Preprint ANL/MCS-P1777-0710, Math. and Comp. Science Division, Argonne National Laboratory

Solving Large Multibody Dynamics Problems on the GPU

Dan Negrut[*] Alessandro Tasora[†] Mihai Anitescu[‡]
Hammad Mazhar[*] Toby Heyn[*] Arman Pazouki[*]

27 July, 2010

This paper describes an approach for the dynamic simulation of complex computer-aided engineering models where large collections of rigid bodies interact mutually through millions of frictional contacts and bilateral mechanical constraints. Thanks to the massive parallelism available on today's GPU boards, 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 (CAE) design, in virtual reality, in video games, and in computer graphics in general, for instance, when physical simulation is used for special effects in 3D movies.

Devices composed of rigid bodies interacting through frictional contacts and

[*]Simulation Based Engineering Lab, Department of Mechanical Engineering, University of Wisconsin, Madison, WI, 53706

[†]Department of Industrial Engineering, University of Parma, V.G.Usberti 181/A, 43100, Parma, Italy

[‡]Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439

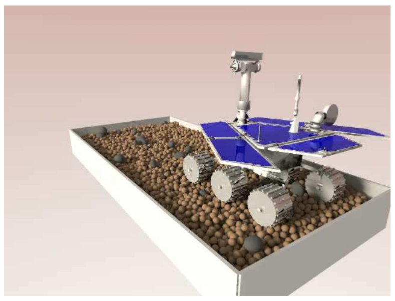


Figure 1: Chrono::Engine [1] simulation of a complex, rigid multibody mechanism with contacts and joints.

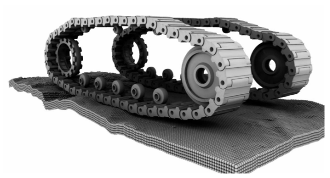


Figure 2: Chrono::Engine simulation of a tracked vehicle on a granular soil. The GPU was used for both dynamics and collision detection between tracks, sprockets, and pebbles [2].

mechanical joints pose numerical solution challenges because of the discontinuous nature of the motion; the dynamics is nonsmooth because of the presence of noninterpenetration, collision, and adhesion constraints. 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, such as in Fig. 3, would require prohibitively long computational times, hindering the effectiveness of multibody dynamics simulation in the CAE landscape. Results reported in [3] indicate that the most widely used commercial software for multibody dynamics runs into significant difficulties when handling simple problems involving hundreds of contact events, and cases with thousands of contacts become intractable. The method embraced in this work can solve efficiently problems with millions of contacts in a sequential CPU implementation, and improved performance can be obtained with the GPU implementation discussed herein.

Until recently, because of a price barrier, taking advantage of the potential of large-scale parallel computing was the privilege of a relatively small number of research groups, thus limiting the spectrum of applications benefiting from efficiency gain induced by parallel computing. This scenario is rapidly changing as a result of a trend set by general-purpose computing on GPUs. Few GPU projects, however, are concerned with the dynamics of multibody systems. The two most significant ones can be traced back to the Havok and the Ageia physics engines. Both are commercial proprietary libraries used in the video-game industry. Given these circumstances, the goal of this work was to implement an open source, general-purpose GPU solver for

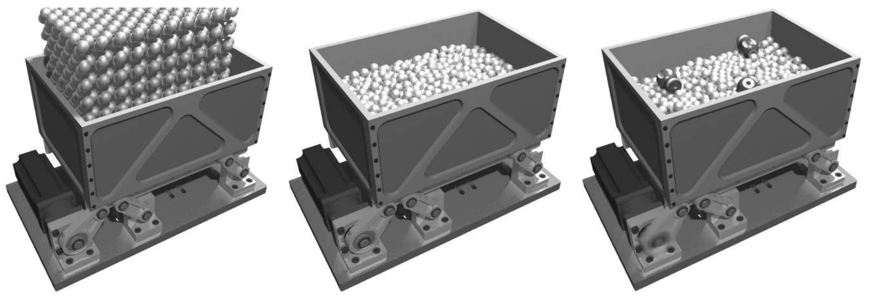


Figure 3: The proposed method can simulate the dynamics of devices with motors, joints and contacts, as in the case of this size segregation machine that shakes thousands of steel spheres.

multibody dynamics backed by rigorous convergence results that guarantee the accuracy of the solution. To this end, the parallel version implemented on the GPU builds on an analytical framework defined in [4, 5], which can robustly accommodate bilaterally constrained rigid bodies undergoing frictional contacts.

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 [6], the approach embraced here relies on a different mathematical framework. Specifically, the algorithm draws on time-stepping procedures producing weak solutions of the differential variational inequality (DVI) problem that describes the time evolution of rigid bodies with impact, contact, friction, and bilateral constraints. When compared to penalty methods, the DVI approach has a greater algorithmic complexity but avoids the small time steps that plague the former approach.

Early numerical methods based on DVI formulations can be traced back to the early 1980s and 1990s [7, 8, 9], while the DVI formulation has been recently classified by differential index [10]. Recent approaches based on time-stepping schemes have included both acceleration-force linear complementarity problem (LCP) approaches [11, 12] and velocity-impulse, LCP-based time-stepping methods [13, 14, 15]. 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 [8, 13]. 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 [16], may exhibit exponential worst-case complexity [17]. 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 [13, 14].

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 proposed, developed, and tested [5]. The method is based on a time-stepping formulation that solves at every step a cone-constrained quadratic optimization problem [18]. 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 $\mathbf{q} = \left[\mathbf{r}_1^T, \epsilon_1^T, \ldots, \mathbf{r}_{n_b}^T, \epsilon_{n_b}^T\right]^T \in \mathbb{R}^{7n_b}$ 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 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) ϵ_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 $\mathbf{v} = \left[\dot{\mathbf{r}}_1^T, \bar{\omega}_1^T, \ldots, \dot{\mathbf{r}}_{n_b}^T, \bar{\omega}_{n_b}^T\right]^T \in \mathbb{R}^{6n_b}$. 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 $\bar{\omega}_i$ into the corresponding quaternion derivative $\dot{\epsilon}_i$ by means of the linear algebra formula $\dot{\epsilon}_i = \frac{1}{2}\mathbf{G}^T(\mathbf{q})\bar{\omega}_i$, with 3x4 matrix $\mathbf{G}(\mathbf{q})$ as defined in [19].

We denote by $\mathbf{f}^A(t, \mathbf{q}, \mathbf{v})$ the set of applied, or external, generalized forces.

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 $\nabla_q \Psi_i = [\partial\Psi_i/\partial\mathbf{q}]^T$.

