Advancing GPU IPC for Stiff Affine-Deformable Simulation

KEMENG HUANG*, Carnegie Mellon University, USA and The University of Hong Kong, TransGP, China
XINYU LU*, TransGP, China
HUANCHENG LIN, Carnegie Mellon University, USA and The University of Hong Kong, TransGP, China
TAKU KOMURA, The University of Hong Kong, TransGP, China
MINCHEN LI, Carnegie Mellon University, USA

Fig. 1. **London bus.** We model a $0.7m \times 1.2m \times 2.3m$ bus, including its frame, gears, and wheels, based on the puzzle instructions from Wooden City®. Despite the *eye-norm* modeling accuracy, our simulation robustly captures the realistic non-smooth motion caused by potential interference in the low-precision gear system and the self-adjustment of motor speed (averaging $3m/s$) relative to resistance (see our supplemental video). The simulation includes 265K surface triangles and an average of 118K rapidly changing contact pairs, achieving efficient performance at $1.56s$ per time step ($\Delta t = 5ms$).

Incremental Potential Contact (IPC) is a widely used, robust, and accurate method for simulating complex frictional contact behaviors. However, achieving high efficiency remains a major challenge, particularly as material stiffness increases, which leads to slower Preconditioned Conjugate Gradient (PCG) convergence, even with the state-of-the-art preconditioners. In this paper, we propose a fully GPU-optimized IPC simulation framework capable of handling materials across a wide range of stiffnesses, delivering consistent high performance and scalability with up to 10× speedup over state-of-the-art GPU IPC methods. Our framework introduces three key innovations: 1) A novel connectivity-enhanced Multilevel Additive Schwarz (MAS) preconditioner on the GPU, designed to efficiently capture both stiff and soft elastodynamics and improve PCG convergence at a reduced preconditioning cost. 2) A C^2-continuous cubic energy with an analytic eigensystem for strain limiting, enabling more parallel-friendly simulations of stiff membranes, such as cloth, without membrane locking. 3) For extremely stiff behaviors where elastic waves are barely visible, we employ affine body dynamics (ABD) with a hash-based multi-layer reduction strategy for fast Hessian assembly and efficient affine-deformable coupling. We conduct extensive performance analyses and benchmark studies to compare our framework against state-of-the-art methods and alternative design choices. Our system consistently delivers the fastest performance across soft, stiff, and hybrid simulation scenarios, even in cases with high resolution, large deformations, and high-speed impacts. Our framework will be fully open-sourced upon acceptance.

CCS Concepts: • **Computing methodologies** → **Physical simulation**; *Parallel algorithms.*

Additional Key Words and Phrases: GPU Programming, Incremental Potential Contact, Elastodynamics, Finite Element Method, Affine Body Dynamics, Preconditioning, Cloth Simulation

*K. Huang and X. Lu contribute equally to this work

Authors' Contact Information: Kemeng Huang, kmhuang@connect.hku.hk, kmhuang819@gmail.com, Carnegie Mellon University, USA and The University of Hong Kong, TransGP, China; Xinyu Lu, lxy819469559@gmail.com, TransGP, China; Huancheng Lin, lamws@connect.hku.hk, Carnegie Mellon University, USA and The University of Hong Kong, TransGP, China; Taku Komura, taku@cs.hku.hk, The University of Hong Kong, TransGP, China; Minchen Li, minchernl@gmail.com, Carnegie Mellon University, USA.

1 INTRODUCTION

Incremental Potential Contact (IPC) [Li et al. 2020a] is a cutting-edge elastodynamic contact simulation method widely used in computer graphics, computational mechanics, robotics, etc. Despite its robustness, accuracy, and differentiability in simulating complex frictional contact behaviors, IPC's efficiency remains a significant bottleneck, limiting its full potential. Several variants have been proposed to address IPC's efficiency issues, often at the expense of accuracy [Lan et al. 2023, 2022b, 2021; Li et al. 2023].

To accelerate IPC without sacrificing accuracy, Huang et al. [2024] introduced GIPC with a GPU-friendly redesign of the numerical algorithms. This included replacing direct factorization with a Preconditioned Conjugate Gradient (PCG) solver and proposing a Gauss-Newton approximation of the barrier Hessian matrices with analytic eigensystems. While GIPC is effective, its efficiency deteriorates significantly as object stiffness increases. This is mainly due to the growing condition number of the global linear system, which requires more PCG iterations to solve. To improve the PCG convergence of stiff material, Wu et al. [2022] proposed the multilevel additive Schwarz (MAS) preconditioner [Wu et al. 2022]. Their approach involves sorting the nodes based on Morton codes and building a hierarchy by grouping the nodes at each level. Despite its effectiveness, the lack of consideration for mesh connectivity during reordering leads to suboptimal domain hierarchy construction. This results in high construction costs, additional overhead for deformable simulations and challenges for GPU optimization.

When simulating stiff elastic thin shells like cloth, another challenge arises, which is membrane locking. With linear triangle elements, the stiff membrane energy (Young's modulus around $10MPa$ for cloth [Penava et al. 2014]) will often result in nonnegligible extra artificial bending resistance, forming sharp creases and plastic appearances in the simulation results (Figure 12 top). Simulating cloth with smaller stiffness can result in more realistic wrinkles, but it will suffer from over-elongation issues. To tackle this challenge, Li et al. [2021] propose to augment soft membrane energy with a barrier-based strain-limiting term to prevent cloth from over-stretching while avoiding membrane locking. This strategy enables realistic cloth simulation within the IPC framework, but the required exact strain limit satisfaction necessitates a backtracking-based line search filtering scheme, as the updated strain has a complicated relation to the step size, making analytic expressions unavailable. Additionally, numerical eigendecomposition is needed for computing a positive semi-definite approximation of the strain-limiting term's Hessian matrix, which further complicates GPU optimization.

For even stiffer problems where elastic waves are barely visible, objects can be treated as rigid [Ferguson et al. 2021] or stiff affine [Lan et al. 2022a] bodies in the IPC framework for a reduced number of degrees of freedom (DOF). But to accurately simulate contact behaviors, surface elements from the original input geometry are used, which also makes simulating rigid-deformable coupled scenarios convenient. Chen et al. [2022] introduced a unified Newton barrier method for stiff affine-deformable simulation, possibly with articulation constraints. However, although some components are GPU-accelerated, the primary simulation processes still execute on the CPU, leading to suboptimal performance. ZeMa [Du et al. 2024] is another GPU IPC framework for stiff affine-deformable simulation, with most processes parallelized on the GPU, except for the linear system, which is solved on the CPU using a direct solver. However, ZeMa lacks a well-optimized contact Hessian assembly algorithm, as it accumulates the 12×12 dense contact Hessian matrices to the affine body DOFs atomically, where conflicting operations can significantly impede the performance, especially when there are a large number of contacts. Moreover, direct solvers often fall short in large-scale simulations.

In summary, there are still plenty of rooms for optimizing linear solver preconditioners, strain limiting, global Hessian matrix assembly, etc., for realizing a highly GPU-optimized IPC framework that can efficiently simulate large-scale affine-deformable coupled scenarios. In this paper, we propose such a framework, achieving up to 10× speedup compared to GIPC via the following 3 major innovations:

- A novel connectivity-enhanced MAS preconditioner on the GPU that achieves improved PCG convergence at a lower precomputation and per-iteration cost (section 4). Our preconditioner consistently performs effective and well-structured aggregations, which supports smaller blocksizes and further GPU optimizations based on warp reduction.
- A C^2-continuous cubic strain-limiting energy with an analytic eigensystem, enabling realistic cloth simulation without membrane locking (section 5). As numerical eigendecomposition and line search filtering for the feasibility of the strain limits are not needed, our model supports highly GPU-parallelized computations.
- A hash-based multi-layer reduction strategy for fast Hessian matrix assembly (section 6). Our strategy significantly reduces the number of numerical operations, and it enables the development of a memory-efficient symmetric blockwise sparse matrix-vector multiplication method to further boost PCG performance.

In section 7, we perform extensive and rigorous performance analyses and benchmark studies to validate our framework and compare it to state-of-the-art GPU IPC systems and alternative design choices that may seem reasonable but suffer from suboptimal performance in practice. Our framework exhibits the fastest performance in soft, stiff, and hybrid simulation scenarios, even with high resolution, extreme deformation, and high-speed impacts. Our system will be fully open-sourced upon acceptance.

2 RELATED WORKS

Contact Simulation. Simulating frictional contact for (nearly) rigid and deformable solids has been an extensively studied topic in both computer graphics and computational mechanics. Starting from a few decades ago, various methods have been developed, ranging from impulse-based methods [Bridson et al. 2002; Harmon et al. 2009], impact zone methods [Harmon et al. 2008; Li et al. 2020b; Tang et al. 2018], and more constraint-based methods [Allard et al. 2010; Kaufman et al. 2008; Macklin et al. 2019; Otaduy et al. 2009; Verschoor and Jalba 2019] to fictitious domain methods [Jiang et al. 2017; Misztal and Bærentzen 2012; Müller et al. 2015; Pagano and Alart 2008], etc. These methods are effective and efficient in many situations, but they lack guarantees on algorithmic convergence and generating penetration-free results, especially when simulating challenging examples where extensive parameter tuning is often also required when varying set-ups. We refer to Andrews et al. [2022] for a comprehensive review.

More recently, Li et al. [2020a] proposed Incremental Potential Contact (IPC), which simulates the nonsmooth frictional contact behaviors of deformable solids by approximating them using smooth constitutive models with bounded error. Equipped with a filter line search scheme and the projected Newton method, penetration-free results and algorithmic convergence are guaranteed within an optimization time integrator. IPC has shown effectiveness in various application scenarios, including cloth reconstruction [Zheng et al. 2024], material modeling of interlocked rigid components [Tang et al. 2023], multi-material coupled simulations [Jiang et al. 2022; Li et al. 2024; Xie et al. 2023], etc. Despite being robust and accurate, improving the efficiency of IPC while maintaining its reliability is still a major challenge.

Accelerating IPC Performance. Over the past few years, several methods have been proposed to enhance IPC's efficiency. Lan et al. [2021] reduced the DOFs of deformable bodies to their medial axis and derived contact and friction forces on the associated slab primitives. With significantly fewer DOFs, the efficiency improved substantially at the expense of accuracy, and penetration-free results were only guaranteed on their medial representation. Lan et al. [2022b] incorporated IPC into the projective dynamics framework

on the GPU, delaying contact constraint set updates and continuous collision detection (CCD) to once per set of inner iterations to efficiently generate penetration-free results. However, their spring-based approximation to the barrier-based contact model limited their performance in challenging scenarios. Lan et al. [2023] applied a stencil-wise coordinate descent method to solve the IPC system. With a penetration-free warm start and graph coloring-based GPU parallelization, their method achieved fast performance. However, as a block coordinate descent method, its performance drops quickly with stiff materials or a large number of contacts. Wang et al. [2023a] and Li et al. [2023] both applied time-splitting techniques to decouple contact and elasticity simulation, achieving faster performance but requiring small time steps for stiff problems due to the lack of unconditional stability. These methods trade accuracy for efficiency using simplified models or specialized strategies, often introducing extra parameters for tuning.

More recently, Huang et al. [2024] introduced GIPC, a fully GPU-optimized IPC method with an inexact Gauss-Newton solver using the MAS preconditioner [Wu et al. 2022]. GIPC also derived approximated contact energy Hessians with analytic eigensystems, avoiding GPU-unfriendly numerical eigendecomposition. Concurrently, Du et al. [2024] proposed ZeMa, another GPU IPC framework supporting the coupling of deformable bodies with stiff affine bodies. Both GIPC and ZeMa achieved significant speed-ups without sacrificing accuracy. However, as discussed earlier, their high performance could unavoidably degrade when there are stiff materials or a large number of contacts. We thus follow this path to keep advancing GPU IPC for stiff elastodynamics without sacrificing accuracy and robustness.

Multiresolution Methods. One of the most critical design choices on the GPU is to apply iterative solvers instead of direct solvers for linear systems. Multigrid is a popular GPU-friendly preconditioner often used in iterative linear solvers, such as the PCG method, for efficiently solving ill-conditioned systems. The major challenge in multigrid methods is the development of effective restriction and prolongation operators, especially for unstructured meshes. Researchers in computer graphics have explored various geometric multigrid methods (GMG) [Wang et al. 2020, 2018; Xian et al. 2019; Zhu et al. 2010] by constructing mesh hierarchies, where numerous implementation issues are often encountered. Compared to GMG, another type of multigird, the algebraic multigrid (AMG) methods [Demidov 2019; Naumov et al. 2015; Tamstorf et al. 2015], do not require explicitly constructing coarser geometries. However, they often require a significant amount of precomputation to analyze the structure of the matrix, making them more expensive when handling problems with varying matrix structures. This inspires Wu et al. [2022] to propose a multilevel additive schwarz (MAS) [Dryja and Widlund 1990] preconditioner, incorporating the idea of domain decomposition to achieve fast convergence with low per-iteration costs when applied to PCG. But its node reordering based on Morton code is not aware of the mesh connectivity, which may lead to suboptimal aggregation. We thus propose a connectivity-enhanced MAS based on METIS [Karypis and Kumar 1998] to tackle this issue and achieve even faster performance.

Strain Limiting and Eigenanalysis. To realistically simulate cloth using linear elements, strain limiting is often applied to avoid membrane locking. Most existing methods [English and Bridson 2008; Goldenthal et al. 2007; Narain et al. 2012; Thomaszewski et al. 2009; Wang et al. 2010] enforce a bound constraint on the strain measure per element using augmented Lagrangian approaches or post-projection techniques. While these methods do not guarantee strict satisfaction of strain limits, they can effectively simulate membrane locking-free behaviors. However, when coupled with frictional contact, these methods often introduce artifacts due to the need for smaller time steps or difficulties in achieving solver convergence. More recently, Li et al. [2021] applied a barrier method to handle strain-limiting constraints, guaranteeing exact constraint satisfaction in a monolithic manner. However, it requires expensive backtracking-based line search filtering and numerical eigendecomposition, which are both GPU-unfriendly. Here, the backtracking line search is responsible for finding configurations that are both feasible and with a smaller objective function. With the development of analytical eigenanalysis on distortion energies, such as the isotropic [Lin et al. 2022; Smith et al. 2019] and anisotropic [Kim et al. 2019] elasticity, membrane [Kim 2020; Panetta 2020], bending [Wang et al. 2023b; Wu and Kim 2023], and contact [Huang et al. 2024; Shi and Kim 2023] energies, they have become popular components of modern GPU simulation methods. We thus propose a cubic strain-limiting energy with analytic eigensystems so that realistic cloth dynamics can be more efficiently simulated without the need of numerical eigendecomposition and backtracking-based line search filtering.

3 BACKGROUND AND PRELIMINARIES

3.1 Unified Incremental Potential

IPC [Li et al. 2020a] formulates implicit time integration of elastodynamic contact as a minimization problem:

$$\mathbf{x}^{t+\Delta t} \approxeq \arg\min_{\mathbf{x}\in\mathbb{R}^{3n}} E_s(\mathbf{x}), \tag{1}$$

where $\mathbf{x}$ is the world-space positions of the n nodes, followed by a velocity update $\mathbf{v}^{t+\Delta t} = (\mathbf{x}^{t+\Delta t} - \mathbf{x}^t)/\Delta t$, taking implicit Euler as an example. The objective function

$$E_s(\mathbf{x}) = \frac{1}{2}(\mathbf{x}-\hat{\mathbf{x}})^T\mathbf{M}_s(\mathbf{x}-\hat{\mathbf{x}}) + \Delta t^2\Psi_s(\mathbf{x}) + B(\mathbf{x}) + D(\mathbf{x}) \tag{2}$$

is the Incremental Potential (IP), where $\hat{\mathbf{x}} = \mathbf{x}^t + \Delta t\mathbf{v}^t + \Delta t^2\mathbf{M}_s^{-1}\mathbf{f}_e$, $\mathbf{M}_s$ is the mass matrix, $\mathbf{f}_e$ is the external force, B is the contact barrier potential, and D is the approximated friction potential. The total elasticity energy $\Psi_s(\mathbf{x}) = \Psi_{vol}(\mathbf{x}) + \Psi_{memb}(\mathbf{x}) + \Psi_{bend}(\mathbf{x}) + \Psi_{strain}(\mathbf{x})$ contains volumetric strain energies ($\Psi_{vol}(\mathbf{x})$) as well as membrane ($\Psi_{memb}(\mathbf{x})$), bending ($\Psi_{bend}(\mathbf{x})$), and strain-limiting ($\Psi_{strain}(\mathbf{x})$) energies for thin shells.

