# Penetration-free Projective Dynamics on the GPU

LEI LAN, Clemson University & University of Utah, USA
GUANQUN MA, Clemson University & University of Utah, USA
YIN YANG, Clemson University, University of Utah & Timestep Inc., USA
CHANGXI ZHENG, Columbia University & Tencent Pixel Lab, USA
MINCHEN LI, University of California, Los Angeles & TimeStep Inc., USA
CHENFANFU JIANG, University of California, Los Angeles & TimeStep Inc., USA

Fig. 1. **Halloween party.** We present a GPU-based deformable simulation algorithm, which plugs an interior-point-like constraint formulation into the projective dynamics framework. We revise the computation for collision projection to accommodate continuous collision detection and barrier-based constraints. This simulation backbone is empowered by a novel GPU algorithm named A-Jacobi for a faster linear solve. With a more efficient root finding (and thus, faster CCD), our algorithm offers the non-intersection guarantee while still maintaining good efficiency. Here we show several snapshots of an interesting simulation consisting of deformable objects of various shapes. Several ghost-like creatures hang on a spooky tree, swaying in the wind. A few monster pumpkins fall and bounce. The total number of DOFs reaches 265K in this experiment. Our GPU algorithms runs between 7.7 to 26.8 FPS, and the time step size is $h = 1/100$ sec.

We present a GPU algorithm for deformable simulation. Our method offers good computational efficiency and penetration-free guarantee at the same time, which are not common with existing techniques. The main idea is an algorithmic integration of projective dynamics (PD) and incremental potential contact (IPC). PD is a position-based simulation framework, favored for its robust convergence and convenient implementation. We show that PD can be employed to handle the variational optimization with the interior point method e.g., IPC. While conceptually straightforward, this requires a dedicated rework over the collision resolution and the iteration modality to avoid incorrect collision projection with improved numerical convergence. IPC exploits a barrier-based formulation, which yields an infinitely large penalty when the constraint is on the verge of being violated. This mechanism guarantees intersection-free trajectories of deformable bodies during the simulation, as long as they are apart at the rest configuration. On the downside, IPC brings a large amount of nonlinearity to the system, making PD slower to converge. To mitigate this issue, we propose a novel GPU algorithm named A-Jacobi for faster linear solve at the global step of PD. A-Jacobi is based on Jacobi iteration, but it better harvests the computation capacity on modern GPUs by lumping several Jacobi steps into a single iteration. In addition, we also re-design the CCD root finding procedure by using a new minimum-gradient Newton algorithm. Those saved time budgets allow more iterations to accommodate stiff IPC barriers so that the result is both realistic and collision-free. Putting together, our algorithm simulates complicated models of both solids and shells on the GPU at an interactive rate or even in real time.

CCS Concepts: • **Computing methodologies** → **Physical simulation**.

Additional Key Words and Phrases: Physics-based simulation, Iterative solver, CCD, Barrier function, GPU

**ACM Reference Format:**
Lei Lan, Guanqun Ma, Yin Yang, Changxi Zheng, Minchen Li, and Chenfanfu Jiang. 2022. Penetration-free Projective Dynamics on the GPU. *ACM Trans. Graph.* 41, 4, Article 69 (July 2022), 16 pages. https://doi.org/10.1145/3528223.3530069

Authors' addresses: Lei Lan, Clemson University & University of Utah, USA, lan6@clemson.edu; Guanqun Ma, Clemson University & University of Utah, USA, guanqum@clemson.edu; Yin Yang, Clemson University, University of Utah & Timestep Inc., USA, yin5@clemson.edu; Changxi Zheng, Columbia University & Tencent Pixel Lab, USA, cxz@cs.columbia.edu; Minchen Li, University of California, Los Angeles & TimeStep Inc., USA, minchernl@gmail.com; Chenfanfu Jiang, University of California, Los Angeles & TimeStep Inc., USA, chenfanfu.jiang@gmail.com.


0730-0301/2022/7-ART69 $15.00
https://doi.org/10.1145/3528223.3530069

## 1 INTRODUCTION

Simulating elastically deformable models is a highly desired feature in many applications. This problem has been extensively studied in the graphics community. The challenges are multifaceted. First, the nonlinear elasticity of large-scale models leads to a high-dimension

optimization problem (i.e., in the variational form [Kane et al. 2000; Martin et al. 2011]) that needs to be solved repetitively at each time step. This computation is time-consuming with existing discretization tools like the finite element method (FEM) [Zienkiewicz et al. 1977]. The interactions among objects bring further complexities. Normally, it is required that two models do not overlap with each other at any simulation instance. Such spatial exclusivity needs extra safeguards for processing object contacts and collisions. Mathematically, this condition can be formulated as complementarity programming [Alizadeh et al. 1997]. For instance, linear complementarity programming (LCP) [Cottle et al. 2009] is a commonly chosen strategy in graphics applications. However, solving with complementarity constraints for a globally optimal solution is difficult and known NP-complete [Chung 1989]. Alternatively, one could resort to soft constraints such as penalty methods [Teschner et al. 2005] to avert the combinatorial search in LCP. The parameter tuning e.g., the stiffness/damping of each penalty term is rather tedious, and penalty methods can still produce intersections, especially if objects have fine and thin geometries (e.g., a piece of cloth) and/or when they undergo high-velocity movements. Such artifacts are not only visually annoying, but they may also negatively impact the downstream applications.

In this paper, we report a new solution aiming to address all the aforementioned challenges *simultaneously*. This is achieved by reworking the simulation pipeline with novel numerical procedures and dedicated GPU implementations. The backbone of our framework is based on projective dynamics (PD) [Bouaziz et al. 2014]. The non-penetration constraint on the other hand, is enforced with *barrier functions* following the paradigm of incremental potential contact (IPC) [Li et al. 2020]. To the best of our knowledge, it is the first successful attempt to algorithmically integrate those two simulation modalities. The core idea is treating PD as a generic nonlinear programming, which is further coupled with a displacement reversion process based on the continuous collision detection (CCD) to ensure per-step simulation snapshots are free of intersections. While PD is friendly with GPUs [Fratarcangeli et al. 2016; Wang 2015], injecting highly stiff barrier energies into the PD system still slows its convergence and thus the overall simulation performance. In addition, expensive CCD processing also consumes significant time budgets. This suggests we have to "gouge" more time from the solver part in order to keep our simulation efficient, preferably at an interactive or real-time frame rate.

Our answer to this dilemma is a novel GPU algorithm named *A-Jacobi*. We note that previous research efforts on GPU simulation focus primarily on how to convert a (sequential) numerical procedure (such as a direct linear solver [Higham 2009]) to a parallel one [Fratarcangeli et al. 2018]. Compared with other computations along the simulation pipeline like contact culling and CCD, existing parallel solvers however may not be able to use up the throughput of a modern GPU. A-Jacobi is designed based on this observation, and it aggregates multiple stationary Jacobi iterations [Kelley 1995] into one step. While per-iteration computation is more expensive, such overhead is invisible to the GPU under a careful implementation. We show that A-Jacobi is still a stationary method, and the Chebyshev acceleration remains effective. With a good precomputation and smart use of local caching in the shared memory, A-Jacobi becomes much faster (160% – 280%× speedup) than state-of-the-art GPU solvers for problems of tens or hundreds thousands of degrees of freedom (DOFs).

In addition to the solver-side improvement, we give a faster root-finding algorithm in CCD. We notice that computing an exact time of impact (TOI) timestamp is not needed in our framework – any time instances slightly before the actual TOI can serve the purpose of keeping all geometries intersection-free. This observation hints some leeways in CCD, which are exploited leading to a minimum-gradient Newton search algorithm. Instead of searching for a root along the current gradient, minimum-gradient Newton, as the name suggests, proceeds with a *locally minimal* gradient so that this secant line always hits the zero-crossing before the tangent line at each search iteration. Therefore the gradient computation can be skipped. This trick squeezes extra computational saves at each time step.

## 2 RELATED WORK

Finding good algorithms to simulate elastically deformable objects has been an active graphics research topic since 1980s [Terzopoulos and Fleischer 1988; Terzopoulos et al. 1987, 1988]. Basically, the simulation seeks a dynamic equilibrium among the external force, inertia force, and the nonlinear internal force. For large-scale models i.e., with hundred thousands of DOFs, the simulator needs to solve a high-dimension nonlinear system at each time step. Therefore, even the underlying numerical tools such as FEM [Zienkiewicz et al. 1977], finite difference method [Zhu et al. 2010], meshless method [Martin et al. 2010; Müller et al. 2005], mass-spring system [Liu et al. 2013], and material point method (MPM) [Gao et al. 2017] are well established, efficiently simulating high-resolution models is still a challenging problem.

Acceleration is often achieved using *model reduction*, which creates a subspace representation of fullspace DOFs. Modal analysis [Choi and Ko 2005; Hauser et al. 2003; Pentland and Williams 1989] and its first-order derivatives [Barbič and James 2005] are often considered as the most effective way for the subspace construction. Displacement vectors from recent fullspace simulations can also be utilized as subspace bases [Kim and James 2009]. Condensation [Teng et al. 2015] and Schur complement [Peiret et al. 2019] based formulation is also highly effective. Sometimes, it is viable to coarsen the geometric shape representation to prescribe the dynamics of a fine model like skin rigging widely used in modern animation systems. Analogously, Capell and colleagues [2002a] deformed an elastic body using an embedded skeleton; Gilles and colleagues [2011] used 6-DOF rigid frames to drive the deformable simulation; Faure and colleagues [2011] used scattered *handles* to model complex deformable models; Lan and colleagues [2020; 2021] exploited *medial axis transform* to build the mesh skeleton; Martin and colleagues [2010] used sparsely-distributed integrators named *elastons* to model the nonlinear dynamics of rod, shell, and solid uniformly. A noticeable benefit of model reduction is the size of the simulation no longer depends on the resolution of the model. On the downside, many reduction methods require expensive precomputations [Yang et al. 2015]. Because of the reduction, the accuracy compromise and the loss of simulation details are inevitable with model reduction.

Another line of contributions approaches to simulation speedup by designing customized numerical procedures. For instance, multi-resolution methods [Capell et al. 2002b; Grinspun et al. 2002] leverage hierarchical basis functions for the discretization. Similarly, a multigrid solver projects fine-grid residual errors onto a coarser grid, where linear or nonlinear iterations become more effective [Tamstorf et al. 2015; Wang et al. 2020; Xian et al. 2019; Zhu et al. 2010]. Hecht and colleagues [2012] proposed a delayed factorization scheme that reuses existing sparse Cholesky factorization as much as possible to save the computation. Quasi-Newton solvers use Hessian approximates, which can also be employed to accelerate the computation [Liu et al. 2017]. The domain decomposition method [Farhat et al. 2000] is a powerful method that breaks the original model into many small domains. It is often used with model reduction in graphics to enable per-domain subspace customization [Barbič and Zhao 2011; Kim and James 2012; Wu et al. 2015; Yang et al. 2013].

GPGPU platforms like nVidia CUDA [Sanders and Kandrot 2010] offer a different perspective to simulation acceleration. As the traditional solvers are intrinsically sequential, the core difficulty is how to map such computation to a parallelizable scheme [Wang and Yang 2016]. Position based dynamics or PBD [Macklin et al. 2016; Müller et al. 2007] reformulates the equation of motion based on particle positions, and the system solve becomes a series of constraint projections. This idea is generalized to PD [Bouaziz et al. 2014], wherein all the constraint projections can be carried out independently. Being a baseline simulator, several GPU methods have been successfully developed based on PD/PBD. For instance, Wang [2015] designed a Jacobi-based PD algorithm, and Chebyshev iteration [Golub and Varga 1961] was used to improve the convergence rate. Gauss-Seidel method has a better convergent behavior, but its GPU implementation is less straightforward and requires a DOF partition with coloring algorithms [Fratarcangeli et al. 2016].

In addition to performance-wise improvement, existing work has also put a lot of efforts improving the quality of the simulation such as more expressive material models [Martin et al. 2011; Smith et al. 2018] or more robust nonlinear programming under extreme-scale deformations [Irving et al. 2004]. Oftentimes, it is required that models do not intersect with each other during the simulation. Interpenetration among virtual objects is visually annoying and could lead to severe errors to follow-up applications like fabrication and robot manipulation. For deformable shapes, this requirement is enforced at all the surface geometric primitives as a set of nonlinear inequality constraints [Bridson et al. 2002; Daviet et al. 2011; Harmon et al. 2009, 2008; Otaduy et al. 2009]. In PBD/PD systems, collisions are often treated as a unilateral constraint . Many methods have been proposed to improve the performance and robustness within PD-like frameworks [Komaritzan and Botsch 2018, 2019; Overby et al. 2017; Wang et al. 2021]. However, the detection algorithms used are discrete, and interpenetration can still be possible for fast-moving geometries.

