

Boba: Batched Simulation for Physics-Based Gaussian Digital Twins

Yihan Pang¹, Hanxiao Jiang², Sushant Kondguli³, Sarita Adve¹, and Shenlong Wang¹

¹ University of Illinois Urbana-Champaign, Urbana, IL, USA
{yihanp2,sadve,shenlong}@illinois.edu

² Columbia University, New York, NY, USA
hanxiao.jiang@columbia.edu

³ Meta Reality Labs Research, USA
sushant.kondguli@gmail.com

Abstract. Digital twins replicate the appearance and physical behavior of real-world objects for interactive simulation in XR, robotics, and gaming. While neural representations such as 3D Gaussians enable high-fidelity twins reconstructed from images, existing systems simulate only one instance at a time and barely reach real-time performance, making large-scale rollout evaluation for robot learning, model predictive control, and multi-user XR impractical. We present Boba, the first batched simulator for physics-based Gaussian digital twins. Our key idea is to separate the static twin template from the dynamic simulation state and co-design the physics, deformation, and rendering/visualization pipelines for batched execution. This structure enables shared template computation across instances while minimizing memory traffic and synchronization overhead. Guided by this principle, we introduce compact surrogate spring-mass models and a set of system-level optimizations—including mixed-precision Gaussian skinning, memory-efficient execution, and shared-memory-aware batching—that significantly reduce compute and memory costs. On an XR-configured Jetson Orin, Boba’s local configuration achieves over $10\times$ single-instance speedup over the prior state-of-the-art PHYSTWIN pipeline. Its distributed configuration, split between the Orin and a server, reaches 25.1 ms end-to-end latency, $13.2\times$ lower than the same PHYSTWIN baseline on Orin, while lowering incremental power by 22.2% relative to Boba’s local configuration. For batched execution, Boba scales to 3,310 FPS on RTX 4090, a $16.2\times$ system-level aggregate-throughput improvement over Boba’s optimized single-instance configuration on the same GPU, and accelerates predictive control for deformable object manipulation, achieving speedups from $26\times$ to an estimated $> 2,000\times$ relative to PHYSTWIN-based planning. These results show that Boba can support power-constrained XR applications and scalable robot planning.

1 Introduction

Digital twins aim to replicate both the appearance and physical behavior of real-world objects, enabling interactive simulation for extended reality (XR), robotics,

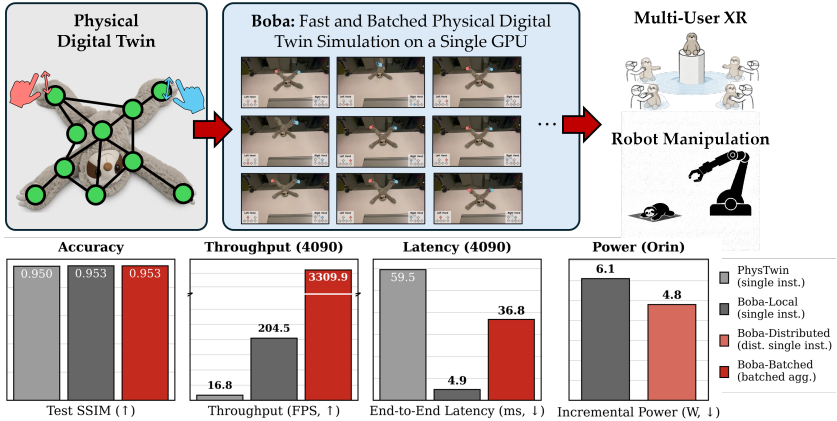


Fig. 1: Boba: Efficient and high-throughput physics-based Gaussian digital twin simulation. By co-designing physics, deformation, and rendering/visualization, Boba preserves fidelity, provides over $10\times$ single-instance speedup relative to the prior state-of-the-art PhysTwin pipeline, and reaches 3,310 FPS average aggregate throughput on 4090. For XR deployment, BOBA-DISTRIBUTED lowers the Jetson Orin’s incremental power by 22.2% relative to BOBA-LOCAL. Together, Boba’s configurations support applications such as multi-user XR and robot manipulation.

and gaming. Recent neural representations such as 3D Gaussians allow high-fidelity digital twins to be reconstructed directly from images, and systems such as PhysGaussian, VR-GS, and PhysTwin [13, 14, 36] couple these representations with deformable physics simulation. However, these pipelines simulate only one instance at a time and barely reach real-time rates, making interactive XR use difficult and limiting applications that require many parallel rollouts, such as robot learning, model predictive control, and multi-user XR systems.

A promising direction to address this limitation is **batched simulation**, where many simulation instances share model parameters and assets while executing in parallel on a GPU. Prior works [7, 19, 20, 27, 30, 32, 35] show that co-designing simulation algorithms and system infrastructure can dramatically increase throughput by executing many environments simultaneously with shared memory and vectorized computation. This paradigm has proven highly effective for reinforcement learning workloads and other rollout-heavy workloads.

However, existing batched simulators primarily target rigid-body physics with lightweight rendering. By contrast, physics-based neural digital twins require substantially more complex pipelines, including deformable dynamics, dense interactions, and neural rendering tightly coupled with simulation. These characteristics introduce large memory footprints, high memory traffic, and synchronization overhead, making straightforward batching inefficient.

In this work, we introduce Boba (Fig. 1), the first batched simulation framework for physics-based Gaussian digital twins, built on top of the PhysTwin representation. Boba instantiates its core optimized pipeline in three configurations: BOBA-LOCAL for optimized local execution, BOBA-DISTRIBUTED for power- and latency-constrained XR deployment with an edge-server split, and

BOBA-BATCHED for high-throughput rollouts. Our key idea is to enable scalable neural digital twin simulation through **template–state separation** and system co-design. The static twin template, such as spring topology, rest geometry, and skinning metadata, is shared across time and across simulation instances, while only dynamic state updates change during execution. This structure allows simulation, deformation, and rendering to be **co-designed for batched execution**, reducing redundant computation, memory traffic, and synchronization overhead. Fig. 2 overviews Boba’s design. Guided by this principle, we address both algorithmic and systems-level bottlenecks. We first introduce compact surrogate spring-mass models that reproduce the behavior of much denser systems while reducing memory and compute by an order of magnitude. We further accelerate Gaussian deformation and rendering through mixed-precision skinning, memory-traffic-aware execution, and fused GPU operations. Finally, we redesign the batched simulation with shared memory layouts and contention-free parallel scheduling for efficient large-scale deformable-twin simulation.

Together, these optimizations enable fast local execution, latency- and power-efficient distributed XR deployment, and high-throughput batched execution. On a ~ 15 W embedded GPU, BOBA-LOCAL brings PhysTwin-style simulation to real-time performance within an XR-class edge-device power budget. For power- and latency-constrained XR deployment, BOBA-DISTRIBUTED reduces latency and incremental power relative to BOBA-LOCAL, leaving more edge-device headroom for richer XR applications and longer device use. Separately, on RTX 4090, BOBA-LOCAL provides the same-GPU reference for BOBA-BATCHED, whose higher throughput accelerates deformable-object predictive control.

Our main contributions are threefold: **(1)** We introduce the **first batched simulator for physics-based Gaussian digital twins**, enabling many episodes of the same reconstructed object to run efficiently on a single GPU. **(2)** We propose a **model–system co-design** for scalable twin simulation, including compact surrogate spring-mass models, efficient Gaussian deformation and rendering/visualization, and shared-memory-aware batching that reduces synchronization and memory bottlenecks. **(3)** We demonstrate system impact across XR deployment, batching, and robot planning: BOBA-LOCAL achieves over $10\times$ **single-instance speedup over PHYSTWIN** across embedded and desktop GPUs, BOBA-DISTRIBUTED achieves a $13.2\times$ **latency improvement** over PHYSTWIN on Orin with 22.2% **lower incremental power** than BOBA-LOCAL, BOBA-BATCHED achieves **3,310 FPS** aggregate throughput in batched execution on RTX 4090 (Fig. 3), and accelerates PHYSTWIN-based predictive control, including a $26\times$ speedup on rope and an estimated $> 2,000\times$ speedup on cloth with matched self-collision.