The IP for simulating affine bodies [Lan et al. 2022a] is defined as

$$E_r(\mathbf{q}) = \frac{1}{2}(\mathbf{q}-\hat{\mathbf{q}})^T\mathbf{M}_r(\mathbf{q}-\hat{\mathbf{q}}) + \Delta t^2\Psi_r(\mathbf{q}) + B(\mathbf{x}(\mathbf{q})) + D(\mathbf{x}(\mathbf{q})), \tag{3}$$

where $\mathbf{q} \in \mathbb{R}^{12\alpha}$ is the reduced space coordinates of the α affine bodies. For affine body j, $\mathbf{q}_j = [\mathbf{p}_j^T, \mathbf{A}_{j1}^T, \mathbf{A}_{j2}^T, \mathbf{A}_{j3}^T]^T$, with $\mathbf{p}_j \in \mathbb{R}^3$ the translation vector and $\mathbf{A}_j \in \mathbb{R}^{3\times3}$ the affine deformation

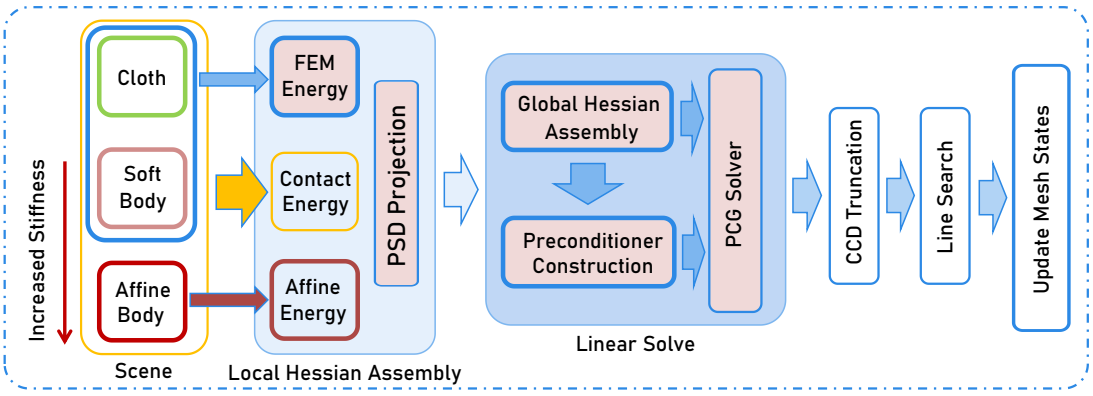


Fig. 2. **An overview of 1 Newton iteration in our unified GPU IPC framework.** Our simulation focuses on cloth, soft bodies with varying stiffness, and stiff affine bodies [Lan et al. 2022a]. Cloth and soft bodies are modeled using FEM. Contact between all objects is handled using IPC [Li et al. 2020a]. We solve the system with a Newton-PCG solver and use Continuous Collision Detection (CCD) to filter the search direction and prevent penetration, applying backtracking line search to ensure energy decrease. Our contributions, highlighted in light pink blocks, include a novel strain limiting energy with an analytic eigensystem for efficient PSD projection, an optimized linear solver with a connectivity-enhanced MAS preconditioner, and a highly-parallelized global Hessian assembly strategy for affine-deformable coupling.

matrix. Here, $\mathbf{x}_i(\mathbf{q}_j) = \mathbf{A}_j\bar{\mathbf{x}}_i + \mathbf{p}_j$ is the current position of full-space node i on affine body j for measuring contact energies, with $\bar{\mathbf{x}}_i$ the rest-state full space coordinates. The Jacobian matrix $\mathbf{J}_{ij} = \frac{\partial \mathbf{x}_i}{\partial \mathbf{q}_j} \in \mathbb{R}^{3\times 12}$ maps information between the reduced and full spaces. For example, the mass matrix $\mathbf{M}_r$ of affine body j is calculated as $\mathbf{M}_r = \sum_i m_i \mathbf{J}_{ij}^T \mathbf{J}_{ij}$, where m_i is mass of each node i. Similarly, the contact gradient of affine body j is calculated as $\nabla_{\mathbf{q}_j} B = \sum_i \mathbf{J}_{ij}^T \nabla_{\mathbf{x}_i} B$. From this, we can see that computing contact forces and Hessian matrices for the reduced DOFs could require accumulating values from a significant amount of contact pairs, which can be expensive.

Combining Equation 2 and Equation 3, denoting nodal positions of unreduced deformable solids as $\mathbf{x}_s$, the unified affine-deformable coupled IP [Chen et al. 2022] is defined as

$$\begin{aligned} E(\mathbf{q}, \mathbf{x}_s) = &\frac{1}{2}(\mathbf{x}_s - \hat{\mathbf{x}}_s)^T \mathbf{M}_s (\mathbf{x}_s - \hat{\mathbf{x}}_s) + \Delta t^2 \Psi_s(\mathbf{x}_s) \\ &+ \frac{1}{2}(\mathbf{q} - \hat{\mathbf{q}})^T \mathbf{M}_r (\mathbf{q} - \hat{\mathbf{q}}) + \Delta t^2 \Psi_r(\mathbf{q}) \\ &+ B([\mathbf{x}_s^T, \mathbf{x}(\mathbf{q})^T]^T) + D([\mathbf{x}_s^T, \mathbf{x}(\mathbf{q})^T]^T), \end{aligned} \tag{4}$$

and we minimize this energy w.r.t. $\{\mathbf{q}, \mathbf{x}_s\}$ per time step to simulate the coupled system:

$$(\mathbf{q}, \mathbf{x}_s)^{t+\Delta t} \approx \underset{\mathbf{x}_s \in \mathbb{R}^{3n}, \mathbf{q} \in \mathbb{R}^{12\alpha}}{\arg\min} \; E(\mathbf{q}, \mathbf{x}_s). \tag{5}$$

To accelerate local Hessian computation on the GPU, we utilize potential energies with analytic eigensystems (subsection 3.2). Specifically, for collision energy B and friction energy D, we employ the eigenanalysis from GIPC [Huang et al. 2024]. The Stable Neo-Hookean model [Smith et al. 2018] is applied for Ψ_{vol}, while the bending energy from [Wu and Kim 2023] is used for Ψ_{bend}. To efficiently assemble Ψ_{memb} and Ψ_{strain}, we use Kim [2020]'s membrane energy and propose a cubic strain-limiting energy with analytic eigensystem, detailed in section 5.

3.2 IP Minimization and Linear Solves

To minimize $E(\mathbf{u})$, where $\mathbf{u} = \{\mathbf{q}, \mathbf{x}\}$, line search methods are often used, which iteratively search along a descent direction by minimizing a local quadratic proxy of the IP in iteration i:

$$E_i(\mathbf{u}) = E(\mathbf{u}_i) + (\mathbf{u} - \mathbf{u}_i)^T \nabla E(\mathbf{u}_i) + \frac{1}{2}(\mathbf{u} - \mathbf{u}_i)^T \mathbf{P}(\mathbf{u}_i)(\mathbf{u} - \mathbf{u}_i). \tag{6}$$

The descent direction is obtained by solving the linear system $\mathbf{P}(\mathbf{u}_i)\mathbf{d} = -\nabla E(\mathbf{u}_i)$, where $\mathbf{d} = \mathbf{u}_i^* - \mathbf{u}_i$ and $\mathbf{u}_i^* = \arg\min_{\mathbf{u}} E_i(\mathbf{u})$. Here, $\mathbf{P}$ is a symmetric positive definite (SPD) proxy matrix approximating $\nabla^2 E$ for fast convergence, with projected Newton [Li et al. 2020a] or Gauss-Newton [Huang et al. 2024] approximations shown effective. The new iterate is computed as $\mathbf{u}_{i+1} = \mathbf{u}_i + \alpha \mathbf{d}$, where α is the step size calculated via backtracking line search, with filtering to ensure feasibility imposed by contact and strain limits [Li et al. 2021]. The minimization terminates when the iterate is sufficiently close to the local minimum, e.g. when $\|\mathbf{d}\| \le \varepsilon_d$, where ε_d is the Newton tolerance.

To solve the linear system $\mathbf{P}(\mathbf{u}_i)\mathbf{d} = -\nabla E(\mathbf{u}_i)$, Cholesky factorization, using e.g. CHOLMOD [Chen et al. 2008], is a popular solution. These direct solvers perform well on ill-conditioned systems but their performance decreases with increasing DOFs, especially for large-scale simulations. Additionally, the high memory cost of direct solvers results in worse performance on the GPU compared to the CPU [Lan et al. 2021]. Conversely, iterative solvers, such as the preconditioned conjugate gradient (PCG) method, iteratively approach the solution via matrix-vector multiplications (SpMV), making them more practical for large-scale simulations.

Although PCG has superior convergence rate compared to many non-Krylov iterative solvers, such as Jacobi and Gauss-Seidel, it still requires many iterations to reach low error when the linear system is ill-conditioned. An effective preconditioner that approximates the inverse of the proxy matrix can make the system better conditioned and easier to solve. MAS [Wu et al. 2022] is currently the best option considering both convergence speed and the overhead of computing

and applying the preconditioner. Additionally, the high computational cost of matrix assembly and SpMV in each iteration often makes PCG the major bottleneck in simulation. Thus, designing GPU-parallel algorithms for PCG is also critical. See Figure 2 for an overview of our simulation framework.

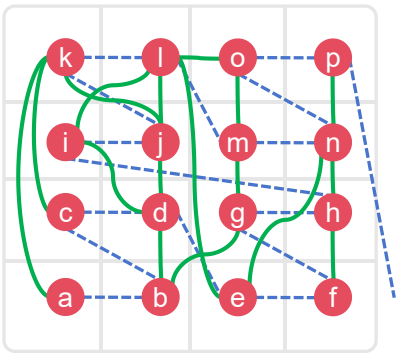


Fig. 3. **A node sorting example using Morton code.** Red disks represent nodes, green lines show topology connections, and dotted blue lines indicate Morton code order. After sorting, nodes will follow the character sequence shown. Note that this is a contrived example for clarity.

4 CONNECTIVITY-ENHANCED MAS PRECONDITIONER WITH GPU OPTIMIZATION

The multilevel additive Schwarz (MAS) preconditioner [Wu et al. 2022] enables efficient linear solver convergence with excellent scalability w.r.t. problem size and stiffness in elastodynamics problems. However, its node reordering has some limitations, which makes it less effective in cases with complex geometry or softer materials. We start by a comprehensive analysis of these limitations (subsection 4.1), and then propose a connectivity-enhanced MAS construction (subsection 4.2) together with further GPU optimizations (subsection 4.3) to achieve a 2.2× faster performance in average compared to Wu et al. [2022].

4.1 Discussions of MAS

MAS constructs the domain hierarchy by first sorting all mesh vertices using Morton codes, and then grouping these vertices into subdomains of size N. Next, connected vertices within each subdomain are merged to create the next-level simulation domain. These merged vertices, called super nodes, are then grouped again to form the domains at subsequent levels until a maximum level is reached, no further merges could be performed, or only a single subdomain is present. During preconditioning, the inverse of each subdomain's Hessian matrix is multiplied with the mapped input vector, and the resulting vectors are mapped and summed to produce the output. With a fixed number of levels, maximizing node connectivity within each subdomain is critical to making dimensionality reduction more effective, which allows information to propagate farther at coarser levels and ensures faster convergence. See our supplemental document for more details.

We identify two major areas for potential improvement in MAS: 1) The convergence of MAS heavily relies on node reordering using Morton codes. While Morton codes capture spatial information, they do not consider mesh connectivity, which can result in slower convergence. 2) To obtain denser node connectivity within each subdomain and improve aggregation efficiency, a subdomain size of 32 vertices is typically used in MAS. However, this leads to significant overhead in terms of precomputation and preconditioning, especially when using double-precision floating-point numbers. Reducing the subdomain size to 16 vertices can decrease computational costs, but it may also slow down convergence, as fewer nodes are merged within each subdomain, potentially resulting in worse overall performance [Wu et al. 2022].

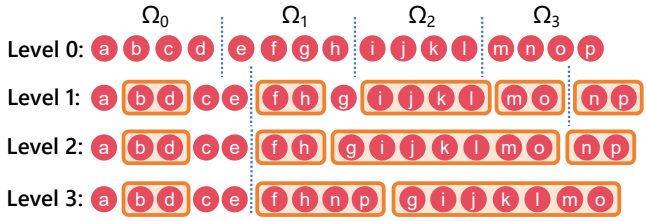


Fig. 4. **MAS aggregation of Figure 3 using Morton code.** Within each subdomain (size 4 here, separated by dashed blue lines), connected nodes are merged into a super node (orange block) at the next coarser level. The order of the super node is determined by the lowest order of the merged nodes. For example, in level 1, nodes b and d are merged and positioned before c because b has a lower order. The subdomains are then reconstructed from sets of 4 consecutive super nodes, and here this process continues until no further merging is possible at a certain level.

Case Study. Let's examine the first issue more closely with an example (Figure 3). In this case, we assume a domain size of 4 for MAS, meaning the nodes are grouped into subdomains $\Omega = \Omega_0 \cup \Omega_1 \cup \Omega_2 \cup \Omega_3$, each with 4 nodes, at level 0, as shown in Figure 4. In $\Omega_0 = \{a, b, c, d\}$, only nodes $\{b, d\}$ are connected. Consequently, at the coarser level 1, these two nodes are merged into a super node. Similarly, nodes $\{f, h\}$, $\{i, j, k, l\}$, $\{m, o\}$, and $\{n, p\}$ are merged at level 1. However, starting from level 1, there is no connectivity between any super nodes in the first subdomain, and only a few can be merged in the other subdomains. This results in 3 levels for these 16 nodes, with inefficient aggregation, leading to poor performance of the MAS preconditioner. In fact, the performance can even be worse than that of a simple block Jacobi preconditioner in practice.

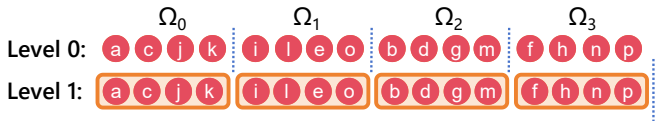


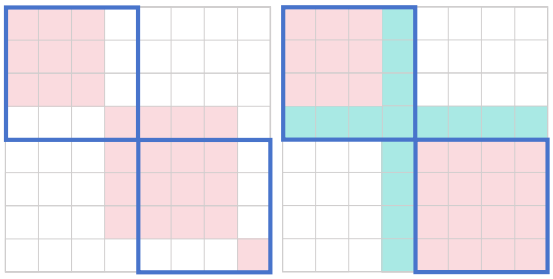
Fig. 5. **Our connectivity-enhanced MAS aggregation of Figure 3.** The orange blocks represent super nodes at the coarser levels. At level 0, our method groups nodes based on mesh connectivity, and so each group becomes 1 super node at level 1, completing the aggregation.

Since the aggregation is based on node connectivity, a natural improvement is to group the nodes directly by their connectivity at level 0, ensuring that the 4 nodes in each subdomain are fully connected and can be merged into a single super node, as shown in Figure 5. In this way, at level 1, there will be only 4 super nodes that cover all the original nodes in the mesh, allowing us to terminate the aggregation with just one subdomain. As a result, this method constructs an efficient two-level MAS that can be used to effectively approximate the global Hessian.

4.2 Connectivity-Enhanced MAS Construction

The example above demonstrates the importance of considering mesh connectivity during node reordering for optimal node aggregation. However, in practice, achieving perfect aggregation is both expensive and challenging. Therefore, we propose a practical connectivity-enhanced strategy that efficiently approximates optimal aggregation. Our core idea is to use METIS [Karypis and Kumar 1998] to partition the mesh nodes before the simulation and then reorder them with padding at level 0, optimizing node connectivity within each subdomain.

METIS is a fast graph partitioning method that can divide mesh nodes into a specific number of partitions, minimizing inter-partition connectivity and often leading to dense connectivity within each partition. However, METIS does not guarantee that each partition will have exactly the same number of nodes, as achieving perfectly balanced partitions is expensive and sometimes impossible. Handling blocks of varying sizes on the GPU can degrade performance and complicate implementation. Thus, we can treat METIS partitioning as a reordering step, and then distribute all nodes evenly into each MAS subdomain. However, this can still result in suboptimal aggregation since the nodes in the same subdomain may come from different METIS partitions as shown in Figure 6a.



(a) (b)

Fig. 6. **Aligning METIS partitions and MAS subdomains.** The red blocks represent subdomains from the METIS partition, the deep blue rectangle shows the MAS subdomain, and the green-blue blocks represent inserted zero entries as padding. (a) An example where the METIS partition sizes are 3 and 4, while the MAS subdomain size is 4, leading to isolated nodes within subdomains and potentially suboptimal aggregation (subsection 4.1). (b) Our solution inserts zero entries to prevent isolated nodes caused by misalignment between the METIS partition and MAS subdomains.

To address this issue and ensure effective aggregation, we propose an index mapping scheme to assign each partition to a separate MAS subdomain and pad the empty entries as illustrated in Figure 6b. We first set the number of METIS partitions M as

$$M = \left\lceil \frac{V}{N - N_o} \right\rceil, \tag{7}$$

where N is the number of nodes in each MAS subdomain, V is the total number of mesh nodes, $\lceil \cdot \rceil$ is the ceiling operator, and N_o is a slack variable that ensures the node count in each METIS partition does not exceed N. During the precomputation stage, we search for the smallest possible N_o, starting from 0. If any partition exceeds N nodes, we increment N_o and repartition. In almost all cases, $N_o = 1$ is sufficient.

To implement this approach, we first construct a mapping array of size $N \cdot M$, which maps each node to a slot in its corresponding MAS subdomain, as shown in Figure 7. This mapping array is then used to build the MAS preconditioner. To ensure correct data access during construction, we also create a remapping array, which helps GPU threads access the correct node data for each entry in the mapping array. Padded entries, where no nodes are mapped, are marked with the value -1 (see Figure 7). The remaining MAS levels are constructed using the original method based on node connectivity. By updating the mapping per time step considering the connectivity introduced by the current contact pairs, our approach reduces connectivity issues and enables more effective aggregation, improving the convergence rate of the MAS-preconditioned linear solver by up to 2.4×.

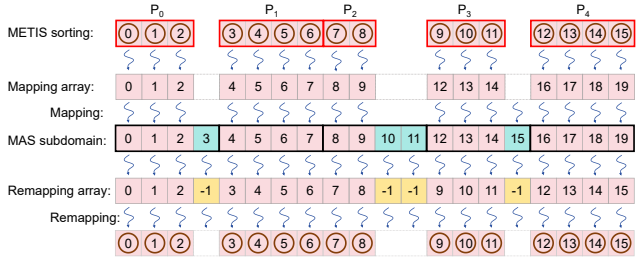


Fig. 7. **Our mapping and remapping scheme.** Nodes in each METIS partition are mapped to a separate MAS subdomain using a mapping array (stored in contiguous memory, aligned here for clarity) with padding shown in blue. Our remapping array maps MAS subdomain nodes to the corresponding mesh nodes for easy access to geometric data.

Another significant advantage of our method is the ability to use smaller subdomain sizes, resulting in substantial performance gains when computing the inverse of diagonal blocks and applying the preconditioner. In our implementation, we use a subdomain size of 16, compared to the size 32 used in [Wu et al. 2022]. Despite the smaller size, our method remains effective as node connectivity within each subdomain is optimized. This enables cost-effective preconditioning while maintaining a high convergence speed, achieving up to 3× speedup in the PCG solve.

4.3 GPU Warp Reduction for Connectivity-Enhanced MAS

Our connectivity-enhanced MAS construction ensures that the nodes within each subdomain at level 0 are densely connected. This

Algorithm 1 Compute MAS preconditioner

Input: Global Hessian matrix **AG** ▷ an array of tuples: [block row index, block column index, 3×3 block]
Output: MAS preconditioner $\mathbf{M}_{MAS}^{-1}$
Method:

for $GlobalThreadId = 0, 1, ..., len(\mathbf{AG}) - 1$ **do in parallel**
  $[r, c, \mathbf{M3}] = \mathbf{AG}(GlobalThreadId)$ ▷ get 3×3 block
  $[domain\_Idx, domain\_Idy] = \mathbf{Map}(r, c)$ ▷ align METIS and MAS subdomain with padding (Figure 6)
  $m = domain\_Idy/N$ ▷ get MAS subdomian id
  $l = 0$ ▷ MAS level, starting from 0
  if $domain\_Idx/N == domain\_Idy/N$ **then** ▷ if nodes r and c belong to the same level-0 MAS subdomain, write **M3** to the level-0 MAS matrix
    $\mathbf{M}_{0,MAS}(m)[domain\_Idx\%N, domain\_Idy\%N] = \mathbf{M3}$
  else ▷ add **M3** to higher level MAS matrix if needed
    for $l = 1, 2, ..., levels - 1$ **do**
      if r and c in the same subdomain at level l **then**
        AtomicAdd **M3** to $\mathbf{M}_{l,MAS}$
      end if
    end for
  end if
end for
for $BId = 0, 1, ..., len(\mathbf{M}_{0,MAS}) - 1$ **do in parallel** ▷ accumulate all 3×3 blocks in $\mathbf{M}_{0,MAS}$ by warp reduction and add it to all higher-level MAS matrices
  $\mathbf{M3} = warpReduction(\mathbf{M}_{0,MAS}(BId))$
  for $l = 1, 2, ..., levels - 1$ **do**
    AtomicAdd **M3** to $\mathbf{M}_{l,MAS}$
  end for
end for
return $\mathbf{M}_{MAS}^{-1}$

property opens up new opportunities for additional GPU optimizations in efficiently computing the proxy Hessian matrix at coarser MAS levels.

