVIDIA_CUDA_ProgrammingGuide_2.3.pdf.* 2009.

15. Stewart, D.E. and J.C. Trinkle, *An implicit time-stepping scheme for rigid-body dynamics with inelastic collisions and Coulomb friction.* International Journal for Numerical Methods in Engineering, 1996. **39**: p. 2673-2691.

16. Moreau, J., *Unilateral contact and dry friction in finite freedom dynamics.* Nonsmooth mechanics and Applications, 1988: p. 1-82.

17. Monteiro-Marques, M., *Differential inclusions in nonsmooth mechanical problems: Shocks and dry friction.* Progress in Nonlinear Differential Equations and Their Applications, 1993. **9**.

18. Pfeiffer, F., M. Foerg, and H. Ulbrich, *Numerical aspects of non-smooth multibody dynamics.* Computer Methods in Applied Mechanics and Engineering, 2006. **195**(50-51): p. 6891-6908.

19. Stewart, D.E., *Rigid-body dynamics with friction and impact.* SIAM Review, 2000. **42**(1): p. 3-39.

20. Stewart, D., *Convergence of a Time Stepping Scheme for Rigid Body Dynamics and Resolution of Painlevé's Problem.* Archive for Rational Mechanics and Analysis, 1998. **145**(3): p. 215-260.

21. Tasora, A. and M. Anitescu, *A Fast NCP Solver for Large Rigid-Body Problems with Contacts, Friction, and Joints.* Multibody Dynamics: Computational Methods and Applications, 2008: p. 45.

22. Anitescu, M. and A. Tasora, *An iterative approach for cone complementarity problems for nonsmooth dynamics.* Computational Optimization and Applications, 2009,accepted.

23. Mark Harris, Shubhabrata Sengupta, and J.D. Owens, *Parallel Prefix Sum (Scan) with CUDA*, in *GPU Gems 3*, H. Nguyen, Editor. 2008, Addison-Wesley. p. 851-876.

24. Heyn, T., H. Mazhar, and D. Negrut, *On the simulation of tracked vehicles operating on granular terrain: a parallel multibody dynamics aproach (to be submitted).* Multibody System Dynamics, 2009.

25. Heyn, T., *Simulation of tracked vehicles on granular terrain leveraging GPU computing, M.S. Thesis*, in *Mechanical Engineering*. 2009, University of Wisconsin-Madison: Madison.

26. Mazhar, H. *Million Body Simulation*. 2009. http://sbel.wisc.edu/Animations/index.htm.

27. Gougar, H., A. Ougouag, and W. Terry, *Advanced core design and fuel management for pebble-bed reactors.* Idaho National Engineering and Environmental Laboratory, INEEL/EXT-04-02245, 2004.

28. Kadak, A. and M. Bazant. *Pebble flow experiments for pebble bed reactors*, 2nd International Topical Meeting on High Temperature Reactor Technology, Beijing, CHINA, September 22-24, 2004.

29. Gougar, H., et al. *Design of pebble-bed reactors using genetic algorithms*. Proceedings of PHYSOR 2004.

30. Ougouag, A., J. Ortensi, and H. Hiruta, *Analysis of an Earthquake-Initiated-Transient in a PBR*. 2009, Tech. Rep. INL/CON-08-14876, Idaho National Laboratory (INL).