2 Related Work

Interactive Physical Digital Twins. The integration of neural scene representations with physics simulation has enabled digital twins that jointly model photorealistic appearance and physically consistent dynamics. Prior work commonly

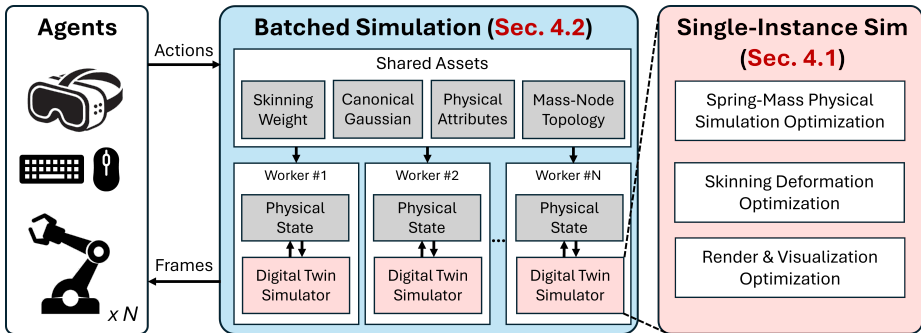


Fig. 2: Overview of Boba. Boba separates shared twin templates from per-instance dynamic state to enable high-throughput physics-based Gaussian digital twin simulation. Static assets are shared across workers, while each worker maintains its own physical state. The optimized single-instance pipeline serves as the base, and the batched framework adapts it across N independent episodes on a single GPU. For clarity, the figure shows Boba’s high-level organization and abstracts per-stage optimizations and the distributed communication path, which are detailed in the text and supplement.

adopts neural 3D parameterizations, including NeRF-based models [2, 5, 15] and Gaussian Splatting representations [14, 36, 39, 40], to build visually faithful and simulation-ready assets. Within this line of research, some methods [5, 14, 36] rely on hand-crafted or manually tuned physical parameters, which can introduce a gap between simulated responses and real-world motion. Other approaches [2, 15, 26, 39, 40] attempt to recover physical properties from video observations, yet often depend on synthetic data, limited excitation patterns, or dense multi-view geometry, limiting robustness and generality.

Recent progress has improved practical applicability. PhysTwin [13] represents a pivotal advance by learning physical digital twins directly from real human-object interaction videos. It integrates a spring-mass dynamics model with Gaussian Splatting for appearance reconstruction, enabling interactive simulation and subsequent robotics applications [38]. Follow-up works [3, 37] extend this paradigm to broader categories of physical effects, further validating the effectiveness of physics-grounded digital twins.

Nevertheless, prior works focus primarily on physical digital twin construction rather than computational scalability. They do not target low-power edge deployment or parallel simulation for MPC and RL. In contrast, we build upon PhysTwin with Boba, a multi-configuration system that targets different points in the throughput-latency-power trade-off, shifting the focus from fidelity alone to practical efficiency for XR and robotics.

Parallel Simulation Systems. High-throughput simulation has long been a focus in physics engine design, where runtime efficiency directly affects scalability and usability. Early GPU-centric engines, including Brax [6], Isaac Gym [18], and MuJoCo XLA (MJX) [34], introduced automatic batching, JIT compilation, and GPU-native execution for rigid-body physics. Later systems, such as Isaac Lab [19], ManiSkill3 [7, 20, 30, 35], Genesis [32], Madrona [27], and NVIDIA Newton [33], extended these ideas to modular, parallel, and differentiable workloads.

Although these systems achieve impressive throughput for rigid-body and modular primitives, they are not designed for batched deformable neural digital twins. Such simulations involve tightly coupled physics-based dynamics and the propagation of motion to large numbers of neural rendering primitives (e.g., Gaussians), introducing memory, synchronization, and numerical challenges beyond traditional engines. Building on PhysTwin [13], our work extends parallel simulation to this regime, supporting both energy-efficient single-instance execution and high-throughput batch rollouts.

3 Preliminaries: Physical Digital Twins

Physical digital twins aim to model real-world objects that both *look realistic* and *behave physically correctly* under interaction, enabling interactive simulation for XR, robotics, and gaming. Unlike static 3D reconstruction, a physical twin must support *dynamics*, *deformation*, and *rendering/visualization* in a unified pipeline. Modern neural twin systems therefore combine a physics simulator, a deformation model, and a differentiable renderer. The same differentiable pipeline is also used for inverse simulation, where observed interactions are used to estimate the twin state and parameters. We use recent *Gaussian-based* twins [13, 14, 36] as a concrete example to introduce notation.

Overview. A physical digital twin consists of two coupled representations. The *physical state* is $\mathbf{x}_t = \{\mathbf{x}_{t,i}\}_{i=1}^N$ with $\mathbf{x}_{t,i} \in \mathbb{R}^3$, describing the positions of physical elements at time t . Each element also carries a set of *physical attributes* $\boldsymbol{\theta}_i$ (e.g., mass, elasticity, damping, or other material parameters) that determine its dynamic behavior. The *visual representation* is a set of G 3D Gaussians with geometry and appearance parameters $\mathbf{g} = \{(\boldsymbol{\mu}_g, \Sigma_g, c_g, \alpha_g)\}_{g=1}^G$, where $\boldsymbol{\mu}_g$ is the Gaussian center, Σ_g encodes its anisotropic shape, c_g denotes color or appearance coefficients, and α_g denotes opacity. A skinning relation $\mathbf{w} = \{w_{ig}\}$ couples the two representations, allowing simulated motion to deform the Gaussians. Interaction is represented by an input $\mathbf{u}_t$, which may encode user-specified forces or constraints in XR, or action commands from an agent in robotics. The twin is executed as an unrolled process over time. At each step, the system advances the physical state using $\mathbf{u}_t$ over a simulation substep Δt , deforms the Gaussians, and renders an image. We use semicolons to separate dynamic state and input variables from fixed twin, rendering, and integration parameters:

$$\hat{I}_{t+1}(\mathbf{x}_t, \mathbf{u}_t; \boldsymbol{\theta}, \mathbf{w}, \mathbf{g}, \pi, \Delta t) = \text{Render}(\text{Deform}(\text{Sim}(\mathbf{x}_t, \mathbf{u}_t; \boldsymbol{\theta}, \Delta t), \mathbf{w}, \mathbf{g}), \pi),$$

where π denotes camera parameters, Sim advances the physical state from $\mathbf{x}_t$ given interaction $\mathbf{u}_t$, physical parameters $\boldsymbol{\theta} = \boldsymbol{\theta}_i$, and substep Δt , Deform maps the state to Gaussian geometry using $\mathbf{w}$ and $\mathbf{g}$, and Render produces an image.

Dynamical Simulation. The simulator maintains discrete physical states $\mathbf{x}_t$, while the underlying object dynamics are governed by a deformable model. A generic form is $m_i \ddot{\mathbf{x}}_i(\tau) = \mathbf{f}_i^{\text{int}}(\mathbf{x}(\tau), \dot{\mathbf{x}}(\tau); \boldsymbol{\theta}) + \mathbf{f}_i^{\text{ext}}(\mathbf{u}(\tau))$, where τ denotes continuous time, $\mathbf{f}^{\text{int}}$ models deformation forces determined by the physical parameters $\boldsymbol{\theta}$,