In GPU MAS [Wu et al. 2022], since nodes in each subdomain can be mapped to an arbitrary number of super nodes at the next coarser level, it is nontrivial to design an efficient reduction scheme for accumulating the matrix entries to assemble the proxy matrix at the next level. Instead, they construct a mapping array to establish connections between mesh nodes and super nodes at different levels, and then apply atomic operations to compute the proxy matrices at higher levels. Take Figure 4 as an example, node $\{f\}$ in level 0 maps to $\{f, h\}$ in level 1 and 2, and then to $\{f, h, n, p\}$ in level 3. The data associated with $\{f\}$ will be added to the super nodes at coarser levels via atomic operations. However, as we see, as the structure becomes coarser, more atomic additions of a larger number of matrix entries are needed, which will result in more writing conflicts and increase computational cost, especially when using double-precision floating-point numbers.

In our case, the nodes within the same level-0 subdomain map to a small number (often the same) super nodes in coarser levels. This allows for the application of a warp reduction method to accumulate the level-0 matrix blocks. The accumulated matrix is then used to construct the sub-matrix of the next coarser levels using atomic operations. This approach effectively reduces the cost of MAS matrix precomputation, since significantly less atomic operations are needed as the level-1 matrix can be efficiently computed using warp reduction. This also allows for more levels in the MAS preconditioner with lower computational cost. See Algorithm 1 and our supplemental document for more details.

5 CUBIC STRAIN-LIMITING ENERGY WITH ANALYTIC EIGENSYSTEM

We develop our cubic strain-limiting energy based on the FEM Baraff-Witkin (FBW) constitutive model [Kim 2020]. Inspired by Li et al. [2021] which augments soft membrane energy with a barrier-based strain-limiting term to avoid membrane locking, we aim to develop a barrier-free strain-limiting energy with analytic eigensystems so that expensive backtracking-based line search filtering for the strain limit and numerical eigendecomposition are both avoided. We will first introduce FBW and our model (subsection 5.1), and then derive the analytic eigensystem of our Hessian matrix (subsection 5.2).

5.1 Formulation

In the FBW method [Kim 2020], the membrane energy is defined as the sum of a stretching term and a shearing term:

$$\Psi_{memb} = \Psi_{stretch} + \Psi_{shear}. \tag{8}$$

The stretching term $\Psi_{stretch}$ is defined using two orthonormal basis, $\mathbf{n}_u = [1, 0]^T$ and $\mathbf{n}_v = [0, 1]^T$, and based on $I_5(\mathbf{F}, \mathbf{n}) = \mathbf{n}^T\mathbf{F}^T\mathbf{F}\mathbf{n}$ [Kim et al. 2019], where $\mathbf{F} = [\mathbf{F}_0|\mathbf{F}_1] \in \mathbb{R}^{3x2}$ is the deformation gradient:

$$\Psi_{stretch} = \lambda a_t(\sqrt{I_5(\mathbf{F}, \mathbf{n}_u)} - 1)^2 + \lambda a_t(\sqrt{I_5(\mathbf{F}, \mathbf{n}_v)} - 1)^2. \tag{9}$$

Here, a_t is the volume weight, i.e. the product of triangle area and the thickness, and λ is the stretching stiffness. In this paper, we simply use the same λ for both $\mathbf{n}_u$ and $\mathbf{n}_v$ directions. But note that different stiffnesses can be applied to model more anisotropic behaviors, which will not affect the computation and eigenanalysis.

We follow Kim [2020] and use $\sqrt{I_5(\mathbf{F}, \mathbf{n})}$ to define our strain-limiting energy, as it effectively represents the stretch factor in the direction $\mathbf{n}$. Directly incorporating I_5 into the barrier-based strain-limiting model of Li et al. [2021] enables reusing the eigenanalysis from Kim [2020]. However, backtracking-based line search filtering, which is not parallel-friendly, is still required since no analytic bound for feasible step sizes is available for the strain. Additionally, because the barrier function diverges rapidly near the strain limit, solver performance becomes highly sensitive to the barrier stiffness, complicating parameter setting. To address this, we apply a cubic penalty function to handle the inequality constraint imposed by the strain limit. This approach allows for slight constraint violations while providing stretching resistance on the same scale as practical settings.

To simplify the explanation, let's consider the $\mathbf{n}_u$ direction in Equation 9 as an example. We construct a C^2-continuous clamped

cubic strain-limiting energy using I_5:

$$\Psi^u_{SL} = \lambda' a_t S(I_5(\mathbf{F}, \mathbf{n}_u))(\sqrt{I_5(\mathbf{F}, \mathbf{n}_u)} - 1)^3, \tag{10}$$

where λ' is the stiffness and S is a piecewise function defined as

$$S(s) = \begin{cases} 1, & s > 1, \\ 0, & 0 \le s \le 1, \end{cases} \tag{11}$$

and so the strain-limiting force is only applied when the cloth is stretched. By similarly defining the cubic term in the $\mathbf{n}_v$ direction, we obtain our complete cubic strain-limiting energy

$$\Psi_{SL} = \Psi^u_{SL} + \Psi^v_{SL}.$$

This energy is well-defined even if the strain limit is exceeded, eliminating the need for line search filtering. Given that the Young's modulus of real fabrics ranges from $1MPa$ to $10MPa$ [Penava et al. 2014] when the cloth is stretched, setting our $\lambda' = 5MPa$ provides stretching resistance at the same scale when the cloth approaches the strain limit, e.g., 5%. Thus, our model prevents over-elongation in practical cloth manipulation scenarios. For $\Psi_{stretch}$, we can now safely use a small stiffness ($\lambda = 0.05MPa$ in all our examples) to avoid membrane locking. Refer to Figure 8 for a plot of our composed membrane energy in the $\mathbf{n}_u$ direction. Since the stiffness for the shearing term Ψ_{shear} primarily depends on fabric type and, based on our observations, has minimal impact on the over-elongation or membrane locking issues, we simply set it to 30% of the stretching stiffness and apply the original eigenanalysis from Kim [2020] to compute its PSD Hessian.

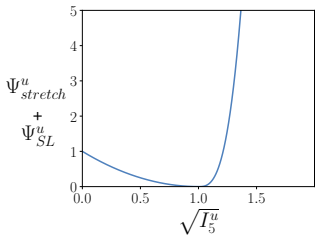


Fig. 8. **Our composed strain energy in the $\mathbf{n}_u$ direction.** Combined with the membrane energy from Kim [2020], our cubic strain-limiting energy enables using soft compression resistance and stiff stretching resistance to avoid membrane locking and over-elongation issues. Here, $\lambda a_t = 1$ and $\lambda' = 100\lambda$ are used for visualization.

5.2 Eigenanalysis

Following Kim [2020]; Kim et al. [2019], we can analytically derive the eigensystem of our I_5-based strain-limiting energy's Hessian matrix with respect to the deformation gradient $\mathbf{F}$. For simplicity, we will omit λ' and a_t in the following discussion.

The eigenvalues of I_5-based energies are:

$$e_1(I_5) = 4I_5 \frac{\partial^2 \Psi}{\partial {I_5}^2} + 2\frac{\partial \Psi}{\partial I_5},$$
$$e_{2,3}(I_5) = 2\frac{\partial \Psi}{\partial I_5}. \tag{12}$$

Since $\mathbf{n}_u$ and $\mathbf{n}_v$ are orthogonal, we can substitute $I_5(\mathbf{F}, \mathbf{n}_u)$ and $I_5(\mathbf{F}, \mathbf{n}_v)$ into Equation 12 separately to obtain all six eigenvalues for our Ψ_{SL} when the cloth is stretched ($I_5 > 1$):

$$e_1(I_5(\mathbf{F}, \mathbf{n}_u)) = 6(\sqrt{I_5(\mathbf{F}, \mathbf{n}_u)} - 1), \tag{13}$$

$$e_{2,3}(I_5(\mathbf{F}, \mathbf{n}_u)) = 3\left(\frac{1}{\sqrt{I_5(\mathbf{F}, \mathbf{n}_u)}} + \sqrt{I_5(\mathbf{F}, \mathbf{n}_u)} - 2\right), \tag{14}$$

$$e_4(I_5(\mathbf{F}, \mathbf{n}_v)) = 6(\sqrt{I_5(\mathbf{F}, \mathbf{n}_v)} - 1), \tag{15}$$

$$e_{5,6}(I_5(\mathbf{F}, \mathbf{n}_v)) = 3\left(\frac{1}{\sqrt{I_5(\mathbf{F}, \mathbf{n}_v)}} + \sqrt{I_5(\mathbf{F}, \mathbf{n}_v)} - 2\right). \tag{16}$$

From these equations, we see that all eigenvalues are positive when $I_5 > 1$, indicating that our cubic strain-limiting energy Ψ_{SL} is convex with respect to $\mathbf{F}$ (recall that $\Psi_{SL} = 0$ when $I_5 \le 1$). Since $\mathbf{F}$ is a linear function of the degrees of freedom x, Ψ_{SL} is also convex with respect to x.

Thus, we can directly compute the PSD Hessian matrix of Ψ_{SL} in the $\mathbf{n}_u$ and $\mathbf{n}_v$ directions separately, applying the chain rule with I_5 as the intermediate variable:

$$\mathbf{H}_{SL,*} = 3S(I_5(\mathbf{F}, \mathbf{n}_*))\left(1 - \frac{1}{\sqrt{I_5(\mathbf{F}, \mathbf{n}_*)}}\right)(\sqrt{I_5(\mathbf{F}, \mathbf{n}_*)} - 1)\mathbf{H}_* + \frac{3S(I_5(\mathbf{F}, \mathbf{n}_*))(I_5(\mathbf{F}, \mathbf{n}_*) - 1)}{I_5(\mathbf{F}, \mathbf{n}_*)^{3/2}}(\mathbf{f}_*\mathbf{f}_*^T). \tag{17}$$

Here, $\mathbf{f}_* = \mathrm{vec}(\mathbf{F}\mathbf{L}_*)$, $\mathbf{L}_* = \mathbf{n}_*(\mathbf{n}_*)^T$, $\mathbf{H}_* = \mathbf{L}_* \otimes \mathbf{1}_{3\times 3}$, $\otimes$ denotes the tensor product, $\mathbf{I}_{3\times 3}$ is the 3×3 identity matrix, and $\mathrm{vec}(\cdot)$ is the vectorization operator [Kim and Eberle 2022], with $*$ representing either u or v. All of these computations can be efficiently parallelized on the GPU, unlike numerical eigendecomposition.

6 FAST GLOBAL HESSIAN OPERATIONS ON THE GPU

In this section, we introduce our multi-layer reduction strategy for fast Hessian matrix assembly in affine-deformable coupling simulations (subsection 6.1). Using a custom hash-based parallel reduction algorithm (subsection 6.2), our strategy significantly reduces the number of numerical operations and write conflicts, and it enables the development of a memory-efficient symmetric sparse block-wise matrix-vector multiplication method (subsection 6.3) to further boost PCG performance.

6.1 Multi-Layer Reduction for ABD Hessian Assembly

As previously mentioned, in affine body simulation, the main bottleneck lies in transforming and accumulating the contact Hessian matrices (collision and friction) from mesh node DOFs to the affine body DOFs within the global Hessian matrix. For contact pairs involving nodes belonging to affine bodies, consider the example of contact between two affine bodies. In this case, each 3×3 block $\mathbf{C}(r, c)$ in the contact Hessian, where r and

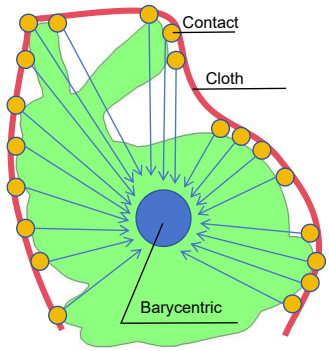


Fig. 9. Example of accumulating contact information for reduced affine bodies.

c are node indices, must be
mapped to a 12×12 matrix for the affine body pair $\{AF(r), AF(c)\}$ via multiplication with the Jacobian matrices $\{\mathbf{J}_r, \mathbf{J}_c\}$. All resulting 12×12 Hessian matrices for each affine body pair are then accumulated [Chen et al. 2022; Lan et al. 2022a].

However, the number of mapped Hessians typically far exceeds the number of affine body pairs (see Figure 9), leading to significant accumulation overhead, let alone the frequent write conflicts. To address this issue, we first accumulate all 3×3 contact Hessian blocks with the same node index pair $\{r, c\}$ through reduction, then apply a single product with $\{\mathbf{J}_r, \mathbf{J}_c\}$ to transform it into a 12×12 matrix. This process yields a sequence of 12×12 matrices for the blocks of the global Hessian corresponding to the affine body pair $\{AF(r), AF(c)\}$. A second layer of reduction is then used to accumulate these 12×12 Hessians in parallel, significantly reducing write conflicts and the number of values to be accumulated, resulting in faster performance.

6.2 Hash-based Parallel Reduction

To further enhance accumulation efficiency, we developed a parallel reduction technique using a hashing method based on either the node index pair $\{r, c\}$ or the affine body index pair $\{AF(r), AF(c)\}$. Each matrix block is encoded with a 64-bit hash value, where the first index forms the higher 32 bits and the second forms the lower 32 bits. Sorting the hash values enables consecutive memory placement of matrix blocks sharing the same hash [Huang et al. 2019, 2020], facilitating efficient parallel accumulation. The key insight here is to build a custom reduction scheme that is based on registers rather than shared memory and avoid global synchronization by taking advantage of warp-level operations [Gao et al. 2018; Huang et al. 2024; Zhao et al. 2020].

Our algorithm is constructed using basic parallel primitives:

(1) Device run-length encoding
(2) Device exclusive sum
(3) Warp segmented reduction

The first two primitives are standard in parallel computing. For warp-level segmented reduction, we use the CUB intrinsic function ***cub::WarpReduce(values, tags)***. Consider a simple example with a warp size of 8: if ***values*** = $[1, 1, 1, 1, 1, 1, 1, 1]$ and ***tags*** = $[1, 0, 0, 1, 0, 0, 1, 0]$, the output will be ***outPut*** = $[3, \bullet, \bullet, 3, \bullet, \bullet, 2, \bullet]$, where • indicates invalid entries. Here, the ***tags*** array specifies start and end positions for reduction in ***values***, with 1 marking the start. Reduction results (valid entries in ***outPut***) are then written to the result array. The overall procedure, *FastHashReduction*, is detailed in Algorithm 2. First, the map array is constructed in parallel, mapping duplicate keys to their unique key indices to ensure reduction results are written to the correct locations (lines 1–8). The *FastSegmentReduction* algorithm (Algorithm 3) then efficiently computes the matrix.

FastHashReduction. Our FastHashReduction algorithm is designed to handle various value types. For ABD Hessian matrices, each value is a 12×12 matrix, while for FEM Hessian matrices, each value is a 3×3 matrix. In a hybrid FEM-ABD model, there are four types of Hessian matrix blocks: ABD-ABD (12×12), FEM-FEM (3×3), ABD-FEM (12×3), and FEM-ABD (3×12). To unify these four block matrix types, we decompose them into multiple 3×3 sub-blocks, resulting in:

(1) 16 hash keys and 16 sub-blocks of 3×3 matrices for ABD-ABD Hessian matrix blocks.
(2) 4 hash keys and 4 sub-blocks of 3×3 matrices for ABD-FEM and FEM-ABD Hessian matrix blocks.

The FEM-FEM Hessian matrix blocks remain unchanged. This approach allows us to launch a single, unified CUDA kernel to reduce all types of local Hessians, maximizing parallelism and minimizing branching.

Algorithm 2 FastHashReduction

Input:
K ▷ an array of sorted hash keys
V ▷ an array of values, sorted by the hash keys
Output:
R ▷ an array of reduced values
Method:
1: $UK, NK \leftarrow$ RunLengthEncode(K) ▷ get unique keys (UK) and their counts (NK)
2: $Num \leftarrow$ len(UK)
3: $ESum \leftarrow$ ExclusiveSum(NK) ▷ get prefix sum, starting from 0
// assign the unique key index to O for each key in K:
4: $P \leftarrow$ Zeros(len(K))
5: **for** $tid = 0, 1, ..., Num - 1$ **do in parallel**
6: $P[ESum[tid] + NK[tid] - 1] \leftarrow 1$
7: **end for**
8: $O \leftarrow$ ExclusiveSum(P) ▷ map each K to unique key
9: $R \leftarrow$ FastSegmentReduction(O,V)

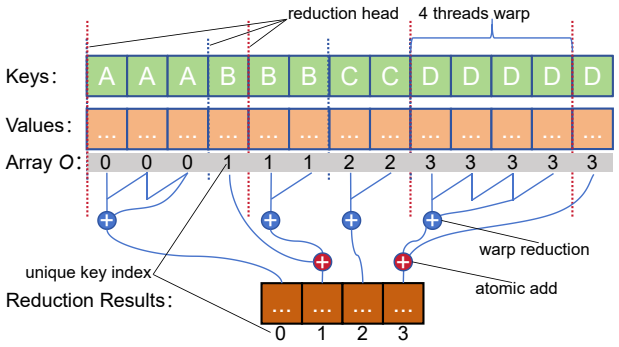


Fig. 10. **Illustration of FastSegmentReduction.** The warp-level reduction is performed within segments, using Array O to identify segment boundaries. For segments spanning multiple warps (B and D), atomic operations are applied to avoid data conflicts, while direct writing is used otherwise.