Recently, Li and colleagues [2020] presented a solution named IPC to tackle inequality constraints induced from collisions and contacts. They explored an interior point method with a logarithmic barrier penalty. Intuitively, this penalty yields an increasingly stronger repulsion force when objects become closer enough to each other. As a result, the overall variational optimization becomes unconstrained. At each Newton solve, a CCD-based line search is followed to ensure all the primitives are free from intersections before any displacement update to be committed. This method has then been successfully employed for reduced simulation [Lan et al. 2021], co-dimensional simulation [Li et al. 2021b], rigid body simulation [Ferguson et al. 2021], embedded FEM [Choo et al. 2021], FEM-MPM coupling [Li et al. 2021a], and even for geometric modeling [Fang et al. 2021]. We are strongly inspired by the versatile applicability of IPC, and seek for its possible GPU implementation so that high-quality non-intersecting simulation can also be executed interactively or even in real time. At first sight, PD seems to be a promising starting point for its convenient implementation and GPU friendliness. However, few preliminary tests suggest otherwise as PD immediately becomes unusable with CCD and IPC barriers. We carefully examine those technical obstacles and revise the vanilla PD framework by correcting its local projection for CCD-based collisions and delaying overaggressive barrier updates. Implementation-wise, we propose a novel aggregated Jacobi solver and faster CCD root finding algorithm. Putting together, we enable efficient simulation of complicated models with rich collision and contact events. Simulations of hundred thousands DOFs in the fullspace now become real-time or nearly real-time, and they are guaranteed to be free of interpenetration.

## 3 BACKGROUND

To make our presentation self-contained, we start with a brief review of PD and IPC, while referring the reader to the related literature [Bouaziz et al. 2014; Li et al. 2020] for more details.

Given a time integration scheme such as implicit Euler, the starting point of PD is a variational optimization in form of:

$$\arg\min_{x} E(x,\dot{x}) + \Psi(x), \quad E = \frac{1}{2h^2}\left\|\mathsf{M}^{\frac{1}{2}}(x-z)\right\|_F^2. \tag{1}$$

Here $x$ denotes the positions of all the vertices. $h$ is the time step size. $E$ is the momentum potential, and $\mathsf{M}$ is the mass matrix. $z = x^* + h\dot{x}^* + h^2\mathsf{M}^{-1}f_{\text{ext}}$ is a known vector based on the previous displacement $x^*$, velocity $\dot{x}^*$, and an external force $f_{\text{ext}}$. $\Psi$, the elastic potential, is typically a summation of various potential terms, accounting for penalties against deformations of shearing, stretching, bending, volume/length changes, etc. Its negated gradient describes the internal elastic force, $f_{int} = -\nabla\Psi(x)$ (under hyperelasticity assumption [Ogden 1997]).

Unlike Newton-type nonlinear programming [Nocedal and Wright 2006], PD decouples Eq. (1) with an auxiliary variable $y_i$. For the $i$-th constraint set, $y_i$ denotes the *target positions* of pertaining vertices. $y_i$ can also be viewed as a high-dimensional point on the constraint manifold, satisfying the constraint $C_i(y_i) = 0$, whose Euclidean distance to its *current position* $x_i$ is minimized. The optimization of Eq. (1) progresses in a local-global alternating manner, which will be referred to as L-G iterations in the rest of the paper. In the local step, we compute $y_i$ for each constraint a.k.a. *local projection*:

$$\arg\min_{y_i} \frac{1}{2}\left\|\mathsf{A}_i\mathsf{S}_i x - \mathsf{B}_i y_i\right\|_F^2, \quad \text{s.t. } C_i(y_i) = 0, \tag{2}$$

where $\mathsf{S}_i$ is a selection matrix choosing DOFs for the constraint $C_i$ out of the position vector $x$ i.e., $x_i = \mathsf{S}_i x$; $\mathsf{A}_i$ and $\mathsf{B}_i$ are both constant, often Laplacian-like, to facilitate the distance measure.

While intrinsically nonlinear, Eq. (2) only involves a small number of DOFs and thus can be solved concurrently for all the constraints. This feature makes the local step particularly suitable for GPUs.

After the local projection, the global step solves a linear system for $x$:

$$\left(\frac{\mathrm{M}}{h^2}+\sum_i \omega_i \mathrm{S}_i^\top \mathrm{A}_i^\top \mathrm{A}_i \mathrm{S}_i\right) x = \frac{\mathrm{M}}{h^2} z + \sum_i \omega_i \mathrm{S}_i^\top \mathrm{A}_i^\top \mathrm{B}_i y_i. \tag{3}$$

Here each constraint is assigned with a weight coefficient $\omega_i$ to embody its "importance". PD treats collision as a constraint as well and sets the target position $y_i$ of the penetrating vertex as its closest projection on the collision plane. The collision constraint also changes the linear system in the global step Eq. (3). Therefore, Eq. (3) has to be re-factorized if a direct solver is in use. Nevertheless, collision processing in PD remains penalty-based, just like using a mass-less linear spring to drag each colliding vertex out of the penetrating plane. It is known that such soft constraint model does not guarantee all the collisions can be successfully resolved at the end of a time step.

IPC [Li et al. 2020] offers a more robust way to process collisions. Without enforcing inequality or complementarity constraints explicitly, IPC exploits a more sophisticated penalty mechanism. Let $d_k$ denote the unsigned distance between the $k$-th pair of surface primitives (i.e., vertex-triangle or edge-edge). IPC uses a *barrier potential* to penalize the collision,

$$B(d_k) = \begin{cases} -(d_k-\hat{d})^2 \ln\left(\frac{d}{\hat{d}}\right), & 0 < d_k < \hat{d} \\ 0, & d_k \geq \hat{d}, \end{cases} \tag{4}$$

where $\hat{d}$ is a user-provided tolerance of the collision resolution. $B(d_k)$ diverges if $d_k < \hat{d}$ and approaches $\infty$ when $d_k \to 0$. Consequently, as long as we ensure $d_k$ being positive at the beginning of a time step (i.e., all primitives are separate), the use of $B(d_k)$ prevents any future interpenetration with an increasingly stronger repulsion.

Our idea is to use the barrier function (Eq. (4)) in PD for the collision resolution, obtaining the optimization problem

$$\arg\min_x E(x,\dot{x}) + \Psi(x) + B(x), \quad B(x) = \sum_k \omega_k B_k(x_k). \tag{5}$$

However, solving this optimization problem is not easy. In order to obtain the specific form of $B$, one needs to compute $d_k$ for all the primitive pairs. If the distance is smaller than $\hat{d}$, $B_k$ activates yielding a highly stiff barrier potential. To battle the strong nonlinearity induced by the barrier function, all the existing IPC implementations [Ferguson et al. 2021; Li et al. 2020, 2021b] rely on precise Hessian information and Newton's method to compute the search direction of displacement update. At each Newton iteration, a line search is also needed to ensure that the energy decreases with the proposed displacement update, and also the new vertex positions remain interpenetration-free. Yet, this solving process does not directly fit the local-global strategy in PD, as we will explain next.

## 4 OUR METHOD

Provided that both PD and IPC are based on the variational optimization, a natural idea is to construct a simulator by piecing the best from each party to enjoy the efficiency of PD with intersection-free guarantee. Yet, a naïve combination of these two techniques does not produce physically correct results, and two fundamental challenges must be addressed.

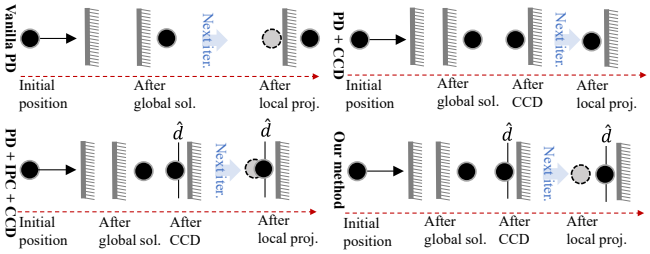


Fig. 2. **Different synergic strategies for PD and CCD.** Appending CCD pruning after each PD global solve does keep the vertex collision-free. However, it also leads to sticking artifacts ignoring the colliding velocity of the vertex. Adding the barrier is not helpful unless the computation of local projection for barrier-based collision constraints is revised properly.

### 4.1 Challenge I: Battle the Stickings

Our first attempt is to follow the filtered line search in IPC and truncate the newly computed vertex positions $x$ with CCD. This process, which we call *CCD pruning*, linearly downscales the vertex position change $\Delta x$ after the global solve, according to TOI to roll back all the vertices to a collision-free state. The result, however, is problematic – the colliding vertices stick to each other even under a strong collision penalty. This pilot study suggests some incompatibilities between PD-like solvers and CCD.

We examine this problem in a simple case study illustrated in Fig. 2: a vertex moves toward a plane and penetrate into it by the end of an L-G iteration. Applying the CCD pruning after the global solve reverts the vertex back to its pre-collision location. This step is not included in the vanilla PD. As a result, a collision constraint is generated at the next L-G iteration, and the corresponding target position (the gray vertex in the figure) yields a penalty drag attracting the vertex to this collision-free position. However, the use of CCD pruning introduces a deadlock: the L-G iterations no longer generate collision constraints (due to the pruning), and the vertex becomes short of momentum to be pushed away from the plane.

Unlike IPC, adding a stiffer barrier energy of Eq. (4) does not resolve this problem. This is explained in the "PD + IPC + CCD" case in Fig. 2. After the CCD pruning, if the distance $d$ between the vertex and the plane is smaller than $\hat{d}$, instead of a collision constraint we generate a *barrier constraint* at the next L-G iteration. If we use this constraint in the PD framework, the vertex target position of this barrier constraint will be set $\hat{d}$-away from the plane (i.e., the nearest towards its current position while being barrier-free). Thus, the distance between the target position and the current position is always $\hat{d} - d$ in a barrier constraint. It is not an issue in the IPC framework because the magnitude of the barrier gradient increases sharply to infinity as $d$ approaches to 0. It eventually yields a sufficiently strong repulsion force to push the vertex back. In contrast, PD is a position-based model, in which a constraint force is never explicitly formulated. Instead, *the concept of the constraint*

*force is manifested as the offset between the target and current positions of the vertex.* Consider again the "vanilla PD" case in Fig. 2. The vertex springs back because of the difference between its current and target positions. As the target position is always on the positive side of the plane, for a given $h$, the deeper penetration we have, the stronger rebound the constraint projection will produce.

Then what role does $\omega$ play? The weight of the constraint has long been understood as its "stiffness". However, unlike methods that explicitly compute internal forces, the stiffness in PD does *not* directly influence the collision rebound of the vertex: increasing the weight coefficient for the collision constraint does not strengthen the rebound of the vertex (e.g., see Fig. 3). The constraint weight makes difference only when the vertex participates in multiple constraints; the target position of the constraint with a bigger weight is to be satisfied under a higher priority than target positions from other (softer) constraints. Therefore the barrier is only helpful to ensure the follow-up L-G iteration is free of the barrier constraint (and free of the collision), but it does not address the sticking issue.

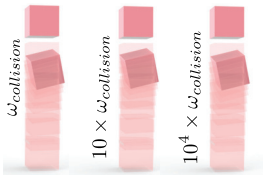


Fig. 3. **Rebound and $\omega$.** The rebound of a falling box does not change noticeably even after we increase the weight coefficient of the collision constraint ($\omega_{collision}$) by four orders.

*Proposed barrier projection.* Based on the above analysis, we propose to revise the local projection of barrier-based collision constraints in the following way. When a barrier potential is activated ($0 < d_k < \hat{d}$), we set the target position for that barrier constraint as the vertex position at the end of the time step if the collision had occurred (which in reality, is forestalled by CCD pruning). Let $t_0$ and $t_1$ be the timestamps before and after the time step. CCD returns TOI $t_I \in (t_0, t_1]$ of the vertex, which approximates the time instance when the collision occurs[1]. Assuming $\dot{x}$ stays constant within the step, the target position of the vertex can then be computed as:

$$y = x + (t_I - t_0)\dot{x} + (t_1 - t_I)\big(\mathsf{I} - (1+\epsilon)nn^\top\big)\dot{x}, \tag{6}$$

where $n \in \mathbb{R}^3$ is a unit vector of the collision normal, $\mathsf{I}$ is an identity matrix, and $\epsilon \in [0, 1]$ represents the restitution of the collision. In Eq. (6), $x + (t_I - t_0)\dot{x}$ estimates the collision position of the vertex. The velocity component in parallel to $n$ is then reflected under the coefficient of restitution. After that, the vertex travels for another $t_1 - t_I$ time to reach its target position. When the collision is between a pair of primitives, we scale the rebound of each party based on their inverse mass so that the momentum is preserved.

## 4.2 Challenge II: Avoid Excessive Pruning

The use Eq. (6) for local projection of barrier constraints resolves sticking artifacts. But the number of iterations needed in each time step becomes large, and per L-G vertex position changes are jittery and bumpy.

[1]In our implementation, $t_I$ is slightly smaller than the actual TOI (more in § 6). In IPC implementation [Li et al. 2020], $t_I = 0.8 \cdot (t_{TOI} - t_0) + t_0$.