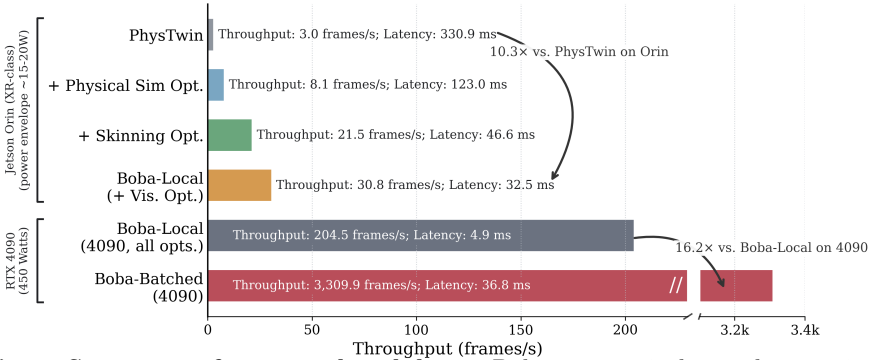


Fig. 3: System performance breakdown. Boba progressively accelerates single-instance execution on Orin, with BOBA-LOCAL achieving a 10.3× speedup over PHYS-TWIN. On RTX 4090, BOBA-LOCAL serves as the same-GPU single-instance baseline for BOBA-BATCHED; BOBA-BATCHED reaches 3,310 FPS average aggregate throughput, a 16.2× improvement over this baseline.

and $\mathbf{f}^{\text{ext}}$ represents external interaction effects induced by the input. The internal model may follow Spring-Mass, FEM, PBD, or MPM [12, 16, 17, 21, 28, 29]; discretizing and integrating with a substep Δt yields $\mathbf{x}_{t+1} = \text{Sim}(\mathbf{x}_t, \mathbf{u}_t; \boldsymbol{\theta}; \Delta t)$.

Deformation. The deformation stage maps the simulated state to the visual representation. For Gaussian twins, simulated motion is transferred to Gaussian parameters through linear blend skinning, which we write abstractly as $\mathbf{g}_{t+1} = \text{Deform}(\mathbf{x}_{t+1}, \mathbf{w}, \mathbf{g})$, where $\mathbf{x}_{t+1} = \text{Sim}(\mathbf{x}_t, \mathbf{u}_t; \boldsymbol{\theta}; \Delta t)$ and $\mathbf{g}_{t+1}$ denotes the deformed Gaussian parameters at time $t + 1$.

Rendering. Images are rendered via splatting by compositing contributions from deformed Gaussians along viewing rays, which we denote as $\text{Render}(\cdot)$.

Inverse Simulation. Given paired observations and interactions $\{(I_t, \mathbf{u}_t)\}_{t=1}^T$, inverse simulation estimates the state trajectory and twin parameters $(\mathbf{x}_{1:T}, \boldsymbol{\theta}, \mathbf{w}, \mathbf{g}, \pi)$ by minimizing $\sum_{t=1}^T \|I_t - \hat{I}_t\|$. Gradients propagate through Render , Deform , and Sim , enabling joint estimation of geometry, appearance, and physical parameters. While this pipeline enables realistic digital twins, the tightly coupled simulation, deformation, and rendering stages introduce substantial compute and memory overhead, motivating scalable simulation methods.

4 Method

Boba is a scalable framework for physics-based Gaussian digital twins of deformable objects, supporting latency- and power-constrained single-instance execution, either locally or with an edge-server split, and high-throughput batched simulation. Our key idea is to restructure and optimize the simulation pipeline around *template-state separation*, where the static twin template, such as spring topology, physical attributes, rest geometry, and skinning metadata, is shared across time and across simulation instances, while only dynamic state variables

change during execution. This design reduces redundant computation, memory traffic, and synchronization overhead that arise when directly scaling the PhysTwin pipeline, where spring-mass simulation and Gaussian deformation introduce substantial per-instance compute and data-movement costs. Guided by this idea, Boba introduces coordinated optimizations across the pipeline, including efficient spring-mass simulation, accelerated Gaussian deformation, optimized rendering/visualization and communication, and batching-aware execution kernels.

We first describe single-instance optimizations for spring-mass simulation and Gaussian deformation, together with the system-level optimizations that support local and distributed execution (Sec. 4.1). We then introduce batching-aware data layouts and kernels for high-throughput execution across many digital twin instances (Sec. 4.2). Detailed rendering/visualization and communication optimizations are provided in the supplementary material. Fig. 2 overviews Boba.

4.1 Single-Instance Efficient Twin Simulation

We first optimize the PhysTwin pipeline for efficient single-instance execution while preserving physical fidelity. Our goal is to reduce per-step compute and memory traffic so that a single digital twin can run interactively under tight latency and power budgets. Observing that deformable simulation exhibits both redundancy and modular computation, we organize Boba’s single-instance optimizations into three groups: (1) spring-mass simulation acceleration through spatiotemporal coarsening, rest-state collision pruning, and locality-aware execution, (2) efficient Gaussian deformation updates through faster rotation extraction, sparse skinning, and mixed-precision computation, and (3) system-level rendering/visualization and communication optimizations that reduce post-deformation overhead and enable BOBA-DISTRIBUTED’s edge-server path. We describe the first two groups in detail below because they form the shared simulation/deformation path used across Boba’s local, distributed, and batched configurations. We summarize the third group in this section, with additional details provided in the supplement.

Spring-Mass Optimization Boba reduces spring-mass computation with spatiotemporal coarsening, rest-state collision pruning, and locality-aware execution. Fig. 4 shows the spatiotemporal coarsening and rest-state collision-pruning steps.

Spatiotemporal Coarsening. We start by reducing the cost of deformable simulation through spatial and temporal coarsening of the spring-mass system. This opportunity arises from redundancy in deformable dynamics: nearby nodes often exhibit correlated motion and similar physical attributes, and deformation observed at the Gaussian level is often insensitive to small variations in node density or integration step. Let $(\mathbf{x}_t, \boldsymbol{\theta}, \mathbf{w}, \mathbf{g}, \Delta t)$ denote the original PhysTwin system, where $\mathbf{x}_t$ stores the spring-mass node positions. Our goal is to construct a compact spring-mass system $(\mathbf{x}'_t, \boldsymbol{\theta}', \mathbf{w}', \mathbf{g}, \Delta t')$ with significantly fewer nodes ($M \ll N$) and a coarser integration step $\Delta t' > \Delta t$ such that the resulting deformation remains visually consistent with the original model. Reducing the node count shrinks the dynamic state from $O(N)$ to $O(M)$ and correspondingly

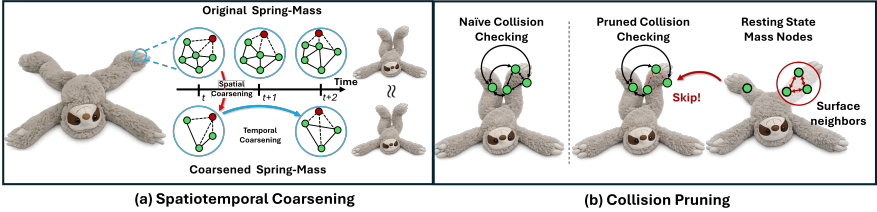


Fig. 4: Spring-mass optimization. (a) **Spatiotemporal coarsening** compresses the spring-mass system by reducing node resolution and simulation steps while preserving the resulting Gaussian deformation. (b) **Rest-state collision pruning** skips collision computations for node pairs that are nearby in rest space.

reduces spring-force evaluation and skinning computation. We therefore perform spatiotemporal coarsening by jointly reducing the spatial resolution M of the spring graph and increasing the simulation timestep to $\Delta t'$. The compact system parameters $(\theta', \mathbf{w}')$ are refit so that the resulting Gaussian trajectories match those of the original model. In practice, we optimize directly on rendered rollouts to ensure that visual behavior remains consistent with the original system.

Rest-state collision pruning. We then exploit locality to reduce unnecessary collision computation. Self-collision detection often dominates runtime in deformable simulation because many node pairs must be tested for potential interaction. Unlike rigid-body simulation, where convex proxies or bounding volumes can cheaply cull collisions, deformable surfaces lack stable proxies that remain valid under large deformation. We observe that nodes that are close in the rest configuration typically correspond to structural adjacency on the object and rarely produce meaningful collision impulses during simulation. We therefore precompute a symmetric rest-space pruning mask over node pairs. A pair (i, j) is flagged if its rest-space distance is below αd_{col} , where d_{col} is the collision distance and $\alpha > 1$ is a safety margin. During simulation, the collision kernel consults this mask and skips impulse computation for flagged pairs. The same pruning rule is applied during training and runtime to avoid train-runtime mismatch.