FastSegmentReduction. In Algorithm 2, we create a mapping array O to associate each key-value pair with its target memory location, as illustrated in Figure 10. Array O is also used to generate tag information for the CUB intrinsic function ***cub::WarpReduce(values, tags)***. The ***tags*** are determined by comparing adjacent entries in O, as described in Algorithm 3. Here, b^-, b, and b^+ represent the

previous, current, and next unique key indices, respectively. The ***IsValid*** flag checks whether the GlobalThreadId falls within valid data range. If $b^- \neq b$, it marks the start of a new segment. The ***IsCrossWarp*** flag indicates if the segment spans multiple warps, necessitating atomic operations to avoid data corruption; otherwise, direct writing is safe. These flags are set per thread, but only the leading thread (head thread) can write to the output array. Therefore, we merge flags across threads in each segment: a head thread is valid if any thread in the segment is valid and marked as cross-warp if any spans multiple warps. This is achieved via segmented warp reduction, using an OR operation for merging (see lines 5-28).

This hash key reduction method efficiently constructs the global Hessian matrix, critical for the subsequent linear system solve and preconditioner computation. Additionally, our Hessian assembly method ensures that 3×3 sub-blocks in each row are sorted in the memory by column index, enabling an efficient SpMV algorithm with warp reduction to enhance PCG solver performance.

6.3 Symmetric Reduce-By-Key (SRBK) SpMV

Sparse matrix-vector multiply (SpMV) is a frequently called subroutine in scientific computing, where a conjugate gradient solver, for example, may invoke hundreds or thousands of SpMV operations for each linear solve. Thus, speeding up SpMV is critical for improving the overall performance. The NVIDIA cuSparse library's SpMV on Block Sparse Row (BSR) format is the fastest for general sparse matrices, but it lacks support for symmetric matrices. Since the SpMV algorithm is memory-bound, we aim to leverage symmetry to reduce memory access and achieve significant speedup.

We propose a symmetric Reduce-By-Key (SRBK) SpMV method (Algorithm 4) that is approximately 2× faster than cuSparse BSR. Our algorithm works on a format called Sorted Symmetric Block Coordinates, storing only the diagonal and upper triangular blocks of the matrix. All block row and column indices are sorted, a free lunch from our FastHashReduction algorithm for assembling the matrix.

Our SRBK SpMV and FastSegmentReduction algorithms have a similar structure, both using a "Reduction By Key" parallel primitive. Sorted row indices serve as hash keys, while the product of the block matrix and the input vector represents the value to be reduced (lines 5-9). The main difference is in how we access the lower triangular part of the matrix, where we atomically add the multiplication results to the output vector (line 11). The data from the lower triangular part is directly transposed from the upper part, reducing global memory access by nearly half, and thus significantly enhancing performance.

Algorithm 3 FastSegmentReduction

Input:
O ▷ The result in line 8 of Algorithm 2
V ▷ An array of values, sorted by the hash keys
Output:
R ▷ An array of reduced values, one value for each hash key
Method:

```
1:  define BD = BLOCK_DIM
2:  define WD = WARP_DIM
3:  BlockCount ← (len(V) + BD - 1) / BD
4:  for GlobalThreadId = 0, 1, ..., BlockCount×BD-1 do in parallel
        // get tags for cub::WarpReduce(values, tags):
5:      LaneId ← GlobalThreadId % WD
6:      b⁻, b, b⁺ ← −1                          ▷ −1 for invalid value
7:      IsValid, IsHead, IsCrossWarp ← 0
8:      Value ← 0
9:      if 0 < GlobalThreadId < len(V) then
10:         b⁻ ← O[GlobalThreadId −1]
11:     end if
12:     if GlobalThreadId < len(V) - 1 then
13:         b⁺ ← O[GlobalThreadId +1]
14:     end if
15:     if GlobalThreadId < len(V) then
16:         b ← O[GlobalThreadId]
17:         Value ← V[GlobalThreadId]
18:         IsValid ← 1
19:     end if
20:     if LaneId == 0 then
21:         IsHead ← 1, IsCrossWarp ← b⁻ == b
22:     else
23:         IsHead ← b⁻ ≠ b
24:         if LaneId = WARP_DIM −1 then
25:             IsCrossWarp ← b⁺ == b
26:         end if
27:     end if
28:     Tuple ← (IsValid, IsCrossWarp)
29:     Tuple ← cub::WarpReduce(Tuple, IsHead)
30:     Value ← cub::WarpReduce(Value, IsHead)
        // write the result to the corresponding memory location:
31:     if IsHead and IsValid then
32:         if IsCrossWarp then
33:             AtomicAdd(R[b], Value)
34:         else
35:             R[b] ← Value
36:         end if
37:     end if
38: end for
```

7 EXPERIMENTS AND ANALYSIS

We present our evaluation and results in this section. The experiments were conducted with a 24-core Intel Core i9 13900KF CPU with 64GB of RAM, and an NVIDIA RTX 4090 GPU with 24GB of RAM. For the simulations, we use modified PCG, with a relative error tolerance of 10^{-4}, which has shown to be sufficient for IPC [Huang et al. 2024].

Our experiments include ablation and comparative studies on preconditioners, strain-limiting energies, SpMV, and GPU IPC methods (subsection 7.1), followed by stress tests (subsection 7.2). The simulation statistics are presented in Table 1, Table 2, and Table 3. Regarding alternative GPU IPC methods, we compared our system against GIPC [Huang et al. 2024], using the same linear solver and

Algorithm 4 Symmetric Reduce-By-Key (SRBK) SpMV

Input:
Rid, **Cid**, **AU** ▷ block row index, block column index, and value of upper-triangular and diagonal blocks
$\mathbf{V}_{input}$ ▷ SpMV input vector
Output:
$\mathbf{V}_{output}$ ▷ SpMV output vector
Method:
1: **define** BD = BLOCK_DIM
2: **define** WD = WARP_DIM
3: BlockCount ← (len(**AU**) + BD - 1) / BD
4: **for** GlobalThreadId = 0, 1, ..., BlockCount×BD−1 **do in parallel**
5: Run line 5-29 in Algorithm 3, replacing O with **Rid**
6: **if** IsValid **then** ▷ GlobalThreadId is within valid data range
7: j ← **Cid**[GlobalThreadId]
8: **H** ← **AU**[GlobalThreadId]
9: Value ← $\mathbf{H} \cdot \mathbf{V}_{input}[j]$ ▷ multiply with upper-triangular and diagonal blocks
10: **if** $b \neq j$ **then** ▷ If not diagonal
11: AtomicAdd($\mathbf{V}_{output}[j]$, $\mathbf{H}^T \cdot \mathbf{V}_{input}[b]$) ▷ multiply with lower-triangular blocks
12: **end if**
13: **end if**
14: Run line 28-37 in Algorithm 3, replacing R with $\mathbf{V}_{output}$
15: **end for**

termination criteria as ours, but with hybrid block-diagonal or GPU MAS [Wu et al. 2022] preconditioner. Additionally, we compared our system to the GPU-based ABD method [Lan et al. 2022a], where only CHOLMOD compiled with Intel MKL LAPACK and BLAS is running on the CPU. Finally, we conducted a comparison with the state-of-the-art affine-deformable coupling IPC framework, ZeMa [Du et al. 2024], which also uses the CPU-based CHOLMOD solver, while leveraging GPU acceleration for other components. All simulations were performed using double-precision floating-point arithmetic.

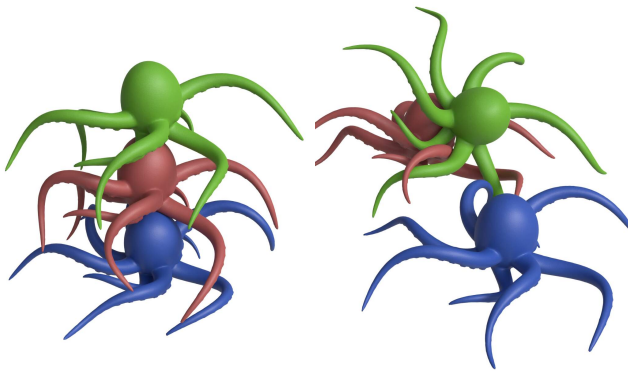


Fig. 11. **Octopus stack.** The simulation uses $\hat{d} = 10^{-3}l$, $\rho = 1000\,\mathrm{kg/m^3}$, $\Delta t = 5 \times 10^{-3}$ s, and a Newton tolerance of $10^{-2}l\Delta t$. This test case validates the performance improvement of our connectivity-enhanced MAS preconditioner, which achieves a 1.8× speedup in linear solve.

7.1 Ablation and Comparative Studies

Preconditioner. To evaluate the performance of our connectivity-enhanced MAS preconditioner, we construct several scenes with different geometric structures (see Figure 11, Figure 13, Figure 19, and Figure 20). The timing breakdown and CG iteration counts are reported in Table 1. For a fair comparison, we implemented alternative preconditioners into our simulation framework, ensuring that all other components remain the same. The block-diagonal preconditioner was also tested as a reference. Our results show that, with more effective aggregation, our method using a subdomain size of 32 provides the most significant improvement in PCG convergence compared to MAS [Wu et al. 2022] (see avg. #cg column in Table 1). Even with smaller subdomain sizes (16), our method converges faster than MAS and performs the best overall, achieving up to 3× speedup in the PCG solve and up to 2.6× faster for the full simulation, thanks to more efficient precomputation and preconditioning.

As seen from the similar average CG iterations across resolutions (the rows for Figure 11 in Table 1), our preconditioner maintains the scalability of MAS, resulting in near-linear total timing with respect to resolution. In Figure 13, the 'Ours (16)' configuration shows only a 1.17× speedup in convergence, as the volumetric structure of the bunnies results in similar METIS and Morton code reorderings, limiting improvement. However, we still observe a 1.58× speedup in PCG time and a 1.47× overall speedup, due to the lower computational cost of our preconditioner compared to MAS.

Strain limiting. Figure 12 presents a cloth simulation example using FBW [Kim 2020] with and without barrier-based [Li et al. 2021] or our cubic strain-limiting energy. The cloth consists of 100K nodes and is tested with various membrane stiffnesses. In cloth simulation, avoiding excessive stretching is crucial, as it can cause unrealistic, overly-elastic behavior. To prevent this, a large Young's modulus (e.g., $5MPa$), within the range of real cloth parameters [Penava et al. 2014], is needed. However, such stiffness can lead to membrane locking in even medium-resolution meshes, resulting in sharp creases and artificial plastic appearances (Figure 12a). Reducing the stiffness to $0.5MPa$ mitigates membrane locking but visible artifacts remain (Figure 12b). Further reducing the stiffness to $50KPa$ smooths wrinkles but causes over-elongation (Figure 12c). Our cubic strain-limiting energy provides sufficient stretching resistance as in real fabrics, avoiding excessive stretching and capturing smooth, realistic wrinkles with a small membrane stiffness of $50KPa$ for compression (Figure 12e, and supplemental video).

We also compare with the barrier-based strain-limiting method in CIPC using their default settings: a 1.1 strain limit and 1 activation threshold. We use a $1KPa$ stiffness for the barrier term, as smaller or larger values both lead to worse conditioned systems. With similar result quality (Figure 12d), our method is 1.24× faster (Table 2), with lower costs in local Hessian construction and line search (see Table 3 for a more detailed breakdown). While our method performs well for practical cloth manipulation scenarios, it does not guarantee strain limit satisfaction in extreme cases like the barrier-based method. Thus, we further compare their behavior in a stress test (Figure 22), detailed later.

Table 1. **Simulation statistics with different preconditioners.** Mesh DOFs: number of vertices (v), tetrahedra (t), and surface triangles (f). PPC, PPR, and CGR represent the time spent on preconditioner pre-computation, preconditioning, and CG iterations, respectively, with PCG = PPC + PPR + CGR. The remaining columns report the time for miscellaneous tasks (misc), total simulation time (timeTot), total CG iterations (#cg), total Newton iterations (#Newton), and the average CG iterations per Newton iteration (avg. #cg). All times are in seconds.

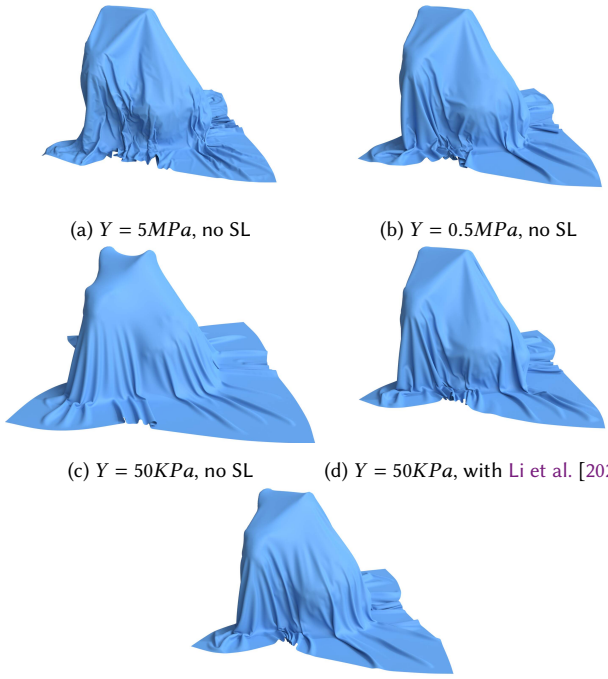
Example	Mesh DOFs: v, t, f	Preconditioner	PPC	PPR	CGR	PCG	Misc	TimeTot	#cg	#Newton	avg. #cg
Figure 11	$42K$, $171K$, $45K$	GPU MAS	38	55	102	195	26	221	6.98e5	2.03e3	344
		Block Diagonal	**0.06**	21	338	359	27	385	2.24e6	2.07e3	1082
		Ours (32)	32	**25**	**60**	117	26	143	**3.45e5**	2.02e3	**171**
		Ours (16)	**10**	29	69	**108**	26	**134**	4.41e5	1.99e3	221
		Speed Up	3.80×	1.90×	-	1.81×	-	1.65×	-	-	1.51×
Figure 11	$114K$, $558K$, $69K$	GPU MAS	105	162	257	524	69	593	9.51e5	2.31e3	412
		Block Diagonal	**0.16**	**43**	582	625	69	694	2.48e6	2.30e3	1079
		Ours (32)	96	79	**127**	302	69	371	**4.37e5**	2.31e3	**189**
		Ours (16)	33	60	145	**238**	69	**307**	5.20e5	2.31e3	225
		Speed Up	3.18×	2.7×	-	2.20×	-	1.93×	-	-	1.84×
Figure 13	$203K$, $1122K$, $50K$	GPU MAS	152	188	229	569	87	656	6.23e5	1.76e3	354
		Block Diagonal	**0.18**	**63**	918	981	88	1069	2.61e6	1.77e3	1474
		Ours (32)	134	137	**180**	451	87	538	**4.54e5**	1.76e3	**258**
		Ours (16)	42	108	210	**360**	87	**447**	5.30e5	1.76e3	301
		Speed Up	3.61×	1.74×	-	1.58×	-	1.47×	-	-	1.17×
Figure 19	$123K$, $330K$, $211K$	GPU MAS	371	836	1029	2236	217	2453	4.38e6	7.93e3	552
		Block Diagonal	**0.44**	**137**	1920	2057	216	2273	8.84e6	7.90e3	1119
		Ours (32)	354	196	**298**	848	215	1063	**9.98e5**	7.86e3	**127**
		Ours (16)	76	215	439	**730**	215	**945**	1.81e6	7.85e3	231
		Speed Up	4.88×	3.89×	-	3.06×	-	2.60×	-	-	2.39×
Figure 20	$79K$, $298K$, $110K$	GPU MAS	86	193	274	553	42	595	1.42e6	1.62e3	877
		Block Diagonal	**0.08**	**57**	798	855	43	898	5.09e6	1.64e3	3105
		Ours (32)	47	89	**124**	260	42	302	**6.84e5**	1.63e3	**420**
		Ours (16)	12	68	136	**216**	41	**257**	7.53e5	1.62e3	465
		Speed Up	7.17×	2.83×	-	2.56×	-	2.31×	-	-	1.89×

Table 2. **Simulation statistics for Figure 12.** Columns show total time for linear solve (PCG) and miscellaneous tasks (Misc), total simulation time (TimeTot), total PCG iterations (#cg), total Newton iterations (#Newton), and average PCG iterations per Newton iteration (#cg per iter). All simulations use the same PCG relative tolerance of 10^{-4} with our connectivity-enhanced MAS preconditioner.

Young's Modulus (Pa)	PCG	Misc	TimeTot	#cg	#Newton	avg. #cg per iter
$5MPa$, no SL (Figure 12a)	2.91e2	1.34e2	4.25e2	7.07e5	4.03e3	175
$0.5MPa$, no SL (Figure 12b)	2.31e2	1.61e2	3.92e2	4.12e5	4.49e3	92
$50KPa$, no SL (Figure 12c)	1.26e2	1.34e2	2.60e2	1.66e5	3.63e3	46
$50KPa$ + CIPC SL (Figure 12d)	1.55e2	2.02e2	3.57e2	2.24e5	3.99e3	56
$50KPa$ + our SL (Figure 12e)	1.45e2	1.43e2	2.88e2	2.06e5	3.76e3	55

Global Hessian assembly. As illustrated in Figure 14, we use the wrecking ball simulation (Figure 18) to conduct an ablation study on our multi-layer reduction strategy (Strategy C) for global Hessian matrix assembly, comparing it against single-layer reduction (Strategy B) and pure atomic operations (Strategy A). In both Strategies A and B, the local Hessian matrices of all contact pairs are first mapped to the affine body DOFs by multiplying the Jacobian matrices. Strategy B then performs a reduction to accumulate these matrices into the global Hessian, while Strategy A uses atomic operations directly for accumulation. To ensure a fair comparison, we also sort the value array prior to atomic accumulation. This step optimizes the distribution of the value array, enabling aggregation optimization in the CUDA intrinsic function ***atomicAdd***, thereby achieving the best possible performance. Even with this optimization, pure atomic operations remain slightly slower than single-layer reduction, as shown in the comparison of Strategies A and B in Figure 14. Our multi-layer reduction strategy achieves an additional nearly 5× speedup over single-layer reduction.

Reduce by Key. We compare our *FastHashReduction* method with the CUB intrinsic function ***cub::DeviceReduce::ReduceByKey*** and plot our speedup relative to the number of values being accumulated (Figure 15). To keep the total usage of GPU memory unchanged, the reduction is performed on 16M 3×3 matrices, and we adjust the number of matrices with the same row or column indices to control the workload. Our method shows superior performance across a wide range of accumulation workload, from 1 to 2^{16}, achieving a maximum speedup of approximately 1.55× at 2^{11}. In FEM simulations, the average number of values with the same row or column indices typically remains below 2^7, where our method achieves a consistent speedup of 1.2 to 1.4×. Additionally, we observe that ***cub::DeviceReduce::ReduceByKey*** fails to compile when reducing the 12×12 affine body Hessian matrix due to shared memory limitations. In contrast, our *FastHashReduction* method remains effective by reusing warp registers, leveraging the element-independent nature of matrix summation.