The root of this problem is *excessive* CCD pruning. Every CCD changes the landscape of the optimized energy by adding new or removing barrier potentials (i.e., Eq. (4)). If such modifications happen too frequently, L-G iterations could yield oscillating and conflicting search patterns, which significantly slow the convergence or even diverge the optimization. An illustrative example is shown in Fig. 4. The vertex moves following a descent direction computed via the current global solve (the red curve), and the next L-G iteration could take it further right. Now, a CCD pruning jumps in and brings a different optimization target (the blue curve). In the next L-G iteration, the vertex moves along the descent direction on the blue curve, which cancels its previous search. If another CCD changes the optimization landscape back to one similar to the red curve, the vertex may keep moving back and forth between those two configurations.

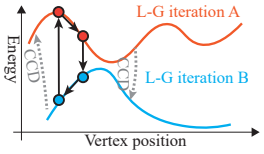


Fig. 4. **Oscillating L-G search.** Excessive CCD pruning alters the target function too frequently, resulting in oscillating search directions and slow convergence.

*A quick discussion.* The fundamental reason behind this issue is we are unsure about the objective function (i.e., which barriers are nonzero in Eq. (5)) when collisions occur. A CCD pruning provides an *estimation* of the target function (i.e., with an active set), and we should evaluate the feasibility with a well-approximated solution to this active set before switching to the next. In IPC [Li et al. 2020], each projected Newton iteration often contributes sufficient improvement over the solution to the current active set (knowing Newton's method has a default step size of one). Therefore this issue is hidden under Newton-based IPC implementation. Other iterative schemes (including our method) with a smaller local convergence rate shall all face this problem.

## 4.3 Projective IPC with Nested Loop

With this side effect of CCD pruning in mind, we now describe our algorithm in detail. As outlined in Alg. 1, we organize the L-G iterations in two levels, namely the outer loop and the inner loop (starting at line 4 and line 7 in Alg.1 respectively). Note that the system is always intersection-free at the beginning of each outer loop. The simulator first evaluates the barrier potential $B_k$ (Eq. (4)) for all surface primitive pairs with $d_k < \hat{d}$ and computes the corresponding target position (i.e., using Eq. (6)). After that, we step into the inner loop using the standard L-G alternating strategy. At each inner loop, the solver computes the momentum potential $E$ (line 8), the elastic potential $\Psi$ (line 9), and updates vertex positions to $\tilde{x}$ via the global solve. Here ElasticProjection($\tilde{x}$) performs local projection for different deformable models such as tetrahedral strain, edge length change, bending curvature, area/volume preservation etc. However, an inner iteration does not update the barrier potential assuming the collision landmark remains unchanged.

The purpose of the inner loop is to offer the solver sufficient compliance for the pursuit of a better local minimum under the latest barrier (collision) configuration. We quit the inner loop when the change rate ($\delta E$) of total variational potential $E + \Psi + B$ w.r.t. the previous inner iteration is lower than a given threshold $\varepsilon_{inner}$, which

**ALGORITHM 1:** Projective IPC solver.

$z \leftarrow x^* + h\dot{x}^* + h^2\mathsf{M}^{-1}f_{ext}$;
$\tilde{x} \leftarrow x^* + h\dot{x}^* + \frac{h^2}{4}\ddot{x}^*$; // $\tilde{x}$ now is a predicted position
$x \leftarrow x^*$, $\Delta x \leftarrow \tilde{x} - x$;
**while** $\|\Delta x\|^2 > \varepsilon_{outer}$ **do**
  // $\varepsilon_{\text{outer}} = 10^{-4}$
  $B \leftarrow \text{BarrierProjection}(x)$; // barrier projection (§ 4.1)
  $\delta E \leftarrow +\infty$; // $\delta E$ is per-iteration potential change rate
  **while** $\delta E > \varepsilon_{inner}$ **do**
    // $\varepsilon_{\text{outer}} = 10^{-2}$
    $E \leftarrow \frac{1}{2h^2}\|\mathsf{M}^{-1}(\tilde{x} - z)\|_F^2$; // update momentum potential
    $\Psi \leftarrow \text{ElasticProjection}(\tilde{x})$;
    $\tilde{x} \leftarrow \text{GlobalSolve}$;
    update $\delta E$;
  **end**
  $\text{CollisionCulling}(\tilde{x})$; // patch-based GPU culling (§ 7.2)
  $t_I \leftarrow \text{CCD}(x, \tilde{x})$; // minimum-gradient Newton method (§ 6)
  $\tilde{x} \leftarrow x + \frac{t_I}{h} \cdot (\tilde{x} - x)$; // per outer loop CCD pruning (§ 4.2)
  $\Delta x \leftarrow \tilde{x} - x$, $x \leftarrow \tilde{x}$; // update $x$ and $\Delta x$
**end**
$\dot{x} \leftarrow \frac{x - x^*}{h}$, $\ddot{x} \leftarrow \frac{\dot{x} - \dot{x}^*}{h}$; // velocity and acceleration update

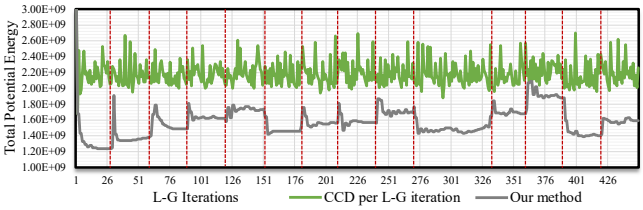


Fig. 5. **Variation of total potential.** This figure plots the variation of the total system potential within a time step if one chooses to perform CCD pruning after each L-G iteration (the green curve) or to use the proposed two-level iteration scheme as in Alg. 1 (the gray curve).

suggests the deformation of the model well equilibrates barrier (collision) constraints generated at the beginning of the inner loop. In fact, we *permit* temporary intersections of vertices within the inner loop, which are truncated before we move to the next outer iteration. To this end, we invoke a CCD between the latest (possibly intersecting) position vector $\tilde{x}$ and $x$ (the most recent intersection-free state). $\tilde{x}$ is then pruned by the CCD, and we advance to the next outer iteration with a renewed barrier potential. When an outer iteration does not bring sufficient improvements over $x$, we consider the optimization convergent and start the same procedure for the next time step.

A concrete example is shown in Fig. 5, which plots the variation of the total system potential $E + \Psi + B$ within a time step when the falling bunny hits the floor. If we apply CCD after each L-G iteration and truncate the vertex displacement accordingly, the total system potential does not converge (or converges very slowly) and oscillates significantly. This observation endorses our previous analysis: changing the active set too often negatively impacts the convergence of barrier-based optimization. Alg. 1 on the other hand, sufficiently lowers the potential at each outer loop (indicated by red dash lines in the figure) to check the feasibility of the active set. While a follow-up CCD pruning often drastically alters the target potential, which can be clearly observed as sharp energy changes after each outer loop, the overall optimization proceeds smoothly and converges when $x$ becomes stabilized.

The L-G iterations in Alg. 1 can be viewed as a generic optimization scheme. Therefore, any existing Hessian-free nonlinear programming such as nonlinear CG, nonlinear Gauss-Seidel, ADMM, quasi-Newton, or Newton-Krylov could fit in Alg. 1. The key adjustment is to avoid overaggressive barrier updates and CCD-based displacement pruning. A major reason we choose PD as our baseline nonlinear solver is its GPU-friendly implementation. The local projection is independent at each constraint set, and many contributions have shown that the linear solve at the global step can also be conveniently mapped to GPU [Fratarcangeli et al. 2016; Wang 2015]. Next, we discuss a new GPU algorithm that further improves the performance of the GPU linear solve.

## 5 AGGREGATED JACOBI SCHEME

While local projection step is trivially parallelizable, the global step is not, as it needs to solve a standard linear system in the form of $\mathsf{A}x = b$, where $\mathsf{A} = \frac{\mathsf{M}}{h^2} + \sum \omega_i \mathsf{S}_i^\top \mathsf{A}_i^\top \mathsf{A}_i \mathsf{S}_i$ and $b = \frac{\mathsf{M}}{h^2} z + \sum \omega_i \mathsf{S}_i^\top \mathsf{A}_i^\top \mathsf{B}_i y_i$ (i.e., Eq. (3)). Wang [2015] showed that each global step can be quickly solved using the Jacobi method, whose convergence is further improved by Chebyshev [Axelsson 1977]. This strategy is a baseline of several follow-up GPU algorithms [Fratarcangeli et al. 2016, 2018; Wang and Yang 2016]. In this section, we describe an aggregated Jacobi scheme (A-Jacobi). The goal is to better exploit the computation capacity of the GPU. In addition, our method is compatible with the Chebyshev-based method [2015] and thereby also benefits from its performance improvement.

### 5.1 A Woodbury Perspective of Jacobi

Jacobi method is a well-known iterative linear solver based on matrix splitting. Given the global matrix A, we partition it into a diagonal matrix $\mathsf{D} = \texttt{diag}(\mathsf{A})$ and an off-diagonal matrix B such that $\mathsf{A} = \mathsf{D} - \mathsf{B}$. The solving procedure starts with an initial guess of $x^{(0)}$ and advances following the recursive relation

$$x^{(k)} = \mathsf{D}^{-1}b + \mathsf{R}x^{(k-1)}, \quad \mathsf{R} = \mathsf{D}^{-1}\mathsf{B}. \tag{7}$$

Note that $\mathsf{D}^{-1}b$ is shared across all iterations. Therefore, the per-iteration computation is dominated by $\mathsf{R}x^{(k-1)}$. R is a sparse matrix and works as a differential operator on each vertex locally. On the GPU, each thread takes care of an inner product between $x^{(k-1)}$ and the corresponding row vector from R. For instance, for the shell model using quadratic bending [Wardetzky et al. 2007], this inner product involves a weighted summation over one-ring neighbors of this vertex on the mesh (e.g., see Fig. 17). While the parallel scheme is straightforward and easy to implement, *per-thread computation is too light and does not even justify the overhead for the thread initialization with CUDA.*

A relevant technique is Woodbury formula [Hager 1989], which forms the foundation of the quasi-Newton family. The Woodbury

equation relates the inverse of a matrix before and after a rank-$k$ update:

$$(\mathsf{D}+\mathsf{UBV})^{-1} = \mathsf{D}^{-1} - \mathsf{D}^{-1}\mathsf{U}(\mathsf{B}^{-1}+\mathsf{V}\mathsf{D}^{-1}\mathsf{U})^{-1}\mathsf{V}\mathsf{D}^{-1}. \quad (8)$$

A special case of Eq. (8), by setting both U and V as identity matrices, can be written as

$$\mathsf{A}^{-1} = \mathsf{D}^{-1} + \mathsf{D}^{-1}\mathsf{B}\mathsf{D}^{-1} + \mathsf{D}^{-1}\mathsf{B}\mathsf{D}^{-1}\mathsf{B}\mathsf{D}^{-1} + \cdots = \sum_{k=0}^{\infty} \mathsf{R}^k\mathsf{D}^{-1}. \quad (9)$$

Right-multiplying $b$ at both sides of Eq. (9) reveals the structure of $x^\star$, the solution of $\mathsf{A}x = b$:

$$x^\star = \mathsf{A}^{-1}b = \sum_{k=0}^{\infty} \mathsf{R}^k\mathsf{D}^{-1}b = x_{(0)} + x_{(1)} + \cdots + x_{(k)} + \cdots, \quad (10)$$

from which we have $x_{(0)} = \mathsf{D}^{-1}b$, $x_{(1)} = \mathsf{R}\mathsf{D}^{-1}b$, $x_{(2)} = \mathsf{R}^2\mathsf{D}^{-1}b$, and so forth. The above series converges as long as the spectral radius $\rho(\mathsf{R}) < 1$. Suppose the initial guess $x^{(0)} = \mathsf{D}^{-1}b + \delta x$. It is easy to see that $x_{(k)}$ and $x^{(k)}$ are related by

$$\begin{aligned}
x^{(0)} &= \mathsf{D}^{-1}b + \delta x,\\
x^{(1)} &= \mathsf{D}^{-1}b + \mathsf{R}(\mathsf{D}^{-1}b + \delta x) = \mathsf{D}^{-1}b + \mathsf{R}\mathsf{D}^{-1}b + \mathsf{R}\delta x,\\
x^{(2)} &= \mathsf{D}^{-1}b + \mathsf{R}\mathsf{D}^{-1}b + \mathsf{R}^2\mathsf{D}^{-1}b + \mathsf{R}^2\delta x,\\
&\cdots\\
x^{(k)} &= \sum_{j=0}^{k} x_{(j)} + \mathsf{R}^k\delta x.
\end{aligned}$$

The initial deviation of $\delta x$ will be eliminated by $\mathsf{R}^k$ for a sufficiently large $k$, and $x^{(k)}$ converges to $x^\star$.