Locality-aware node ordering. We improve memory efficiency by exploiting spatial locality in the spring-mass system. Spring-force evaluation is often memory-bound because each spring references two endpoint nodes whose positions and velocities may be far apart in memory, leading to scattered global-memory reads on the GPU. To improve locality, we reorder nodes using a Morton (Z-order) key computed from their rest-space positions, which maps nearby 3D coordinates to nearby memory indices. This produces a permuted state vector $\tilde{\mathbf{x}} = P\mathbf{x}$ in which spatially nearby nodes occupy contiguous memory locations. We remap all spring endpoints to the reordered node indices and group springs into spatial blocks for evaluation. This permutation does not change the algorithmic complexity of spring-force computation, but it significantly improves memory-coalescing behavior and cache efficiency during GPU execution. Because the access pattern is determined by spring endpoint indices rather than the nodes' current positions, this preprocessing is performed only once and reused throughout simulation.

Skinning Optimization Boba accelerates skinning, where mass-node motion is propagated to Gaussian primitives via linear blend skinning (LBS). Each node needs a rigid transform, with translation from the spring-mass update and rotation from local deformation. At the scale of thousands of nodes and hundreds of thousands of Gaussians, LBS is bottlenecked by rotation extraction and memory traffic. Boba addresses these bottlenecks with three optimizations: rotation extraction with temporal reuse, sparse skinning with bounded influence sets, and mixed-precision LBS.

Rotation extraction with temporal reuse. The standard approach estimates node rotations using polar decomposition via an SVD of the 3×3 deformation matrix $\mathbf{F}$. While mathematically lightweight, general-purpose SVD implementations can incur substantial memory overhead because they materialize working copies and allocate temporary buffers, which becomes a bottleneck when executed for many nodes and batched instances. Boba replaces the SVD with a rotation-only solver based on the polar-decomposition identity. We compute $\mathbf{C} = \mathbf{F}^\top \mathbf{F}$, obtain $\mathbf{C}^{-1/2}$ through symmetric eigendecomposition with eigenvalue clamping, and recover the rotation as $\mathbf{R} = \mathbf{F}\mathbf{C}^{-1/2}$. The solver is implemented as a fused GPU kernel that reuses neighborhood data already loaded in memory and writes results directly in quaternion form. We further exploit temporal coherence by reusing rotations across simulation substeps. For each node, we cache the previous deformation matrix and rotation, and recompute the rotation only when the Frobenius-norm change $\|\mathbf{F} - \mathbf{F}_{\text{cache}}\|_F$ exceeds a threshold. This avoids redundant rotation extraction when local deformation changes minimally.

Sparse and mixed-precision skinning. We reduce the cost of Gaussian skinning by exploiting both sparsity and mixed precision in linear blend skinning (LBS). First, we limit the number of node influences used for each Gaussian. Since deformation is typically smooth at the Gaussian scale, blending transforms from a large number of mass nodes yields diminishing returns. Boba therefore selects the K_{lbs} nearest nodes in rest space and computes normalized skinning weights on this reduced set, significantly lowering both computation and memory traffic. Second, we adopt a mixed-precision implementation of LBS to reduce memory bandwidth. Cached node rotations and rest-pose skinning quantities are stored in FP16, while numerically sensitive operations—including deformation matrix construction, the rotation reuse test, and eigen-based rotation extraction—remain in FP32 for stability. Translation blending is also performed in FP32 to avoid accumulated rounding errors. The final Gaussian poses are cast back to FP32 before rendering to maintain compatibility with the Gaussian splatting renderer.

Rendering/visualization and distributed deployment. Boba also optimizes the post-deformation system path used to render and deliver updated twin states. In the local and distributed single-instance configurations, Boba keeps frame compositing and display transfer on the GPU where possible, reducing CPU-GPU synchronization and memory copies. In the distributed configuration, static template data are cached on both devices, and only compact dynamic Gaussian pose

updates are transmitted from the server to the edge device. These optimizations reduce rendering/visualization and communication overhead without changing the simulated state. Additional details are in the supplement.

4.2 High-Throughput Batched Digital Twin Simulation

Boba supports a throughput-oriented setting where many digital twin instances are simulated concurrently. Prior batched simulators primarily target rigid-body environments with low-dimensional per-episode state. In contrast, neural digital twins maintain high-dimensional deformation, geometry, and rendering state, making straightforward per-instance replication prohibitively expensive in memory and bandwidth. Moreover, scatter-heavy physics kernels suffer from severe atomic contention when executed across many instances. Boba addresses these challenges through a batched design that separates shared templates from dynamic state and co-designs force assembly for parallel execution.

Template-state separation. We adopt the template-state decomposition common in parallel simulation systems [27, 34] and extend it to deformable Gaussian digital twins. The shared template stores static data including spring topology, rest geometry, skinning metadata, Gaussian parameters, and rest-derived precomputations. Per-instance dynamic state—such as node positions, velocities, and cached rotations—is packed into contiguous batched arrays. GPU kernels index instance-specific segments through offsets while reusing the shared template. This avoids duplicating large static metadata across instances and enables efficient batched simulation with modest memory growth.

Conflict-free force accumulation. Spring-mass dynamics require accumulating forces from all incident springs at each node. In batched settings, naive implementations rely on atomic additions because many incident springs may concurrently add force contributions to the same node within each instance; repeated across many instances, these atomics become a major bottleneck. Boba eliminates atomics by decoupling force computation from force accumulation. First, a spring-parallel kernel computes each spring force once and stores it in a temporary buffer F_{temp} . Second, a node-parallel kernel gathers the forces of its incident springs using a precomputed adjacency map and accumulates them locally. Each adjacency entry stores a signed spring index indicating whether the node adds or subtracts that spring’s contribution. This gather-based assembly eliminates atomic contention and allows force accumulation to scale with node degree rather than the total number of springs. Combined with template sharing, the design enables high-throughput batched simulation of deformable digital twins.

5 Experiments

We evaluate Boba through three questions: **(1) Efficiency:** *Can it accelerate single-instance physics-based Gaussian digital twin simulation while preserving physical and visual fidelity, and can distributed execution further reduce latency*

and incremental power for XR-class deployment? **(2) Scalability:** Can it scale deformable digital twin simulation to large batch sizes with high GPU efficiency, avoiding bottlenecks such as atomic contention? **(3) System Impact:** Can it support high-throughput robot-control rollouts and real-time XR interaction under tight latency and power budgets? We build on PhysTwin as the base deformable twin model, but Boba’s optimizations, including spatiotemporal coarsening, efficient Gaussian deformation, and batching-aware simulation kernels, generalize to a broader class of physics-based Gaussian digital twins.

5.1 Experiment Settings

Hardware and network. We evaluate Boba on two hardware classes: an XR-class edge device and a desktop-class GPU system. For the edge device, we use an NVIDIA Jetson Orin [24], configured to emulate an XR-class compute budget by restricting CPU/GPU compute resources; we refer to this device as *Orin*. For the desktop-class GPU system, we use a workstation with an Intel i9 CPU [10] and an NVIDIA RTX 4090 GPU [22], which we refer to as *4090*. In the distributed setting, Orin serves as the edge device and the 4090 system serves as the server; Orin connects over Wi-Fi and the server connects via Ethernet. Full hardware and network details are in the supplement.

Evaluated Capabilities. We evaluate BOBA under two capabilities: single-instance execution and batched execution.

Single-instance execution. We evaluate three single-instance deployment configurations: (i) PHYSTWIN, which runs the original pipeline locally and is evaluated on both Orin and 4090; (ii) BOBA-LOCAL, which runs Boba’s optimized single-instance pipeline locally on both Orin and 4090 without batching-specific optimizations; and (iii) BOBA-DISTRIBUTED, Boba’s edge-server variant, with Orin as the edge device and 4090 as the server. Both BOBA-LOCAL and BOBA-DISTRIBUTED use an optimized rendering/visualization pipeline, and BOBA-DISTRIBUTED additionally uses communication-specific optimizations. Details of the communication and rendering/visualization are provided in the supplementary material.