Constraints are consistent at velocity level provided that $\nabla\Psi_i^T \mathbf{v} + \frac{\partial\Psi_i}{\partial t} = 0$, where $\nabla\Psi_i^T = \nabla_q \Psi_i^T \mathbf{L}(\mathbf{q})$.

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.

Note that a signed distance function, differentiable at least up to some value of the interpenetration, can be easily defined if bodies are smooth and convex. However, this situation is not always possible, for instance when dealing with concave or faceted shapes often used to represent parts of mechanical devices.

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. We use the classical Coulomb friction model to define these forces [14]. 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 [13]. Given two bodies in contact A and B, 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 imposes the following nonlinear constraints:

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

where $\mathbf{v}_{i,T}$ is the relative tangential velocity. The constraint $\langle \mathbf{F}_{i,T}, \mathbf{v}_{i,T} \rangle = -||\mathbf{F}_{i,T}||\,||\mathbf{v}_{i,T}||$ requires that the tangential force be opposite to the tangential velocity. Note that the friction force depends on the friction coefficient $\mu_i \in \mathbb{R}^+$.

An equivalent convenient way of expressing this constraint is by using the maximum dissipation principle:

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

In fact, the first-order necessary Karush-Kuhn-Tucker conditions for the minimization problem (1) correspond to the Coulomb model above [20, 9].

The complete model

Considering the effects of both the set $\mathcal{A}$ of frictional contacts and the set $\mathcal{B}$ of bilateral constraints, the time evolution of the dynamical system is governed by the following differential variational inequality (a differential problem with set-valued functions and complementarity constraints):

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

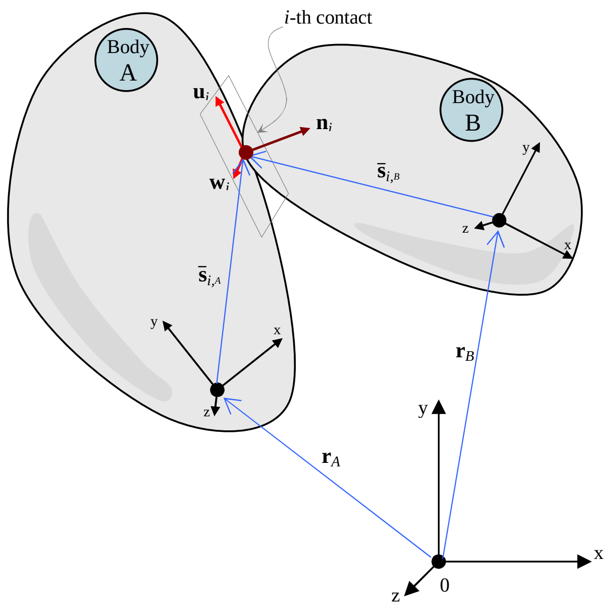


Figure 4: Contact i between two bodies $A, B \in \{1, 2, \dots, n_b\}$

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

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

where we use $\mathbf{A}_{i,p} = [\mathbf{n}_i, \mathbf{u}_i, \mathbf{w}_i]$ as the $\mathbb{R}^{3\times 3}$ matrix of the local coordinates of the ith contact and introduce the vectors $\bar{\mathbf{s}}_{i,A}$ and $\bar{\mathbf{s}}_{i,B}$ as contact point positions in body coordinates, with skew matrices $\tilde{\mathbf{s}}_{i,A}$ and $\tilde{\mathbf{s}}_{i,B}$; see Fig. 4.

3 The time-stepping scheme

We formulate the dynamical problem in terms of measure differential inclusions [15], whose numerical solution can be obtained by using the following time-stepping scheme based on the solution of a complementarity problem at each time step.

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 [4]:

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

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

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

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

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

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 [21]. 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 [18] when the step size $h \to 0$.

Several numerical methods can be used to solve (4)–(7) [22]. Our approach casts the problem as a monotone optimization problem by introducing a relaxation over the complementarity constraints, replacing Eq. (6) with $i \in \mathcal{A}: 0 \le \frac{1}{h}\Phi_i(\mathbf{q}^{(l)}) + \mathbf{D}_{i,n}^T\mathbf{v}^{(l+1)} - \mu_i\sqrt{(\mathbf{v}^T\,\mathbf{D}_{i,u})^2 + (\mathbf{v}^T\,\mathbf{D}_{i,w})^2} \perp \gamma_n^i \ge 0$. The solution of the modified time-stepping scheme will approach the solution of the same measure differential inclusion for $h \to 0$ as the original scheme [18].

Previous work [5] showed 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 [5, 23], we note that the algorithm makes use of the following vectors:

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

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

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

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

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

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

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

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

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

The iterative process uses the metric projector $\Pi_{\Upsilon_i}(\cdot)$ [4], which is a non-expansive map $\Pi_{\Upsilon_i} : \mathbb{R}^3 \to \mathbb{R}^3$ acting on the triplet of multipliers associated with the ith contact. Thus, if the multipliers fall into the friction cone, they are not modified; if they are in the polar cone, they are set to zero; in the remaining cases they are projected orthogonally onto the surface of the friction cone. The overrelaxation factor ω and η_i parameters are adjusted to control the convergence. Interested readers are referred to [5] for a proof of the convergence of this method.

The previous algorithm has been implemented on serial computer architectures and proved to be reliable and efficient. In the following, the time-consuming part of the methodology, the CCP iteration, will be reformulated to take advantage of the parallel computing resources available on GPU boards.

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. The rest of 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, rigid multibody dynamics solver on the GPU

Modern GPU processors can execute thousands of threads in parallel, providing teraflops-level computing speed. These processors, usually devoted to the execution of pixel shading fragments for three-dimensional visualization, can be exploited also for scientific computation thanks to development environments such as CUDA from NVIDIA, which provide C++ functions to easily manage GPU data buffers and *kernels*, that is, operations to executed in parallel on the data. The proposed algorithm fits well into the GPU multithreaded model because the computation can be split into multiple threads each acting on a single contact, or kinematic constraint, or rigid body depending on the stage of the computation.

Buffers for data structures

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.

Special care should be paid to minimize the memory overhead caused by repeated transfers of large data structures. We organized data structures in a way that minimizes the number of fetch and store operations and maximizes the arithmetic intensity of the kernel code, as recommended by the CUDA development guidelines.

The data structure for the contacts has been mapped into columns of four floats, as shown in Fig. 5. 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

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

with the adoption of the following 3x3 matrices:

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

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 $\mathbf{D}^T_{i,v_{AB}}$ 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:

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

Note that those four parts of the $\Delta\mathbf{v}_i$ terms are not stored in the ith 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.

