NSF GRANT # 0700191
NSF PROGRAM NAME: CMMI

Simulation of Multibody Dynamics Leveraging New Numerical Methods and Multiprocessor Capabilities

Dan Negrut
Simulation Based Engineering Lab, Department of Mechanical Engineering
University of Wisconsin, Madison, WI, 53706

Laurent Jay
Department of Mathematics
University of Iowa, Iowa-City, IA, 52242

Alessandro Tasora
Department of Industrial Engineering
University of Parma, V.G.Usberti 181/A, 43100, Parma, Italy

Mihai Anitescu
Mathematics and Computer Science Division
Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439

Hammad Mazhar, Toby Heyn, Arman Pazouki
Simulation Based Engineering Lab, Department of Mechanical Engineering
University of Wisconsin, Madison, WI, 53706

Abstract This paper describes an approach for the dynamic simulation of computer-aided engineering models where large collections of rigid bodies interacting through millions of frictional contacts and bilateral mechanical constraints. Thanks to the massive parallelism available on modern GPUs, we are able to simulate sand, granular materials, and other complex physical scenarios with one order of magnitude speedup when compared to a sequential CPU–based implementation of the discussed algorithms.

1. Introduction, Problem Statement, and Context

The ability to efficiently and accurately simulate the dynamics of rigid multibody systems is relevant in computer-aided engineering design, virtual reality, video games, and computer graphics. Devices composed of rigid bodies interacting through frictional contacts and mechanical joints pose numerical solution challenges because of the discontinuous nature of the motion. Consequently, even relatively small systems composed of a few hundred parts and constraints may require significant computational effort. More complex scenarios such as vehicles running on pebbles and sand as in Fig. 1 and Fig. 2, soil and rock dynamics, and flow and packing of granular materials are particularly challenging and prone to very long simulation times. Results reported in [3] indicate that the most popular rigid body software for engineering simulation, which uses an approach based on the so called Discrete Element Method, runs into significant difficulties when handling problems involving thousands of contact events.

Until recently, the high cost of parallel computing limited the analysis of such large systems to a small number of research groups. This is rapidly changing owing in large part to general-purpose computing on the GPU. Another example of commercially available rigid body dynamics software is NVIDIA's PhysX [4]. This software is commonly used in real-time applications where performance (rather than accuracy) is the primary goal. The goal of our effort was to somewhat de-emphasize the efficiency attribute and instead implement an open source, general-purpose physics-based GPU solver for multibody dynamics backed by convergence results that guarantee the accuracy of the numerical solution.

Unlike the so-called penalty or regularization methods, where the frictional interaction can be represented by a collection of stiff springs combined with damping elements that act at the interface of the two bodies [5], the approach embraced here draws on time-stepping procedures producing weak solutions of the differential varia-

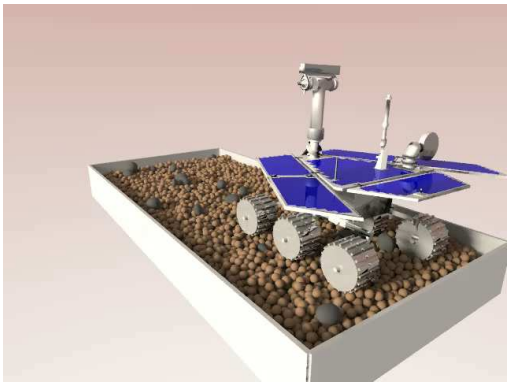

Figure 1: Chrono::Engine [1] simulation of a complex, rigid multi-body mechanism with contacts and joints.

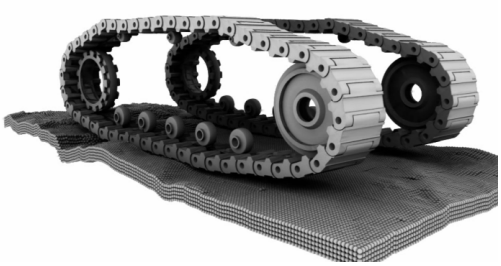

Figure 2: Chrono::Engine simulation of a tracked vehicle on a granular soil. The GPU was used for both dynamics and collision detection between tracks, sprockets, and pebbles [2].

tional inequality (DVI) problem, which describes the time evolution of rigid bodies with impact, contact, friction, and bilateral constraints. Early numerical methods based on DVI formulations can be traced back to the early 1980s and 1990s [6, 7, 8]. Recent approaches based on time-stepping schemes have included both acceleration-force linear complementarity problem (LCP) approaches [9, 10] and velocity-impulse, LCP-based time-stepping methods [11, 12, 13]. The LCPs, obtained as a result of the introduction of inequalities accounting for nonpenetration conditions in time-stepping schemes, coupled with a polyhedral approximation of the friction cone, must be solved at each time step in order to determine the system state configuration as well as the Lagrange multipliers representing the reaction forces [7, 11]. If the simulation entails a large number of contacts and rigid bodies, as is the case for granular materials, the computational burden of classical LCP solvers can become significant. Indeed, a well-known class of numerical methods for LCPs based on *simplex methods*, also known as *direct* or *pivoting* methods [14], may exhibit exponential worst-case complexity [15]. Moreover, the three-dimensional Coulomb friction case leads to a nonlinear complementarity problem (NCP). The use of a polyhedral approximation to transform the NCP into an LCP introduces unwanted anisotropy in friction cones and significantly augments the size of the numerical problem [11, 12].

In order to circumvent the limitations imposed by the use of classical LCP solvers and the limited accuracy associated with polyhedral approximations of the friction cone, a parallel fixed-point iteration method with projection on a convex set has been developed [16]. The method is based on a time-stepping formulation that solves at every step a cone-constrained quadratic optimization problem [17]. The time-stepping scheme has been proved to converge in a measure differential inclusion sense to the solution of the original continuous-time DVI. This paper illustrates how this problem can be solved in parallel by exploiting the parallel computational resources available on NVIDIA's GPU cards.

2. Core Method The formulation of the equations of motion, that is, the equations that govern the time evolution of a multibody system, is based on the so-called absolute, or Cartesian, representation of the attitude of each rigid body in the system. The state of the system is denoted by the generalized positions $\mathbf{q} = \left[\mathbf{r}_1^T, \boldsymbol{\varepsilon}_1^T, \ \ldots, \mathbf{r}_{n_b}^T, \boldsymbol{\varepsilon}_{n_b}^T\right]^T \in \mathbb{R}^{7n_b}$ and their time derivatives $\dot{\mathbf{q}} = \left[\dot{\mathbf{r}}_1^T, \dot{\boldsymbol{\varepsilon}}_1^T, \ldots, \dot{\mathbf{r}}_{n_b}^T, \dot{\boldsymbol{\varepsilon}}_{n_b}^T\right]^T \in \mathbb{R}^{7n_b}$, where n_b is the number of bodies, $\mathbf{r}_j$ is the absolute position of the center of mass of the jth body, and the quaternions (Euler parameters) $\boldsymbol{\varepsilon}_j$ are used to represent rotation and to avoid singularities. Instead of using quaternion derivatives in $\dot{\mathbf{q}}$, it is more advantageous to work with angular velocities expressed in the local (body-attached) reference frames; in other words, the method described will use the vector of generalized velocities $\mathbf{v} = \left[\dot{\mathbf{r}}_1^T, \bar{\boldsymbol{\omega}}_1^T, \ldots, \dot{\mathbf{r}}_{n_b}^T, \bar{\boldsymbol{\omega}}_{n_b}^T\right]^T \in \mathbb{R}^{6n_b}$. Note that the generalized velocity can be easily obtained as $\dot{\mathbf{q}} = \mathbf{L}(\mathbf{q})\mathbf{v}$, where $\mathbf{L}$ is a linear mapping that transforms each $\bar{\boldsymbol{\omega}}_i$ into the corresponding quaternion derivative $\dot{\boldsymbol{\varepsilon}}_i$ by means of the linear algebra formula $\dot{\boldsymbol{\varepsilon}}_i = \frac{1}{2}\mathbf{G}^T(\mathbf{q})\bar{\boldsymbol{\omega}}_i$, with 3x4 matrix $\mathbf{G}(\mathbf{q})$ as defined in [18]. We denote by $\mathbf{f}^A(t, \mathbf{q}, \mathbf{v})$ the set of applied, or external, generalized forces.