Eq. (10) suggests that a solution of $\mathsf{A}x = b$ cannot be obtained unless the series is sufficiently expanded. Unfortunately, one Jacobi iteration only extends this series by one term. The corresponding computation is lightweight, but it is also unable to fully exploit the GPU resource. More importantly, the entire Jacobi procedure remains sequential since the next iteration cannot occur without completing the current one.

## 5.2 A-Jacobi and Its GPU Implementation

Our A-Jacobi is based the following intuition: we want each iteration to use a higher order expansion of Eq. (10) of multiple $x_{(k)}$ terms. In addition, we ensure that per-iteration computation is still parallelizable and scaled to fully exploit the GPU capacity. Concretely, we further expand Eq. (7):

$$\begin{aligned}
x^{(k)} &= \mathsf{D}^{-1}b + \mathsf{R}x^{(k-1)} = \mathsf{D}^{-1}b + \mathsf{R}\mathsf{D}^{-1}b + \mathsf{R}^2x^{(k-2)}\\
&= (\mathsf{R}^0 + \mathsf{R})\mathsf{D}^{-1}b + \mathsf{R}^2x^{(k-2)} = \sum_{j=0}^{\ell-1} \mathsf{R}^j\mathsf{D}^{-1}b + \mathsf{R}^\ell x^{k-\ell}.
\end{aligned}$$

We refer to $\ell$, the largest repeat count of R in the above derivation, as the *order* of A-Jacobi and re-index the recursion:

$$x^{(k)} = \sum_{j=0}^{\ell-1} \mathsf{R}^j\mathsf{D}^{-1}b + \mathsf{R}^\ell x^{(k-1)} = \widetilde{\mathsf{D}}b + \widetilde{\mathsf{R}}x^{(k-1)}. \quad (11)$$

One can easily see that the first-order A-Jacobi is just the regular Jacobi method with $x_{(k)} \leftarrow \mathsf{R}\mathsf{D}^{-1}x_{(k-1)}$ being computed one by one.

An $\ell$-order A-Jacobi computes a batch of $\ell$ terms in one iteration, which can be unfolded as:

$$\begin{bmatrix} x_{(k)} \\ x_{(k+1)} \\ x_{(k+2)} \\ \vdots \end{bmatrix} \leftarrow \begin{bmatrix} \widetilde{\mathsf{R}} & & & \\ & \widetilde{\mathsf{R}} & & \\ & & \widetilde{\mathsf{R}} & \\ & & & \ddots \end{bmatrix} \begin{bmatrix} x_{(k-\ell)} \\ x_{(k-\ell+1)} \\ x_{(k-\ell+2)} \\ \vdots \end{bmatrix}. \quad (12)$$

The key question is how to assemble $\widetilde{\mathsf{R}}$ and evaluate the matrix-vector product of $\widetilde{\mathsf{R}}x^{(k)}$ efficiently on the GPU.

To answer this question, we first need to understand how the global matrix A is constructed and updated during the simulation. If all the constraint sets stay unchanged, A is constant; D and B are constant; and R is constant too. In this case, we can just precompute $\widetilde{\mathsf{R}}$. This is conceptually similar to pre-factorize A with a direct solver (i.e., an alternative form of precomputing $\mathsf{A}^{-1}$) as did in the original PD algorithm [Bouaziz et al. 2014].

On the other hand, our simulation framework processes collisions implicitly using IPC barriers. Each CCD could possibly induce a different A, and a precomputed $\widetilde{\mathsf{R}}$ is not available with contacts and self-collisions. Fortunately, a bulk of $\widetilde{\mathsf{R}}$ assembly is still precomputable. This is because an IPC barrier only alters diagonal elements in A, leaving its off-diagonal part i.e., B collision-invariant. Consider the first-order A-Jacobi: $\mathsf{R}x^{(k)} = \mathsf{D}^{-1}\mathsf{B}x^{(k)}$ for a nonzero vector $x^{(k)}$. It is easy to see that:

$$[\mathsf{R}x^{(k)}]_i = D_{ii}^{-1}B_{ij}x_j^{(k)}. \quad (13)$$

Here $B_{ij}$ is the element at the $i$-th column and the $j$-th row of matrix B, and we use the summation convention for brevity. The above relation tells that each element in $x^{(k+1)} = \mathsf{R}x^{(k)}$ is a linear function of $B_{ij}$, scaled by $D_{ii}^{-1}$. We also know that B is sparse, and the summation $B_{ij}x_j^{(k)}$ only needs to loop over incident vertices and edges of vertex $i$ such that $B_{ij}$ corresponds to an edge, and $x_j^{(k)}$ is associated with a vertex.

When the order of A-Jacobi increases to two (i.e., $\ell = 2$), we have a similar equation of:

$$[\mathsf{R}^2x^{(k)}]_i = \overbrace{D_{ii}^{-1}D_{ss}^{-1}\underbrace{B_{is}B_{sj}}_{\text{A-product}}}^{\text{A-coefficient}} x_j^{(k)}. \quad (14)$$

Eq. (14) simply says $[\mathsf{R}^2x^{(k)}]_i$ becomes a quadratic function of $B_{ij}$ in second-order A-Jacobi. While $D_{ii}^{-1}$ and $D_{ss}^{-1}$ are changing under different collision/contact configurations, $B_{is}B_{sj}$ can be precomputed. We hereby refer to such a product of multiplying different $B_{ij}$ elements as an *aggregated product* or A-product. After scaled by $D_{ii}^{-1}$ and $D_{ss}^{-1}$, an A-product becomes an *aggregated coefficient* i.e., A-coefficient.

The shared summation index $s$ in Eq. (14) also suggests we should traverse not only the immediately adjacent vertices and edges but also the ones incident to them i.e., two-ring neighbors. In general, as the order of A-Jacobi grows, the order of the polynomial increases accordingly, covering a wider neighborhood. The spectral radius $\rho(\mathsf{R}^\ell)$ is also better shaped than $\rho(\mathsf{R})$ as long as $\rho(\mathsf{R}) < 1$. Interestingly, it is noteworthy that the actual computation load of one

$\ell$-order A-Jacobi iteration is *heavier* than $\ell$ regular Jacobi iterations. This is because calculating $\mathsf{R}(\mathsf{R}x)$ always involves fewer floating point operations than $(\mathsf{RR})x$. Therefore the advantage of A-Jacobi is only tangible with a dedicated GPU implementation. This however is quite common in the design of GPU algorithms – we exploit the capacity of GPU to consume redundant computations and trade them for observable performance gains. Please find more implementation details of A-Jacobi solver in the Appendix A.

### 5.3 Weighted A-Jacobi and Chebyshev

Weighted Jacobi is a common Jacobi variation by using a damped iteration matrix $\mathsf{R} \leftarrow \mathsf{I} - w\mathsf{D}^{-1}\mathsf{A}$ (in this case $\mathsf{D} \leftarrow w\mathsf{D}$ is also scaled by $w$). It is easy to see that we can also aggregate multiple weighted Jacobi iterations following the same procedure of A-Jacobi. For stiffer problems, damped iteration matrix has a smaller spectral radius, which will be further enhanced by A-Jacobi because $\rho(\widetilde{\mathsf{R}}) = \rho^{\ell}(\mathsf{R}) < \rho(\mathsf{R})$.

A-Jacobi is compatible with Chebyshev. From Eq. (10), it is easy to see that the sequence of $x_{(0)}, x_{(1)}, x_{(2)}$ etc. from regular Jacobi iterations spans a Krylov subspace of R. Each new search direction is generated by further multiplying the current one with R. The Chebyshev method aims to improve the current search direction based on the previous Krylov basis vectors. This mechanism also applies to A-Jacobi: the Krylov basis vectors are now expanded by $\widetilde{\mathsf{R}}$. Therefore, Chebyshev can also be used with A-Jacobi to further accelerate its convergence.

## 6 FAST ROOT APPROXIMATE IN CUBIC CCD

CCD plays a pivotal role in our framework and is the enabler of the non-intersection guarantee. CCD occurs between a pair of surface primitives, edge-edge or vertex-triangle. The trajectories of primitives are linearized within a time step, and a CCD query tells whether and when their trajectories overlap. This query can be formulated as a root-finding problem of a cubic equation [Provot 1997]. Our framework invokes CCD at the end of each outer loop (line 14, Alg. 1), and we need to perform multiple CCDs in one time step. Existing CCD algorithms aim to find TOI between the primitive pair, either analytically [Brochu et al. 2012] or numerically [Harmon et al. 2009]. We notice that however, an exact TOI is not really needed in our framework. At TOI, two primitives become just in contact i.e., $d_k = 0$. Under this situation, the barrier potential of Eq. (4) is ill-defined. Therefore once a collision is confirmed, we always backtrack to an earlier time instance before TOI. For instance, Li and colleagues [2020] used $0.8t_{TOI}$ in their IPC implementation. In this section, we describe a faster root finding algorithm, which does not return an exact TOI but a moment slightly ahead of it.

We consider a triangle-level CCD a cubic root finding problem of:

$$d(t) = 0, \quad \text{for } d = At^3 + Bt^2 + Ct + D. \tag{15}$$

As long as $A \neq 0$, this function has at least one real root. In reality, we only need to know if there exists a real root within the interval of $(t_0, t_1]$, where $t_0$ and $t_1$ denote the starting and ending moments of the current time step. Another piece of useful information is that $At^3 + Bt^2 + Ct + D > 0$ at $t = t_0$, because each time step always starts from an intersection-free state. Being a cubic polynomial, $d(t)$ has two local extrema: one local minimum ($t_{min}$) and one local maximum ($t_{max}$). Those two extrema subdivide the function domain into three monotonic intervals. The basic idea of our algorithm is to first identify which interval TOI may reside in, and an altered Newton-Raphason search i.e., the minimum-gradient Newton is designed to find an approximate root smaller than the actual TOI.

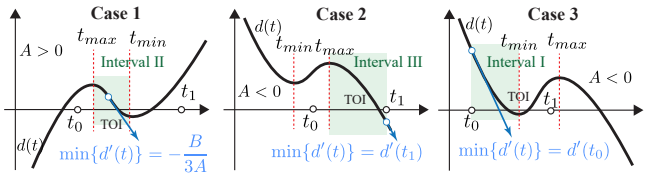


Fig. 6. **Fast root approximate.** We propose a fast root approximate algorithm, based on the minimum-gradient Newton. Our rationale is twofold: 1) the exact TOI is not needed in our framework, and 2) the TOI can only occur in a monotonically decreasing interval (colored in light green in the figure) regardless if $A > 0$ or $A < 0$, since $d(t_0)$ is always positive.

To identify the right monotonic interval, we first compute $t_{min}$ and $t_{max}$, which are simply two roots of the quadratic function $d'(t) = 0$ or $3At^2 + 2Bt + C = 0$. As shown in Fig. 6, we investigate three possible cases:

- **Case 1 $A > 0$** If $A > 0$, we know $t_{max} \leq t_{min}$. In this case, we further compute $d(t_{min})$. If $d(t_{min}) > 0$, we can conclude that there is no collision between $t_0$ and $t_1$. Otherwise, we also need to compare $t_0$ with $t_{min}$. If $t_0 < t_{min}$, TOI is between $[t_{max}, t_{min}]$ (i.e., interval II in the figure). On the other hand, if $t_0 \geq t_{min}$ and as we also know $d(t_0) > 0$, primitives are moving away from each other w.r.t. time, and no collision will be produced.
- **Case 2 $A < 0$ and $d(t_{min}) \geq 0$** $A < 0$ implies $t_{min} \leq t_{max}$. Therefore TOI could land either before $t_{min}$ or after $t_{max}$. If the condition of $d(t_{min}) \geq 0$ holds, TOI can only occur after $t_{max}$. In other words, a valid TOI can only exist in the interval of $(t_{max}, t_0]$, which is interval III in the figure.
- **Case 3 $A < 0$ and $d(t_{min}) < 0$** The last case is when $A < 0$ but $d(t_{min}) < 0$. In this situation, we also need to confirm $t_0$ is before $t_{min}$. TOI is between the interval of $(t_0, t_{min})$ i.e., interval I if and only if $t_0 < t_{min}$. Otherwise, TOI is still in the interval of $(t_{max}, t_0]$ or interval III, if it exists.

Based on the above case studies, it is now clear that the TOI should always sit in one of monotonically decreasing intervals regardless the sign of $A$. This is because $d(t_0) > 0$ is positive, and the curve must travel downwards in order to generate a zero-crossing. This property leaves us only three options i.e., three intervals highlighted in the figure. Pinpointing the TOI interval facilitates our numerical root finding. This procedure begins with the starting $t$ of the TOI interval. Instead of evaluating $d'(t)$ at each iteration, we use a fixed gradient during the search. Doing so certainly saves a lot of computations for $d'(t)$, but how to make sure the resulting root approximate is smaller than the actual root?