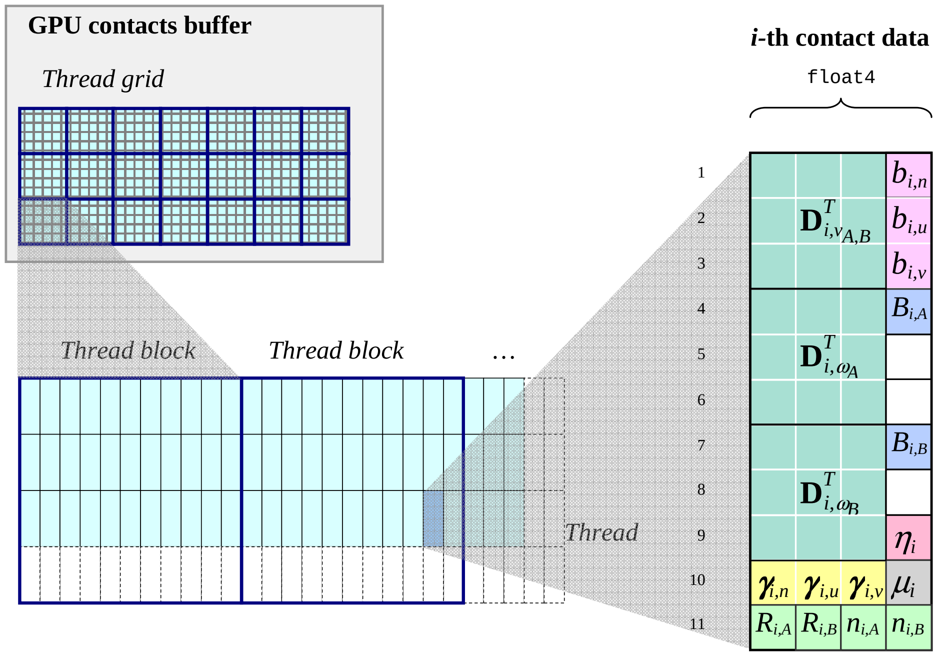
The constraints buffer, shown in Fig. 6, is based on a similar concept. Jacobians $\nabla\Psi_i$ of all scalar constraints are stored in a sparse format, each corresponding to four rows $\nabla\Psi_{i,v_A}$, $\nabla\Psi_{i,\omega_A}$, $\nabla\Psi_{i,v_B}$, $\nabla\Psi_{i,\omega_B}$. Therefore the product $\nabla\Psi_i^T\mathbf{v}^r$ in Eq. (13) can be performed as the scalar value $\nabla\Psi_i^T\mathbf{v}^r = \nabla\Psi^T_{i,v_A}\dot{\mathbf{r}}_{A_i} + \nabla\Psi^T_{i,\omega_A}\omega_{A_i} + \nabla\Psi^T_{i,v_B}\dot{\mathbf{r}}_{B_i} + \nabla\Psi^T_{i,\omega_B}\omega_{B_i}$. Also, the four parts of the sparse vector $\Delta\mathbf{v}_i$ can be computed and stored as

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

Figure 7 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.

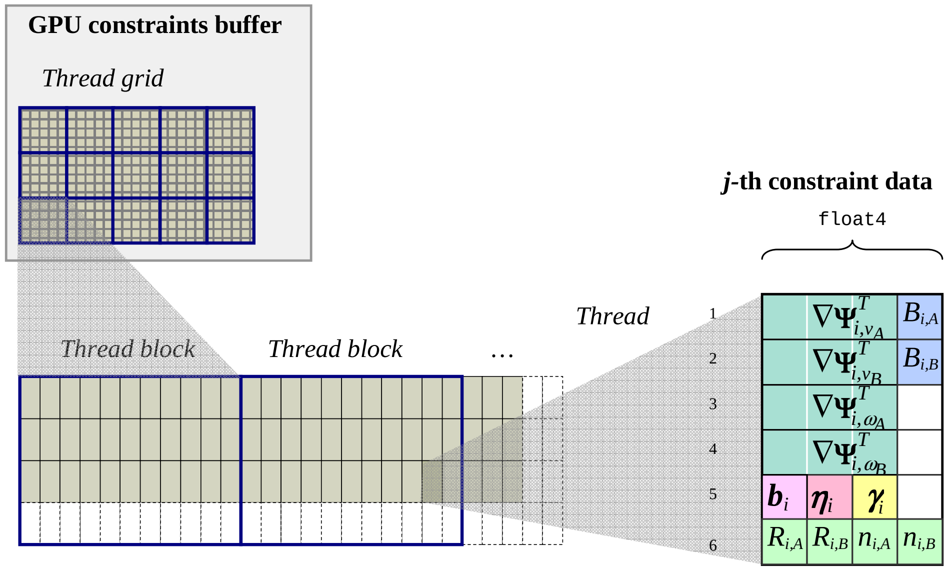
The parallel algorithm

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



GPU contacts buffer
Thread grid
Thread block
Thread block
...
Thread
i-th contact data
float4
1
2
3
4
5
6
7
8
9
10
11
$\mathbf{D}^T_{i,v_{A,B}}$
$b_{i,n}$
$b_{i,u}$
$b_{i,v}$
$\mathbf{D}^T_{i,\omega_A}$
$B_{i,A}$
$\mathbf{D}^T_{i,\omega_B}$
$B_{i,B}$
η_i
$\gamma_{i,n}$ $\gamma_{i,u}$ $\gamma_{i,v}$ μ_i
$R_{i,A}$ $R_{i,B}$ $n_{i,A}$ $n_{i,B}$

Figure 5: Grid of data structures for frictional contacts, in GPU memory.



GPU constraints buffer
Thread grid
Thread block
Thread block
...
Thread
j-th constraint data
float4
1
2
3
4
5
6
$\nabla\mathbf{\Psi}^T_{i,v_A}$ $B_{i,A}$
$\nabla\mathbf{\Psi}^T_{i,v_B}$ $B_{i,B}$
$\nabla\mathbf{\Psi}^T_{i,\omega_A}$
$\nabla\mathbf{\Psi}^T_{i,\omega_B}$
b_i η_i γ_i
$R_{i,A}$ $R_{i,B}$ $n_{i,A}$ $n_{i,B}$

Figure 6: Grid of data structures for scalar constraints, in GPU memory.