(a) $Y = 5MPa$, no SL (b) $Y = 0.5MPa$, no SL

(c) $Y = 50KPa$, no SL (d) $Y = 50KPa$, with Li et al. [2021]

(e) $Y = 50KPa$, with our SL

Fig. 12. **Membrane locking and strain limiting.** The cloth consists of $100K$ vertices and $200K$ triangles, simulated with FEM, while the bunny is simulated using ABD. The simulation parameters are: $\Delta t = 0.01$ s, Poisson's ratio $\nu = 0.49$, density $\rho = 200\,\mathrm{kg/m^3}$, cloth thickness 10^{-3} m, friction coefficient $\mu = 0.1$, Newton tolerance $\varepsilon_d = 10^{-2} l\Delta t$, membrane Young's modulus $Y = \{5\times10^6, 5\times10^5, 5\times10^4\}$ Pa, and distance threshold $\hat{d} = 10^{-3} l$, where l is the scene diagonal length. Li et al. [2021] (d) and our method (e) simulate fine wrinkling details without membrane locking (a, b) or over-elongation (c), with our method being 1.24× faster (Table 2).

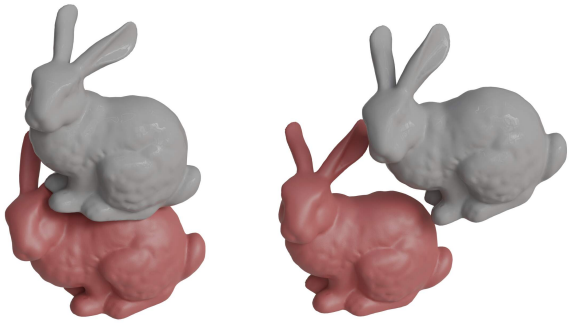


Fig. 13. **Stiff and soft bunnies.** The top bunny is nearly rigid (Young's modulus 10^8 Pa), and the bottom red one is deformable (Young's modulus 10^6 Pa). Each bunny consists of $101K$ vertices and $561K$ tetrahedra. The simulation uses $\hat{d} = 5\times10^{-4} l$, $\rho = 1000\,\mathrm{kg/m^3}$, $\Delta t = 5\times10^{-3}$ s, and a Newton tolerance of $10^{-2} l\Delta t$. This test compares GIPC, ZeMa, and our method in hybrid scenarios. Our method shows a 4× speedup when simulating both bunnies using FEM and a 10× speedup when using ABD+FEM.

SpMV. We compare the performance of different SpMV methods using the stiff and soft bunnies example shown in Figure 13, simulating both bunnies with FEM. As shown in Figure 16, the BSR SpMV (1×) from cuSparse[1] is treated as a baseline. Without matrix sorting, the Triplet SpMV (0.16×) implemented by atomically adding the matrix-vector products and the MatrixFree SpMV (0.18×) from GIPC both suffer from an *atomic operation flood*, which undermines the parallelism. On the other hand, sorting-based methods have a significant speedup, benefiting from the reduction of atomic operations. The traditional CSR SpMV (0.75×) from cuSparse has an almost 5× speedup compared to the unsorted methods. The Block Coordinates (BCOO) SpMV from MUDA[2] and the BSR SpMV from cuSparse further improves the performance by applying blockwise matrix-vector multiplication. Our SRBK SpMV method (1.85×) further mitigates the memory-bound issue of BSR SpMV and boosts the performance by almost 2×.

To evaluate the scalability of our method, we compare its speedup factor to BSR on matrices with varying numbers of non-zero 3×3 blocks per row. As shown in Figure 17, the speedup factor increases from 0.93× to 1.9× as the number of non-zero blocks approaches 2^3 from 0. For average non-zero block counts between 2^3 and 2^6 – the typical range for FEM global systems – the speedup factor fluctuates between 1.8× and 2×, which demonstrates strong scalability.

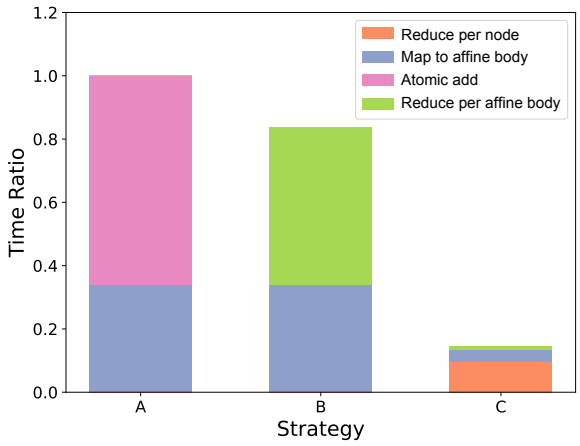


Fig. 14. **Ablation study on multi-layer reduction for Hessian assembly.** We compare our multi-layer reduction strategy (C) against single-layer reduction (B) and pure atomic operations (A) in the wrecking ball simulation (Figure 18). Despite sorting the value array for optimal ***atomicAdd*** performance, using atomic operations is slightly slower than single-layer reduction, and ours achieves nearly 5× speedup over Strategy B.

Comparison with GPU ABD. To evaluate the efficiency of our framework in simulating stiff affine bodies, we replicate the wrecking ball example (Figure 18) from ABD [Lan et al. 2022a]. In our setup, the sphere and the boxes are assigned the same density, ensuring that the number of contacts remains large as less boxes will

[1] https://developer.nvidia.com/cusparse
[2] https://github.com/MuGdxy/muda

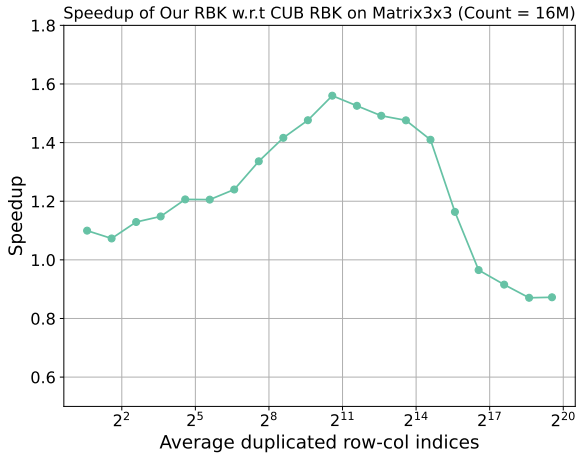


Fig. 15. **Speedup of FastHashReduction vs. CUB ReduceByKey.** Speedup factor is shown relative to the average number of matrices sharing the same row or column indices. A peak speedup of 1.55× occurs around 2^{11}. For typical duplication levels in FEM simulations (up to 2^7), FastHashReduction maintains a consistent speedup between 1.2× and 1.4×.

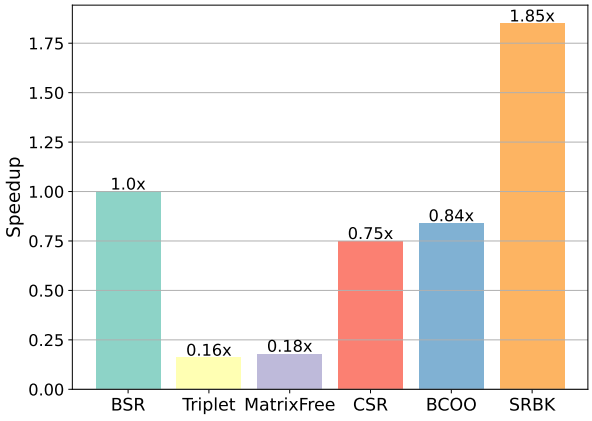


Fig. 16. **Timing comparison of SpMV methods.** The BSR method from cuSparse serves as the baseline. Our method (SRBK) achieves a maximum speedup of 1.85× over the baseline.

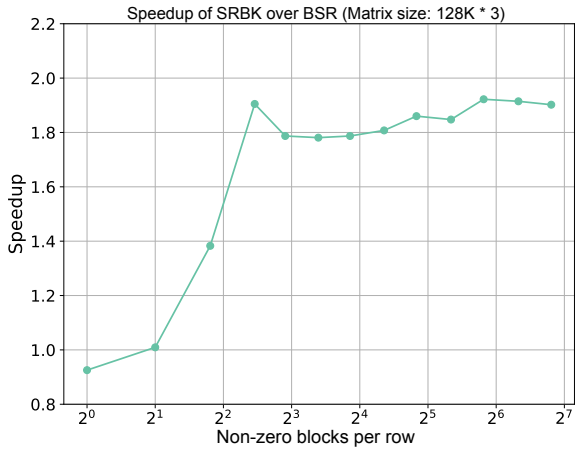


Fig. 17. **Scalability of our SRBK SpMV.** Using the BSR from cuSparse as the baseline, we compare our SpMV's speedup over BSR on matrices with different numbers of non-zero blocks per row. The speedup increases from 0.93× to 1.9× as non-zero blocks per row approach 2^3. For typical FEM ranges (2^3–2^6 blocks), speedup stabilizes between 1.8× and 2×, demonstrating our strong scalability.

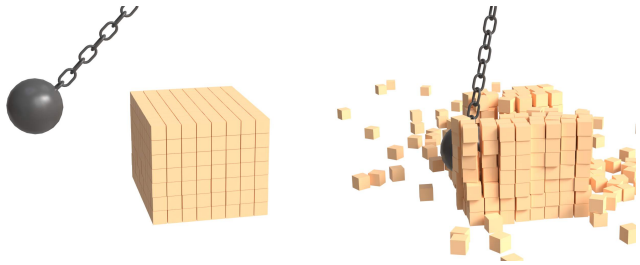


Fig. 18. **Wrecking Ball.** This example compares GPU ABD [Lan et al. 2022a] and our method using only stiff affine bodies. The simulation parameters are $\hat{d} = 10^{-3}l$, $\rho = 1000$ kg/m^3, $\Delta t = 0.01$ s, $\mu = 0.05$, static friction tolerance $10^{-3}l$, and Newton tolerance $10^{-2}l\Delta t$. Our method achieves a 2.6× speedup.

be punched out. This configuration makes the problem more challenging for the simulation framework. Our method utilizes PCG to solve the linear systems, using the 12×12 diagonal blocks as the preconditioner. In this scenario, ABD requires 37.2 ms per Newton iteration, whereas our method only takes 14.4 ms, resulting in a 2.6× speedup.

Comparison with GIPC and ZeMa on hybrid scenarios. Here, we compare our method with GIPC [Huang et al. 2024] and ZeMa [Du et al. 2024] on the stiff and soft bunnies example (Figure 13). In this simulation, the Young's modulus is set to 10^6 Pa for the bottom bunny and 10^8 Pa for the top bunny. Our framework can treat the top bunny as a stiff affine body and the bottom bunny with FEM, while GIPC needs to simulate both using FEM. Results in Table 3 indicate that our coupling framework achieves a 10× overall speedup compared to GIPC. To analyze the sources of this acceleration, we applied our contributions incrementally. 'Ours (SRBK)' uses only our SRBK method, matching other components to GIPC, which already yields a 2.43× speedup with more efficient Hessian assembly and SpMV. 'Ours (CEMAS+SRBK)' further adds connectivity-enhanced (CE) MAS preconditioner to 'Ours (SRBK)', retaining other components, and shows an additional 1.47× overall speedup due to faster PCG convergence enabled by CEMAS, resulting in a 3.57× improvement compared to GIPC. Note that in both 'Ours (SRBK)' and 'Ours (CEMAS+SRBK)' cases, we simulate both bunnies using FEM. ZeMa performs the worst in this case, as direct solvers are not efficient for simulations with a large number of DOFs. Thus, for general IPC simulations, GIPC remains the best alternative. Therefore, in subsequent experiments, we compare our methods only against GIPC.

We further compare our method with GIPC on the examples shown in Figure 11, Figure 12, Figure 19, Figure 20, Figure 22, and

Table 3. **Simulation statistics for all examples.** Columns show time for energy gradient and Hessian computation (Hess), linear system solves (LSolve), CCD (CCD), backtracking line search (LineS), miscellaneous tasks (Misc), and the entire simulation (TimeTot), followed by iteration counts, and finally, speedup relative to GIPC. All times are in seconds. Here, we compare our method with GIPC [Huang et al. 2024], ZeMa [Du et al. 2024], and the barrier-based SL [Li et al. 2021]. We also show how each of our innovation contributes to the overall speedup.

	Mesh DOFs: v, t, f	Framework	Hess	LSolver	CCD	LineS	Misc	TimeTot	#cg	#Newton	avg. #cg	**Speed Up**
Figure 11	114K, 558K, 69K	GIPC	63	1134	5	13	6	1221	9.41e5	2.30e3	409	1.00×
	114K, 558K, 69K	Ours (SRBK)	45	524	5	13	6	593	9.51e5	2.31e3	412	2.06×
	114K, 558K, 69K	Ours (CEMAS+SRBK)	45	238	5	13	6	**307**	5.20e5	2.31e3	**225**	3.98×
Figure 12	109K, 80K, 200K	GIPC	106	606	20	54	6	792	2.34e5	3.89e3	60	1.00×
	109K, 80K, 200K	Ours (SRBK)	75	489	20	55	6	645	2.37e5	3.86e3	61	1.23×
	109K, 80K, 200K	Ours (CEMAS+SRBK)	75	184	20	55	6	340	2.61e5	3.95e3	66	2.33×
	109K, 80K, 200K	CIPC (CEMAS+SRBK)	104	202	21	91	6	424	2.89e5	4.20e3	69	-
	100K, 0, 200K	Ours (ABD+CEMAS+SRBK)	67	145	19	51	6	**288**	2.06e5	3.76e3	**55**	2.75×
Figure 13	203K, 1122K, 50K	GIPC	90	1481	4	14	7	1596	6.23e5	1.76e3	354	1.00×
	114K, 561K, 50K	ZeMa	42	54810	2.7	9	7	54871	-	1.74e3	-	0.03×
	203K, 1122K, 50K	Ours (SRBK)	63	569	3	14	7	656	6.23e5	1.76e3	354	2.43×
	203K, 1122K, 50K	Ours (CEMAS+SRBK)	63	360	3	14	7	447	5.30e5	1.76e3	301	3.57×
	114K, 561K, 50K	Ours (ABD+CEMAS+SRBK)	37	104	2.7	8.5	7	**159**	1.73e5	1.73e3	**100**	10.03×
Figure 19	123K, 330K, 211K	GIPC	137	3910	35	83	8	4173	4.31e6	7.87e3	548	1.00×
	123K, 330K, 211K	Ours (SRBK)	91	2236	35	83	8	2453	4.38e6	7.93e3	552	1.70×
	123K, 330K, 211K	Ours (CEMAS+SRBK)	90	730	35	82	8	**945**	1.81e6	7.85e3	**231**	4.42×
Figure 20	79K, 298K, 110K	GIPC	24	1080	5	13	8	1130	1.42e6	1.62e3	877	1.00×
	79K, 298K, 110K	Ours (SRBK)	16	553	5	13	8	595	1.42e6	1.62e3	877	1.90×
	79K, 298K, 110K	Ours (CEMAS+SRBK)	16	216	5	12	8	257	7.53e5	1.62e3	465	4.40×
	78K, 290K, 110K	Ours (ABD+CEMAS+SRBK)	14	148	4	8	7	**181**	4.50e5	1.41e3	**319**	6.24×
Figure 21	70K, 241K, 106K	GIPC	692	10686	114	637	14	12143	7.30e6	1.58e4	462	1.00×
	70K, 241K, 106K	Ours (SRBK)	460	4511	113	594	14	5692	7.58e6	1.58e4	480	2.13×
	70K, 241K, 106K	Ours (CEMAS+SRBK)	452	1547	112	597	14	2722	3.37e6	1.57e4	214	4.46×
	53K, 133K, 106K	Ours (ABD+CEMAS+SRBK)	407	696	103	535	12	**1753**	1.48e6	1.46e4	**101**	6.93×
Figure 22	105K, 10K, 202K	GIPC	210	2473	54	211	9	2957	3.10e6	9.43e3	329	1.00×
	105K, 10K, 202K	Ours (SRBK)	142	762	51	205	9	1069	3.13e6	9.27e3	338	2.76×
	105K, 10K, 202K	Ours (CEMAS+SRBK)	145	451	52	207	9	864	8.21e5	9.50e3	**86**	3.42×
	105K, 10K, 202K	CIPC (CEMAS+SRBK)	219	812	59	329	10	1429	1.75e6	9.69e3	180	-
	105K, 0, 202K	Ours (ABD+CEMAS+SRBK)	161	468	47	178	9	**863**	7.78e5	8.51e3	91	3.42×

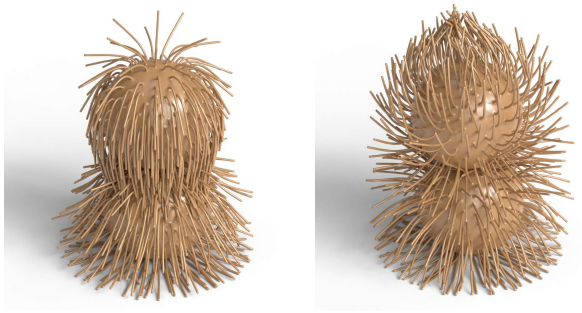


Fig. 19. **Furry ball.**This example compares GIPC [Huang et al. 2024] and our method. The simulation parameters are $\hat{d} = 4 \times 10^{-4} l$, $\rho = 1000$ kg/m^3, $Y = 1MPa$, $\nu = 0.49$, $\Delta t = 0.01$ s, and $\varepsilon_d = 10^{-2} l \Delta t$. Our method achieves a 4.42× speedup.

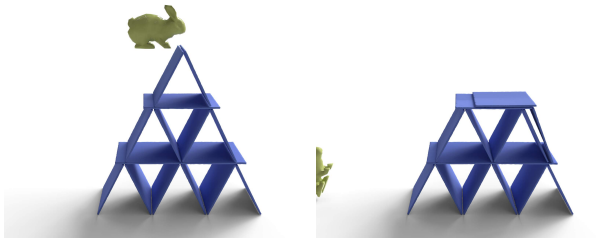


Fig. 20. **Board House.** This example compares GIPC [Huang et al. 2024] with our method. Simulation parameters are $\hat{d} = 1 \times 10^{-4} l$, $\rho = 1000\,\text{kg/m}^3$, $Y = 0.1$ GPa, $\nu = 0.49$, $\Delta t = 0.01$ s, and $\varepsilon_d = 10^{-2} l \Delta t$. Our method achieves a 4.4× speedup using FEM and 6.24× using ABD.