2.1. Bilateral constraints Bilateral constraints represent kinematic pairs, for example spherical, prismatic or revolute joints, and can be expressed as algebraic equations constraining the relative position of two bodies. Assuming a set $\mathcal{B}$ of constraints is present in the system, they lead to the scalar equations $\Psi_i(\mathbf{q}, t) = 0, \quad i \in \mathcal{B}$. Assuming smoothness of constraint manifold, $\Psi_i(\mathbf{q}, t)$ can be differentiated to obtain the Jacobian $\nabla_q \Psi_i = [\partial \Psi_i / \partial \mathbf{q}]^T$.

The notation $\nabla\Psi_i^T = \nabla_q \Psi_i^T \mathbf{L}(\mathbf{q})$ will be used in what follows.

2.2. Contacts with friction Given a large number of rigid bodies with different shapes, modern collision-detection algorithms are able to find efficiently a set of contact points, that is, points where a *gap function* $\Phi(\mathbf{q})$ can be defined for each pair of near-enough shape features. Where defined, such a gap function must satisfy the nonpenetration condition $\Phi(\mathbf{q}) \geq 0$ for all contact points.

When a contact i is active, that is, $\Phi_i(\mathbf{q}) = 0$, a normal force and a tangential friction force act on each of the two bodies at the contact point. In terms of notation, $\mathcal{A}$ will denote the set of all active contacts for a given configuration $\mathbf{q}$ of the system at time t_l. In fact, $\mathcal{A}(\mathbf{q}, \varepsilon)$ includes even potential contacts between bodies that are at t_l within a distance ε of each other and might collide during the time step from t_l to t_{l+1}. If no collision occurs, the algorithm will lead to zero normal/tangential forces for inactive collisions that were conservatively added to $\mathcal{A}(\mathbf{q}, \varepsilon)$.

We use the classical Coulomb friction model to define these forces [12]. If the contact is not active, that is, $\Phi_i(\mathbf{q}) > 0$, no contact or friction forces exist. This implies that the mathematical description of the model leads to a complementarity problem [11]. Consider two bodies A and B in contact, as shown in Fig. 3. Let $\mathbf{n}_i$ be the normal at the contact pointing toward the exterior of the body of lower index, which by convention is considered to be body A. 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 orthonormal vectors. The frictional contact force is impressed on the system by means of multipliers $\widehat{\gamma}_{i,n} \geq 0$, $\widehat{\gamma}_{i,u}$, and $\widehat{\gamma}_{i,w}$, which lead to the normal component of the force $\mathbf{F}_{i,N} = \widehat{\gamma}_{i,n}\mathbf{n}_i$ and the tangential component of the force $\mathbf{F}_{i,T} = \widehat{\gamma}_{i,u}\mathbf{u}_i + \widehat{\gamma}_{i,w}\mathbf{w}_i$. The Coulomb model is expressed by using the maximum dissipation principle:

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

2.3. The complete model The time evolution of the dynamical system is governed by the following differential variational inequality [16]:

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

The tangent space generators $\mathbf{D}_i = [\mathbf{D}_{i,n}, \mathbf{D}_{i,u}, \mathbf{D}_{i,w}] \in \mathbb{R}^{6n_b \times 3}$ are sparse and are defined given a pair of contacting bodies A and B as

$$\mathbf{D}_i^T = \begin{bmatrix} \mathbf{0} & \dots & -\mathbf{A}_{i,p}^T & \mathbf{A}_{i,p}^T \mathbf{A}_A \tilde{\mathbf{s}}_{i,A} & \mathbf{0} & \dots \\ \mathbf{0} & \dots & \mathbf{A}_{i,p}^T & -\mathbf{A}_{i,p}^T \mathbf{A}_B \tilde{\mathbf{s}}_{i,B} & \mathbf{0} & \dots \end{bmatrix}, \quad (3)$$

where, using the notation in Fig. 3, $\mathbf{A}_A$ is the orientation matrix associated with body A, $\mathbf{A}_{i,p} = [\mathbf{n}_i, \mathbf{u}_i, \mathbf{w}_i]$ is the $\mathbb{R}^{3\times3}$ matrix of the local coordinates of the ith contact, the vectors $\overline{\mathbf{s}}_{i,A}$ and $\overline{\mathbf{s}}_{i,B}$ are the contact point positions in body coordinates. A tilde $\tilde{\mathbf{x}}$ over a vector $\mathbf{x} \in \mathbb{R}^3$ represents the skew symmetric matrix associated with the outer product of two vectors [18].

3. The time-stepping scheme Given a position $\mathbf{q}^{(l)}$ and velocity $\mathbf{v}^{(l)}$ at the time step $t^{(l)}$, the numerical solution is found at the new time step $t^{(l+1)} = t^{(l)} + h$ by solving the following optimization problem with equilibrium constraints [19]:

$$\mathbf{M}(\mathbf{v}^{(l+1)} - \mathbf{v}^{(l)}) = h\mathbf{f}(t^{(l)}, \mathbf{q}^{(l)}, \mathbf{v}^{(l)}) + \sum_{i \in \mathcal{B}} \gamma_{i,b} \nabla\Psi_i + + \sum_{i \in \mathcal{A}} \left(\gamma_{i,n}\mathbf{D}_{i,n} + \gamma_{i,u}\mathbf{D}_{i,u} + \gamma_{i,w}\mathbf{D}_{i,w}\right), \quad (4)$$

$$i \in \mathcal{B}: \quad \tfrac{1}{h}\Psi_i(\mathbf{q}^{(l)}, t) + \nabla\Psi_i^T \mathbf{v}^{(l+1)} + \tfrac{\partial\Psi_i}{\partial t} = 0 \quad (5)$$

$$i \in \mathcal{A}: \quad 0 \leq \tfrac{1}{h}\Phi_i(\mathbf{q}^{(l)}) + \mathbf{D}_{i,n}^T \mathbf{v}^{(l+1)} \perp \gamma_n^i \geq 0, \quad (6)$$

$$(\gamma_{i,u}, \gamma_{i,w}) = \underset{\mu_i \gamma_{i,n} \geq \sqrt{\gamma_{i,u}^2 + \gamma_{i,w}^2}}{\operatorname{argmin}} \mathbf{v}^{(l+1),T} \left(\gamma_{i,u}\mathbf{D}_{i,u} + \gamma_{i,w}\mathbf{D}_{i,w}\right) \quad (7)$$

$$\mathbf{q}^{(l+1)} = \mathbf{q}^{(l)} + h\mathbf{L}(\mathbf{q}^{(l)})\mathbf{v}^{(l+1)}. \quad (8)$$

Here, γ_s represents the constraint impulse of a contact constraint; that is, $\gamma_s = h\widehat{\gamma}_s$, for $s = n, u, w$. The $\frac{1}{h}\Phi_i(\mathbf{q}^{(l)})$ term achieves constraint stabilization; its effect is discussed in [20]. Similarly, the term $\frac{1}{h}\Phi_i(\mathbf{q}^{(l)})$ achieves stabilization for bilateral constraints. The scheme converges to the solution of a measure differential inclusion [17] when the step size $h \to 0$.