Batched-instance execution. This scenario targets throughput-oriented execution where the server executes many instances concurrently. We compare: (i) PHYSTWIN-MULTIPROC, which scales by launching one independent single-instance process per simulated instance, with each process maintaining its own pipeline state and memory allocations; and (ii) BOBA-BATCHED, which executes multiple instances within a single process using Boba’s single-instance optimization stack together with batched-only extensions for shared state, conflict-free force accumulation, and batch-aware rendering/visualization. The batch-aware rendering/visualization path is detailed and evaluated in the supplement.

Datasets & Metrics. We use the same objects as PhysTwin [13]. For **single-instance evaluation**, we use the action sequences provided by PhysTwin. We report *end-to-end latency*; for the XR local/distributed setting, we additionally

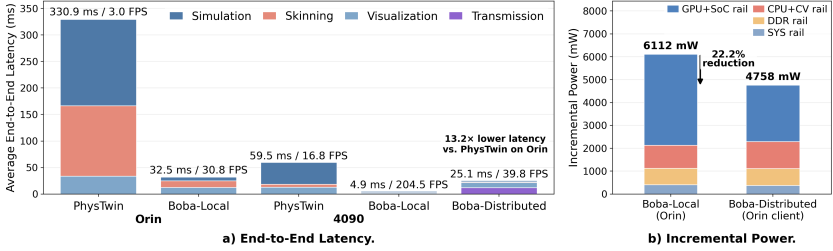


Fig. 5: Latency and incremental power across single-instance deployment settings.

report *incremental power*, measured on the Orin as workload-induced power above the idle baseline; server-side power is not included. For local configurations, latency includes simulation, skinning, and rendering/visualization on the target device. For BOBA-DISTRIBUTED, latency includes server-side simulation/skinning, transmission, and edge-side rendering/visualization. Power measurement details are provided in the supplementary material. For **single-instance quality evaluation**, we use the two quality tasks from PhysTwin: reconstruction & resimulation and future prediction. We report 3D metrics using single-direction Chamfer Distance and tracking error against manually annotated ground-truth points from PHYSTWIN, and 2D metrics using PSNR, SSIM, LPIPS, and silhouette IoU, evaluated at the center viewpoint and averaged across frames and cases.

For **batched-instance evaluation**, actions are generated using a random interaction generator that mimics human manipulation. Each batch element is an independent digital-twin instance with its own state and action. We report *aggregate throughput*, *per-step latency*, and *maximum batch capacity*. A batch step denotes applying one action input to each of the N instances, advancing simulation and deformation/skinning, and producing one rendered/visualized frame per instance; per-step latency is its wall-clock time. Aggregate throughput is total FPS across all N instances. For each case, we sweep batch size, measure aggregate FPS over the full run at each size, report the maximum, and average these maxima across cases. Maximum batch capacity is the largest feasible batch size for each case and batched implementation.

5.2 Single-Instance Results

Latency. Fig. 5a shows end-to-end latency across configurations. On Orin, BOBA-LOCAL reduces latency from 330.9 ms for PHYSTWIN to 32.5 ms (10.2 $\times$). On 4090, the same optimization stack reduces latency from 59.5 ms to 4.9 ms (12.1 $\times$), indicating that our local optimizations transfer across platforms. BOBA-DISTRIBUTED offloads simulation and skinning to the server while the edge applies the streamed updates and performs rendering/visualization, reducing latency to 25.1 ms. This is a further 1.3 $\times$ improvement over BOBA-LOCAL on Orin and a 13.2 $\times$ end-to-end improvement over PHYSTWIN on Orin, despite communication.

Power. Fig. 5b shows incremental power for BOBA-LOCAL and BOBA-DISTRIBUTED. BOBA-DISTRIBUTED reduces this power compared with BOBA-LOCAL on Orin, from 6112 mW to 4758 mW (22.2%). The largest measured reduction occurs on

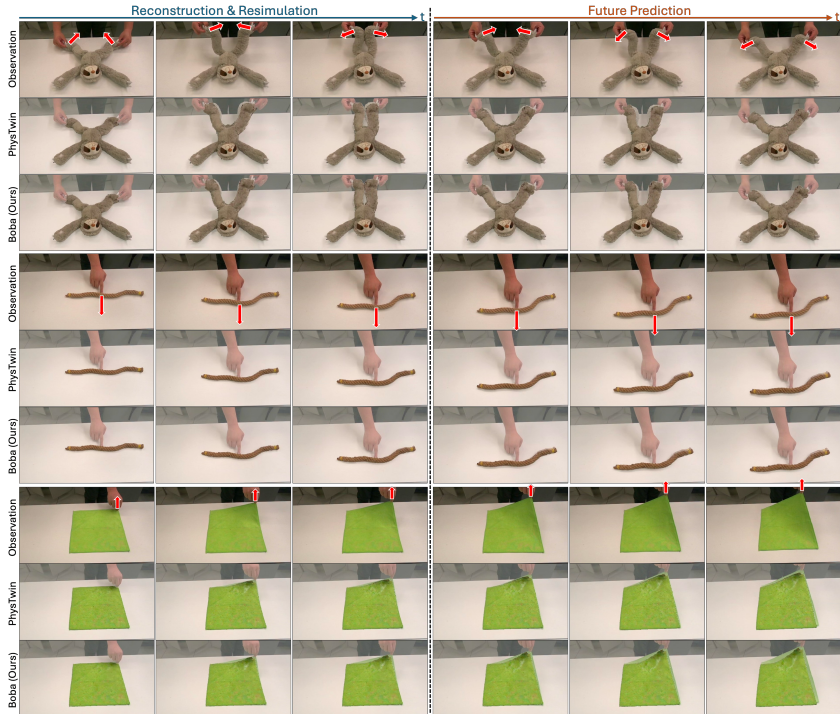


Fig. 6: Visual fidelity. Boba closely matches the PHYSTWIN baseline across reconstruction/resimulation and future prediction, with real observations shown as reference. Results use BOBA-LOCAL; local hardware affects performance but not the visual output.

Orin’s combined GPU+SoC rail, which reports the GPU and on-chip system logic together. Since Boba’s simulation and skinning stages are GPU-heavy, this reduction is consistent with moving these stages off the edge device, although the shared rail does not let us attribute the reduction to GPU and SoC components separately. Other reported rails, including CPU, memory, and system power, change more modestly, indicating that the remaining communication and state-reconstruction work does not offset the compute-side power savings. Overall, BOBA-DISTRIBUTED reduces latency while lowering device-side power, leaving more on-device compute headroom for concurrent XR tasks.

Fidelity. Fig. 6 and Table 1 summarize visual comparisons and objective fidelity metrics. Visually, Boba closely matches PHYSTWIN across reconstruction & resimulation and future prediction while preserving consistent dynamics. Across objective metrics, BOBA-LOCAL closely matches PHYSTWIN in both 3D and image-space metrics, indicating that our compute optimizations do not materially degrade physical or perceptual fidelity. BOBA-DISTRIBUTED, our distributed deployment design, remains close to BOBA-LOCAL across both reconstruction & resimulation and future prediction. This indicates that its communication and rendering/visualization pipeline introduces no measurable fidelity loss while enabling lower-latency and lower-power deployment.

Table 1: Quantitative results on reconstruction & resimulation and future prediction.

Task Method	Reconstruction & Resimulation						Future Prediction					
	CD↓	Track↓	IoU↑	PSNR↑	SSIM↑	LPIPS↓	CD↓	Track↓	IoU↑	PSNR↑	SSIM↑	LPIPS↓
PhysTwin	0.005	0.009	87.4	28.7	0.966	0.025	0.012	0.022	71.4	23.3	0.950	0.049
Boba-Local/Boba-Batched	0.006	0.009	89.6	28.4	0.963	0.031	0.014	0.022	75.5	23.6	0.953	0.050
Boba-Distributed	0.006	0.009	90.2	28.6	0.964	0.030	0.014	0.022	75.6	23.7	0.953	0.050

Table 2: Batch capacity and throughput by parallelization strategy.