Figure 21, with statistics reported in Table 3. Generally, 'Ours (SRBK)' achieves a 1.23×–2.76× speedup over GIPC. Adding our CEMAS preconditioner further improves performance, providing an additional 1.24×–2.6× speedup as shown by 'Ours (CEMAS+SRBK)', leading to an overall improvement of 2.33×–4.46× compared to GIPC. Notably, this is achieved without approximating stiff materials as affine bodies, which can further boost performance up to 2.75×–10× faster than GIPC. In Figure 12, comparing 'Ours (SRBK)' and 'Ours (CEMAS+SRBK)' reveals that while CEMAS does not improve PCG convergence due to similar ordering with Morton code sorting in GIPC, it significantly reduces PCG solver time due to lower construction and preconditioning costs. Here, PCG costs represent a smaller portion of total timing than in other examples

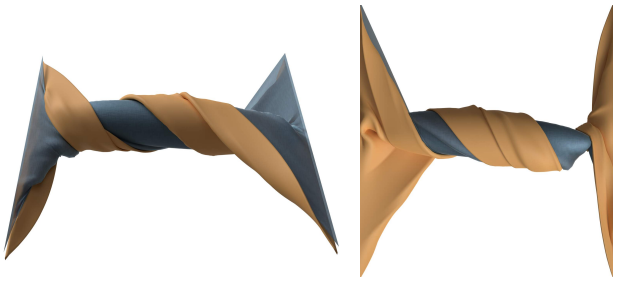


Fig. 21. **Mat-cloth twist.** Stress test on extreme deformation with cloth, volumetric mat, and a stiff affine ellipsoid (inside). Our method simulates this stiff challenging scenario efficiently with 1.8s per time step ($\Delta t = 0.01s$), achieving a 6.9× speedup compared to GIPC.

due to fewer PCG iterations required, resulting in a modest 1.23× speedup with SRBK alone. We will discuss Figure 22 and Figure 21 in detail in the next subsection on stress tests.

7.2 Stress Tests

Mat-cloth twist. To further validate the robustness and efficiency of our framework, we perform a simulation coupling volumetric solids, cloth, and stiff affine bodies, as shown in Figure 21. The Young's modulus of the cloth and the mat are $5MPa$ and $0.5MPa$ respectively. They both have a Poisson's ratio of 0.49 and the friction coefficient of 0.4. We twist the cloth and mat by rotating their left and right sides for 2 rounds in 10 seconds, wrapping a stiff affine ellipsoid between them, creating extreme stretching, bending, and frictional contacts. The maximum number of contact pairs is 567K, averaging 351K per time step. The full simulation completed in 29 minutes for 1000 time steps (see the detailed time breakdown in Table 3). Our framework exhibit a 6.93× speedup compared to GIPC in this case.

Box Pile. We then create a challenging example to stress test multibody contact and strain limiting (SL) by dropping an $8 \times 15 \times 8$ pile of $0.16\,\mathrm{m}^3$ boxes with $\rho = 1000\,\mathrm{kg/m}^3$, Young's modulus of 0.1 GPa, and Poisson's ratio of 0.49 onto a $1.5\,\mathrm{m}^2$ thin shell with $100K$ vertices, $\rho = 200\,\mathrm{kg/m}^3$, and a thickness of 1 mm, as shown in Figure 22. The boxes weigh approximately 4 t in total; thus, with a 5 MPa stiffness for our SL energy, this is similar to dropping a sedan onto a bedsheet, which would result in fracture and is thus out of the scope of our simulator. Therefore, we raise our SL energy stiffness to 50 GPa to hold the heavy box pile on the thin shell. Despite the high SL stiffness, our method robustly captures realistic wrinkling behavior. The fast-changing contact pairs, caused by the increasing falling speed of the boxes, significantly challenge the nonlinear solver, resulting in many Newton iterations per time step. Our method handles this challenging simulation efficiently, achieving 3.45 s per time step ($\Delta t = 0.01$ s).

We also compare our method with the SL method in CIPC, listed as 'CIPC (SL+CEMAS+SRBK)' in Table 3. Due to the high-speed box collisions, CIPC's SL requires a sufficiently large barrier stiffness to prevent the cloth from stretching near the SL limit; otherwise, the logarithmic function may cause numerical issues, leading to optimization failure. Here, we set the barrier SL stiffness to 0.1 MPa. However, this higher stiffness results in more expensive linear solves due to worse system conditioning, and CIPC's SL further incurs additional time from numerical eigendecomposition, singular value decomposition for local Hessian calculations, and backtracking line search to ensure SL feasibility, making it slower than our method.

In this example, we observe significantly faster PCG convergence with a diagonal preconditioner than with the MAS preconditioner in GIPC. Thus, we apply the diagonal preconditioner for both GIPC and 'Ours (SRBK)'. This difference arises because Morton code-based reordering may spread nodes from the same box into different groups. Without connections between boxes, nodes from different boxes in the same group cannot be effectively merged, slowing down MAS performance. In contrast, our CEMAS minimizes inter-group connections, effectively grouping nodes from the same box, enabling faster PCG convergence as shown in 'Ours (CEMAS+SRBK)'. Interestingly, approximating the boxes as stiff affine bodies does not significantly improve performance in this scenario, as each box's small DOF count allows efficient computation with pure FEM.

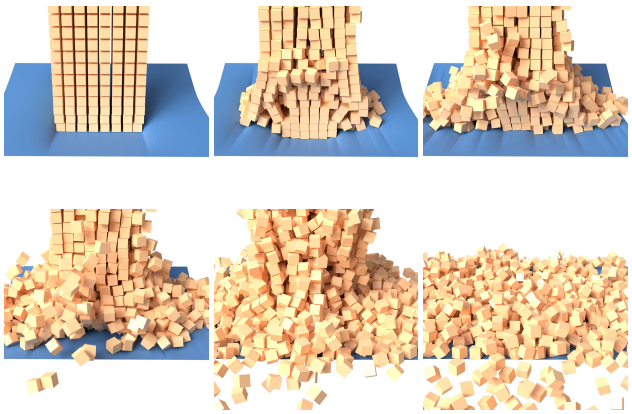


Fig. 22. **Box pile.** $8 \times 15 \times 8$ boxes dropped onto a high-resolution cloth with 100K nodes, showing our capability of efficiently simulating high-speed impacts, rapidly changing contacts, and fine wrinkling details in the same scene at 3.45s per time step.

3D puzzle bus simulation. We digitalize a 3D puzzle, the London Bus from Wooden City® (Figure 23). We extract the part contours in the illustrations of the puzzle instructions to create the 2D sketches in AutoCAD®. The 3D parts are created and assembled in SolidWorks ®. Our *eye-norm* modeling accuracy of the 3D parts, as well as the rough assembling accuracy, is so far from the real-world vehicle manufacture standard that the gear teeth are not even well aligned, leading to a poor dynamic balance condition on the high-speed rotating parts. Thus, it is challenging to predict the behavior of the bus using the traditional analytic motion analysis (e.g., Mechanical Linkage Analysis), but our framework can still simulate it robustly with the yellow gear acting as the motor (see our supplemental document for details). In fact, our simulator robustly captures the realistic non-smooth motion caused by potential interference in the low-accuracy gear system and self-adjustment of motor speed (averaging $3m/s$) relative to resistance (see Figure 1 and our supplemental video).

Fig. 23. **Diagonal view of the London bus.** The yellow gear acts as the motor, driving the bus forward through rolling friction between the wheels and the ground. The wheels are simulated as rubber using FEM, while all other parts are modeled as stiff affine bodies. Detailed simulation setup is provided in Table 4.

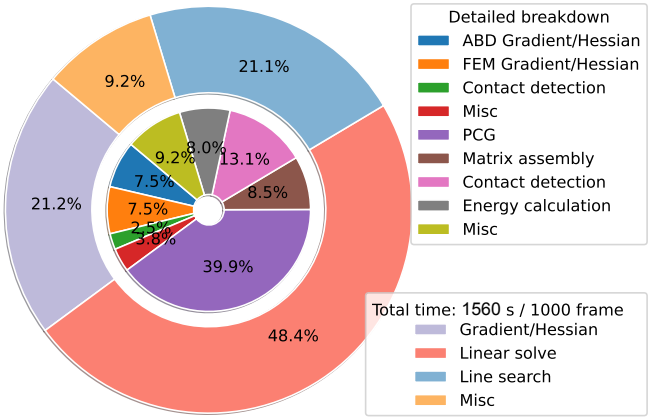


Fig. 24. **Timing breakdown of the London bus simulation.** The outer ring shows the high-level breakdown into Gradient/Hessian (21.2%), Linear solve (48.4%), Line search (5.8%), and Misc (24.6%). The inner ring provides a more detailed breakdown.

A timing breakdown of the simulation is shown in Figure 24. Since the bus is entirely driven by frictional contact forces between the FEM wheels and the ground, resulting in an average of 118K rapidly changing contact pairs, the scene complexity leads to a large number of Newton iterations. The median number of Newton iterations per time step is 29, with a maximum of 786. Despite this challenging scenario, our simulator achieves an average of 1.56 seconds per frame. Detailed simulation setup is provided in Table 4.

8 CONCLUSION AND DISCUSSION

In this paper, we presented a novel, fully GPU-optimized Incremental Potential Contact (IPC) framework capable of simulating materials of a wide range of stiffness with consistent high performance and scalability. Our framework introduces several significant advancements over previous methods, achieving up to 10× speedup compared to the state-of-the-art GIPC. Our contributions include: 1) A novel connectivity-enhanced MAS preconditioner optimized for the GPU, which significantly improves convergence at a lower preconditioning cost; 2) A cubic strain-limiting energy with an analytic eigensystem, designed to simulate stiff membranes such as cloth without membrane locking; 3) An innovative hash-based multi-layer reduction strategy, enabling fast matrix assembly for efficient affine-deformable coupling. Our framework's robust performance is demonstrated through extensive benchmark studies and rigorous performance analyses, showing superior results in various scenarios, including soft, stiff, and hybrid simulations. The framework efficiently handles high resolution, large deformations, and high-speed impacts, maintaining accuracy and robustness. We believe our advancements address critical challenges in IPC simulation, providing a scalable and high-performance solution for complex frictional contact behaviors. Our comprehensive approach not only improves the efficiency of IPC but also retains the method's reliability and accuracy, making it a valuable tool for future research and applications in a variety of fields.

Table 4. **Simulation setup of the London Bus.** FEM Y/P refers to the Young's modulus and Poisson ratio; Gear fr. and Wheel fr. mean the friction coefficient of gear-gear contact and wheel-ground contact, respectively.

Mesh: v, t, f	Δt	FEM Y/P	FEM density	ABD stiffness	ABD density
$(191K, 676K, 265K)$	$5 \times 10^{-3}s$	$7MPa/0.49$	$500kg/m^3$	$100MPa$	$600kg/m^3$
FEM: v,t	Gear fr.	Wheel fr.	Relative $\hat{d}$	Bus Dimensions	
$(31K, 121K)$	1×10^{-3}	0.99	5×10^{-4}	$0.7m \times 1.2m \times 2.3m$	

Limitations and future work. Our preconditioner was designed under the assumption that the mesh connectivity would not change during the simulation. We precompute the partition when loading the meshes, which means our current design is not suitable for simulations with adaptive mesh refinement and/or fracture mechanics, where the mesh topology may change in any frame. Additionally, the main performance bottleneck of IPC comes from two main factors: the expensive computational cost of each Newton iteration and the large number of Newton iterations required in cases with challenging contact scenarios. Our preconditioner design and GPU optimizations have primarily focused on minimizing the cost of each Newton iteration. However, the slow convergence of the optimization in challenging cases can still result in slower overall performance compared to simulation methods that trade accuracy for efficiency. Therefore, it is very meaningful to keep investigating better preconditioners that can effectively handle simulations with mesh topology changes and to explore novel nonlinear solvers or discretization strategies that can significantly improve the convergence of the nonlinear solver.

REFERENCES

Jérémie Allard, François Faure, Hadrien Courtecuisse, Florent Falipou, Christian Duriez, and Paul G Kry. 2010. Volume contact constraints at arbitrary resolution. In *ACM SIGGRAPH 2010 papers*. 1–10.

Sheldon Andrews, Kenny Erleben, and Zachary Ferguson. 2022. Contact and Friction Simulation for Computer Graphics. In *ACM SIGGRAPH 2022 Courses* (Hybrid Event, Vancouver, Canada) *(SIGGRAPH '22)*. Association for Computing Machinery, New York, NY, USA, Article 2, 124 pages.

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.

Yanqing Chen, Timothy A Davis, William W Hager, and Sivasankaran Rajamanickam. 2008. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software (TOMS)* 35, 3 (2008), 1–14.

Yunuo Chen, Minchen Li, Lei Lan, Hao Su, Yin Yang, and Chenfanfu Jiang. 2022. A unified newton barrier method for multibody dynamics. *ACM Transactions on Graphics (TOG)* 41, 4 (2022), 1–14.

Denis Demidov. 2019. AMGCL: An efficient, flexible, and extensible algebraic multigrid implementation. *Lobachevskii Journal of Mathematics* 40 (2019), 535–546.

Maksymilian Dryja and Olof B Widlund. 1990. *Multilevel additive methods for elliptic finite element problems*. Citeseer.

Wenxin Du, Siqiong Yao, Xinlei Wang, Yuhang Xu, Wenqiang Xu, and Cewu Lu. 2024. Intersection-Free Robot Manipulation With Soft-Rigid Coupled Incremental Potential Contact. *IEEE Robotics and Automation Letters* 9, 5 (2024), 4487–4494.

Elliot English and Robert Bridson. 2008. Animating developable surfaces using nonconforming elements. In *ACM SIGGRAPH 2008 papers*. 1–5.

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).

Ming Gao, Xinlei Wang, Kui Wu, Andre Pradhana, Eftychios Sifakis, Cem Yuksel, and Chenfanfu Jiang. 2018. GPU optimization of material point methods. *ACM Transactions on Graphics (TOG)* 37, 6 (2018), 1–12.

Rony Goldenthal, David Harmon, Raanan Fattal, Michel Bercovier, and Eitan Grinspun. 2007. Efficient simulation of inextensible cloth. In *ACM SIGGRAPH 2007 papers*. 49–es.

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.

Kemeng Huang, Floyd M Chitalu, Huancheng Lin, and Taku Komura. 2024. GIPC: Fast and stable Gauss-Newton optimization of IPC barrier energy. *ACM Transactions on Graphics* (2024).

Kemeng Huang, Jiming Ruan, Zipeng Zhao, Chen Li, Changbo Wang, and Hong Qin. 2019. A general novel parallel framework for SPH-centric algorithms. *Proceedings of the ACM on computer graphics and interactive techniques* 2, 1 (2019), 1–16.

Kemeng Huang, Zipeng Zhao, Chen Li, Changbo Wang, and Hong Qin. 2020. Novel hierarchical strategies for SPH-centric algorithms on GPGPU. *Graphical Models* 111 (2020), 101088.

Yupeng Jiang, Yidong Zhao, Clarence E Choi, and Jinhyun Choo. 2022. Hybrid continuum–discrete simulation of granular impact dynamics. *Acta Geotechnica* 17, 12 (2022), 5597–5612.

Zhongshi Jiang, Scott Schaefer, and Daniele Panozzo. 2017. Simplicial complex augmentation framework for bijective maps. *ACM Transactions on Graphics* 36, 6 (2017).

George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392.

Danny M Kaufman, Shinjiro Sueda, Doug L James, and Dinesh K Pai. 2008. Staggered projections for frictional contact in multibody systems. In *ACM SIGGRAPH Asia 2008 papers*. 1–11.

Theodore Kim. 2020. A Finite Element Formulation of Baraff-Witkin Cloth. 39, 8 (2020), 171–179.

Theodore Kim, Fernando De Goes, and Hayley Iben. 2019. Anisotropic elasticity for inversion-safety and element rehabilitation. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–15.

Theodore Kim and David Eberle. 2022. Dynamic deformables: implementation and production practicalities (now with code!). In *ACM SIGGRAPH 2022 Courses* (Vancouver, British Columbia, Canada) *(SIGGRAPH '22)*. Association for Computing Machinery, New York, NY, USA, Article 7, 259 pages.

Lei Lan, Danny M Kaufman, Minchen Li, Chenfanfu Jiang, and Yin Yang. 2022a. Affine body dynamics: fast, stable and intersection-free simulation of stiff materials. *ACM Transactions on Graphics (TOG)* 41, 4 (2022), 1–14.

Lei Lan, Minchen Li, Chenfanfu Jiang, Huamin Wang, and Yin Yang. 2023. Second-order Stencil Descent for Interior-point Hyperelasticity. *ACM Transactions on Graphics (TOG)* 42, 4 (2023), 1–16.

Lei Lan, Guanqun Ma, Yin Yang, Changxi Zheng, Minchen Li, and Chenfanfu Jiang. 2022b. Penetration-free projective dynamics on the GPU. *ACM Transactions on Graphics (TOG)* 41, 4 (2022), 1–16.

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.

Cheng Li, Min Tang, Ruofeng Tong, Ming Cai, Jieyi Zhao, and Dinesh Manocha. 2020b. P-cloth: interactive complex cloth simulation on multi-GPU systems using dynamic matrix assembly and pipelined implicit integrators. *ACM Transactions on Graphics (TOG)* 39, 6 (2020), 1–15.

Minchen Li, Zachary Ferguson, Teseo Schneider, Timothy Langlois, Denis Zorin, Daniele Panozzo, Chenfanfu Jiang, and Danny M. Kaufman. 2020a. Incremental Potential Contact: Intersection- and Inversion-free Large Deformation Dynamics. *ACM Trans. Graph. (SIGGRAPH)* 39, 4, Article 49 (2020).

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

Xuan Li, Yu Fang, Lei Lan, Huamin Wang, Yin Yang, Minchen Li, and Chenfanfu Jiang. 2023. Subspace-Preconditioned GPU Projective Dynamics with Contact for Cloth Simulation. In *SIGGRAPH Asia 2023 Conference Papers*. 1–12.

Xuan Li, Minchen Li, Xuchen Han, Huamin Wang, Yin Yang, and Chenfanfu Jiang. 2024. A Dynamic Duo of Finite Elements and Material Points. In *ACM SIGGRAPH 2024 Conference Papers*.

Huancheng Lin, Floyd M Chitalu, and Taku Komura. 2022. Isotropic ARAP Energy Using Cauchy-Green Invariants. *ACM Trans. Graph.* 41, 6 (2022), 275–1.