The proposed approach casts the problem as a monotone optimization problem through a relaxation over the complementarity constraints, replacing Eq. (6) with

$$i \in \mathcal{A}: 0 \leq \tfrac{1}{h}\Phi_i(\mathbf{q}^{(l)}) + \mathbf{D}_{i,n}^T \mathbf{v}^{(l+1)} - \\ - \mu_i \sqrt{(\mathbf{v}^T \mathbf{D}_{i,u})^2 + (\mathbf{v}^T \mathbf{D}_{i,w})^2} \perp \gamma_n^i \geq 0.$$

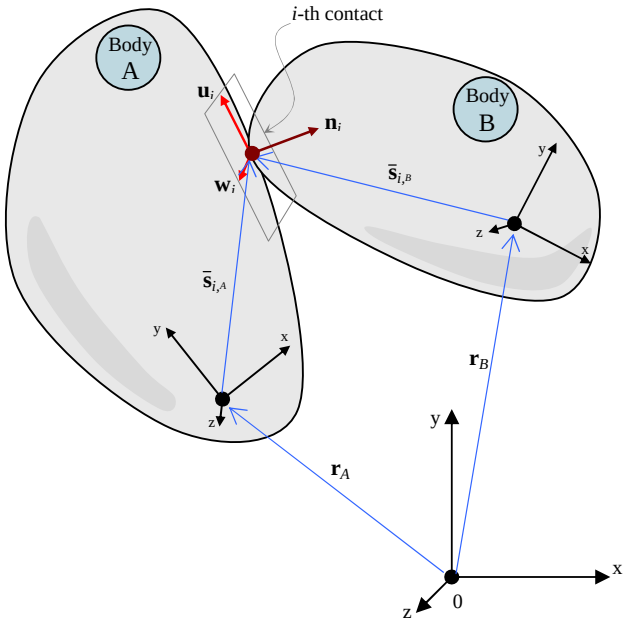

Figure 3: Contact i between two bodies $A, B \in \{1, 2, \ldots, n_b\}$

The solution of the modified time-stepping scheme will approach the solution of the same measure differential inclusion for $h \to 0$ as the original scheme [17], yet, in some situations, for large h, μ, or relative velocity $\mathbf{v}^{(l+1)}$; i.e., when not in an asymptotic regime, this relaxation can introduce motion oscillations. It was shown in [16] that the modified scheme is 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 [16, 21], we note that the algorithm makes use of the following vectors:

$$\tilde{\mathbf{k}} \equiv \mathbf{M}\mathbf{v}^{(l)} + h\mathbf{f}(t^{(l)}, \mathbf{q}^{(l)}, \mathbf{v}^{(l)}) \quad (9)$$

$$\mathbf{b}_i \equiv \left\{\tfrac{1}{h}\Phi_i(\mathbf{q}^{(l)}), 0, 0\right\}^T \quad i \in \mathcal{A}, \quad (10)$$

$$b_i \equiv \tfrac{1}{h}\Psi_i(\mathbf{q}^{(l)}, t) + \tfrac{\partial \Psi_i}{\partial t}, \quad i \in \mathcal{B}. \quad (11)$$

The solution, in terms of dual variables of the CCP (the multipliers), is obtained by iterating the following contraction maps until convergence [19]:

$$\forall i \in \mathcal{A}: \quad \gamma_i^{r+1} = \Pi_{\Upsilon_i}\left[\gamma_i^r - \omega\eta_i\left(D_i^T\mathbf{v}^r + \mathbf{b}_i\right)\right] \quad (12)$$

$$\forall i \in \mathcal{B}: \quad \gamma_i^{r+1} = \Pi_{\Upsilon_i}\left[\gamma_i^r - \omega\eta_i\left(\nabla\Psi_i^T\mathbf{v}^r + b_i\right)\right]. \quad (13)$$

At each iteration r, before repeating (12) and (13), also the primal variables (the velocities) are updated as

$$\mathbf{v}^{r+1} = \mathbf{M}^{-1}\left(\sum_{z\in\mathcal{A}} D_z\gamma_z^{r+1} + \sum_{z\in\mathcal{B}} \nabla\Psi_z\gamma_z^{r+1} + \tilde{\mathbf{k}}\right). \quad (14)$$

Note that the superscript $(l+1)$ was omitted. Interested readers are referred to [16] for a proof of the convergence of this method.

4. Algorithms, Implementations, and Evaluations

A detailed analysis of the computational bottlenecks in the proposed multibody dynamics analysis method reveals that the CCP solution and the prerequisite collision detection represent, in this order, the most compute-intensive tasks of the numerical solution at each integration (simulation) time step. This section concentrates on two approaches that expose a level of fine-grained parallelism that allows an efficient implementation of these two tasks on the GPU.

4.1. Parallel multibody dynamics GPU solver

Buffers for data structures The data structures on the GPU are implemented as large arrays (*buffers*) to match the execution model associated with NVIDIA's CUDA. Four main buffers are used: the contacts buffer, the constraints buffer, the reduction buffer, and the bodies buffer. The data structure for the contacts has been mapped into columns of four floats, as shown in Fig. 4. Each contact will reference its two touching bodies through the two pointers B_A and B_B, in the fourth and seventh rows of the contact data structure. There is no need to store the entire $\mathbf{D}_i$ matrix for the ith contact because it has zero entries

for most of its part, except for the two 12x3 blocks corresponding to the coordinates of the two bodies in contact. In fact, once the velocities of the two bodies $\dot{\mathbf{r}}_{A_i}$, ω_{A_i} and $\dot{\mathbf{r}}_{B_i}$, ω_{B_i} have been fetched, the product $\mathbf{D}_i^T\mathbf{v}^r$ in Eq. (12) can be performed as

$$\mathbf{D}_i^T\mathbf{v}^r = \mathbf{D}_{i,v_A}^T\dot{\mathbf{r}}_{A_i} + \mathbf{D}_{i,\omega_A}^T\omega_{A_i} + \mathbf{D}_{i,v_B}^T\dot{\mathbf{r}}_{B_i} + \mathbf{D}_{i,\omega_B}^T\omega_{B_i} \quad (15)$$

with the adoption of the following 3x3 matrices:

$$\begin{array}{rclrcl} \mathbf{D}_{i,v_A}^T & = & -\mathbf{A}_{i,p}^T, & \mathbf{D}_{i,\omega_A}^T & = & \mathbf{A}_{i,p}^T\mathbf{A}_A\tilde{\mathbf{s}}_{i,A} \\ \mathbf{D}_{i,v_B}^T & = & \mathbf{A}_{i,p}^T, & \mathbf{D}_{i,\omega_B}^T & = & -\mathbf{A}_{i,p}^T\mathbf{A}_B\tilde{\mathbf{s}}_{i,B.} \end{array} \quad (16)$$

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

The velocity update vector $\Delta\mathbf{v}_i$, needed for the sum in Eq. (14) also is sparse: it can be decomposed into small subvectors. Specifically, given the masses and the inertia tensors of the two bodies m_{A_i}, m_{B_i} and $\mathbf{J}_{A_i}$, $\mathbf{J}_{B_i}$, the term $\Delta\mathbf{v}_i$ will be computed and stored in four parts as follows:

$$\begin{array}{ll} \Delta\dot{\mathbf{r}}_{A_i} = m_{A_i}^{-1}\mathbf{D}_{i,v_A}\Delta\gamma_i^{r+1}, & \Delta\omega_{A_i} = \mathbf{J}_{A_i}^{-1}\mathbf{D}_{i,\omega_A}\Delta\gamma_i^{r+1} \\ \Delta\dot{\mathbf{r}}_{B_i} = m_{B_i}^{-1}\mathbf{D}_{i,v_B}\Delta\gamma_i^{r+1}, & \Delta\omega_{B_i} = \mathbf{J}_{B_i}^{-1}\mathbf{D}_{i,\omega_B}\Delta\gamma_i^{r+1}. \end{array} \quad (17)$$

Note that those four parts of the $\Delta\mathbf{v}_i$ terms are not stored in the *i*th contact data structure or in the data structure of the two referenced bodies (because multiple contacts may refer the same body, 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. (14). This will be discussed shortly.