Method	Avg. Batch Capacity	Min Batch Capacity	Max Batch Capacity	Throughput (FPS)
PhysTwin-MultiProc	11.0	4	27	44.7
Boba-Batched (Template-share only)	369.2	213	688	2392.8
Boba-Batched (+ Atomic-free)	368.3	210	688	3309.9

5.3 Batch Scalability and Ablations

Batch scalability. As shown in Table 2, batching-specific optimizations substantially improve scalability and throughput. Relative to PHYSTWIN-MULTIPROC, template sharing alone increases average batch capacity from 11.0 to 369.2 (33.6 $\times$) and improves throughput from 44.7 to 2392.8 frames/s (53.5 $\times$). Adding atomic-free scheduling on top yields the full BOBA-BATCHED design: capacity remains nearly unchanged, with average capacity changing only from 369.2 to 368.3 and maximum capacity remaining 688, while increasing throughput to 3309.9 frames/s, a further 38.3% improvement over template sharing alone. The slight capacity decrease comes from the extra index mapping required by the atomic-free path, but this small memory cost is outweighed by the throughput gain. Overall, template sharing drives capacity gains, while atomic-free scheduling further improves throughput by avoiding atomic contention.

Compute and Batching Breakdown. Fig. 3 summarizes local optimization results on Orin and 4090 and batched execution on 4090 in throughput and latency. We exclude BOBA-DISTRIBUTED to isolate local and batched execution from edge-server communication; detailed per-optimization results are in the supplement. On Orin, PHYSTWIN runs at 3.0 frames/s; physical simulation, skinning, and rendering/visualization optimizations raise throughput to 8.1, 21.5, and 30.8 frames/s, yielding a 10.3 $\times$ end-to-end speedup for BOBA-LOCAL. On 4090, the same local stack improves PHYSTWIN from 16.8 to 204.5 frames/s, providing the optimized same-GPU single-instance reference. BOBA-BATCHED adds batching-specific execution and the throughput-oriented batched rendering/visualization path detailed in the supplement, reaching 3309.9 aggregate frames/s, a 16.2 $\times$ system-level throughput improvement over this reference.

5.4 Application

As shown in Fig. 7, Boba supports downstream applications with different system requirements; here we focus on robot planning. Using BOBA-BATCHED as the rollout simulator in the same MPC setting as PHYSTWIN, where MPC samples candidate action sequences, simulates rollouts, and executes the action that best moves the deformable object toward the target, we observe substantial gains from

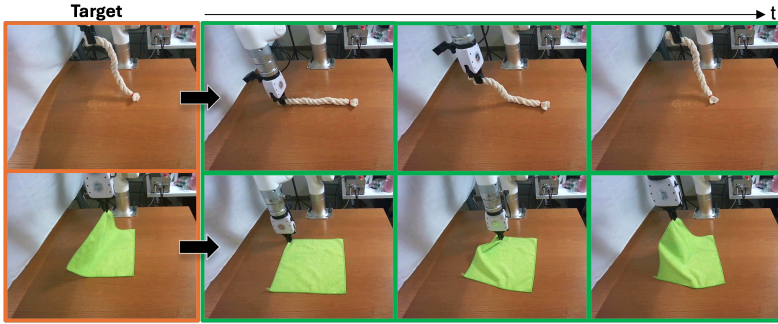


Fig. 7: Robot-planning application. BOBA-BATCHED serves as the MPC rollout simulator for rope and cloth manipulation, enabling high-throughput parallel rollout.

Boba’s batching-friendly design. BOBA-BATCHED achieves a $26\times$ speedup on rope. For cloth, under matched collision settings, PHYSTWIN-based planning with self-collision takes over 24 hours; based on the first 100 samples, BOBA-BATCHED provides an estimated speedup of over $2000\times$. Even without self-collision, it still achieves more than $16\times$ speedup. This gap primarily comes from avoiding repeated collision-graph construction and serialization overhead in PHYSTWIN, highlighting Boba’s advantage for realistic, larger-scale robot-planning workloads. The supplement also shows a Quest 3 XR prototype for controller-based manipulation of a simulated Gaussian physical twin in a reconstructed indoor scene, and a PPO training example using Boba as a high-throughput RL simulator.

6 Conclusion

We introduced **Boba**, a scalable framework for physics-based Gaussian digital twins of deformable objects. Built on template–state separation, Boba co-designs simulation, deformation, rendering/visualization, and batching to reduce overhead while preserving PHYSTWIN-level fidelity. Its three configurations cover complementary regimes: BOBA-LOCAL achieves over $10\times$ speedup over PHYSTWIN on XR-class Orin; BOBA-DISTRIBUTED, using an edge-server split, reaches 25.1 ms latency, a $13.2\times$ improvement over PHYSTWIN on Orin, with 22.2% lower incremental power than BOBA-LOCAL; and BOBA-BATCHED reaches 3,310 FPS aggregate throughput on RTX 4090 and accelerates robot planning, with a measured $26\times$ speedup on rope and an estimated $> 2,000\times$ speedup on cloth with matched self-collision. These results position Boba as a practical substrate for power-constrained XR interaction and scalable deformable-object robot planning.

7 Acknowledgments

The authors thank the anonymous reviewers for their valuable feedback and Benjamin Goldfisher for early project assistance. This work was supported in part by Meta Platforms Technologies, LLC and the National Science Foundation (NSF) grants 2217144, 2312102, 2331878, 2340254, 2414227, and 2404385.

Bibliography

- [1] tegrastats utility. NVIDIA Docs (2024), https://docs.nvidia.com/drive/drive_os_5.1.6.1L/nvlib_docs/index.html#page/DRIVE_OS_Linux_SDK_Development_Guide/Utilities/util_tegrastats.html 25
- [2] Chen, H.y., Tretschk, E., Stuyck, T., Kadlecsek, P., Kavan, L., Vouga, E., Lassner, C.: Virtual elastic objects. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 15827–15837 (2022) 4
- [3] Chen, Y., Hu, Y., Sun, L., Kusnur, T., Herlant, L., Jiang, C.: Empm: Embodied mpm for modeling and simulation of deformable objects. arXiv preprint arXiv:2601.17251 (2026) 4
- [4] Fang, G., Wang, B.: Mini-splatting: Representing scenes with a constrained number of gaussians. In: European conference on computer vision. pp. 165–181. Springer (2024) 22
- [5] Feng, Y., Shang, Y., Li, X., Shao, T., Jiang, C., Yang, Y.: Pie-nerf: Physics-based interactive elastodynamics with nerf. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 4450–4461 (2024) 4
- [6] Freeman, C.D., Frey, E., Raichuk, A., Girgin, S., Mordatch, I., Bachem, O.: Brax—a differentiable physics engine for large scale rigid body simulation. arXiv preprint arXiv:2106.13281 (2021) 4
- [7] Gu, J., Xiang, F., Li, X., Ling, Z., Liu, X., Mu, T., Tang, Y., Tao, S., Wei, X., Yao, Y., Yuan, X., Xie, P., Huang, Z., Chen, R., Su, H.: Maniskill2: A unified benchmark for generalizable manipulation skills. In: International Conference on Learning Representations (2023) 2, 4
- [8] Hanson, A., Tu, A., Lin, G., Singla, V., Zwicker, M., Goldstein, T.: Speedy-splat: Fast 3d gaussian splatting with sparse pixels and sparse primitives. In: Proceedings of the Computer Vision and Pattern Recognition Conference. pp. 21537–21546 (2025) 22
- [9] Hanson, A., Tu, A., Singla, V., Jayawardhana, M., Zwicker, M., Goldstein, T.: Pup 3d-gs: Principled uncertainty pruning for 3d gaussian splatting. In: Proceedings of the Computer Vision and Pattern Recognition Conference. pp. 5949–5958 (2025) 22
- [10] Intel: Intel core i9-13900kf processor. Intel Products (2022), <https://www.intel.com/content/www/us/en/products/sku/230497/intel-core-i913900kf-processor-36m-cache-up-to-5-80-ghz/specifications.html>, accessed: 30 June 2026 11, 24
- [11] Jia, Y., Zhang, H., Zhang, Z., Wu, J., Yu, M., Wang, Z., Jiang, D., Li, Z., Cao, C., Yu, Z., et al.: Gs-playground: A high-throughput photorealistic simulator for vision-informed robot learning. arXiv preprint arXiv:2604.25459 (2026) 22