Miles Macklin, Kenny Erleben, Matthias Müller, Nuttapong Chentanez, Stefan Jeschke, and Viktor Makoviychuk. 2019. Non-smooth newton methods for deformable multi-body dynamics. *ACM Transactions on Graphics (TOG)* 38, 5 (2019), 1–20.

Marek Krzysztof Misztal and Jakob Andreas Bærentzen. 2012. Topology-adaptive interface tracking using the deformable simplicial complex. *ACM Transactions on Graphics (TOG)* 31, 3 (2012), 1–12.

Matthias Müller, Nuttapong Chentanez, Tae-Yong Kim, and Miles Macklin. 2015. Air meshes for robust collision handling. *ACM Transactions on Graphics (TOG)* 34, 4 (2015), 1–9.

Rahul Narain, Armin Samii, and James F O'brien. 2012. Adaptive anisotropic remeshing for cloth simulation. *ACM transactions on graphics (TOG)* 31, 6 (2012), 1–10.

Maxim Naumov, Marat Arsaev, Patrice Castonguay, Jonathan Cohen, Julien Demouth, Joe Eaton, Simon Layton, Nikolay Markovskiy, István Reguly, Nikolai Sakharnykh, et al. 2015. AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods. *SIAM Journal on Scientific Computing* 37, 5 (2015), S602–S626.

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.

Stéphane Pagano and Pierre Alart. 2008. Self-contact and fictitious domain using a difference convex approach. *International journal for numerical methods in engineering* 75, 1 (2008), 29–42.

Julian Panetta. 2020. Analytic eigensystems for isotropic membrane energies. *arXiv preprint arXiv:2008.10698* (2020).

Željko Penava, Diana Šimić-Penava, and Ž Knezic. 2014. Determination of the elastic constants of plain woven fabrics by a tensile test in various directions. *Fibres & Textiles in Eastern Europe* (2014).

Alvin Shi and Theodore Kim. 2023. A Unified Analysis of Penalty-Based Collision Energies. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 6, 3 (2023), 1–19.

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

Breannan Smith, Fernando De Goes, and Theodore Kim. 2019. Analytic eigensystems for isotropic distortion energies. *ACM Transactions on Graphics (TOG)* 38, 1 (2019), 1–15.

Rasmus Tamstorf, Toby Jones, and Stephen F McCormick. 2015. Smoothed aggregation multigrid for cloth simulation. *ACM Transactions on Graphics (TOG)* 34, 6 (2015), 1–13.

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.

Pengbin Tang, Stelian Coros, and Bernhard Thomaszewski. 2023. Beyond Chainmail: Computational Modeling of Discrete Interlocking Materials. *ACM Transactions on Graphics (TOG)* 42, 4 (2023), 1–12.

Bernhard Thomaszewski, Simon Pabst, and Wolfgang Strasser. 2009. Continuum-based strain limiting. In *Computer Graphics Forum*, Vol. 28. Wiley Online Library, 569–576.

Mickeal Verschoor and Andrei C Jalba. 2019. Efficient and accurate collision response for elastically deformable models. *ACM Transactions on Graphics (TOG)* 38, 2 (2019), 1–20.

Huamin Wang, James O'Brien, and Ravi Ramamoorthi. 2010. Multi-resolution isotropic strain limiting. *ACM Transactions on Graphics (TOG)* 29, 6 (2010), 1–10.

Tianyu Wang, Jiong Chen, Dongping Li, Xiaowei Liu, Huamin Wang, and Kun Zhou. 2023a. Fast GPU-Based Two-Way Continuous Collision Handling. *ACM Transactions on Graphics* 42, 5 (2023), 1–15.

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.

Zhendong Wang, Longhua Wu, Marco Fratarcangeli, Min Tang, and Huamin Wang. 2018. Parallel multigrid for nonlinear cloth simulation. In *Computer Graphics Forum*, Vol. 37. Wiley Online Library, 131–141.

Zhendong Wang, Yin Yang, and Huamin Wang. 2023b. Stable Discrete Bending by Analytic Eigensystem and Adaptive Orthotropic Geometric Stiffness. 42, 6 (2023).

Botao Wu, Zhendong Wang, and Huamin Wang. 2022. A GPU-based multilevel additive schwarz preconditioner for cloth and deformable body simulation. *ACM Transactions on Graphics (TOG)* 41, 4 (2022), 1–14.

Haomiao Wu and Theodore Kim. 2023. An Eigenanalysis of Angle-Based Deformation Energies. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 6, 3 (2023), 1–19.

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.

Tianyi Xie, Minchen Li, Yin Yang, and Chenfanfu Jiang. 2023. A Contact Proxy Splitting Method for Lagrangian Solid-Fluid Coupling. *ACM Transactions on Graphics (TOG)* 42, 4 (2023), 1–14.

Zipeng Zhao, Kemeng Huang, Chen Li, Changbo Wang, and Hong Qin. 2020. A Novel Plastic Phase-Field Method for Ductile Fracture with GPU Optimization. In *Computer Graphics Forum*, Vol. 39. Wiley Online Library, 105–117.

Yang Zheng, Qingqing Zhao, Guandao Yang, Wang Yifan, Donglai Xiang, Florian Dubost, Dmitry Lagun, Thabo Beeler, Federico Tombari, Leonidas Guibas, et al. 2024. PhysAvatar: Learning the Physics of Dressed 3D Avatars from Visual Observations. *arXiv preprint arXiv:2404.04421* (2024).

Yongning Zhu, Eftychios Sifakis, Joseph Teran, and Achi Brandt. 2010. An efficient multigrid method for the simulation of high-resolution elastic solids. *ACM Transactions on Graphics (TOG)* 29, 2 (2010), 1–18.

Technical Supplement to "Advancing GPU IPC for Stiff Affine-Deformable Simulation"

KEMENG HUANG*, Carnegie Mellon University, USA and The University of Hong Kong, TransGP, China
XINYU LU*, TransGP, China
HUANCHENG LIN, Carnegie Mellon University, USA and The University of Hong Kong, TransGP, China
TAKU KOMURA, The University of Hong Kong, TransGP, China
MINCHEN LI, Carnegie Mellon University, USA

This document provides supplementary details about the technical background, derivations, and implementation of our methods.

CCS Concepts: • **Computing methodologies** → **Physical simulation**; *Parallel algorithms.*

Additional Key Words and Phrases: GPU Programming, Incremental Potential Contact, Elastodynamics, Finite Element Method, Affine Body Dynamics, Preconditioning, Cloth Simulation

Contents

1 QUANTIZATION OF NEWTON ITERATION COST

In Newton's method for nonlinear optimization, the computational costs of assembling and solving the linear system are critical to overall performance. These costs can be quantified as

$$T^{\text{nt}} = T^{\text{ga}} + T^{\text{pa}} + N^{\text{pcg}} \cdot T^{\text{pcg}}, \tag{1}$$

where T^{nt} is the time spent on the linear system per Newton iteration, T^{ga} is the cost of assembling the linear system, T^{pa} is the preconditioner construction time, N^{pcg} is the number of iterations in the preconditioned conjugate gradient (PCG) solver, and T^{pcg} is the time per PCG iteration. Non-bottleneck components, such as gradient/Hessian computation and collision detection, are excluded here.

The time per PCG iteration, T^{pcg}, can be further broken down as

$$T^{\text{pcg}} = T^{\text{spmv}} + T^{\text{ap}} + T^{\text{dot}} + T^{\text{axpby}}, \tag{2}$$

where T^{spmv}, T^{ap}, T^{dot}, and T^{axpby} represent the time costs of sparse matrix-vector multiplication, applying the preconditioner, dot products, and vector operations of the form $\mathbf{y} = \alpha \cdot \mathbf{x} + \beta \cdot \mathbf{y}$, respectively.

We can refine the quantization of T^{nt} by introducing structure quality metrics: Q^{gs} for the coefficient matrix of the linear system, Q^{ps} for the preconditioner, and Q^{pi} for the preconditioner's quality in approximating the inverse of the coefficient matrix. This gives:

$$\begin{aligned} T^{\text{nt}} =& T^{\text{ga}}(Q^{\text{gs}}) + T^{\text{pa}}(Q^{\text{pi}}, Q^{\text{ps}}) \\ &+ N^{\text{pcg}}(Q^{\text{pi}}) \cdot (T^{\text{spmv}}(Q^{\text{gs}}) + T^{\text{ap}}(Q^{\text{ps}}) + T^{\text{dot}} + T^{\text{axpby}}). \end{aligned} \tag{3}$$

Typically, Q^{gs} is positively correlated with T^{ga} but negatively correlated with T^{spmv}. For instance, compared to a system with a duplicated and unsorted format, a uniquely sorted and compressed format incurs a higher assembly cost but enables faster matrix-vector multiplication. Similar relations apply to Q^{ps}, T^{pa}, and T^{ap}. Generally, Q^{pi} is positively correlated with T^{pa} but negatively correlated with N^{pcg}—a higher-quality preconditioner requires more time to construct but leads to faster PCG convergence. This can result in significant differences in PCG iteration counts between preconditioners. T^{dot} and T^{axpby} are unaffected by Q^{gs}, Q^{ps}, or Q^{pi}, as they depend solely on the number of degrees of freedom (DoFs) in the system.

Thus, improving the performance of each Newton iteration requires enhancing Q^{gs}, Q^{ps}, and Q^{pi} to reduce N^{pcg}, T^{spmv}, and T^{ap} while ensuring that T^{ga} and T^{pa} do not increase significantly.

2 THE CONSTRUCTION OF CONNECTIVITY-ENHANCED MAS

The Multilevel Additive Schwarz (MAS) method is the state-of-the-art preconditioner for PCG solvers in elastodynamic contact simulations [Wu et al. 2022]. However, its Morton code based node reordering may result in slow performance in challenging cases. To further speedup PCG convergence, we proposed a connectivity enhanced MAS based on METIS [Karypis and Kumar 1998] in the main paper. In this section, we will start by introducing the background of MAS in subsection 2.1, and then detail our METIS-based node reordering approach in subsection 2.2, followed by an explanation of our GPU implementation for coarse space construction in subsection 2.3.

2.1 Multilevel Additive Schwarz (MAS) Background

For clarity and self-containment, we include the mathematical definition of MAS from Wu et al. [2022] in this subsection.

*K. Huang and X. Lu contribute equally to this work

Authors' Contact Information: Kemeng Huang, kmhuang@connect.hku.hk, kmhuang819@gmail.com, Carnegie Mellon University, USA and The University of Hong Kong, TransGP, China; Xinyu Lu, lxy819469559@gmail.com, TransGP, China; Huancheng Lin, lamws@connect.hku.hk, Carnegie Mellon University, USA and The University of Hong Kong, TransGP, China; Taku Komura, taku@cs.hku.hk, The University of Hong Kong, TransGP, China; Minchen Li, minchernl@gmail.com, Carnegie Mellon University, USA.

Algorithm 1 METIS-based Node Reordering

Input:
Pos ▷ position of nodes
Ele ▷ node index in each element
DNum ▷ the number of partitions
Output:
sorted_Pos ▷ updated order for each node
sorted_Ele ▷ updated new node index in each element
map ▷ our mapping array
remap ▷ our remapping array
Method:

$P \leftarrow$ Len(Pos) ▷ get the number of nodes
// find neighbor count and indices j for each node:
$\{Neighbors[P]\{j\}, Counsts[P]\} \leftarrow$ **NeighborTraversal**(Ele)
$Offsets[P+1] \leftarrow$ ExclusiveSum$(Counts)$
// expand the neighbor information into an array for parallelization, which also aligns with the input format required by METIS:
for each node i **do**
  $offset = Offsets[i]$
  for each neighbor j of node i **do**
    $Adjs[offset] = j$
    $AdjW[offset++] = Counts[j]$
  end for
end for
// record the part index for each node after METIS partitioning:
$PIdx[P] \leftarrow$ **METIS_Partition**$(Adjs, AdjW, Offsets, DNum)$
// sort mesh nodes according to the part index of nodes:
$idx[N] \leftarrow \{0, 1, 2, ..., P\}$
$\{sorted\_PIdx, sorted\_idx\} \leftarrow$ sort $\{PIdx, idx\}$ using $PIdx$
for $i = 0, 1, ..., P-1$ **do**
  $sorted\_Pos[i] = Pos[sorted\_index[i]]$
end for
// update elements according to the new index of each node:
for each element k **do**
  for each node index t in element k **do**
    $sorted\_Ele[k][t] = sorted\_index[sorted\_Ele[k][t]]$
  end for
end for
// construct **map** and **remap** array:
remap$[DNum \times domain\_size] \leftarrow$ initialize all entries with -1
$s = 0$
for $i = 0, 1, ..., P-1$ **do**
  remap$[domain\_size \times sorted\_PIdx[i] + (s++)] = i$
  if i<node_number-2 **then**
    if sorted_PIdx[i] != sorted_PIdx[i+1] **then**
      $s = 0$
    end if
  end if
end for
$s = 0$
for each entry i in *remap* array **do**
  if remap[i] == s **then**
    map$[s++] = i$;
  end if
end for

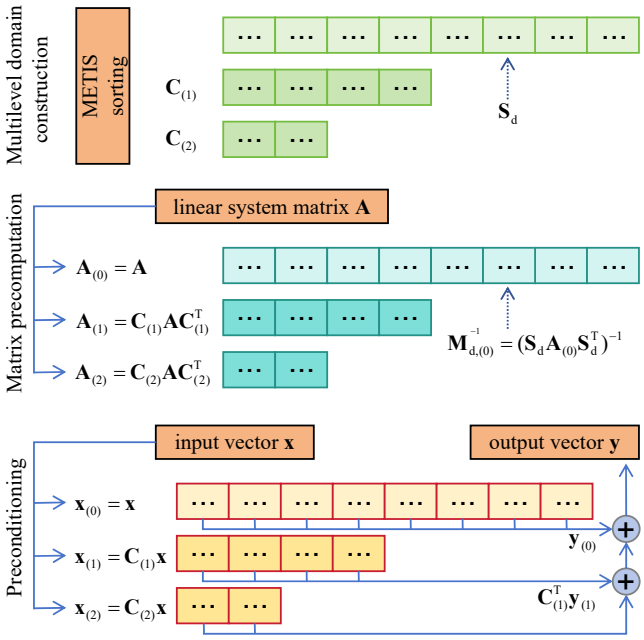


Fig. 1. ***Illustration of MAS preconditioner.***

Let $\mathbf{A}$ be the system matrix, and let Ω_d represent a set of domains covering all nodes Ω, such that $\Omega = \bigcup \Omega_d$. The selection matrix $\mathbf{S}_d \in \mathbb{R}^{3N_d \times 3N}$ extracts the N_d nodes in domain d from the N nodes in Ω. The additive Schwarz (AS) preconditioner is defined as:

$$\mathbf{M}_{(0)}^{-1} = \sum_d \mathbf{S}_d^T \mathbf{M}_{\mathbf{d},(0)}^{-1} \mathbf{S}_d, \tag{4}$$

where $\mathbf{M}_{\mathbf{d},(0)} = \mathbf{S}_d \mathbf{A} \mathbf{S}_d^T \in \mathbb{R}^{3N_d \times 3N_d}$ is a symmetric positive definite matrix within domain d. We do not use overlapping domains here to improve parallel efficiency, making $\mathbf{M}_{(0)}^{-1}$ equivalent to a block diagonal preconditioner.

To further accelerate convergence in large linear systems, a multilevel approach can be adopted. In a two-level MAS, the MAS preconditioner is given by:

$$\mathbf{M}_{2,\mathrm{MAS}}^{-1} = \mathbf{M}_{(0)}^{-1} + \mathbf{C}_{(1)}^T \mathbf{M}_{(1)}^{-1} \mathbf{C}_{(1)}, \tag{5}$$

where $\mathbf{C}_{(1)} \in \mathbb{R}^{3N_1 \times 3N}$ is a coarsening matrix, and $\mathbf{M}_{(1)}$ is a single-level AS preconditioner for the coarsened system matrix $\mathbf{A}_{(1)} = \mathbf{C}_{(1)} \mathbf{A} \mathbf{C}_{(1)}^T$. The second term in Equation 5 is often referred to as the coarse space correction. By further coarsening $\mathbf{M}_{(0)}$, we can extend this to a general MAS preconditioner:

$$\mathbf{M}_{\mathrm{MAS}}^{-1} = \mathbf{M}_{(0)}^{-1} + \sum_l^L \mathbf{C}_{(l)}^T \mathbf{M}_{(l)}^{-1} \mathbf{C}_{(l)}, \tag{6}$$

where $\mathbf{C}_{(l)} \in \mathbb{R}^{3N_{(l)} \times 3N}$ is the coarsening matrix at level l. Unlike hierarchical multilevel methods, these coarseners directly map level 0 to each coarse level, allowing parallel preconditioning across all levels instead of sequentially.

A key challenge in MAS is the efficient construction of coarseners $C_{(l)}$. In GPU MAS [Wu et al. 2022], this process relies on Morton code sorting, which, while efficient, may present issues discussed in the main paper. We address these by incorporating mesh connectivity using METIS for sorting.

2.2 METIS-based Node Reordering

Based on the discussion in the main paper, considering mesh connectivity when reordering nodes is crucial for effective aggregation, which MAS [Wu et al. 2022] based on Morton code does not address. Instead, we partition the mesh nodes using METIS [Karypis and Kumar 1998] and map the partitions, often containing different numbers of nodes, to subdomains with padding, aiming for a dense connectivity within each subdomain to facilitate more effective aggregation at coarser levels.

Further details of our METIS-based sorting can be found in Algorithm 1. In Lines 1–10, *Adjs* is an expanded array of Neighbors, and *AdjW* is the corresponding weight array. We use the number of neighbors as weights to emphasize nodes with more connections. The *Offsets* array helps identify the neighbors for different nodes in *Adjs*. Using *Adjs*, *AdjW*, and *Offsets*, along with the required number of parts, we can perform the partition using the METIS library in Line 11. METIS then produces an array *PIdx*, which indicates the part index for each node, helping to facilitate sorting in Lines 12–21. As mentioned in the main paper, the METIS partitions may result in different number of nodes, which may results in isolation issues if directly mapped to MAS subdomains. To address this, we propose a map and remap method with padding in the main paper, with detailed implementation provided in Lines 22–39.

2.3 Coarse Space Construction on the GPU

This subsection details our GPU implementation for coarse space construction, focusing primarily on the level-1 (L1) coarse space, while higher levels are constructed similarly.