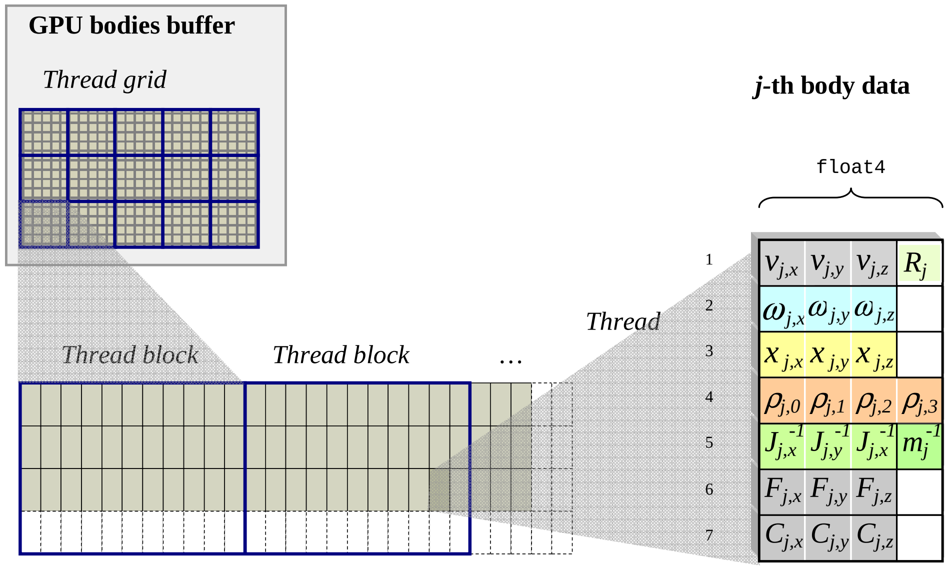

GPU bodies buffer
Thread grid
Thread block
Thread block
...
Thread
j-th body data
float4
1
2
3
4
5
6
7
$v_{j,x}$ $v_{j,y}$ $v_{j,z}$ R_j
$\omega_{j,x}$ $\omega_{j,y}$ $\omega_{j,z}$
$x_{j,x}$ $x_{j,y}$ $x_{j,z}$
$\rho_{j,0}$ $\rho_{j,1}$ $\rho_{j,2}$ $\rho_{j,3}$
$J^{-1}_{j,x}$ $J^{-1}_{j,y}$ $J^{-1}_{j,x}$ m^{-1}_j
$F_{j,x}$ $F_{j,y}$ $F_{j,z}$
$C_{j,x}$ $C_{j,y}$ $C_{j,z}$

Figure 7: Grid of data structures for rigid bodies, in GPU memory.

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 [24], for a similar problem. We adopted an alternative method, with higher generality, based on the *parallel segmented scan* algorithm [25] that operates on an intermediate reduction buffer (Fig. 8); 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. 8, 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.

Since collision detection is the biggest computational overhead after the CCP solution, we also developed a GPU-based parallel code for collision detection, obtaining a 20x speedup factor when compared to the serial code of the Bullet library. The GPU collision code requires the use of multiple kernels and complex data structures that we cannot describe here because

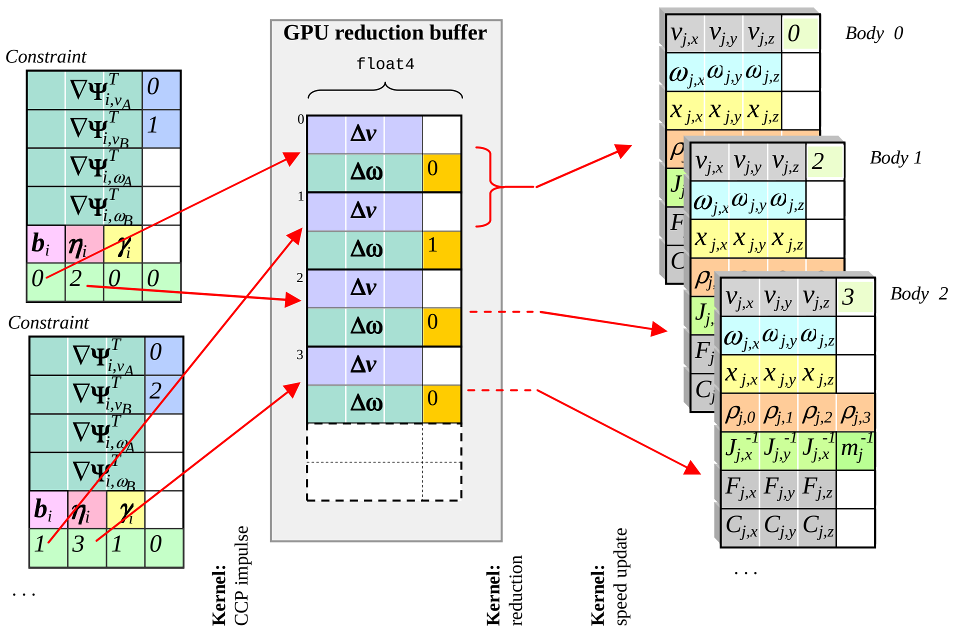


Figure 8: The reduction buffer avoids race conditions in parallel updates of the same body state.

of limited space; details are available in [26].

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 or host*) 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. (*Host, serial*) If needed, copy contact and body data structures from host memory to GPU buffers. Copy also constraint data (residuals b_i and Jacobians) into the constraint buffer.

3. (*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.

4. (*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 $\mathbf{b}_i = \{\frac{1}{h}\Phi_i(\mathbf{q}), 0, 0\}^T$.

5. (*GPU, body-parallel*) **CCP force kernel**. For each body j, initialize body velocities: $\dot{\mathbf{r}}_j^{(l+1)} = h\, m_j^{-1}\mathbf{F}_j$ and $\omega_j^{(l+1)} = h\, \mathbf{J}_j^{-1}\mathbf{C}_j$.

6. (*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.

7. (*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.

8. (*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.

9. (*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.

10. Repeat from step 6 until convergence or until number of CCP steps reached $r > r_{max}$.

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

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

4.2 Parallel collision detection algorithm

The collision detection algorithm implemented performs a two-level spatial subdivision. The first partitioning occurs at the CPU level, which leads to a relatively small number of large *boxes*. The second partitioning of each of these boxes occurs at the GPU level, which leads to a large number of small *bins*. The collision detection occurs in parallel at the bin level. Specifically, an exhaustive collision detection process is carried out by one GPU thread to check for collisions between all the bodies that happen to intersect the associated bin. Since the bin size can be made arbitrarily small, the number of possible collisions inside the bin is kept small. Figure 9 outlines the software and hardware stack associated with this methodology.