The constraints buffer, shown in Fig. 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. (13) can be performed as the scalar value $\nabla\Psi_i^T\mathbf{v}^r = \nabla\Psi_{i,v_A}^T\dot{\mathbf{r}}_{A_i} + \nabla\Psi_{i,\omega_A}^T\omega_{A_i} + \nabla\Psi_{i,v_B}^T\dot{\mathbf{r}}_{B_i} + \nabla\Psi_{i,\omega_B}^T\omega_{B_i}$. Also, the four parts of the sparse vector $\Delta\mathbf{v}_i$ can be computed and stored as

$$\begin{array}{ll} \Delta\dot{\mathbf{r}}_{A_i} = m_{A_i}^{-1}\nabla\Psi_{i,v_A}\Delta\gamma_i^{r+1}, & \Delta\omega_{A_i} = \mathbf{J}_{A_i}^{-1}\nabla\Psi_{i,\omega_A}\Delta\gamma_i^{r+1} \\ \Delta\dot{\mathbf{r}}_{B_i} = m_{B_i}^{-1}\nabla\Psi_{i,v_B}\Delta\gamma_i^{r+1}, & \Delta\omega_{B_i} = \mathbf{J}_{B_i}^{-1}\nabla\Psi_{i,\omega_B}\Delta\gamma_i^{r+1}. \end{array} \quad (18)$$

Figure 6 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 $\mathbf{C}_j$. Note that to speed the iteration, it is advantageous to store the inverse of the mass and inertias rather than their original values, because the operation $\mathbf{M}^{-1}\mathbf{D}_i\Delta\gamma_i^{r+1}$ must be performed multiple times.

The parallel algorithm A parallelization of computations in Eq. (12) and Eq. (13) is easily implemented, by simply assigning one contact per thread (and, similarly, one constraint per thread). In fact the results of these computations would not overlap in memory, and two parallel threads will never need to write in the same memory location at the same time. These are the two most numerically intensive steps of the CCP solver, called the *CCP contact iteration kernel* and the *CCP constraint iteration kernel*.

However, the sums in Eq. (14) cannot be performed with embarrassingly-parallel implementations: it may happen that two or more contacts need to add their velocity updates to the same rigid body. A possible approach to overcome this problem is presented in [22], for a similar problem. We adopted an alternative method, with higher generality, based on the *parallel segmented scan* algorithm [23] that operates on an intermediate reduction buffer (Fig. 7); this method sums the values in the buffer using a binary-tree approach that keeps the computational load well balanced among the many thread processors. In the example of Fig. 7, 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 segmented reduction.

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

Algorithm 1: Time Stepping Using GPU

1. (*GPU, see section 4.2.*) Perform collision detection between bodies, obtaining $n_{\mathcal{A}}$ possible contact points within a distance δ, as contact positions $s_{i,A}$, $s_{i,B}$ on the two touching surfaces, and normals $\mathbf{n}_i$.
2. (*GPU, body-parallel*) **Force kernel**. For each body, compute forces $\mathbf{f}(t^{(l)}, \mathbf{q}^{(l)}, \mathbf{v}^{(l)})$, if any (for example, gravity). Store these forces and torques into F_j and C_j.
3. (*GPU, contact-parallel*) **Contact preprocessing kernel**. For each contact, given contact normal and position, compute in place the matrices $\mathbf{D}_{i,v_A}^T$, $\mathbf{D}_{i,\omega_A}^T$, and $\mathbf{D}_{i,\omega_B}^T$. Then compute η_i and the contact residual $\mathbf{b}_i = \{\frac{1}{h}\Phi_i(\mathbf{q}), 0, 0\}^T$.
4. (*GPU, body-parallel*) **CCP force kernel**. For each body j, initialize body velocities: $\dot{\mathbf{r}}_j^{(l+1)} = h\, m_j^{-1}\mathbf{F}_j$

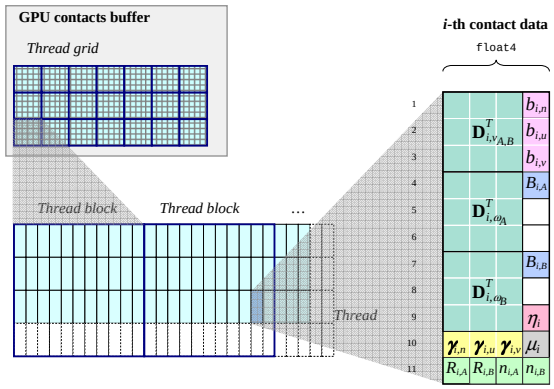

Figure 4: Grid of data structures for frictional contacts, in GPU memory.

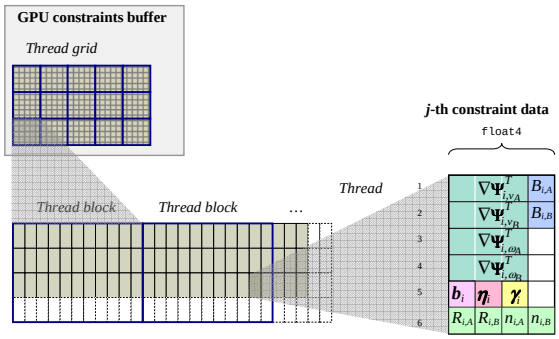

Figure 5: Grid of data structures for scalar constraints, in GPU memory.

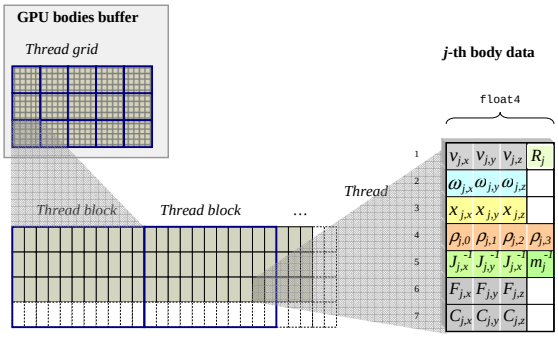

Figure 6: Grid of data structures for rigid bodies, in GPU memory.

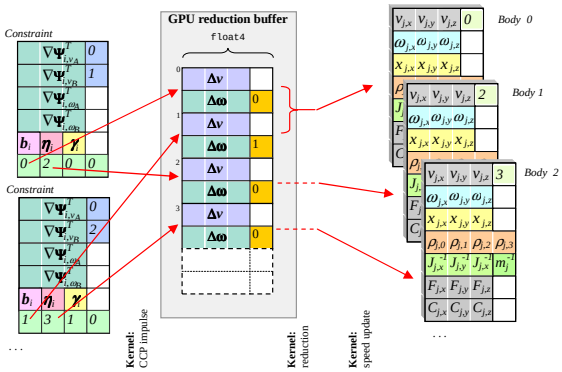

Figure 7: The reduction buffer avoids race conditions in parallel updates of the same body state.

and $\omega_j^{(l+1)} = h\,\mathbf{J}_j^{-1}\mathbf{C}_j$.