In the classic Newton-Raphson method, each search finds the zero-crossing along the tangent line of the function. As long as the magnitude of our search slope is bigger than the current curve slope, our search will always hit $d = 0$ before the Newton search. Since the root search is only performed in three possible (decreasing) intervals, we can just use the a locally minimal gradient within the

Table 1. **Time statistics.** This table reports detailed statistics of our experiments. # **Bdy** is the total number of deformable bodies in the example. # **DOF** gives the total number of simulation DOFs. # **Ele.** and # **Tri./Edg.** are the total numbers of elements (tetrahedron/triangle elements), surface triangles and edges. The latter two factors contribute the the complexity of culling and collision processing. # **Con.** and # **Bar.** report the total number of elastic constraints and average number of barrier constraints during the simulation. $\ell$ is the order of A-Jacobi. # **L-G** is the average number of L-G iterations needed for each time step. # **Out.** is the average outer loop count, knowing that each outer loop could need several L-G iterations. # **A-J** reports, on average, the total number of A-Jacobi iterations used for each time step. The next four columns are timing information. All are in **milliseconds**. **A-J** is the total time used for the A-Jacobi solver. **CCD** is the average time needed for collision culling and CCD processing. **Bar.** shows the timing for generating all the barrier constraints, and **Misc.** stands for other computation costs e.g., variables initialization, convergence check etc. **FPS** reports the FPS range during the simulation.

| Test | # Bdy | # DOF | # Ele. | # Tri./Edg. | # Con. | # Bar. | $\ell$ | # L-G | # Out. | # A-J | A-J | CCD | Bar. | Misc. | FPS |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Falling dinosaur (Fig. 10) | 4 | 69K | 103K | 23K/35K | 106K | 6K | 3 | 40 | 8 | 122 | 21.4 | 11.4 | 5.1 | 1.1 | 21.1 / 115.5 |
| Dragon (Fig. 13, top) | 1 | 80K | 100K | 39K/58K | 100K | 8K | 3 | 44 | 14 | 134 | 35.1 | 15.7 | 4.5 | 1.3 | 14.6 / 119.3 |
| Armadillo (Fig. 13, bottom) | 3 | 78K | 77K | 45K/68K | 99K | 11K | 3 | 32 | 10 | 96 | 29.9 | 8.9 | 2.8 | 1.5 | 19.4 / 112.0 |
| Tiered skirt (Fig. 14, bottom) | 1 | 91K | 59K | 84K/127K | 120K | 42K | 3 | 46 | 11 | 228 | 41.4 | 19.3 | 7.7 | 1.3 | 12.2 / 29.1 |
| Rubber helicopters (Fig. 11) | 3 | 150K | 224K | 46K/69K | 224K | 14K | 3 | 45 | 11 | 176 | 38.7 | 19.2 | 7.6 | 1.8 | 11.2 / 87.1 |
| Bone dragon (Fig. 15) | 4 | 109K | 89K | 69K/103K | 92K | 8K | 3 | 63 | 12 | 340 | 48.5 | 15.2 | 4.5 | 1.8 | 12.4/48.4 |
| "Animal crossing" (Fig. 16) | 21 | 218K | 293K | 86K/128K | 293K | 16K | 2 | 50 | 8 | 257 | 39.5 | 19.1 | 3.8 | 1.8 | 13.4/46.3 |
| Halloween party (Fig. 1) | 13 | 249K | 265K | 159K/239K | 287K | 9K | 2 | 93 | 17 | 263 | 85.3 | 32.4 | 9.8 | 2.2 | 7.7/26.8 |

interval in our root search (i.e., a smaller gradient has a bigger slope). Because $d(t)$ is a cubic polynomial, the variation of the gradient is quadratic. If TOI is at interval I of $(t_0, t_{min}]$, the locally minimal gradient is $\min\{d'(t)\} = d'(t_0)$. This is because $d'(t_{min}) = 0$, and the gradient (which is a negative quantity) monotonically increases in this interval. Similarly, if TOI is at interval III of $(t_{max}, t_1]$, the locally minimal gradient is $\min\{d'(t)\} = d'(t_1)$. In this interval, the gradient monotonically decreases from zero (at $t = t_{max}$) all the way to $t = t_1$. Lastly, if the TOI interval is interval II of $[t_{min}, t_{max}]$ the locally minimal gradient is the global minimum of the gradient, which can be directly computed as $-B/3A$ by setting $d''(t) = 3At + B = 0$.

By exploiting the fact that computing a precise TOI is unnecessary, we further economize the CCD processing. We find that our algorithm is up to 60% faster than additive CCD [Li et al. 2021b; Zhang et al. 2006], at a cost of an inexact TOI. Meanwhile, we directly use this approximated root to truncate $\Delta x$ (line 15, Alg. 1).

## 7 EXPERIMENTAL RESULTS

We implemented our algorithm on a desktop PC with an 8-core intel i9 CPU and an nVidia 3090 GPU. Most of our experiments include tens to hundreds thousands DOFs. We found that interesting animations with rich local effects under frequent interactions can be well captured at this scale. *Most of those experiments run in real-time over 24 FPS in most animation frames.* When intense, clustered contacts occur i.e., under sharp external forces or massive bodies collisions, the FPS could drop as more outer loops are needed. Because of the barrier constraint and CCD pruning at each outer loop, the simulation does not generate any interpenetration even for fine, thin and fast-moving shapes. Meanwhile, our simulation is typically three-order faster than regular CPU-based (multi-threaded) IPC simulation. Unlike reduced simulation methods [Lan et al. 2020, 2021], our algorithm runs in fullspace capturing all the dynamical details. We uniformly scale the model to fit into a $1 \times 1 \times 1$ box and set $\hat{d} = 0.005$ in all experiments. That is to say, if the size of the model is about one meter, a barrier constraint will be generated if another object becomes closer than five millimeters. The time step size is 1/100 sec. Simulating one time step could take several outer loops, which is composed of multiple L-G iterations. Each L-G iteration needs to run a few A-Jacobi iterations for the global solve. Detailed statistics of the simulation settings, models, and timings are reported in Tab. 1. Most experiments can also be found in the supplementary video.

### 7.1 Performance of A-Jacobi

First of all, we would like check the performance of the proposed A-Jacobi method, and how it is compared with other commonly used linear solvers. As shown in Fig. 7, we simulate a square table cloth (with 30K DOFs) under the low-frequency gravity force and plot the convergence curves of different solvers. We also compare convergent behaviors among solvers when a sharp local external force is applied. In this comparison, we pick a representative frame during the simulation e.g., when the external force is applied. We also solve the global system using a direct LU solver, which is regarded as $x^\star$ for the relative error measure. The solvers tested in this experiment include regular Jacobi, Gauss-Seidel (GS), GS with SOR (successive over-relaxation), and preconditioned conjugate gradient (PCG). We use the Jacobi preconditioner similar to [Wang and Yang 2016] for the PCG solver.

The difference between vanilla Jacobi and A-Jacobi is easy to follow. Each A-Jacobi iteration is equivalent to running multiple Jacobi iterations. Therefore, it is not a surprise to see A-Jacobi reduces the error faster. GS is another popular linear solver, and it is also based on matrix splitting. GS has been used with PD [Fratarcangeli et al.

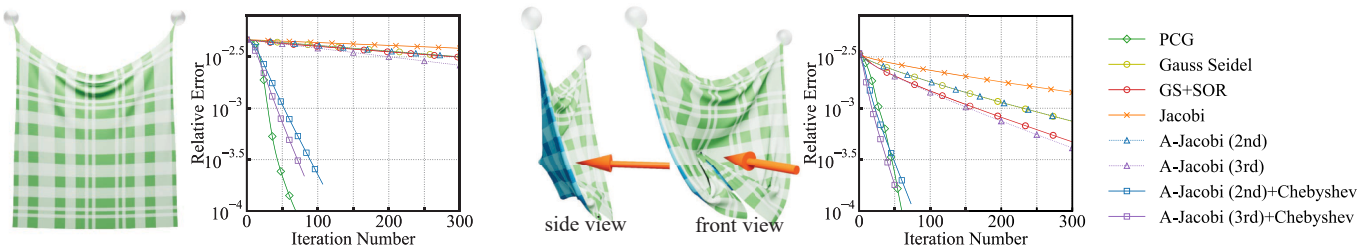


Fig. 7. **Convergence plots.** We compare convergent behaviors of A-Jacobi and other commonly-used iterative linear solvers. We plot the relative error of solving the global step system (Eq. (3)) using Jacobi, Gauss Seidel, Gauss Seidel and SOR, PCG, as well as second- and third-order A-Jacobi. With our GPU implementation, one A-Jacobi iteration is nearly as efficient as one regular Jacobi iteration, making A-Jacobi the fastest converging solver in the benchmark.

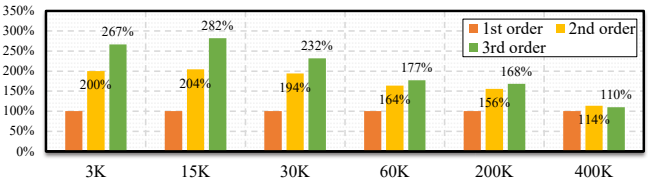


Fig. 8. **A-Jacobi performance.** This figure visualizes the performance of A-Jacobi under different problem sizes on a 3090 GPU. The red bars represent the first-order A-Jacobi, which is equivalent to the regular Jacobi method. Therefore it always has the same performance (i.e., 100%). Second- and third-order A-Jacobis are more efficient in general. For instance, for a 15K-DOF simulation system, third-order A-Jacobi is nearly three times faster (2.82×). When the problem size further increases, the advantage of A-Jacobi diminishes because unfolding the computation at each vertex reaches the capacity of the GPU.

2016] on the GPU. Conceptually, we can also build an aggregated version of GS method. Unlike Jacobi, parallel GS requires on-the-fly vertex coloring under different collision events. This step makes the assembly of $\tilde{\mathbf{R}}$ less practical in reality. PCG is arguably one of the strongest iterative linear solvers. A PCG iteration always returns the optimal search direction and step size within the current Krylov subspace. Therefore Chebyshev is not compatible with PCG. There exist GPU-based PCG implementations [Helfenstein and Koko 2012]. However, in order to obtain the optimal search direction and search step size, inner product computations are frequently needed in PCG, which stand as a major bottleneck. Hence the computation time of each PCG iteration is much slower (by one or two orders) than Jacobi and A-Jacobi on the GPU. With Chebyshev acceleration, the convergent rate of A-Jacobi is similar to PCG, but the parallelization of A-Jacobi can be conveniently tuned to achieve the optimal balance between the thread concurrency and cost.

What is the best order of A-Jacobi we should use? The answer depends on the DOF count of the simulation and the GPU specifications. To better explore this question, we visualize the "performance" of A-Jacobi in Fig. 8, for simulation instances of different sizes. The so-called performance here refers to how fast a solver can expand the series of Eq. (10), compared with the vanilla Jacobi iteration. Clearly, the first-order A-Jacobi always has the same performance (100%) as the regular Jacobi method. Second- and third-order A-Jacobi can better exploit the hardware resource and exhibit stronger efficiency in general. For reasonable-scale problems, we often observe multifold performance gains using A-Jacobi. When the problem size further goes up, the advantage of A-Jacobi becomes less obvious as a regular Jacobi iteration already consumes significant hardware resource. The complexity growth however, is largely linear, and using multiple GPUs can easily boost the simulation performance further. Nevertheless, this is not the focus of this paper, and we will investigate more along this direction as our future work.

## 7.2 Patch-based GPU Collision Culling

Another important step along the pipeline is collision culling. When aiming for interactive or real-time simulations, culling becomes a "double-edged blade" as we would like to avoid unnecessary CCD computations as much as possible while still maintaining a low cost for the entire culling procedure. In other words, we seek for a good trade-off between culling effectiveness and efficiency on the GPU. Due to restrained time budgets, we do not use primitive-level BVHs which allow one to identify all the colliding primitives at the leaf level. Our method is inspired by I-Cloth [Tang et al. 2018] but is more generic and can be used for any deformable shapes.

Specifically, we first subdivide the surface geometry of the model into *patches*. This is a well-studied problem in computational geometry, and many excellent algorithms are available (e.g., variational surface approximation [Wu and Kobbelt 2005]). Here we only need a speedy and robust patch generation and do not concern too much about the subdivision quality. To this end, we simply obtain the patch partition based on a regular voxelization and group all the surface triangles within a voxel cell as a patch (as shown in Fig. 9). Note that we do not require triangles of a patch being connected topologically. After that, we build a BVH (of hierarchical AABBs) based on the voxelized model such that each of its leaf node houses a surface patch. From this perspective, our culling strategy can be considered as the combination of BVH and voxel-based spatial hashing.

In our implementation, the BVH is a binary tree so that it can be compactly encoded as a linear array, and the pointer references of parent/children are also straightforward. The structure of the BVH does not change during the simulation, only the AABBs at each BVH level will be updated. We also pre-build a list of all the primitive pairs of each patch for detecting potential within-patch contacts and self-collisions. At the simulation run time, patch-based culling eventually leads to a list of overlapping patches. We then

Fig. 9. **Patch-based collision culling.** Our implementation includes a simple and effective culling modality. The input model is subdivided into patches based on a regular voxelization. Patch BVH facilitates a broad phase culling, which generates a list of overlapping patches. Because intra-patch primitive pairs are pre-built, primitive-level CCD is then followed to compute $t_I$ for all the inter- and intra-patch primitive pairs.