Four OpenMP threads control the four GPUs available on the computer. The coarse-grained partitioning at the CPU level is straightforward: the volume occupied by the objects is partitioned into boxes whose edges are aligned with a global Cartesian reference frame. Typically, this operation results in hundreds of boxes, which are subsequently assigned in a round robin fashion to each of the four GPUs. For instance, if there are 125 boxes it is expected that on average each of the four GPUs will have to process about 31 or 32 boxes. Objects that span two or more boxes are automatically assigned to each box when the data is sent down for fine-level partitioning on the GPU. A mechanism is in place on the GPU side to avoid double counting of potential collisions in this case. The specifics of the GPU collision detection are discussed in detail in the following subsections. Once the collision data has been computed for each box, it is merged together on a single CPU thread.

4.2.1 Stages of GPU collision detection algorithm

A high-level overview of the GPU-based collision detection is as follows. The collision detection process starts by identifying the intersections between objects and bins. The object-bin pairs are subsequently sorted by bin id. Next, each bin's starting index is determined so that the bins' objects can be traversed sequentially. All objects inside a bin are subsequently checked against each other for collisions by one GPU thread. This high-level process is implemented in a sequence of nine stages, each of which is discussed next. Figure 10 shows what a typical set of data used for collision detection looks like and will be used in what follows to explain the proposed approach.

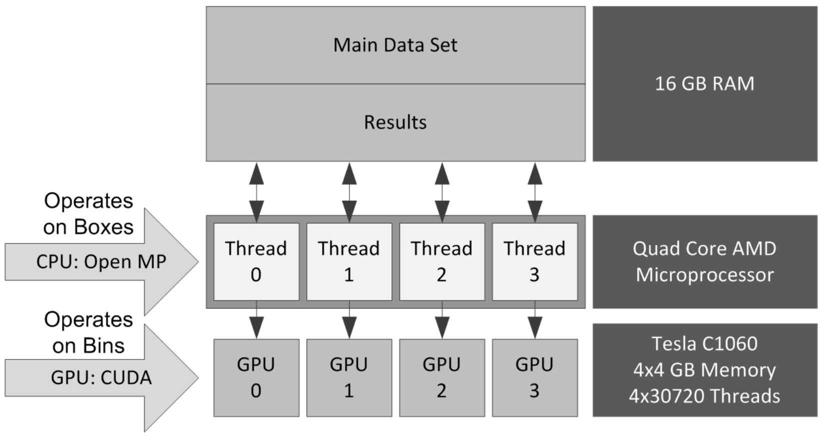


Figure 9: Software and hardware stack

Stage 1 The collision detection process begins by identifying all object-to-bin intersections. As Figure 11 shows, an object (body) can "touch" more than one bin; there is no limit to how many intersections take place. The proposed approach, for performance reasons, had bins sized so that spheres would intersect only up to eight bins; however, because the actual number of intersections between an object and a bin is computed, an object can intersect any number of bins. This feature allows the proposed algorithm to be extended to any geometry.

In this stage, the bounds of the simulation space are calculated first. Both the largest and outermost objects are determined, allowing the required bin size to be calculated. In order for the grid and bins to remain uniform, each side of the grid, like a cube, has equal length. The bin size is set to be twice as large as the radius of the largest object, which ensures that each sphere can touch a maximum of eight bins. The issue of choosing an optimal bin size is further discussed in [26] as it relates to efficient use of the GPU. If one knows the bin size, the number of bins used in the collision detection process can be determined.

Next, the minimum and maximum bounding points of each object are determined and placed in their respective bins. For example, Fig. 11 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 is then saved into

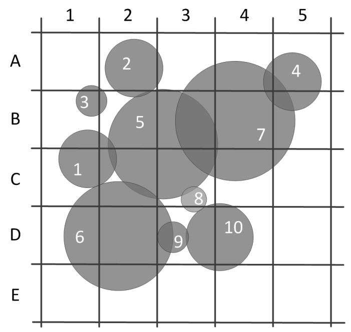


Figure 10: Two-dimensional example used to introduce the nine stages of the collision detection process. The grid is aligned to a global Cartesian reference frame.

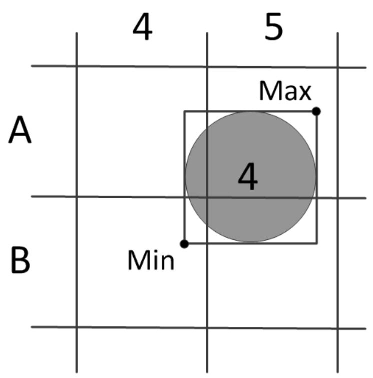


Figure 11: Minimum and maximum bounds of object, based on spatial subdivision in Fig. 10.

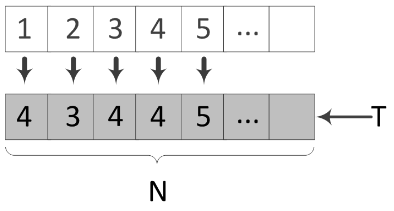


Figure 12: Array **T** with N entries, based on spatial subdivision in Fig. 10.

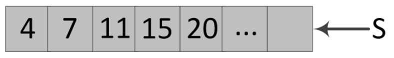


Figure 13: Result of prefix sum operation on **T**, based on spatial subdivision in Fig. 10. Each entry represents an object's offset based on the number of bins it touches.

an array, **T** (see Fig. 12), which is the size of the number of objects N. As a result of this stage, array **T** contains at index i the number of bins that object i touches.

Stage 2 An inclusive parallel prefix sum is next performed on **T**, which was created in Stage 1. A parallel prefix sum (scan) operation takes an array of N elements and returns a second array in which element i is the sum of the first i entries of the original array [25]. The CUDA-based Thrust library implementation [27] of the scan algorithm used operates on **T** to return in **S** the memory offset for each object in **T** (see Fig. 13). Specifically, if one needs to determine what bins body b intersects, $\mathbf{S}[b-1]$ provides the memory offset used in Stage 3.

Stage 3 In Stage 3, an array **B** (see Fig. 16), is 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 in the uniform grid. Each element in **B** is a key-value pair of two unsigned integers. The key in this pair is the bin number and the value is the object number. The bin number is calculated as described in the pseudocode of Fig. 15, where $[i, j, k]$, in 3D, are the coordinates of the bin in the uniform grid. This process is equivalent to a 3D geometric hash function and ensures that each bin number is unique. Additional checks make sure that the bin number is within the valid bounds of the uniform grid. As Fig. 14 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 process used to determine the intersections is essentially the same as Stage 1 with the difference being that intersections are written rather than just being counted. 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 In Stage 4, the key-value array **B** is sorted by its key, that is, by bin id. This stage utilizes a GPU-based radix sort from the Thrust library [27]. This stage 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.