5. (*GPU, contact-parallel*) **CCP contact iteration kernel**. For each contact i, do $\gamma_i^{r+1} = \lambda\ \Pi_{\Upsilon_i}\left(\gamma_i^r - \omega\eta_i\left(\mathbf{D}_i^T\mathbf{v}^r + \mathbf{b}_i\right)\right) + (1-\lambda)\gamma_i^r$. Note that $\mathbf{D}_i^T\mathbf{v}^r$ is evaluated with sparse data, using Eq. (15). Store $\Delta\gamma_i^{r+1} = \gamma_i^{r+1} - \gamma_i^r$ in the contact buffer. Compute sparse updates 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, constraint-parallel*) **CCP constraint iteration kernel**. For each constraint i, do $\gamma_i^{r+1} = \lambda\ \left(\gamma_i^r - \omega\eta_i\left(\nabla\Psi_i^T\mathbf{v}^r + b_i\right)\right) + (1-\lambda)\gamma_i^r$. Store $\Delta\gamma_i^{r+1} = \gamma_i^{r+1} - \gamma_i^r$ in the contact buffer. Compute sparse updates 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, reduction-slot-parallel*) **Segmented reduction kernel**. Sum all the $\Delta\dot{\mathbf{r}}_i$, $\Delta\omega_i$ terms belonging to the same body, in the reduction buffer.

8. (*GPU, body-parallel*) **Body velocity updates kernel**. For each j body, add the cumulative velocity updates that can be fetched from the reduction buffer, using the index R_j.

9. Repeat from step 5 until convergence or until number of CCP steps reached $r > r_{max}$.

10. (*GPU, body-parallel*) **Time integration kernel**. For each j body, perform time integration as $\mathbf{q}_j^{(l+1)} = \mathbf{q}_j^{(l)} + h\mathbf{L}(\mathbf{q}_j^{(l)})\mathbf{v}_j^{(l+1)}$.

11. (*Host, serial*) If needed, copy body, contact, and constraint data structures from the GPU to host memory.

4.2. Parallel collision detection algorithm The 3D collision detection algorithm implemented performs a two-level spatial subdivision using axis aligned bounding boxes. The first partitioning occurs at the CPU level and yields a relatively small number of large *boxes*. The second partitioning of each of these boxes occurs at the GPU level leading to a large number of small *bins*. The GPU 3D collision detection, which handles spheres, ellipsoids,

and planes, occurs in parallel at the bin level. Any other geometries are represented as a collection of these primitives using a padding (decomposition) process presented in detail in [2]. Several kernel calls build on each other to eventually enable, in a one-thread-per-bin GPU parallel fashion, an exhaustive collision detection process in which thread *i* checks for collisions between all the bodies that happen to intersect the associated bin *i*. This requires $o(b_i^2)$ computational effort, where b_i represents the number of bodies touching bin *i*. The value of b_i is controlled by an appropriate selection of the bin size. Figure 8 illustrates a typical collision detection scenario and is used in what follows to outline the nine stages of the proposed approach.

Stage 1. The process begins by identifying all object-to-bin intersections. As Figure 9 shows, an object (body) can intersect, or touch, more than one bin; there is no limit to how many such intersections take place. The minimum and maximum bounding points of each object are determined and placed in their respective bins. For example, Fig. 9 shows that object 4's minimum point lies in B4 and its maximum point in A5. The entire object must fit between the minimum and maximum points; therefore the number of bins that the object intersects can be determined quickly by counting the number of bins between the two points in each axis and multiplying them. In this case the number is 4. This number of bins touched by each body is saved into an array, **T** (see Fig. 10), of size equal to the number of objects *N*.

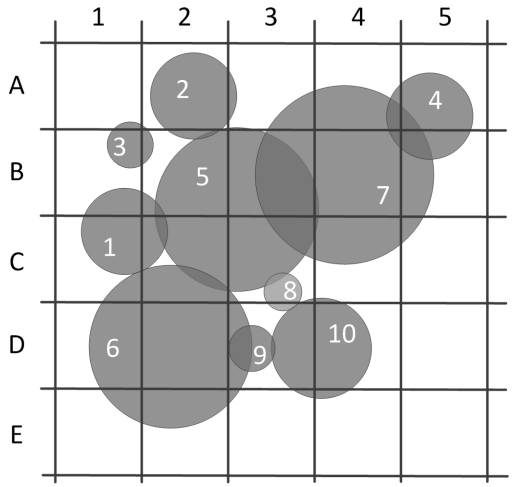

Figure 8: Two-dimensional example used to introduce the nine stages of the collision detection process. The grid is aligned to a global Cartesian reference frame.

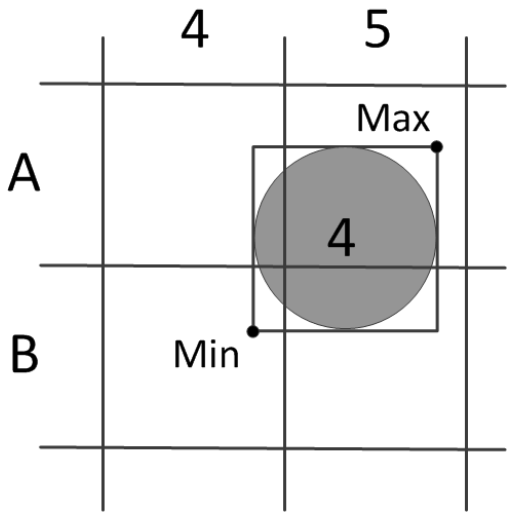

Figure 9: Minimum and maximum bounds of object, based on spatial subdivision in Fig. 8.

Stage 2. Next, we perform an inclusive parallel prefix sum on **T**. The CUDA-based Thrust library implementation [24] of the scan algorithm operates on **T** to return in **S** (see Fig. 11) the memory offset information.

Stage 3. An array **B** (see Fig. 12), is first allocated of size equal to the value of the last element in **S**. This value is equal to the total number of object-bin intersections. Each element in **B** is set to a key-value pair of two unsigned integers. The key is the bin id and the value is the object id. As Fig. 18 shows, objects not fully contained within the outer edge of the grid are restricted so that their maximum bound cannot be greater than the bounds of the uniform grid. The per-body parallel process used to determine the object-bin; i.e., value-key, pairs is essentially the same as in Stage 1 with the caveat that this information is now saved in **B** rather than just being counted in **T**. In this stage, the memory offsets contained in **S** are used so that the thread associated with each body can write data to the correct location in **B**.

Stage 4. The key-value array **B** is sorted in this stage by key, that is, by bin id. This effectively inverts the body-to-bin mapping to a bin-to-body mapping by grouping together all bodies in a given bin for further processing. The stage draws on the GPU-based radix sort from the Thrust library [24].