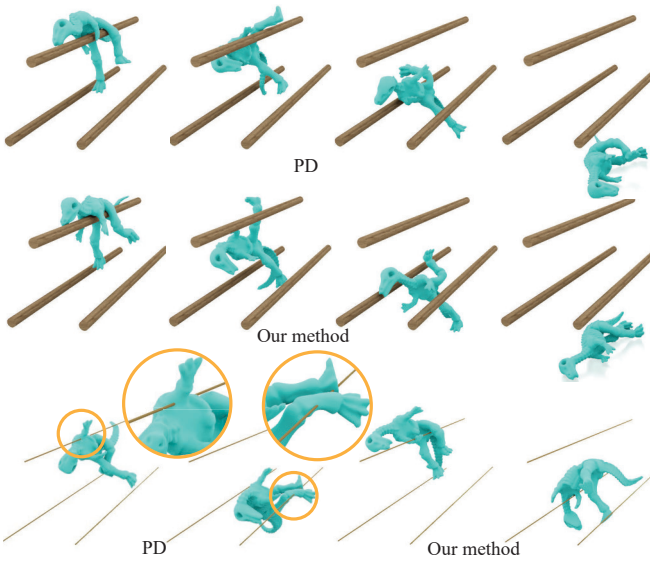


Fig. 10. **Falling dinosaur.** Our method is PD-based. Therefore the simulation results of the falling dinosaur using both algorithms are similar, if all the collisions are well resolved. This can be verified from the snapshots in the top two rows. The biggest advantage of our method is the guaranteed collision resolution based on barrier constraints. After we reduce the diameter of the rod by 90%, PD fails to detect the pass-through of the rod during dinosaur's falling, while our method yields realistic collision responses.

*exhaustively* compute CCD for all primitive parings from two overlapping patches. Together with the within-patch primitive pair list, we generate the final list of barrier constraints.

### 7.3 Comparison with Fullspace PD and IPC

Since our method combines the most favorable features from both PD and IPC, it is of great interest to examine how is our method compared with each of those two competitors. We first show a side by side comparison between our method and PD for simulating a falling dinosaur with 103K elements. The snapshots of the resulting animation are given in the top two rows of Fig. 10. Basically, the visual difference between our method and PD is hardly perceivable as long as PD successfully resolves the collisions between the dinosaur and wooden rods. After we shrink the cross-section of the rods by 90%, interpenetration between the thin rod and the dinosaur is more visible because of the failure of discrete collision detection. Our method uses CCD for barrier-based collision processing, and it is therefore reliable and robust even for thin geometries. If we give up CCD and barrier constraint, A-Jacobi solver is able to push the simulation performance to exceed 200 FPS for this example.

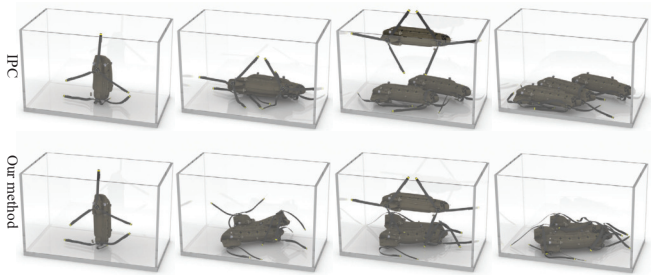


Fig. 11. **Rubber helicopters.** We simulate three rubber helicopters colliding in a glass tank. This is a fullspace simulation and involves over 150K DOFs and 224K elements. Both IPC and our method produce plausible animations with the guarantee of interpenetration-free. Being a GPU algorithm, our method is over 2,000× faster than IPC.

We also compare our method with fullspace IPC [Li et al. 2020]. Such comparison is not strictly "apple-to-apple" since our method is PD-based and less capable of capturing an accurate hyperelastic behavior. To minimize this discrepancy, we implement an as-rigid-as-possible (ARAP) material [Igarashi et al. 2005] with IPC instead of using the invertible neo-Hookean model [Smith et al. 2018]. The IPC solver runs on an 8-core i9 CPU. We used intel MKL library [Wang et al. 2014] and enabled multi-threading whenever possible (e.g., in CCD, culling and system solve). For the same falling dinosaur experiment of Fig. 10, IPC also produces a high-quality intersection-free animation regardless of the size of the rod. The difference between IPC and our method is indiscernible. On average, IPC takes 1.85 min to simulate one frame. Being a GPU simulator, our method is over 2,600× faster than IPC. Fig. 11 shows another comparison using our method and fullspace IPC. In this experiment, three rubber helicopters of 224K elements fall into a glass tank and collide with each other tightly. The concave geometry of the helicopter also yields a lot of self-collisions. Both IPC and our method robustly resolve all the collisions using the barrier-based penalty, but our method is 2,000× faster (2.23 min per frame with IPC; 65.4 ms per

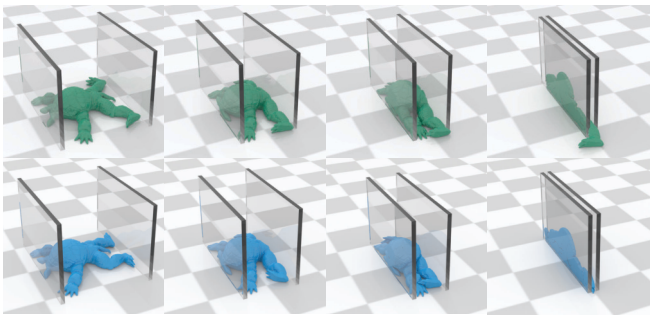

Fig. 12. **Flatten Armadillo.** Our method remains stable under extreme-scale deformation with massive close contacts. We note that the simulation results of IPC (top) and our method (bottom) are similar, and both do not have any interpenetration. In this example, our method is 67× faster on average. The total number of outer loops goes up as the boards close: from 8 outer iterations (leftmost) to 54 outer iterations (rightmost). The FPS therefore, drops from 67.9 to 1.3.

frame with our method). If we choose to run our solver on the CPU with multithreading, the average computation time for one time step is 3.2 sec, still much faster than IPC. Again, we would like to mention that this speedup benchmark may be misleading and imprecise, since two simulations run on different hardwares and with different algorithms. Nevertheless, this experiment should more or less showcase the performance gap between CPU and GPU simulations, as long as the GPU can be reasonably leveraged.

### 7.4 Robustness under Close Contacts

Under close contacts, many vertices are engaged with each other leading to a stiffer system. Here we evaluate the robustness of our method by flatting the Armadillo with two glassy boards. We compare our method and IPC with the ARAP energy. The results are shown in Fig. 12. Both our method and IPC generate plausible results, which are visually similar to each other. With highly intense self-collisions, the speedup of our method is smaller than other experiments e.g., Fig. 11. This is because the convergence of PD-like solver is faster, including our method, slows down when the optimization approaches to smaller residual errors. In this example, the full-space IPC will need about 21 sec on average to simulate one step, while our method uses about 312 ms – still 67× faster though.

### 7.5 Interpenetration-free Simulation in Real-time

The superior performance of A-Jacobi allows us to simulate deformable objects of complicated geometries in real time under interactive user manipulations. Here we show three of such examples of a dragon (Fig. 13 top), an Armadillo (Fig. 13 bottom), and a multi-layer skirt in Fig. 14. The dragon model consists of 100K elements. We interactively use external forces to drag it back and forth, hitting the rods from various angles. In the Armadillo example, two pieces of cloth fall to the Armadillo. There are in total 77K elements in the simulation. The user drags the cloth and the Armadillo to trigger intense inter-body collisions and self-collisions. The example of Fig. 14 is a tiered skirt of multiple layers. This simulation

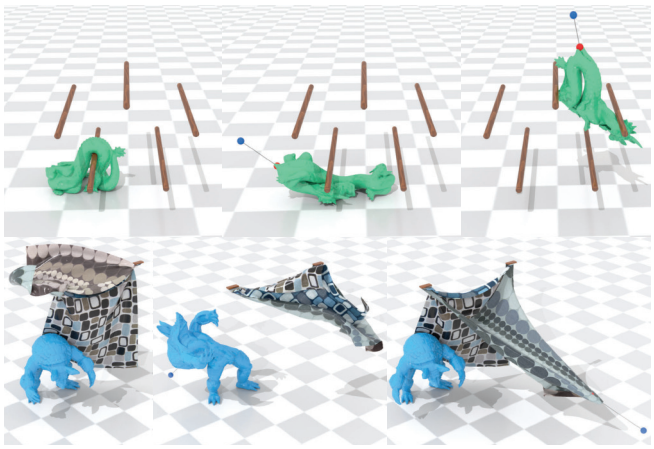

Fig. 13. **Interpenetration-free simulation in real time.** Our framework delivers a real-time frame rate while simulating complex models. Here we show two deformable animations that are generated by interactive user manipulations. Thanks to the barrier-based collision processing, the simulation is always free of interpenetration under any user inputs.

involves over 91K DOFs and exhibits rich collision patterns at different layers under external forces and user inputs. Our method is able to robustly process all of those examples in real time. It is noteworthy that a recent cloth simulation algorithm proposed by Wu and colleagues [2020] is also GPU-based. It has a higher run-time FPS than our algorithm "on the surface" for Fig. 14. Such excellent performance relies on a dedicated collision detection and culling method specifically crafted for less-extensible cloth. Our algorithm targets on general deformable simulation problems and thus cannot enjoy such cloth-only optimizations and speedups. We believe, by combing A-Jacobi and the repulsion method from [Wu et al. 2020], the state-of-the-art cloth simulation could be further improved.

### 7.6 More examples

Lastly, we report three more comprehensive examples involving both shell-like shapes and elastic solids. Fig. 15 demonstrates a simulation example of highly concave models i.e., the bone dragon. In this case, three bone dragons drop to a table cloth (89K elements in total), whose four corners are fixed. The impacts of the dragons generate multiple localized dents on the cloth. The skeletons of dragon wings entangle with the cloth leading to wrinkles at edges. Those effects are accurately captured by our simulation with a run-time FPS between 12.4 to 48.4. In the second example (Fig. 16), we throw a batch of 11 cartoon animal characters into a container with many rubber straps in the space. There are in total 293K elements in this example. The interactions among animals and rubber straps yield interesting animation effects, and our simulation FPS ranges from 13.4 to 46.3. The third example involves multiple objects as shown in Fig. 1. Several ghost-like creatures are hanging on a spooky tree. The "ghost" is modeled as a piece of cloth embedded with additional volume preserving constraints at the head. The tree sways under the wind. After wind stops, several monster pumpkins fall on the tree and bounce on the grass. This simulation consists of over 265K elements, and the run-time FPS varies between 7.7 to 26.8.

Fig. 14. **Tiered skirt.** Our method also produces high-quality interpenetration-free cloth animations. The tiered skirt has over 91K simulation DOFs and multiple layers. Our solver robustly resolves all the collisions and self-collisions of the skirt. The simulation FPS is between 12.2 to 29.1.

Fig. 15. **Bone dragons on cloth.** The bone dragon has a sharply concave shape. We drop three of them into a piece of table cloth. The four corners of the cloth is fixed. After hitting on the cloth, bone dragons slide down on the cloth and fall on the floor. This simulation is beyond the capability of conventional PD or PBD based frameworks as CCD is a must to ensure the models are free of interpenetration. The total number of DOFs of the simulation is over 108K. Our algorithm robustly simulates this scene and the runtime FPS ranges between 12.4 to 48.4.

Fig. 16. **"Animal crossing".** A pack of soft animal characters fall and hit several rubber straps. There are 21 deformable objects in this example consisting of 293K elements. The falling animals induce interesting deformation effects: a couple of straps get entangled with the animals and are pulled to the floor. The runtime simulation FPS is between 13.4 to 46.3.

## 8 CONCLUSION

In this paper, we present a GPU algorithm for an efficient simulation of elastic objects. Our method demonstrates the feasibility of using GPU solvers to tackle barrier-augmented simulations, which are currently coped with Newton's method exclusively due to the high nonlinearity of the barrier constraints. We have made several non-trivial revisions over the vanilla PD framework to handle sticking artifacts and slow convergence. Our simulation is empowered by a new GPU solver named A-Jacobi, which further enhances the performance of GPU linear solve. Together with a faster CCD processing, we manage to simulate complicated scenes on the GPU at an interactive rate. Compared with the CPU-based IPC system, our method is three orders faster without any DOF reductions. The simulation is free of expensive precomputation/prepossessing and gives all details of local deformations while retaining the non-penetration guarantee. Such a combination of convenience, speed, robustness, and quality is not common in existing methods.