- [12] Jiang, C., Schroeder, C., Teran, J., Stomakhin, A., Selle, A.: The material point method for simulating continuum materials. In: *Acm siggraph 2016 courses*, pp. 1–52 (2016) [6](#)
- [13] Jiang, H., Hsu, H.Y., Zhang, K., Yu, H.N., Wang, S., Li, Y.: Phystwin: Physics-informed reconstruction and simulation of deformable objects from videos. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. pp. 7219–7230 (2025) [2](#), [4](#), [5](#), [11](#)
- [14] Jiang, Y., Yu, C., Xie, T., Li, X., Feng, Y., Wang, H., Li, M., Lau, H., Gao, F., Yang, Y., et al.: Vr-gs: A physical dynamics-aware interactive gaussian splatting system in virtual reality. In: *ACM SIGGRAPH 2024 Conference Papers*. pp. 1–1 (2024) [2](#), [4](#), [5](#)
- [15] Li, X., Qiao, Y.L., Chen, P.Y., Jatavallabhula, K.M., Lin, M., Jiang, C., Gan, C.: Pac-nerf: Physics augmented continuum neural radiance fields for geometry-agnostic system identification. *arXiv preprint arXiv:2303.05512* (2023) [4](#)
- [16] Liu, T., Bargteil, A.W., O’Brien, J.F., Kavan, L.: Fast simulation of mass-spring systems. *ACM Transactions on Graphics (TOG)* **32**(6), 1–7 (2013) [6](#)
- [17] Macklin, M., Müller, M., Chentanez, N.: Xpbd: position-based simulation of compliant constrained dynamics. In: *Proceedings of the 9th International Conference on Motion in Games*. pp. 49–54 (2016) [6](#)
- [18] Makoviychuk, V., Wawrzyniak, L., Guo, Y., Lu, M., Storey, K., Macklin, M., Hoeller, D., Rudin, N., Allshire, A., Handa, A., State, G.: Isaac gym: High performance gpu-based physics simulation for robot learning (2021) [4](#)
- [19] Mittal, M., Roth, P., Tigue, J., Richard, A., Zhang, O., Du, P., Serrano-Muñoz, A., Yao, X., Zurbrugg, R., Rudin, N., Wawrzyniak, L., Rakhsha, M., Denzler, A., Heiden, E., Borovicka, A., Ahmed, O., Akinola, I., Anwar, A., Carlson, M.T., Feng, J.Y., Garg, A., Gasoto, R., Gulich, L., Guo, Y., Gussert, M., Hansen, A., Kulkarni, M., Li, C., Liu, W., Makoviychuk, V., Malczyk, G., Mazhar, H., Moghani, M., Murali, A., Noseworthy, M., Poddubny, A., Ratliff, N., Rehberg, W., Schwarke, C., Singh, R., Smith, J.L., Tang, B., Thaker, R., Trepte, M., Wyk, K.V., Yu, F., Millane, A., Ramasamy, V., Steiner, R., Subramanian, S., Volk, C., Chen, C., Jawale, N., Kuruttukulam, A.V., Lin, M.A., Mandlekar, A., Patzwaldt, K., Welsh, J., Zhao, H., Anes, F., Lafleche, J.F., Moënné-Loccoz, N., Park, S., Stepinski, R., Gelder, D.V., Amevor, C., Carius, J., Chang, J., Chen, A.H., de Heras Ciechowski, P., Daviet, G., Mohajerani, M., von Mural, J., Reutsky, V., Sauter, M., Schirm, S., Shi, E.L., Terdiman, P., Vilella, K., Widmer, T., Yeoman, G., Chen, T., Grizan, S., Li, C., Li, L., Smith, C., Wiltz, R., Alexis, K., Chang, Y., Chu, D., Fan, L.J., Farshidian, F., Handa, A., Huang, S., Hutter, M., Narang, Y., Pouya, S., Sheng, S., Zhu, Y., Macklin, M., Moravanszky, A., Reist, P., Guo, Y., Hoeller, D., State, G.: Isaac lab: A gpu-accelerated simulation framework for multi-modal robot learning. *arXiv preprint arXiv:2511.04831* (2025), <https://arxiv.org/abs/2511.04831>, accessed: 30 June 2026 [2](#), [4](#)

- [20] Mu, T., Ling, Z., Xiang, F., Yang, D., Li, X., Tao, S., Huang, Z., Jia, Z., Su, H.: Maniskill: Generalizable manipulation skill benchmark with large-scale demonstrations. arXiv preprint arXiv:2107.14483 (2021) [2](#), [4](#)
- [21] Müller, M., Heidelberger, B., Hennix, M., Ratcliff, J.: Position based dynamics. *Journal of Visual Communication and Image Representation* **18**(2), 109–118 (2007) [6](#)
- [22] NVIDIA: GeForce RTX 4090 Family. NVIDIA Products (2022), <https://www.nvidia.com/en-us/geforce/graphics-cards/40-series/rtx-4090/>, accessed: 30 June 2026 [11](#), [24](#)
- [23] NVIDIA: NVIDIA RTX PRO 6000 Blackwell Series. NVIDIA Products (2022), <https://www.nvidia.com/en-us/products/workstations/professional-desktop-gpus/rtx-pro-6000-family/>, accessed: 30 June 2026 [24](#)
- [24] NVIDIA: NVIDIA AGX Orin. NVIDIA Robotics and Edge AI (2024), <https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/>, accessed: 30 June 2026 [11](#), [24](#)
- [25] NVIDIA: Software-based power consumption modeling. NVIDIA Jetson Linux Developer Guide (2024), <https://docs.nvidia.com/jetson/archives/r35.1/DeveloperGuide/text/SD/PlatformPowerAndPerformance.html>, accessed: 30 June 2026 [25](#)
- [26] Qiao, Y.L., Gao, A., Lin, M.: Neuphysics: Editable neural geometry and physics from monocular videos. *Advances in Neural Information Processing Systems* **35**, 12841–12854 (2022) [4](#)
- [27] Shacklett, B., Rosenzweig, L.G., Xie, Z., Sarkar, B., Szot, A., Wijmans, E., Koltun, V., Batra, D., Fatahalian, K.: An extensible, data-oriented architecture for high-performance, many-world simulation. *ACM Trans. Graph.* **42**(4) (2023) [2](#), [4](#), [10](#)
- [28] Sifakis, E., Barbic, J.: Fem simulation of 3d deformable solids: a practitioner’s guide to theory, discretization and model reduction. In: *Acm siggraph 2012 courses*, pp. 1–50 (2012) [6](#)
- [29] Sulsky, D., Chen, Z., Schreyer, H.L.: A particle method for history-dependent materials. *Computer methods in applied mechanics and engineering* **118**(1-2), 179–196 (1994) [6](#)
- [30] Tao, S., Xiang, F., Shukla, A., Qin, Y., Hinrichsen, X., Yuan, X., Bao, C., Lin, X., Liu, Y., kai Chan, T., Gao, Y., Li, X., Mu, T., Xiao, N., Gurha, A., Rajesh, V.N., Choi, Y.W., Chen, Y.R., Huang, Z., Calandra, R., Chen, R., Luo, S., Su, H.: Maniskill3: Gpu parallelized robotics simulation and rendering for generalizable embodied ai. *Robotics: Science and Systems* (2025) [2](#), [4](#)
- [31] Team, A.: Alvr - air light vr. GitHub (2026), <https://github.com/alvr-org/ALVR>, accessed: 30 June 2026 [31](#)
- [32] Team, G.A.: The role of simulation in scalable robotics, genesis world 1.0, and the path forward. *Genesis AI Blog* (May 2026), <https://www.genesis.ai/blog/the-role-of-simulation-in-scalable-robotics-genesis-world-10-and-the-path-forward>, accessed: 30 June 2026 [2](#), [4](#)