Stage 5. Next, we identify in parallel the start of each bin in the sorted array **B** by using the pseudocode in Fig. 16. The number of threads used to this end is equal to the number of elements in **B**; i.e., the number of object-bin interactions. Each thread reads the current and previous bin value; if these values differ, then the start of a bin has been detected. The first thread reads only the first element and records it as the initial value. The starting posi-

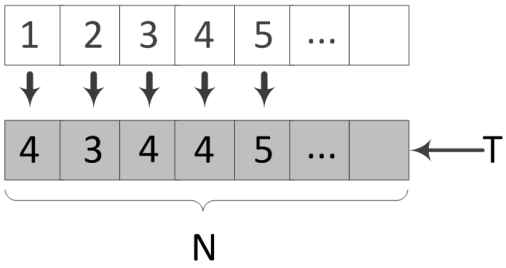

Figure 10: Array **T** with *N* entries, based on spatial subdivision in Fig. 8.

Figure 11: Result of prefix sum operation on **T**, based on spatial subdivision in Fig. 8. Each entry represents an object's offset based on the number of bins it touches.

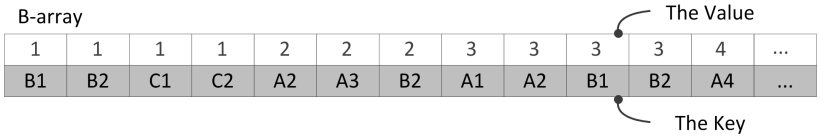

Figure 12: Array **B**, based on spatial subdivision in Fig. 8.

tions for each bin are written into an array **C** of key-value pairs of size equal to the number of bins in the 3D grid. When the start of a bin is found in array **B**, the thread and bin id are saved as the key and value, respectively. This pair is written to the element in **C** indexed by the bin id. Note that not all bins are active. Inactive bins (i.e., bins touched by zero or one bodies), are set to 0xffffffff, the largest possible value for an unsigned integer on a 32-bit, X86 architecture. Figure 14 shows the outcome of this stage.

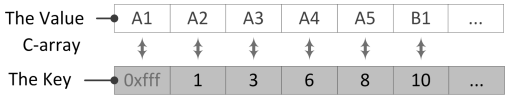

Figure 14: Array **C**, based on spatial subdivision in Fig. 8.

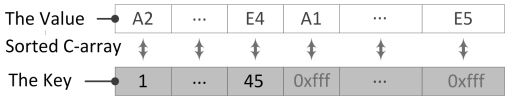

Figure 15: Sorted array **C**, based on spatial subdivision in Fig. 8.

```
For each thread index:
  If index<number of active Bins:
    if index > 0:
      if Current bin number != Previous bin number
        Bin start = index
          else if index=0:
            Bin start = 0
```

Figure 16: Pseudocode: Bin starting index computation.

Stage 6. The array **C** is next radix-sorted [24] by key. Consequently, inactive bins (identified by the 0xffffffff entries, represented for brevity as 0xfff in Fig. 15) "migrate" to the end of the array.

Stage 7. The total number of active bins is determined next by finding the index in the sorted array **C** of the first occurrence of 0xffffffff. Determining this index allows memory and thread usage to be allocated accurately thus having no threads wasted on inactive bins. One GPU thread is assigned in this stage to each active bin to perform an exhaustive, brute-force, bin-parallel collision detection for the purpose of *only counting* the collision events. By carefully selecting the bin size, the number of objects being tested for collisions is expected to be small; i.e., on average, in the range of 3 to 4 objects per bin. After counting the total number of collisions in its bin, the thread writes that tally into an unsigned integer array **D** of size equal to the number of active bins.

More involved, the algorithm for counting and subsequently computing ellipsoid collision information is described in detail in [25]. For spheres, the algorithm checks for collisions by calculating the distance between the objects. Contacts can occur only when the distance between the spheres' centers is less than or equal to the sum of their radii. Because one object could be contained within more than one bin, checks were implemented to prevent double counting. Since the midpoint of a collision volume can be contained only within one bin, only one thread (associated with that bin) will register/count a collision event. For example, in order to determine the midpoint of the collision volume we use the vector from centroid of object 4 to the centroid of object 7; see Fig. 17. The points where this vector intersects each object defines a segment; the location of the middle of this segment is used to decide the unique bin that claims ownership of the contact. If one object is completely inside the other, the midpoint of the collision volume is the centroid of the smaller object. Using this process, the number of collisions are counted for each bin and written to **D**. This stage is outlined in the pseudocode in Fig. 19.

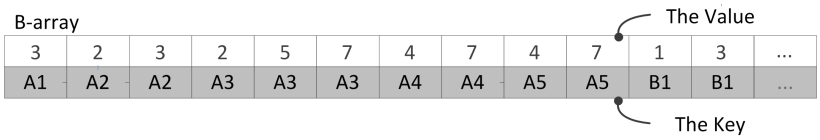

Figure 13: Sorted array **B**, based on spatial subdivision in Fig. 8.

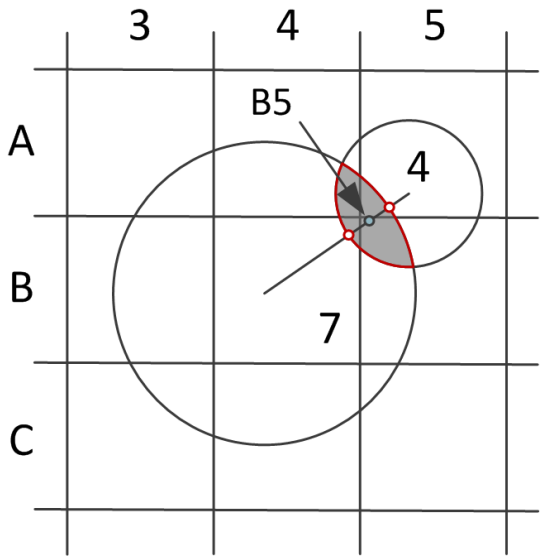

Figure 17: Center of collision volume. Based on spatial subdivision in Fig. 8.

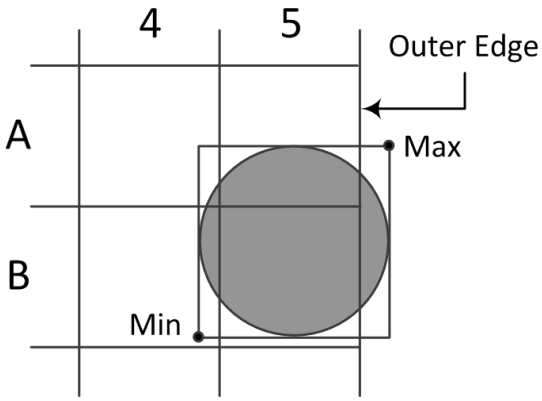

Figure 18: Max bound is constrained to bin A5.

```
For each thread index:
  If index<last:
    For posA=bin start && posA<bin end:
      For posB=posA+1 && posB<bin end:
        centerDist = distance between center of A and B
        rAB =Radius of A plus Radius of B
        if centerDist<=rAB:
          if centerDist+radius of A)<radius of B):
            collision center=bin of object A
          if centerDist+radius of B)<radius of A):
            collision center=bin of object B
          if(current bin=collision center)
            D[index]++;
```

Figure 19: Pseudocode: Determine number of collisions.

```
ObjectA=A
 ObjectB=B
 Normal=-midpoint/centerDist
 Collision point on B(x)= B.x+(B.w/centerDist)*(A.x-B.x)
 (repeat for y and z)…
 Collision point on A(x)= A.x+(A.w/centerDist)*(B.x-A.x)
 (repeat for y and z)…
```

Figure 20: Pseudocode: Computing collision data.

Stage 8. We perform next an inclusive parallel prefix scan operation [24] on **D**. This returns an array **E** whose last element is the total number of collisions in the uniform grid, a value that allows an exact amount of memory to be allocated in the next stage.