Our implementation leverages the CUDA library and efficiently utilizes CUDA warps. Testing with a maximum domain size of 32 (matching the warp size) ensures synchronized threads, as the warp is the fundamental execution unit. To address the artifact shown in Figure 6(b) of Wu et al. [2022], we adopt an aggregation strategy that considers node connectivity. Specifically, we evaluate each candidate supernode at L1, splitting it into multiple supernodes if nodes are not fully connected. To accomplish this, we assign a hash value to each node within a domain. Nodes sharing a hash value are connected, transforming the splitting process into a hashing operation, which we explain below.

Initial Hash Encoding. To identify all connected nodes, we first examine directly connected neighbors. In Algorithm 2, threads are launched based on the size of the remap array, ensuring no isolated nodes (as discussed in the main text; see Figure 6). First, we retrieve the actual node index using the remap array (line 4). This allows us to identify neighboring indices accurately (lines 11–14). Using the map array, we locate the global domain indices ('mapped_id') in the domain array for each neighbor (refer to Figure 7 in the main text). If a neighbor is in the same domain, we set the corresponding bit in a 32-bit hash integer to 1. This encoding uses a 32-bit integer, sufficient for domain sizes up to 32. For instance, if a node in the first domain has index 1 and neighbors {2, 4, 5, 100} with mapped indices {2, 4, 5, 110}, only {2, 4, 5} are within the same domain, resulting in a bit representation of 00110110 for node 1's hash value. To prevent recalculation of visited neighbors at higher levels, we update the neighbor information as specified in lines 21–26. Algorithm 2 thus generates the initial hash value.

Contact Handling. The initial hash captures only mesh connectivity and does not account for contact or collision information. Here, we describe our GPU approach for incorporating contact. Our hash encoding simplifies this step: we update the hash value based on

Algorithm 2 Initial hash encoding

Input:
- $Counsts$ ▷ neighbor counts per node
- $Offsets$ ▷ same with the 'Offsets' in Algorithm 1
- $Adjs$ ▷ same with the 'Adjs' in Algorithm 1

Output:
- con_hashs ▷ compressed domain connection
- $Counsts$ ▷ updated 'Counsts'
- $Adjs$ ▷ updated 'Adjs'

Method:

1: **for** $tid = 0, 1, ..., domain\_number \times domain\_size - 1$ **do**
2: $domain\_id = tid/domain\_size$
3: $lane\_id = tid\%domain\_size$ ▷ node index inside domain
4: $node\_id =$ **remap**$[tid]$ ▷ get real node index
5: **if** $node\_id < 0$ **then**
6: return ▷ filter the ghost node in subdomain
7: **end if**
8: $neighbor\_num = Counsts[node\_id]$
9: $con\_hashs[node\_id] = 1 << lane\_id$ ▷ init hash value
10: $remained\_neighbor = 0$
11: $offset = Offsets[node\_id]$ ▷ get the first neighbor index in compacted neighbor array $adjs$
12: $loop\_index = 0$
13: **for** $loop\_index < neighbor\_num$ **do**
14: $neighbor\_id = adjs[offset + loop\_index]$
// get neighbor node index in the preconditioner domain:
15: $mapped\_id =$ **map**$[neighbor\_id]$
16: $neighbor\_domain\_id = mapped\_id/domain\_size$
17: **if** neighbor_domain_id == domain_id **then**
// update the hash value using the corresponding index information of neighbors within the same sub-domain:
18: $neighbor\_lane\_id = neighbor\_id\%domain\_size$
19: $con\_hashs[node\_id] \mid= (1 << neighbor\_lane\_id)$
20: **else**
// update the remaining neighbor information:
21: $adjs[offset + remained\_neighbor] = neighbor\_id$
22: $remained\_neighbor = remained\_neighbor + 1$
23: **end if**
24: $loop\_index = loop\_index + 1$
25: **end for**
26: $Counsts[tid] = remained\_neighbor$
27: **end for**

Algorithm 3 Contact handling

Input:
con_hashs[*N*] ▷ compressed domain connection
coarse_table[*L*][*N*] ▷ super node index in higher level
collision_paris ▷ collision node index
l ▷ level index
Output:
con_hashs ▷ updated 'con_hashs'
Method:

1: **for** *tid* = 0, 1, ..., Len(*collision_pairs*)-1 **do**
2: *col_pair* = *collision_pairs*[*tid*] ▷ get collision pair
3: *con_Mask*[*col_pair.size*] ← Initialized with zero ▷ the temporary hash value for collision nodes
4: **for** *i* = 0, 1, ..., Len(*col_pair*)-1 **do**
5: **if** *l* == 0 **then**
6: *m_pair*[*i*] = *map*[*col_pair*[*i*]] ▷ map the actual node index in the collision pair to the MAS domain index
7: **else**
8: *m_pair*[*i*] = *coarse_table*[*col_pair*[*i*] + (*l* − 1) × *N*] ▷ retrieve the corresponding super node index at higher levels
9: **end if**
10: **end for**
11: **for** *i* = 0, 1, ..., Len(*col_pair*)-1 **do**
12: **for** *j* = *i* + 1; *j* < *col_pair.size*; *j* = *j* + 1 **do**
// fetch all node pairs within the collision stencil to construct their collision connectivity information:
13: *nid_0* = *m_pair*[*i*]; *nid_1* = *m_pair*[*j*]
14: **If** *nid_0* == *nid_1* **then** *continue*
15: **if** *nid_0*/*domain_size* == *nid_1*/*domian_size* **then**
// update the temporary hash value with collision nodes within the same MAS subdomain:
16: *con_Mask*[*i*] |= (1 << (*nid_0*%*domian_size*))
17: *con_Mask*[*j*] |= (1 << (*nid_1*%*domian_size*))
18: **end if**
19: **end for**
20: **end for**
// merge the temporary collision hash value with the domain connection hash value:
21: **for** *i* = 0, 1, ..., Len(*col_pair*)-1 **do**
22: **atomicOr**(&*con_hashs*[*i*], *con_Mask*[*i*])
23: **if** *l* == 0 **then**
24: **atomicOr**(&*con_hashs*[*col_pair*[*i*]], *con_Mask*[*i*])
25: **else**
26: **atomicOr**(&*con_hashs*[*m_pair*[*i*]], *con_Mask*[*i*])
27: **end if**
28: **end for**
29: **end for**

contact pair information. In Algorithm 3, threads are launched for each contact pair, starting with the initial hash value from Algorithm 2. The variable *collision_pair* stores the indices of colliding nodes, which are mapped to domain or supernode indices at higher levels (lines 4–10). The *coarse_table* maps L0 node indices to corresponding supernode indices at higher levels, detailed in Algorithm 5. Each node in the contact pair is treated as fully connected; if both nodes belong to the same domain at the current level, the corresponding bit in the local hash is set to 1 (lines 12–22). Finally, we atomically merge the local hash into each node's hash to incorporate contact information. Note that colliding nodes may reside in different domains at lower levels but are usually aggregated at higher levels.

Algorithm 4 Connection expansion

Input:
con_hashs[*N*] ▷ compressed domain connection
remap ▷ 'remap' array
Output:
con_hashs[*N*] ▷ updated 'conn_hashs'
supNode_num[*DNum*] ▷ super node count per domain
Method:

1: **for** *tid* = 0, 1, ..., *domain_number* × *domain_size* − 1 **do**
2: *domain_id* = *tid*/*domain_size*
3: *btd_id* = *tid*%*blockSize* ▷ local thread id inside the CUDA block
4: *bDomain_id* = (*btd_id*)/*domain_size* ▷ get the local subdomain index within CUDA thread block
5: *lane_id* = *tid*%*domian_size*
6: *node_id* = **remap**[*tid*]
7: *Masks*[*blockSize*] ← **shared memory** ▷ for loading the hash value for sharing within each subdomain
8: *counts*[*blockSize*/*domain_size*] ← **shared memory** ▷ for counting the number of super nodes per subdomain
9: **If** *node_id* < 0 **then** *continue*
10: **If** *lane_id* == 0 **then** *counts*[*bDomain_id*] = 0
11: *conMask* = *con_hashs*[*node_id*]
12: *Masks*[*btd_id*] = *conMask*
13: *visited* = (1 << *lane_id*)
14: **while** *conMask* ≠ (1 << *domain_size*) − 1 **do**
15: *todo* = (*visited* ⊕ *conMask*) ▷ 32-bit integer marking the connected nodes that have been traversed
16: **If** *todo* == 0 **then** *break* ▷ all connected nodes have been visited
17: *next* = **__ffs**(*todo*) − 1 ▷ fetch the remaining connected node with the minimum index
18: *visited* |= (1 << *next*) ▷ mark the visited node
19: *conMask* |= *Masks*[*next* + *bDomain_id* × *domain_size*] ▷ merge the neighbor's connected nodes that are not included in the current hash value
20: **end while**
21: *con_hashs*[*node_id*] = *conMask*
22: *prefix* = **__popc**(*conMask* **And** ((1 << *lane_id*) − 1))
// count the super nodes in each MAS subdomain:
23: **if** *prefix* == 0 **then**
24: **atomicAdd**(&*counts*[*bDomian_id*], 1)
25: **end if**
26: **if** *lane_id* == 0 **then**
27: *supNode_num*[*domain_id*] = *counts*[*bDomain_id*]
28: **end if**
29: **end for**

Expanded Hash Value. After encoding connectivity and collision information, we merge hash values for nodes indirectly connected within a domain, minimizing the number of connection segments. In Algorithm 4, each node accesses neighboring hash values in shared memory (line 15) for merging. Since the hash is based on domain indices, we reverse map each bit set to 1 to access the neighbor's hash values (lines 16–26; **__ffs** finds the position of the least significant set bit). Consequently, connected nodes share the same hash value, where bit positions indicate their local domain indices. After merging, the number of unique hash values in the domain determines the supernode count at the next level (lines 27–33; **__popc** counts set bits in a 32-bit integer).

L1 Construction. At this point, all connected nodes within a domain share the same hash value, and the number of supernodes per domain is known. We construct the final L1 coarse space, consisting of two mapping arrays: *coarseTable* and *goNext*. *coarseTable* maps node indices to supernode indices, while *goNext* maps each node to its matrix or vector index. Although *coarseTable* provides supernode indices, *goNext* ensures consistent domain size, even when domains lack sufficient nodes, by setting empty entries to zero. Additionally, since all domains are stored in a single array, *goNext* aligns node and array indices. In Algorithm 5, we identify the thread assigned to the node with the lowest domain index for each hash value. These threads match the supernode count, setting a shared 32-bit integer per domain and marking bit positions based on domain index. Counting bits set to 1 before each bit position determines the supernode index, with the global supernode offset allowing correct settings in *coarseTable* and *goNext*.

This completes our GPU approach for L1 coarse space construction. Higher levels can be constructed similarly, and our source code will be made available upon acceptance.

3 LONDON BUS SIMULATION DETAILS

We partition the bus into several components:

(1) the bus frame, a relatively static structure that supports other parts;
(2) the gear system transferring the motion and power from the motor to the wheels;
(3) the back wheels contacting the ground and driving the bus through rolling friction; and
(4) the front wheels contacting the ground but acting as resistance to the bus.

We pack the bus frame and gear axles together into one affine body [Lan et al. 2022] and keep every gear as a separate affine body. The constraints between the gear axles and the gears are achieved directly by the contact force, which is purely physical. The wheels are simulated using FEM, with a $7MPa$ Young's modulus and a 0.49 Poisson's ratio, a common rubber material in reality. Other configurations are listed in the main paper. The time step size limitation ($5ms$) comes from the fastest rotating speed of the gears.

To drive the bus, we designed an ideal motor that applies any extra kinetic energy we want to a single affine body. A generalized motor energy P for a single affine body is defined as

$$P = \frac{1}{2}\|\dot{\mathbf{q}}^P\|^2_{\mathbf{M}^P}, \tag{7}$$

Algorithm 5 L1 construction

Input:
supNode_num ▷ super node number is each L0 subdomain
con_hashs ▷ the output 'con_hashs' in Algorithm 4
remap
Output:
goNext ▷ map to the global super node index
coarseTable ▷ map to the local super node index in higher levels
sNodes ▷ the number of super nodes in L1
Method:

1: $Offsets \leftarrow$ ExclusiveSum($supNode\_num$) ▷ start from 0
2: **for** $tid = 0, 1, ..., domain\_number \times domain\_size - 1$ **do**
3: $dom\_id = tid/domain\_size$
4: $btd\_id = tid\%blockSize$
5: $bDomain\_id = (btd\_id)/domain\_size$ ▷ CUDA blockSize is set with $domain\_size^2$
6: $lane\_id = tid\%domian\_size$
7: $electMask[domain\_size] \leftarrow$ **shared memory space** ▷ used to encode each subdomain with supernode information
8: $PrefixId[domain\_size^2] \leftarrow$ **shared memory space** ▷ used to record the supernode index within the subdomain
9: **If** $lane\_id == 0$ **then** $electMask[bDomain\_id] = 0$
10: **if** $is\_the\_last\_thread(tid)$ **then**
11: $sNodes = Offsets[dom\_id] + supNode\_num[dom\_id]$ ▷ get the number of super nodes in L1
12: **end if**
13: $node\_id =$ **remap**$[tid]$
14: **If** $node\_id < 0$ **then** *continue*
15: $conMask = con\_hashs[node\_id]$
16: $prefix =$ **__popc**$(conMask$ **And** $((1 << lane\_id) - 1))$ ▷ locate the thread of the leaf node with the minimum node index for each supernode
// only the located thread can set the bit value for the *electMast* of each subdomain, so the order of the bits with a value of 1 can be used to determine the order of supernodes within each subdomain:
17: **if** $prefix == 0$ **then**
18: **atomicOr** $(\&electMask[bDomian\_id], 1 << lane\_id)$
19: **end if**
20: $PrefixId[btd\_id] =$ **__popc**$(electMask[bDomian\_id]$ **And** $((1 << lane\_id) - 1))$
21: $PrefixId[btd\_id] = PrefixId[tid\%block\_size] + Offsets[domain\_id]$
22: $elect\_lane =$ **__ffs**$(conMask) - 1$
23: $lane\_prefix = Prefix[elect\_lane + domian\_size \times bDomain\_size]$
24: $coarseTable[0][node\_id] = lane\_prefix$
25: $goNext[node\_id] = lane\_prefix + \lceil N/domain\_size \rceil \times domain\_size$
26: **end for**

where $\dot{\mathbf{q}}^P$ is the effective velocity of the motor driver, $\mathbf{M}^P$ is the power mass matrix, which we expand to $K^P \cdot \mathbf{M}$, where K^P is a parameter indicating how strong the motor is relative to the body mass, and $\mathbf{M}$ is the mass matrix of the affine body.

The time-discretized version of this energy has a form of soft position constraints weighted by the power mass matrix:

$$\hat{P}(\mathbf{q}) = \frac{1}{2\Delta t^2}\|\mathbf{q}^P - \mathbf{q}\|^2_{\mathbf{M}^P}, \tag{8}$$

where $\mathbf{q}^P$ is the target state the motor is trying to drive the system towards and Δt is the time step size. To explain it in more details, at the beginning of time step $t+1$, we have the state $\mathbf{q} = \mathbf{q}^t$, and we can add the motor energy to the system by simply adding $\hat{P}(\mathbf{q}^t)$, where $(\mathbf{q}^P - \mathbf{q}^t)/\Delta t \approx \dot{\mathbf{q}}^P$. Then, by solving $\mathbf{q}^{t+1}$, the added motor energy decreases to $\hat{P}(\mathbf{q}^{t+1})$ (since $\mathbf{q}^{t+1}$ will get closer to $\mathbf{q}^P$ than $\mathbf{q}^t$), and the lost energy transfers to other forms in the system, e.g. kinetic energy, heat, etc.

To compute $\mathbf{q}^P$, we first calculate the movement in material space as

$$\bar{\mathbf{x}}_i^P = \bar{\mathbf{x}}_i + \Delta\bar{\mathbf{x}}_i^P, \tag{9}$$

where $\bar{\mathbf{x}}_i$ is the initial position of the i-th node, $\Delta\bar{\mathbf{x}}_i^P$ is the movement in the material space, and $\bar{\mathbf{x}}_i^P$ is the final position of the i-th node in the material space. Due to the linearity of the affine transformation, we can then calculate the target world-space position as

$$\mathbf{x}^P = \mathbf{J}(\bar{\mathbf{x}}_i^P)\mathbf{q}^t, \tag{10}$$

by changing the basis of the affine deformation modes in $\mathbf{J}$ while keeping using $\mathbf{q}^t$ (note that $\mathbf{x}^t = \mathbf{J}(\bar{\mathbf{x}}_i)\mathbf{q}^t$).

Next, based on the relation

$$\mathbf{x}^P = \mathbf{J}(\bar{\mathbf{x}}_i)\mathbf{q}^P, \tag{11}$$

we can now solve for $\mathbf{q}^P$ by choosing 4 mesh nodes that are not coplanar to form $\mathbf{J}(\bar{\mathbf{x}}_i)$. As illustrated in Figure 2, we pick $\bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_0$ as the rotation axis of the motor, and $\bar{\mathbf{x}}_2 - \bar{\mathbf{x}}_0$ and $\bar{\mathbf{x}}_3 - \bar{\mathbf{x}}_0$ to form two orthogonal vectors to the rotation axis.

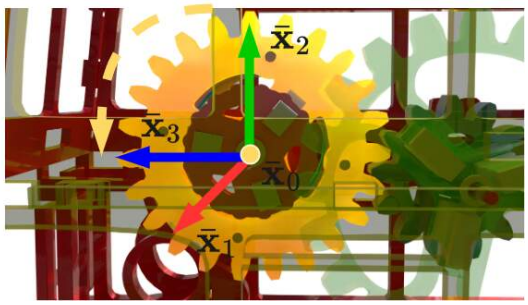


Fig. 2. ***Illustration of the local frame for calculating*** $\mathbf{q}^P$. *Here,* $\bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_0$ *is picked as the rotation axis of the motor, and* $\bar{\mathbf{x}}_2 - \bar{\mathbf{x}}_0$ *and* $\bar{\mathbf{x}}_3 - \bar{\mathbf{x}}_0$ *are two orthogonal vectors to the rotation axis.*

REFERENCES

George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392.

Lei Lan, Danny M Kaufman, Minchen Li, Chenfanfu Jiang, and Yin Yang. 2022. Affine body dynamics: fast, stable and intersection-free simulation of stiff materials. *ACM Transactions on Graphics (TOG)* 41, 4 (2022), 1–14.

Botao Wu, Zhendong Wang, and Huamin Wang. 2022. A GPU-based multilevel additive schwarz preconditioner for cloth and deformable body simulation. *ACM Transactions on Graphics (TOG)* 41, 4 (2022), 1–14.