Our method also has some limitations, for which we will carefully investigate in the near future. First of all, it is known that PD-based models are not fully physically accurate. This issue however should be resolved by porting our algorithm to other nonlinear programmings like ADMM [Narain et al. 2016], which could be further paired with Anderson acceleration [Zhang et al. 2019]. Alternatively, we can also follow a recent contribution from Macklin and Müller [2021] that directly formulates hyperelastic material

models with position-based frameworks. Being a position-based simulator, our method is sensitive to the time step size. Increasing $h$ slows the convergence as PD only converges faster than Newton's method at first few iterations. This limitation however, could be addressed by using sub-step [Macklin et al. 2019]. While our method has an encouraging performance in general, the run-time FPS is impacted by the frequency of collision events. This is understandable since collisions are treated as highly nonlinear barrier constraints in our framework. However, it should be possible to further flatten the FPS fluctuation by designing collision-aware preconditioners or multi-level solvers. The mechanism of A-Jacobi naturally fits multi-GPU systems to further push the simulation performance for higher-resolution models. Exploiting deep learning based method such as in [Luo et al. 2018; Shen et al. 2021] is also an interesting and promising future direction for us.

## ACKNOWLEDGMENTS

We thank anonymous reviewers for their constructive comments. Yin Yang, Lei Lan, and Guanqun Ma are partially supported by National Science Foundation (No. 2011471, 2016414). Chenfanfu Jiang and Minchen Li are are partially supported by National Science Foundation (No. 2153851, 2153863, 2023780) and Department of Energy (No. ORNL 4000171342).

## REFERENCES

Farid Alizadeh, Jean-Pierre A Haeberly, and Michael L Overton. 1997. Complementarity and nondegeneracy in semidefinite programming. *Mathematical programming* 77, 1 (1997), 111–128.

Owe Axelsson. 1977. Solution of linear systems of equations: iterative methods. In *Sparse matrix techniques*. Springer, 1–51.

Jernej Barbič and Yili Zhao. 2011. Real-time large-deformation substructuring. *ACM transactions on graphics (TOG)* 30, 4 (2011), 1–8.

Jernej Barbič and Doug L James. 2005. Real-time subspace integration for St. Venant-Kirchhoff deformable models. In *ACM Trans. Graph. (TOG)*, Vol. 24. ACM, 982–990.

Sofien Bouaziz, Sebastian Martin, Tiantian Liu, Ladislav Kavan, and Mark Pauly. 2014. Projective dynamics: Fusing constraint projections for fast simulation. *ACM transactions on graphics (TOG)* 33, 4 (2014), 1–11.

Robert Bridson, Ronald Fedkiw, and John Anderson. 2002. Robust treatment of collisions, contact and friction for cloth animation. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. 594–603.

Tyson Brochu, Essex Edwards, and Robert Bridson. 2012. Efficient geometrically exact continuous collision detection. *ACM Transactions on Graphics (TOG)* 31, 4 (2012), 1–7.

Steve Capell, Seth Green, Brian Curless, Tom Duchamp, and Zoran Popović. 2002a. Interactive skeleton-driven dynamic deformations. In *ACM Trans. Graph. (TOG)*, Vol. 21. ACM, 586–593.

Steve Capell, Seth Green, Brian Curless, Tom Duchamp, and Zoran Popović. 2002b. A multiresolution framework for dynamic deformations. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*. 41–47.

Min Gyu Choi and Hyeong-Seok Ko. 2005. Modal warping: Real-time simulation of large rotational deformation and manipulation. *IEEE Trans. on Visualization and Computer Graphics* 11, 1 (2005), 91–101.

Jinhyun Choo, Yidong Zhao, Yupeng Jiang, Minchen Li, Chenfanfu Jiang, and Kenichi Soga. 2021. A barrier method for frictional contact on embedded interfaces. arXiv:2107.05814 [math.NA]

Sung-Jin Chung. 1989. NP-completeness of the linear complementarity problem. *Journal of optimization theory and applications* 60, 3 (1989), 393–399.

Richard W Cottle, Jong-Shi Pang, and Richard E Stone. 2009. *The linear complementarity problem*. SIAM.

Gilles Daviet, Florence Bertails-Descoubes, and Laurence Boissieux. 2011. A hybrid iterative solver for robustly capturing coulomb friction in hair dynamics. In *Proceedings of the 2011 SIGGRAPH Asia Conference*. 1–12.

Yu Fang, Minchen Li, Chenfanfu Jiang, and Danny M Kaufman. 2021. Guaranteed globally injective 3D deformation processing. *ACM Trans. Graph.(TOG)* 40, 4 (2021).

Charbel Farhat, Michael Lesoinne, and Kendall Pierson. 2000. A scalable dual-primal domain decomposition method. *Numerical linear algebra with applications* 7, 7-8 (2000), 687–714.

François Faure, Benjamin Gilles, Guillaume Bousquet, and Dinesh K Pai. 2011. Sparse meshless models of complex deformable solids. In *ACM Trans. Graph. (TOG)*, Vol. 30. ACM, 73.

Zachary Ferguson, Minchen Li, Teseo Schneider, Francisca Gil-Ureta, Timothy Langlois, Chenfanfu Jiang, Denis Zorin, Danny M Kaufman, and Daniele Panozzo. 2021. Intersection-free rigid body dynamics. *ACM Transactions on Graphics* 40, 4 (2021), 183.

Marco Fratarcangeli, Valentina Tibaldo, and Fabio Pellacini. 2016. Vivace: A practical gauss-seidel method for stable soft body dynamics. *ACM Transactions on Graphics (TOG)* 35, 6 (2016), 1–9.

Marco Fratarcangeli, Huamin Wang, and Yin Yang. 2018. Parallel iterative solvers for real-time elastic deformations. In *SIGGRAPH Asia 2018 Courses*. 1–45.

Ming Gao, Andre Pradhana Tampubolon, Chenfanfu Jiang, and Eftychios Sifakis. 2017. An adaptive generalized interpolation material point method for simulating elastoplastic materials. *ACM Trans. Graph. (TOG)* 36, 6 (2017), 223.

Benjamin Gilles, Guillaume Bousquet, Francois Faure, and Dinesh K Pai. 2011. Frame-based elastic models. *ACM Trans. Graph. (TOG)* 30, 2 (2011), 15.

Gene H Golub and Richard S Varga. 1961. Chebyshev semi-iterative methods, successive overrelaxation iterative methods, and second order Richardson iterative methods. *Numer. Math.* 3, 1 (1961), 157–168.

Eitan Grinspun, Petr Krysl, and Peter Schröder. 2002. CHARMS: A simple framework for adaptive simulation. *ACM transactions on graphics (TOG)* 21, 3 (2002), 281–290.

William W Hager. 1989. Updating the inverse of a matrix. *SIAM review* 31, 2 (1989), 221–239.

David Harmon, Etienne Vouga, Breannan Smith, Rasmus Tamstorf, and Eitan Grinspun. 2009. Asynchronous contact mechanics. In *ACM SIGGRAPH 2009 papers*. 1–12.

David Harmon, Etienne Vouga, Rasmus Tamstorf, and Eitan Grinspun. 2008. Robust treatment of simultaneous collisions. In *ACM SIGGRAPH 2008 papers*. 1–4.

Kris K Hauser, Chen Shen, and James F O'Brien. 2003. Interactive Deformation Using Modal Analysis with Constraints.. In *Graphics Interface*, Vol. 3. 16–17.

Florian Hecht, Yeon Jin Lee, Jonathan R Shewchuk, and James F O'Brien. 2012. Updated sparse cholesky factors for corotational elastodynamics. *ACM Trans. Graph. (TOG)* 31, 5 (2012), 123.

Rudi Helfenstein and Jonas Koko. 2012. Parallel preconditioned conjugate gradient algorithm on GPU. *J. Comput. Appl. Math.* 236, 15 (2012), 3584–3590.

Nicholas J Higham. 2009. Cholesky factorization. *Wiley Interdisciplinary Reviews: Computational Statistics* 1, 2 (2009), 251–254.

Takeo Igarashi, Tomer Moscovich, and John F Hughes. 2005. As-rigid-as-possible shape manipulation. *ACM transactions on Graphics (TOG)* 24, 3 (2005), 1134–1141.

Geoffrey Irving, Joseph Teran, and Ronald Fedkiw. 2004. Invertible finite elements for robust simulation of large deformation. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*. 131–140.

Couro Kane, Jerrold E Marsden, Michael Ortiz, and Matthew West. 2000. Variational integrators and the Newmark algorithm for conservative and dissipative mechanical systems. *International Journal for numerical methods in engineering* 49, 10 (2000), 1295–1325.

Carl T Kelley. 1995. *Iterative methods for linear and nonlinear equations*. SIAM.

Theodore Kim and Doug L James. 2009. Skipping steps in deformable simulation with online model reduction. In *ACM Trans. Graph. (TOG)*, Vol. 28. ACM, 123.

Theodore Kim and Doug L James. 2012. Physics-based character skinning using multidomain subspace deformations. *IEEE transactions on visualization and computer graphics* 18, 8 (2012), 1228–1240.

Martin Komaritzan and Mario Botsch. 2018. Projective skinning. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1, 1 (2018), 1–19.

Martin Komaritzan and Mario Botsch. 2019. Fast projective skinning. In *Motion, Interaction and Games*. 1–10.

Lei Lan, Ran Luo, Marco Fratarcangeli, Weiwei Xu, Huamin Wang, Xiaohu Guo, Junfeng Yao, and Yin Yang. 2020. Medial Elastics: Efficient and Collision-Ready Deformation via Medial Axis Transform. *ACM Transactions on Graphics (TOG)* 39, 3 (2020), 1–17.

Lei Lan, Yin Yang, Danny Kaufman, Junfeng Yao, Minchen Li, and Chenfanfu Jiang. 2021. Medial IPC: accelerated incremental potential contact with medial elastics. *ACM Transactions on Graphics (TOG)* 40, 4 (2021), 1–16.

Minchen Li, Zachary Ferguson, Teseo Schneider, Timothy Langlois, Denis Zorin, Daniele Panozzo, Chenfanfu Jiang, and Danny M Kaufman. 2020. Incremental potential contact: Intersection- and inversion-free, large-deformation dynamics. *ACM transactions on graphics* 39, 4 (2020).

Minchen Li, Danny M. Kaufman, and Chenfanfu Jiang. 2021b. Codimensional Incremental Potential Contact. *ACM Trans. Graph. (SIGGRAPH)* 40, 4, Article 170 (2021).

Xuan Li, Yu Fang, Minchen Li, and Chenfanfu Jiang. 2021a. BFEMP: Interpenetration-free MPM–FEM coupling with barrier contact. *Computer Methods in Applied Mechanics and Engineering* (2021), 114350.

Tiantian Liu, Adam W. Bargteil, James F. O'Brien, and Ladislav Kavan. 2013. Fast Simulation of Mass-spring Systems. *ACM Trans. Graph. (TOG)* 32, 6 (2013), 214:1–214:7.

Tiantian Liu, Sofien Bouaziz, and Ladislav Kavan. 2017. Quasi-newton methods for real-time simulation of hyperelastic materials. *Acm Transactions on Graphics (TOG)* 36, 3 (2017), 1–16.

Ran Luo, Tianjia Shao, Huamin Wang, Weiwei Xu, Xiang Chen, Kun Zhou, and Yin Yang. 2018. NNWarp: Neural network-based nonlinear deformation. *IEEE transactions on visualization and computer graphics* 26, 4 (2018), 1745–1759.

Miles Macklin and Matthias Muller. 2021. A Constraint-based Formulation of Stable Neo-Hookean Materials. In *Motion, Interaction and Games*. 1–7.

Miles Macklin, Matthias Müller, and Nuttapong Chentanez. 2016. XPBD: position-based simulation of compliant constrained dynamics. In *Proceedings of the 9th International Conference on Motion in Games*. 49–54.

Miles Macklin, Kier Storey, Michelle Lu, Pierre Terdiman, Nuttapong Chentanez, Stefan Jeschke, and Matthias Müller. 2019. Small Steps in Physics Simulation. In *Proceedings of the 18th Annual ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Los Angeles, California) *(SCA '19)*. Association for Computing Machinery, New York, NY, USA, Article 2, 7 pages. https://doi.org/10.1145/3309486.3340247

Sebastian Martin, Peter Kaufmann, Mario Botsch, Eitan Grinspun, and Markus Gross. 2010. Unified simulation of elastic rods, shells, and solids. In *ACM Trans. Graph. (TOG)*, Vol. 29. ACM, 39.

Sebastian Martin, Bernhard Thomaszewski, Eitan Grinspun, and Markus Gross. 2011. Example-based elastic materials. In *ACM SIGGRAPH 2011 papers*. 1–8.

Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. 2007. Position based dynamics. *Journal of Visual Communication and Image Representation* 18, 2 (2007), 109–118.

Matthias Müller, Bruno Heidelberger, Matthias Teschner, and Markus Gross. 2005. Meshless deformations based on shape matching. In *ACM Trans. Graph. (TOG)*, Vol. 24. ACM, 471–478.