Stage 9. The final stage of the collision detection algorithm computes the actual contact information. To this end, an array of contact information structures **F** is allocated with a size equal to the value of the last element in **E**. The collision pairs are then found by using the algorithm outlined in Stage 7. Instead of simply counting the number of collisions, actual contact information is *computed and written* to its respective place in **F**; see pseudocode in Fig. 20.

5. Final Evaluation The GPU iterative solver and the GPU collision detection outlined herein have been embedded in our C++ simulation software Chrono::Engine. We tested the GPU-based parallel method with benchmark problems and compared it with the serial implementation in terms of efficiency.

For the results in Table 1, we simulated densely packed spheres that flow from a silo. The CPU was an Intel Xeon 2.66 GHz; the GPU was an NVIDIA Tesla C1060. The simulation time increases linearly with the number of bodies in the model. The GPU algorithm is at least one order of magnitude faster than the serial algorithm.

The test of Fig. 21 simulates 1 million rigid bodies inside a tank being shaken horizontally. This represents to date the largest multibody dynamics problem solved on one GPU card. The track system shown in Fig. 2 was exercised on granular terrain that was made up of more than 480,000 bodies.

6. Validation against and comparison with state-of-the-art sequential collision detection A first set of experiments was carried out to validate the implementation of the algorithm using various collections of spheres that display a wide spectrum of collision scenarios: disjoint spheres, spheres fully containing other spheres, spheres barely touching each other, and spheres that are in contact but not full containment. The first column of Table 2 reports the number of objects for five scenarios. For each scenario the error between the reference algorithm and the implemented algorithm is reported for the total number of contacts identified, the average error and standard deviation of the contact distance, contact unit normal, and point of contact. The reference algorithm used for validation was the sequential (nonparallel) collision detection implementation available in the open source, state-of-the-art Bullet Physics Engine [26].

These results demonstrate that the error in the proposed algorithm, when compared to the CPU implementation, is of the order of single precision round-off error. This is traced back to the fact that the CPU-based algorithm performs computations in double precision, while the GPU algorithm uses single precision arithmetic. For all scenarios the number of contacts was the same in both the CPU and GPU analyses.

A second set of numerical experiments was carried out to gauge the efficiency of the parallel CD algorithm developed. The reference used was the same sequential CD implementation from Bullet Physics Engine. The CPU used in this experiment (relevant for the Bullet implementation) was AMD Phenom II Black X4 940, a quad core 3.0 GHz processor that drew on 16 GB of RAM. The GPU used was NVIDIA's Tesla C1060. The operating system used was the 64bit version of Windows 7. Two scenarios were considered. The first scenario determined how many contacts a single GPU could determine before running out of memory. As Fig. 23 shows, approximately 22 million contacts were determined in less than 4 seconds. The second scenario gauged the relative speedup gained with respect to a serial implementation. The first test stopped when dealing with about 6 million contacts (see horizontal axis of Fig. 24), when Bullet ran into memory management issues. While providing a sufficient level of accuracy, the single precision GPU algorithm, tailored to deal with sphere-to-sphere contact only, led to a relative speedup of up to 180. We want to emphasize that the 180 factor does not reflect a GPU vs. CPU issue. The Bullet engine was the solution used prior to using the proposed method. It was a much more versatile collision detection engine, which in retrospect was unnecessarily used in double precision.

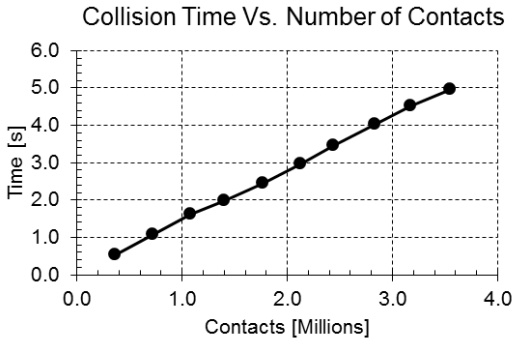

Figure 22: Collision time as the number of contacts increases.

6.1. Scaling analysis Our second set of experiments was designed to illustrate the scaling of the parallel numerical solution and collision detection. The vehicle used to this end was the simulation of a cylindrical tank that had a constant height with the radius varying with the number

Number of Bodies	CPU CCP [s]	GPU CCP [s]	Speedup CCP	Speedup CD
16,000	7.11	0.57	12.59	4.67
32,000	16.01	1.00	16.07	6.14
64,000	34.60	1.97	17.58	10.35
128,000	76.82	4.55	16.90	21.71

Table 1: Benchmark test of the GPU CCP solver and GPU collision detection.

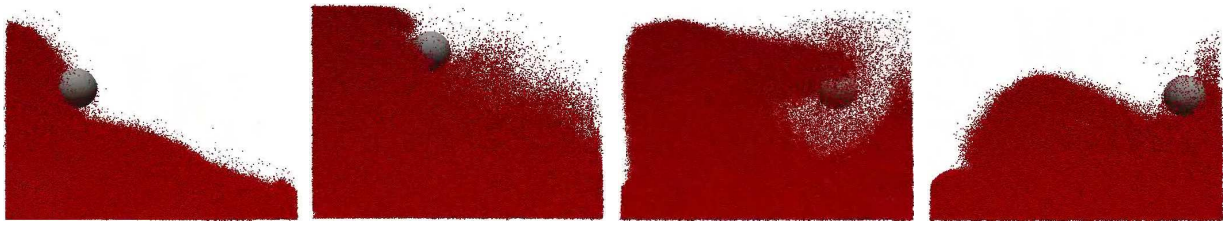

Figure 21: Light ball floating on 1 million rigid bodies moving around in a tank while interacting through friction and contact.

Table 2: Errors computed by taking the Euclidean norm of the difference between the collision data from Bullet and the collision detection algorithm discussed. AE stands for Average Error. SD stands for Standard Deviation

Spheres $[\times 10^6]$	Contacts	Contact Dist. Error [m]		Contact Normal Error [m]		Contact Point Error [m]	
		AE $[\times 10^{-7}]$	SD $[\times 10^{-4}]$	AE $[\times 10^{-10}]$	SD $[\times 10^{-7}]$	AE $[\times 10^{-6}]$	SD $[\times 10^{-3}]$
1	462,108	1.46	2.48	0.82	2.21	2.73	2.98
2	1,015,556	0.74	2.91	1.91	2.15	2.37	3.35
3	1,379,397	1.69	3.52	2.75	2.26	3.58	4.09
4	1,530,309	5.49	4.14	2.33	2.24	1.94	4.78
5	1,995,548	6.35	4.38	1.09	2.23	3.10	5.09

of spheres added to the tank. Specifically, the number of spheres in the tank was increased with each simulation without increasing the fill-in depth of the tank. Instead, the radius of the cylinder was increased for each simulation based on the number of spheres and their packing factor. Each test was run using an NVIDIA Tesla C1060 until the number of collisions and the compute time per solution time step reached steady state. The results presented in Table 3 and graphed in Fig. 22 indicate that the overall algorithm scales linearly. Furthermore, the results suggest that the bulk of the computation at each time step was taken by the GPU dynamics solver, with a small amount of time taken up by the collision detection. These collision detection times are longer than the raw times presented earlier due to the pre- and post-processing required by the physics engine as it organizes data on the GPU for use between the solver and collision detection.