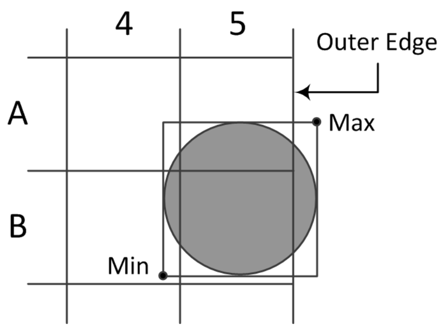
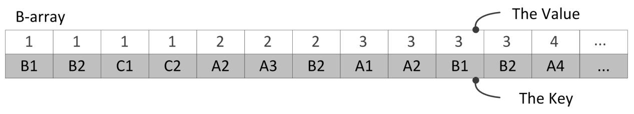


Figure 14: Max bound is constrained to bin A5.

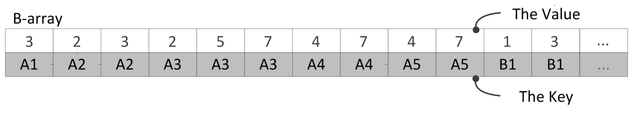
```
SIDE: number of bins on side of uniform grid (stages 1 and 3)
$BinNumber = i + j * SIDE.x + k * SIDE.x * SIDE.y$
```

Figure 15: Pseudocode: Bin number computation.


B-array
The Value
The Key

1	1	1	1	2	2	2	3	3	3	3	4	...
B1	B2	C1	C2	A2	A3	B2	A1	A2	B1	B2	A4	...

Figure 16: Array **B**, based on spatial subdivision in Fig. 10.


B-array
The Value
The Key

3	2	3	2	5	7	4	7	4	7	1	3	...
A1	A2	A2	A3	A3	A3	A4	A4	A5	A5	B1	B1	...

Figure 17: Sorted array **B**, based on spatial subdivision in Fig. 10.

```
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 18: Pseudocode: Bin starting index computation.

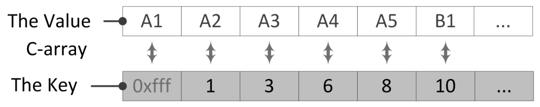


Figure 19: Array **C**, based on spatial subdivision in Fig. 10.

Stage 5 Once the key value pairs are sorted by bin id, the start of each bin needs to be determined. The total number of elements in array **B** is known and is equal to the total number of object-bin interactions. The process for this stage is outlined in the pseudocode of Fig. 18. Each element in the array is processed in parallel by one thread. Each of these threads 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 positions 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. 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. This simplifies the sorting process in the next stage, since such bins cannot host any contacts. Figure 19 shows the outcome of this stage.

Stage 6 In Stage 6, the bins that are not used are pushed to the end of the array. To accomplish this, array **C** is sorted by key and invalid entries (the 0xffffffff entries, represented for brevity as 0xfff in the figure) are moved to the end of the array; see Fig. 20. This stage allows **C** to be traversed sequentially, so that the number of active bins can be determined for the next stage. To this end a second radix sort is utilized. Once sorted, the array is processed in parallel, and the index of the last valid entry in the array is determined. No bins after this index will be processed.

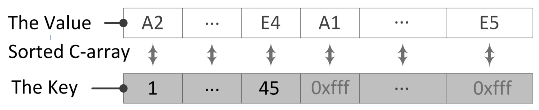


Figure 20: Sorted array **C**, based on spatial subdivision in Fig. 10.

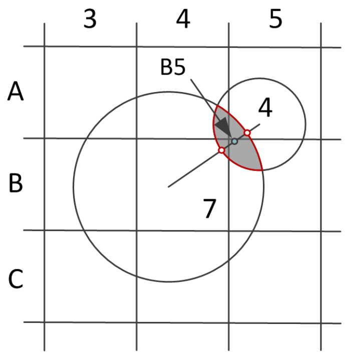


Figure 21: Center of collision volume. Based on spatial subdivision in Fig. 10.

Stage 7 At this point, one GPU thread is assigned to each active bin to perform an exhaustive, brute-force, bin-parallel collision detection. This is effective because the number of objects being tested for collisions has become relatively small. First, the total number of active bins is determined by finding the index in the sorted **C** array where the bin value is a valid number and the next value is an invalid 0xffffffff. Because the values were sorted in the previous stage, there is only one place in the array where this can occur. Determining this value allows memory and thread usage to be allocated accurately. In this manner no threads will be wasted on passive bins, that is, bins that are touched by one or no object at all. In this stage, each thread computes the total number of collisions in its bin and writes that number to an array **D** of unsigned integers with a size equal to the number of active bins. The bin starting number (from Stage 5) is read for the current and next bins, the starting value for the next bin being the ending value for the current one, allowing the list of objects to be iterated through.

The algorithm used to check for collisions between spheres does so by calculating the distance between both objects. Because all objects are spheres, contacts can occur only when the distance between each object's centroid is less than or equal to the sum of their radii. Because one object could be contained within more than one bin, caution was required to prevent re-

```
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 22: Pseudocode: Determine number of collisions.

peated detection of the same collision. For example, if two objects intersect within two separate bins, each thread processing its respective bin shouldn't find the exact same collision pair. Therefore, several conditions need to be satisfied in order to guarantee unique collisions.

The principle used is simple; the midpoint of a collision volume can be contained only within one bin. Therefore, only one thread will find a collision pair. For example, in order to determine the midpoint of the collision volume, the vector going from centroid of object 4 to the centroid of object 7 is determined; see Fig. 21. Then the point where this vector intersects each object is determined. The midpoint between these two points is used as the midpoint of the collision volume. 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 of Fig. 22.

Stage 8 Once the number of collisions per bin is returned, an inclusive prefix scan operation is performed on it. This stage returns an array **E** whose last element is the total number of collisions in the uniform grid. This allows an exact amount of memory to be allocated in the next stage. The Thrust scan algorithm [27] was again used for this stage.