Rahul Narain, Matthew Overby, and George E Brown. 2016. ADMM ⊇ projective dynamics: fast simulation of general constitutive models.. In *Symposium on Computer Animation*, Vol. 1. 2016.

Jorge Nocedal and Stephen Wright. 2006. *Numerical optimization*. Springer Science & Business Media.

Raymond W Ogden. 1997. *Non-linear elastic deformations*. Courier Corporation.

Miguel A Otaduy, Rasmus Tamstorf, Denis Steinemann, and Markus Gross. 2009. Implicit contact handling for deformable objects. In *Computer Graphics Forum*, Vol. 28. Wiley Online Library, 559–568.

Matthew Overby, George E Brown, Jie Li, and Rahul Narain. 2017. ADMM ⊇ projective dynamics: Fast simulation of hyperelastic models with dynamic constraints. *IEEE Transactions on Visualization and Computer Graphics* 23, 10 (2017), 2222–2234.

Albert Peiret, Sheldon Andrews, József Kövecses, Paul G Kry, and Marek Teichmann. 2019. Schur complement-based substructuring of stiff multibody systems with contact. *ACM Transactions on Graphics (TOG)* 38, 5 (2019), 1–17.

Alex Pentland and John Williams. 1989. Good vibrations: Modal dynamics for graphics and animation. In *SIGGRAPH Comput. Graph.*, Vol. 23. ACM.

Xavier Provot. 1997. Collision and self-collision handling in cloth model dedicated to design garments. In *Computer Animation and Simulation'97*. Springer, 177–189.

Jason Sanders and Edward Kandrot. 2010. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.

Siyuan Shen, Yang Yin, Tianjia Shao, He Wang, Chenfanfu Jiang, Lei Lan, and Kun Zhou. 2021. High-order Differentiable Autoencoder for Nonlinear Model Reduction. *arXiv preprint arXiv:2102.11026* (2021).

Breannan Smith, Fernando De Goes, and Theodore Kim. 2018. Stable neo-hookean flesh simulation. *ACM Transactions on Graphics (TOG)* 37, 2 (2018), 1–15.

Rasmus Tamstorf, Toby Jones, and Stephen F McCormick. 2015. Smoothed aggregation multigrid for cloth simulation. *ACM Trans. Graph. (TOG)* 34, 6 (2015), 245.

Min Tang, Tongtong Wang, Zhongyuan Liu, Ruofeng Tong, and Dinesh Manocha. 2018. I-Cloth: Incremental collision handling for GPU-based interactive cloth simulation. *ACM Transactions on Graphics (TOG)* 37, 6 (2018), 1–10.

Yun Teng, Mark Meyer, Tony DeRose, and Theodore Kim. 2015. Subspace condensation: Full space adaptivity for subspace deformations. *ACM Transactions on Graphics (TOG)* 34, 4 (2015), 1–9.

Demetri Terzopoulos and Kurt Fleischer. 1988. Deformable models. *The visual computer* 4, 6 (1988), 306–331.

Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer. 1987. Elastically deformable models, In Proceedings of the 14th annual conference on Computer graphics and interactive techniques. *ACM Siggraph Computer Graphics* 21, 4, 205–214.

Demetri Terzopoulos, Andrew Witkin, and Michael Kass. 1988. Constraints on deformable models: Recovering 3D shape and nonrigid motion. *Artificial intelligence* 36, 1 (1988), 91–123.

Matthias Teschner, Stefan Kimmerle, Bruno Heidelberger, Gabriel Zachmann, Laks Raghupathi, Arnulph Fuhrmann, M-P Cani, François Faure, Nadia Magnenat-Thalmann, Wolfgang Strasser, et al. 2005. Collision detection for deformable objects. In *Computer graphics forum*, Vol. 24. Wiley Online Library, 61–81.

Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 167–188.

Huamin Wang. 2015. A chebyshev semi-iterative approach for accelerating projective and position-based dynamics. *ACM Transactions on Graphics (TOG)* 34, 6 (2015), 1–9.

Huamin Wang and Yin Yang. 2016. Descent methods for elastic body simulation on the GPU. *ACM Trans. Graph. (TOG)* 35, 6 (2016), 212.

Qisi Wang, Yutian Tao, Eric Brandt, Court Cutting, and Eftychios Sifakis. 2021. Optimized processing of localized collisions in projective dynamics. In *Computer Graphics Forum*, Vol. 40. Wiley Online Library, 382–393.

Xinlei Wang, Minchen Li, Yu Fang, Xinxin Zhang, Ming Gao, Min Tang, Danny M Kaufman, and Chenfanfu Jiang. 2020. Hierarchical optimization time integration for cfl-rate mpm stepping. *ACM Transactions on Graphics (TOG)* 39, 3 (2020), 1–16.

Max Wardetzky, Miklós Bergou, David Harmon, Denis Zorin, and Eitan Grinspun. 2007. Discrete quadratic curvature energies. *Computer Aided Geometric Design* 24, 8-9 (2007), 499–518.

Jianhua Wu and Leif Kobbelt. 2005. Structure Recovery via Hybrid Variational Surface Approximation.. In *Comput. Graph. Forum*, Vol. 24. 277–284.

Longhua Wu, Botao Wu, Yin Yang, and Huamin Wang. 2020. A Safe and Fast Repulsion Method for GPU-based Cloth Self Collisions. *ACM Transactions on Graphics (TOG)* 40, 1 (2020), 1–18.

Xiaofeng Wu, Rajaditya Mukherjee, and Huamin Wang. 2015. A unified approach for subspace simulation of deformable bodies in multiple domains. *ACM Transactions on Graphics (TOG)* 34, 6 (2015), 1–9.

Zangyueyang Xian, Xin Tong, and Tiantian Liu. 2019. A scalable galerkin multigrid method for real-time simulation of deformable objects. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–13.

Yin Yang, Dingzeyu Li, Weiwei Xu, Yuan Tian, and Changxi Zheng. 2015. Expediting precomputation for reduced deformable simulation. *ACM Trans. Graph. (TOG)* 34, 6 (2015).

Yin Yang, Weiwei Xu, Xiaohu Guo, Kun Zhou, and Baining Guo. 2013. Boundary-aware multidomain subspace deformation. *IEEE transactions on visualization and computer graphics* 19, 10 (2013), 1633–1645.

Juyong Zhang, Yue Peng, Wenqing Ouyang, and Bailin Deng. 2019. Accelerating ADMM for efficient simulation and optimization. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–21.

Xinyu Zhang, Minkyoung Lee, and Young J Kim. 2006. Interactive continuous collision detection for non-convex polyhedra. *The Visual Computer* 22, 9 (2006), 749–760.

Yongning Zhu, Eftychios Sifakis, Joseph Teran, and Achi Brandt. 2010. An efficient multigrid method for the simulation of high-resolution elastic solids. *ACM Trans. Graph. (TOG)* 29, 2 (2010), 16.

Olgierd Cecil Zienkiewicz, Robert Leroy Taylor, Perumal Nithiarasu, and JZ Zhu. 1977. *The finite element method*. Vol. 3. McGraw-hill London.

## A GPU A-JACOBI IMPLEMENTATION

As mentioned, A-Jacobi exploits modern GPU hardware to hide redundant computation from the end user. As a result, a careful implementation of this solver is critical to obtain a performance gain. Here, we discuss several noteworthy implementation details.

- **Operator assembly** Because B is constant, we always precompute A-products before the simulation. Knowing $\widetilde{\mathsf{R}}$ is a local operator, the possible combinations of $B_{ij}$ in A-products are never exhaustive but only involve a subset of edges within the neighborhood, up to the order of A-Jacobi (see Fig. 17). More precisely, each valid combination of an $\ell$-order A-product echoes an $\ell$-hop path from the vertex. As soon as the barrier constraints are updated, D becomes known, and we precompute all the A-coefficients, which are shared by A-Jacobi iterations. The second precomputation happens at the simulation run time and is carried out on the GPU in parallel.
- **Sparsity suppression** B includes several deformation measures, and the magnitudes/scales of them could differ significantly from each other. Given a piece of less extensible cloth, its stretching stiffness (i.e., $\omega_i$ coefficient in Eq. (3)) is several orders stronger than the bending stiffness. Different measures also have different sparsity patterns. For instance, calculating the stretching at a vertex only needs to visit its incident vertices and edges, while bending needs the information of all the vertices

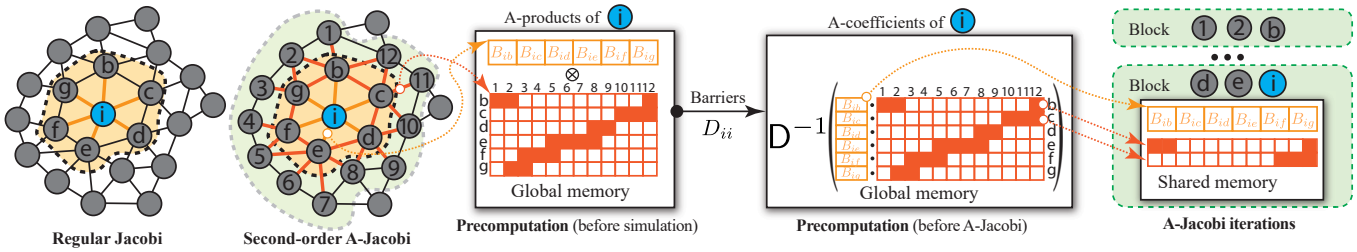


Fig. 17. **GPU implementation of A-Jacobi.** Computing the matrix product of $\widetilde{\mathrm{R}}$ is similar to applying a Laplacian-like operator over the neighborhood of a vertex: the first-order A-Jacobi is equivalent to the regular Jacobi iterating vertex's one-ring neighbors, and the second-order A-Jacobi covers two-ring neighbors of the vertex etc. Edges on the mesh house the corresponding weights, which are non-zero elements in B. In our GPU implementation, we first precompute all the non-zero A-products offline and load them into the GPU global memory. If an A-product is much smaller than others (by two or three orders) due to the disparity of different constraint stiffness, the corresponding computation will be skipped. During the simulation run time, we update A-coefficients in parallel after barrier constraints are built, knowing A-coefficients are unchanged for the entire inner solve. Before an A-Jacobi iteration starts, we need to group adjacent vertices in one CUDA block to leverage their overlapped neighborhoods. A-coefficients of all the vertices within the block are then fetched into local shared memory in parallel to reduce memory latency. The computation at each vertex is further split into several segments so that the parallelism of the computation can be fully scaled to match the performance of the GPU.

in its adjacent triangles. To this end, we split each A-product of $B_{is}B_{sj}$ as:

$$B_{is}B_{sj} = \left(B_{is}^{bend} + B_{is}^{stretch}\right)\left(B_{sj}^{bend} + B_{sj}^{stretch}\right) \approx B_{is}^{bend}B_{sj}^{stretch} + B_{is}^{stretch}B_{sj}^{bend} + B_{is}^{stretch}B_{sj}^{stretch}. \quad (16)$$

It may appear that there are more addends after discarding higher-order bending terms. The sparsity of $\widetilde{\mathrm{R}}$ is actually improved, and fewer additions/multiplications are needed in A-Jacobi.

- **Improve memory performance** From Eq. (14), one can see that the major computation of an A-Jacobi iteration is the accumulation of scaled A-coefficients. In this procedure, the latency of (GPU) memory access is one to two orders slower than the floating point multiplication and addition, if one directly retrieves A-coefficients from the global memory. In addition to the pre-computation, it is more important to reduce the frequency of the global memory access. We again leverage the locality of the $\widetilde{\mathrm{R}}$: if two vertices are adjacent to each other, their $\widetilde{\mathrm{R}}$ operations also share many A-coefficients. Hence, we group multiple nearby vertices (16 in our implementation) into one CUDA block and pre-fetch their A-coefficients into the shared memory in parallel. The shared memory is a register-like fast local storage. Thanks to overlapping neighborhoods of those vertices, A-Jacobi iterations only need to visit the shared memory and become much faster.
- **Scale up** Compared with regular Jacobi, an A-Jacobi iteration needs to traverse a wider neighborhood of each vertex. This increases per-thread computational overhead but does not improve the parallelism of the iteration – the computation is still unfolded at vertices. To this end, we subdivide the accumulation of A-coefficients into multiple smaller segments to further scale the parallelization. In our implementation, we decompose the weighted summation of A-coefficients in Eq. (14) into four sub-summations. In practice, the total number of launching threads can be manipulated to match the hardware capacity, and the computation at each thread can also be tuned to justify the thread initialization cost.

The best order of A-Jacobi depends on the hardware and the size of the simulation problem. Most experiments reported in this paper are of tens or hundreds thousand of DOFs. In this context, we find that second- or third-order Jacobi are most effective on our hardware. With A-Jacobi, one could use multiple GPUs to further push the performance of large-scale simulations as A-Jacobi can always "deplete" hardware resource and convert it to faster FPS. This feature is not available with existing GPU solvers as most of them unfold computations at vertices.