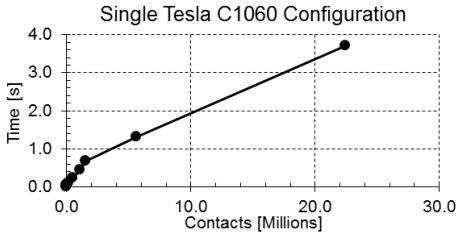

Figure 23: Collision time vs. contacts detected. This graph shows that when the algorithm is executed on a single GPU it scales linearly.

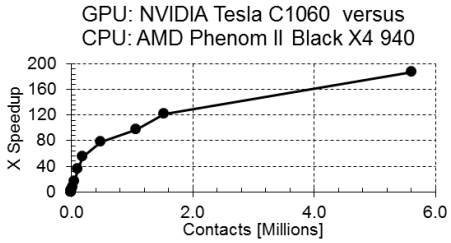

Figure 24: Overall speedup when comparing the CPU algorithm to the GPU algorithm. The maximum speedup achieved was approximately 180 times.

7. Future Directions The GPU dynamics engine proposed is more than one order of magnitude faster than a previously developed sequential implementation. The largest GPU simulation run to date had approximately 1.1 million bodies. Two barriers prevented the simulation of larger systems. First, we exhausted the GPU memory; second, we noticed a convergence stalling in the Gauss-Jacobi algorithm for CCP problems with more than 15 million variables. In order to address these aspects we are developing a distributed computing framework that leverages multiple GPUs, and we are investigating a minimal residual type Krylov method for the CCP solution. For the latter, GPU sparse preconditioning remains an open question.

Acknowledgments We would like to thank Richard Tonge for the substantial feedback and assistance he provided in generating this manuscript. Financial support for D. Negrut was provided in part by the National Science Foundation Awards CMMI–0700191 and CMMI–0840442. Financial support for A.Tasora was provided in part by the Italian Ministry of Education under the PRIN grant 2007Z7K4ZB. Mihai Anitescu was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. We thank NVIDIA and Microsoft for sponsoring our research programs in the area of high-performance computing.

References

[1] Tasora A. Chrono::Engine, An Open Source Physics–Based Dynamics Simulation Engine. Available online at www.deltaknowledge.com/chronoengine, 2006.

[2] T. Heyn. *Simulation of Tracked Vehicles on Granular Terrain Leveraging GPU Computing*. M.S. thesis, Department of Mechanical Engineering, University of Wisconsin–Madison, http://sbel.wisc.edu/documents/TobyHeynThesis_final.pdf, 2009.

[3] J. Madsen, N. Pechdimaljian, and D. Negrut. Penalty versus complementarity-based frictional contact of rigid bodies: A CPU time comparison. Technical Report TR-2007-06, Simulation-Based Engineering Lab, University of Wisconsin, Madison, 2007.

[4] PhysX. NVIDIA PhysX for Developers. Available online at http://developer.nvidia.com/object/physx.html, 2010.

[5] Peng Song, Jong-Shi Pang, and Vijay Kumar. A semi-implicit time-stepping model for frictional compliant contact problems. *International Journal of Numerical Methods in Engineering*, 60(13):267–279, 2004.

[6] Jean J. Moreau. Standard inelastic shocks and the dynamics of unilateral constraints. In G. Del Piero and F. Macieri, editors, *Unilateral Problems in Structural Analysis*, pages 173–221, New York, 1983. CISM Courses and Lectures no. 288, Springer–Verlag.

[7] P. Lotstedt. Mechanical systems of rigid bodies subject to unilateral constraints. *SIAM Journal of Applied Mathematics*, 42(2):281–296, 1982.

Table 3: Total time taken per time step at steady state and the number of contacts associated with it.

Objects $[\times 10^6]$	Total Time [sec]	GPU Collision Detection [sec]	GPU Solver	Contacts
0.2	12.1190	1.0758	10.5881	718,377
0.4	23.2806	1.9746	20.4606	1,403,784
0.6	35.0433	2.9785	30.7971	2,124,639
0.8	46.9516	4.0234	41.2297	2,838,832
1.0	58.1518	4.9473	51.1686	3,548,594

[8] M. D. P. Monteiro Marques. *Differential Inclusions in Nonsmooth Mechanical Problems: Shocks and Dry Friction*, volume 9 of *Progress in Nonlinear Differential Equations and Their Applications*. Birkhäuser Verlag, Basel, 1993.

[9] David Baraff. Issues in computing contact forces for non-penetrating rigid bodies. *Algorithmica*, 10:292–352, 1993.

[10] Jong-Shi Pang and Jeffrey C. Trinkle. Complementarity formulations and existence of solutions of dynamic multi-rigid-body contact problems with Coulomb friction. *Mathematical Programming*, 73(2):199–226, 1996.

[11] David E. Stewart and Jeffrey C. Trinkle. An implicit time-stepping scheme for rigid-body dynamics with inelastic collisions and Coulomb friction. *International Journal for Numerical Methods in Engineering*, 39:2673–2691, 1996.

[12] Mihai Anitescu and Florian A. Potra. Formulating dynamic multi-rigid-body contact problems with friction as solvable linear complementarity problems. *Nonlinear Dynamics*, 14:231–247, 1997.

[13] David E. Stewart. Rigid-body dynamics with friction and impact. *SIAM Review*, 42(1):3–39, 2000.

[14] Richard W. Cottle and George B. Dantzig. Complementary pivot theory of mathematical programming. *Linear Algebra and Its Applications*, 1:103–125, 1968.

[15] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. In *Computer Graphics (Proceedings of SIGGRAPH)*, pages 23–34, 1994.

[16] M. Anitescu and A. Tasora. An iterative approach for cone complementarity problems for nonsmooth dynamics. *Computational Optimization and Applications*, 47(2):207–235, 2010.

[17] Mihai Anitescu. Optimization-based simulation of nonsmooth rigid multibody dynamics. *Mathematical Programming*, 105(1):113–143, 2006.

[18] E. J. Haug. *Computer-Aided Kinematics and Dynamics of Mechanical Systems Volume-I*. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[19] A. Tasora. A Fast NCP Solver for Large Rigid-Body Problems with Contacts. In C.L. Bottasso, editor, *Multibody Dynamics: Computational Methods and Applications*, pages 45–55. Springer, 2008.

[20] Mihai Anitescu and Gary D. Hart. A constraint-stabilized time-stepping approach for rigid multibody dynamics with joints, contact and friction. *International Journal for Numerical Methods in Engineering*, 60(14):2335–2371, 2004.

[21] A. Tasora, D. Negrut, and M. Anitescu. Large-scale parallel multi-body dynamics with frictional contact on the graphical processing unit. *Journal of Multibody Dynamics*, 222(4):315–326, 2008.

[22] T. Harada. Real-time rigid body simulation on GPUs. *GPU Gems*, 3:611–632, 2007.

[23] S. Sengupta, M. Harris, Y. Zhang, and J.D. Owens. Scan primitives for GPU computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, page 106. Eurographics Association, 2007.

[24] J. Hoberock and N. Bell. Thrust: A Parallel Template Library. Available online at http://code.google.com/p/thrust/, 2009.

[25] A. Pazouki, H. Mazhar, and D. Negrut. Parallel ellipsoid collision detection with application in contact dynamics-DETC2010-29073. In Shuichi Fukuda and John G. Michopoulos, editors, *Proceedings to the 30th Computers and Information in Engineering Conference*. ASME International Design Engineering Technical Conferences (IDETC) and Computers and Information in Engineering Conference (CIE), 2010.

[26] Physics Simulation Forum. Bullet Physics Library. Available online at http://www.bulletphysics.com/Bullet/wordpress/bullet, 2008.