Stage 9 The final step of the collision detection algorithm is to write the contact information to the contact pair array. Concretely, 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. At this point, instead of simply counting the number of collisions, actual contact information is written to its respective

```
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 23: Pseudocode: Computing collision data.

place in **F**; see the pseudocode of Fig. 23. Additional contact information can be computed if necessary in this stage.

5 Final Evaluation

The GPU iterative solver and the GPU collision detection 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 method in terms of computing time.

For the results in Table 1, we simulated densely packed spheres that flow from a silo. The CPU is an Intel Xeon 2.66 GHz; the GPU is 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.

Other stress tests were performed with even larger amounts of spheres, such as in the benchmark of Fig. 30. Similarly, the test of Fig. 29 simulates 1 million rigid bodies inside a tank being shaken horizontally (the amount of available RAM on a single GPU board restricted us from going beyond that limit).

Using the proposed GPU method, we are already able to simulate granular soil (pebbles, sand) under the tracks of a vehicle; see Fig. 2. In fact our GPU collision detection code is able to handle nonconvex shapes by performing spherical decomposition. In order to simulate larger scenarios, with smaller grains of sand, future efforts will address the possibility of using domain decomposition, with clusters of multiple GPU boards on multiple hosts.

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: Stress test of the GPU CCP solver and GPU collision detection.

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

5.1 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 in the test for five scenarios. For each test 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 [28].

These results showed that the error in the proposed algorithm, when compared to the CPU implementation, is minimal and is due to floating-point error. The CPU-based algorithm relies on double precision, while the GPU algorithm relies on single precision. While this had an effect on the overall contact data, the number of contacts was the same. Furthermore, the small errors reported above show that no collisions were missed by the algorithm. Because the data was sorted, if a contact had been missed, subsequent contacts would have also been incorrect, leading to large error values.

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. Three scenarios were considered. The first one gauged the relative speedup gained with respect to the serial implementation. This test stopped when dealing with about 6 million contacts (see horizontal axis of Fig. 24), when Bullet ran into memory management issues. The plot illustrates that the relative speedup is up to 180. The second scenario determined how many contacts a single GPU could determine with this algorithm before running short on memory. As Fig. 25 shows, approximately 22 million contacts were determined in less than 4 seconds. This was followed by a third scenario, where the problem size was increased up to 1.6 billion contacts; see Fig. 26. This experiment relied on the software/hardware stack outlined in Fig. 9. Specifically, the test combined the use of OpenMP, for multiple GPU management, with CUDA, for GPU-level computation management.

5.2 Collision detection scaling for relevant dynamics application

Our second set of experiments was designed to illustrate how the proposed algorithm performs when interfaced with a physics based dynamics simulation package. The goal was to understand how the algorithm scaled when the objects were tightly packed rather than randomly distributed as in the previous test. The simulation consisted of a cylindrical tank that had a constant height with the radius varying with the number of spheres added to the tank. Specifically, the number of spheres in the tank was increased with each simulation without increasing the depth of the tank. Instead, the radius of

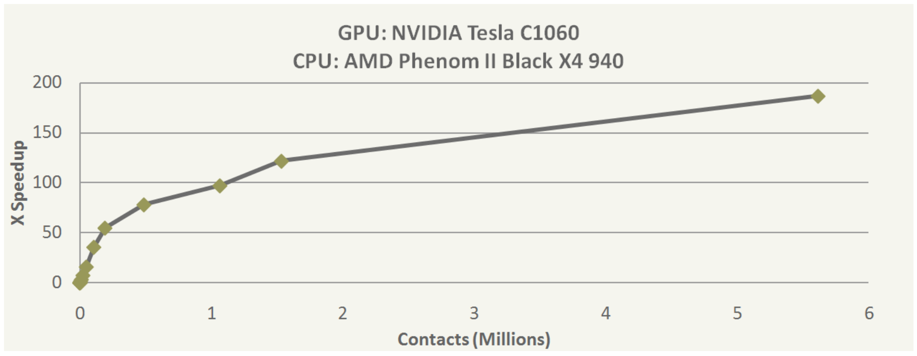


Figure 24: Overall speedup when comparing the CPU algorithm to the GPU algorithm. The maximum speedup achieved was approximately 180 times.

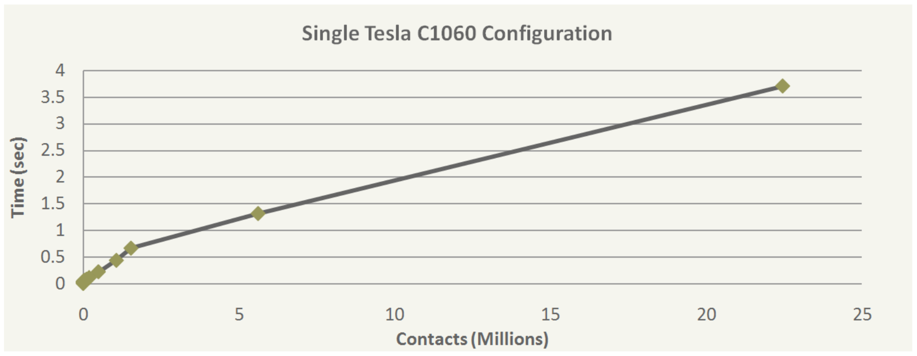


Figure 25: Collision time vs. contacts detected. This graph shows that when the algorithm is executed on a single GPU it scales linearly.

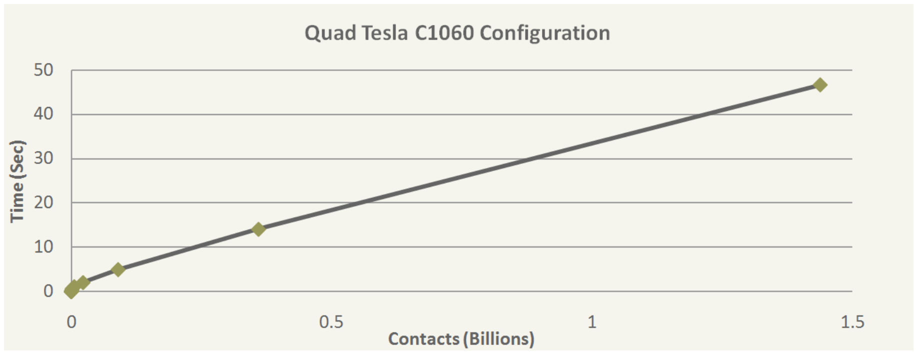


Figure 26: Collision time vs. contacts detected. This graph shows that the multi-GPU algorithm scales linearly and can detect more than a billion contacts in less than a minute.

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
0.1	6.1972	0.5436	5.4243	361,440
0.2	12.1190	1.0758	10.5881	718,377
0.3	18.2708	1.6183	15.9482	1,080,069
0.4	23.2806	1.9746	20.4606	1,403,784
0.5	29.2565	2.4568	25.7773	1,765,772
0.6	35.0433	2.9785	30.7971	2,124,639
0.7	40.5938	3.4695	35.6405	2,439,241
0.8	46.9516	4.0234	41.2297	2,838,832
0.9	52.6227	4.5272	46.1909	3,178,228
1.0	58.1518	4.9473	51.1686	3,548,594

the cylinder, which had to increase, was determined for each simulation using the number of spheres and their packing factor. Each test was run using an NVIDIA Tesla C1060 until the number of collisions and thus the simulation time per time step reached steady state. The open source dynamics engine Chrono::Engine was used for this simulation [1]. The GPU solver in Chrono::Engine was used in conjunction with the proposed algorithm.

In this simulation spheres were first dropped into the tank at fixed intervals until the number of objects in the tank reached the desired amount, at which time a hole in the bottom of the tank was opened. The simulation was then run until steady state flow was reached. The results are presented in Table 3 and graphed in Fig. 27 and Fig. 28. They indicate that even in a dynamics application, the collision detection algorithm scales linearly. Furthermore, the results show that the bulk of each time step was spent on the GPU dynamics solver portion of the simulation, with a small amount of time taken up by the collision detection step. These times are larger than the raw collision detection times presented earlier due to the pre- and post-processing required by the physics engine as it transfers and organizes data for use between the solver and collision detection.

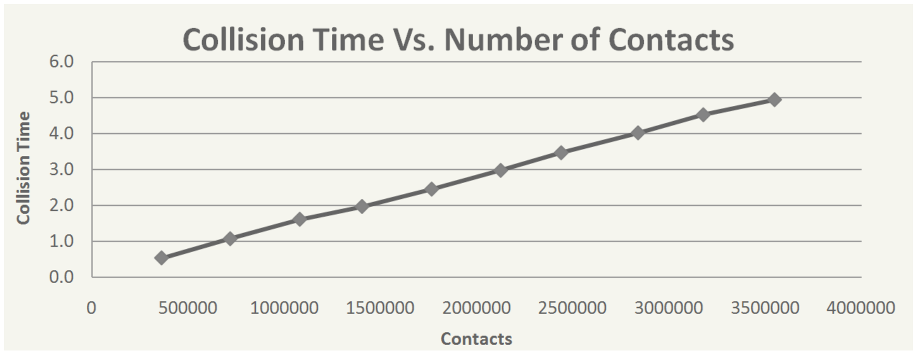


Figure 27: Collision time as the number of contacts increases.

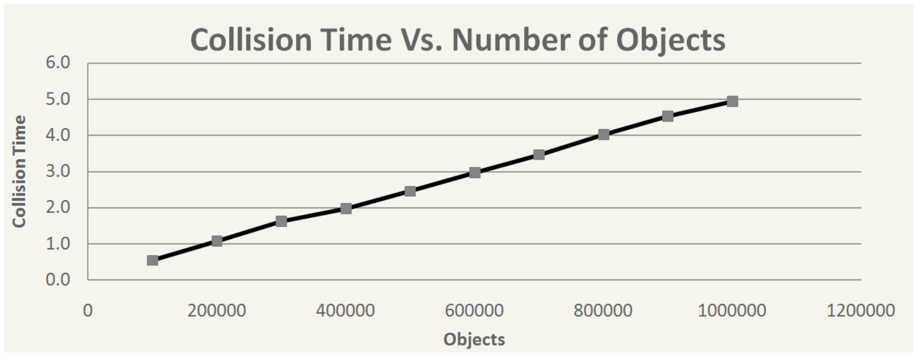


Figure 28: Collision time as the number of objects increases.

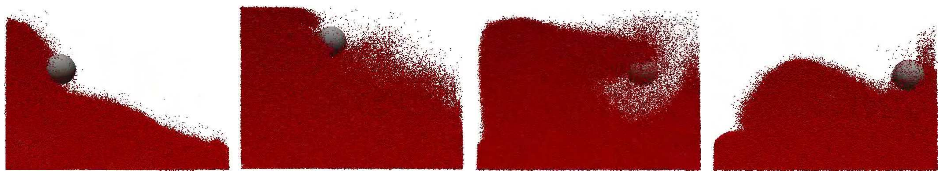


Figure 29: Light ball floating on 1 million rigid bodies moving around in a tank while interacting through friction and contact.

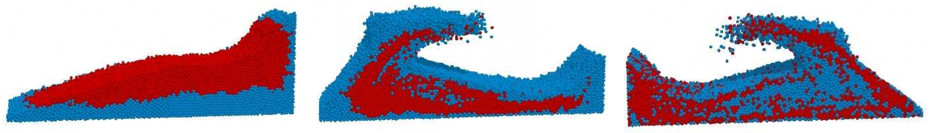


Figure 30: Benchmark: mixing of two granular materials, approximately 40,000 bodies.

6 Future Directions

A parallel numerical method has been proposed for the simulation of multibody mechanical systems with frictional contacts and bilateral constraints. The parallel method is based on an iterative approach that falls within the mathematical framework of measure differential inclusions and is backed by a rigorous convergence analysis.

Results obtained with the proposed method demonstrate that the GPU version of the dynamics solver is about 20x faster than the CPU version. A similar speedup has been obtained for the collision detection.

Acknowledgments

Financial support for D. Negrut is provided in part by the National Science Foundation Career Award CMMI–0840442. Financial support for A.Tasora is 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] Toby 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-2007-05, Simulation-Based Engineering Lab, University of Wisconsin, Madison, 2007.

[4] 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.

[5] M. Anitescu and A. Tasora. An iterative approach for cone complementarity problems for nonsmooth dynamics. *Computational Optimization and Applications*, 2008, in press.

[6] 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.

[7] 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.

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

[9] 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.

[10] Jong-Shi Pang and David Stewart. Differential variational inequalities. *Mathematical Programming*, 113(2):345–424, 2008.

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

[12] 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.

[13] 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.

[14] 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.

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

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

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

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

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

[20] J. J. Moreau. Unilateral contact and dry friction in finite freedom dynamics. In J. J. Moreau and P. D. Panagiotopoulos, editors, *Nonsmooth Mechanics and Applications*, pages 1–82, Berlin, 1988. Springer-Verlag.

[21] 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.

[22] M. Buck and E. Schömer. Interactive rigid body manipulation with obstacle contacts. *The Journal of Visualization and Computer Animation*, 9(4):243–257, 1998.

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

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

[25] 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.

[26] H. Mazhar, T. Heyn, and D. Negrut. A parallel method for large scale collision detection on the GPU. *Multibody Systems Dynamics, submitted*, 2010.

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

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