- [33] The Newton Contributors: Newton: Gpu-accelerated physics simulation for robotics and simulation research. <https://github.com/newton-physics/newton> (Apr 2025), accessed: 30 June 2026 4, 26
- [34] Todorov, E., Erez, T., Tassa, Y.: Mujoco: A physics engine for model-based control. In: 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 5026–5033. IEEE (2012). <https://doi.org/10.1109/IROS.2012.6386109>, accessed: 30 June 2026 4, 10
- [35] Xiang, F., Qin, Y., Mo, K., Xia, Y., Zhu, H., Liu, F., Liu, M., Jiang, H., Yuan, Y., Wang, H., et al.: Sapien: A simulated part-based interactive environment. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. pp. 11097–11107 (2020) 2, 4
- [36] Xie, T., Zong, Z., Qiu, Y., Li, X., Feng, Y., Yang, Y., Jiang, C.: Physgaussian: Physics-integrated 3d gaussians for generative dynamics. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 4389–4398 (2024) 2, 4, 5
- [37] Yang, Y., Zhang, Z., Zhang, X., Zeng, Y., Li, H., Zuo, W.: Physworld: From real videos to world models of deformable objects via physics-aware demonstration synthesis. arXiv preprint arXiv:2510.21447 (2025) 4
- [38] Zhang, K., Sha, S., Jiang, H., Loper, M., Song, H., Cai, G., Xu, Z., Hu, X., Zheng, C., Li, Y.: Real-to-sim robot policy evaluation with gaussian splatting simulation of soft-body interactions. ICRA (2026) 4
- [39] Zhang, T., Yu, H.X., Wu, R., Feng, B.Y., Zheng, C., Snavely, N., Wu, J., Freeman, W.T.: Physdreamer: Physics-based interaction with 3d objects via video generation. In: European Conference on Computer Vision. pp. 388–406. Springer (2024) 4
- [40] Zhong, L., Yu, H.X., Wu, J., Li, Y.: Reconstruction and simulation of elastic objects with spring-mass 3d gaussians. In: European Conference on Computer Vision. pp. 407–423. Springer (2024) 4

Supplemental Material

8 Boba System Variants

Figure 8 summarizes the three deployment variants supported by Boba. From left to right, the figure shows **Boba-Local**, **Boba-Distributed**, and **Boba-Batched**. These variants differ in where the pipeline stages execute and in whether the system targets local single-machine execution, edge-server deployment, or high-throughput batched simulation. The application illustrations in the figure are representative examples rather than exclusive use cases; the same deployment modes can support a range of XR and robotic workloads.

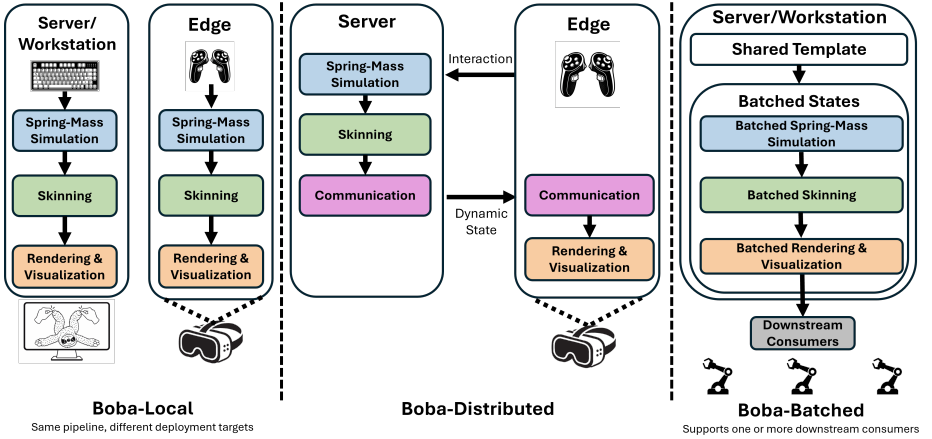


Fig. 8: Deployment variants supported by Boba. **Left: Boba-Local** runs the full Boba pipeline on a single machine, including spring-mass simulation, skinning, and rendering/visualization, supporting standalone interactive deployment on either an edge device or a workstation. **Middle: Boba-Distributed** splits the pipeline between a server and an edge device: the server performs simulation and skinning, while the edge device executes rendering/visualization, supporting interactive deployment with lower end-to-end latency and reduced device-side power. **Right: Boba-Batched** executes one batched pipeline over many instances, including batched spring-mass simulation, skinning, and rendering/visualization, on a GPU to maximize aggregate throughput.

Boba-Local (left) runs the full Boba pipeline on a single machine, including spring-mass simulation, skinning, and rendering/visualization. For XR deployment, this machine is typically the edge device, supporting local execution when distributed offloading is unavailable or undesirable. The same single-machine pipeline can also be executed on a server or workstation, where it provides the optimized single-instance baseline used to evaluate batched execution.

Boba-Distributed (middle) splits the pipeline between the server and the edge device to further reduce end-to-end latency while lowering device-side power,

which is especially important for battery-limited XR deployments. The server performs spring-mass simulation and skinning, while the edge device executes rendering/visualization. This design reduces device-side compute and power consumption, while leaving more edge-device compute headroom for concurrent XR tasks. Static template data are cached on both devices, and only compact dynamic Gaussian pose updates are streamed to the edge for rendering/visualization.

Boba-Batched (right) targets high-throughput batched execution on a single GPU. Rather than launching separate pipelines for each instance, BOBA-BATCHED runs one batched pipeline over many independent per-instance states while reusing a shared template. The batched pipeline includes batched spring-mass simulation, batched skinning, and batched rendering/visualization, maximizing aggregate throughput with modest memory growth. The resulting rollout data and visual outputs can support downstream consumers such as robot planning, RL training, and multi-user XR.

9 Rendering/Visualization Optimization

9.1 Rendering Optimization

This subsection details the rendering-side optimizations used in Boba. We organize them by scope. First, we describe renderer-level optimizations that reduce rasterization work and benefit both single-instance and batched execution. We then describe Boba’s template–state rendering layout, which enables static Gaussian attributes to be shared efficiently in BOBA-BATCHED. Finally, we describe throughput-oriented batched rendering choices, including object-adaptive Gaussian pruning and a batched camera path.

Rasterization work reduction. Boba reduces redundant rasterization work through several renderer-level optimizations. First, opacity-aware projection tightens the projected tile bounds for low-opacity Gaussians, reducing unnecessary tile work. Second, an ellipse-aware tile filter checks whether the actual projected Gaussian ellipse overlaps a tile rather than accepting all tiles in the rectangular AABB, reducing false tile intersections. Third, for the common rendering configuration with RGB output and degree-3 spherical harmonics, Boba fuses projection with SH color materialization, avoiding a separate Python/Torch gather and SH evaluation step. These optimizations reduce rendering cost before and during rasterization and are useful for both single-instance and batched rendering.

Template–state-aware batched renderer. The main paper describes template–state separation for batched simulation; here we apply the same principle inside the Gaussian splatting backend. Static Gaussian attributes shared across instances, such as SH/features, opacity, scale, and rest-space metadata, are kept in shared template buffers. Per-instance dynamic pose attributes produced by skinning, such as deformed centers and orientations, are stored in batched state tensors and accessed through instance identifiers during projection and rasterization. This lets BOBA-BATCHED render one image per instance without materializing a full copy of the Gaussian attribute arrays for every rollout.

Object-adaptive Gaussian pruning. Inspired by recent efficient 3D Gaussian pruning strategies [4, 8, 9], Boba uses pruning as a throughput-oriented preprocessing step for batched rendering. Rather than applying a single pruning ratio across all objects, we use an object-adaptive strategy: we observe that cloth scenes are more sensitive to aggressive pruning, while non-cloth objects tolerate stronger reduction with limited visual degradation. Based on this tradeoff, non-cloth objects retain 30% of Gaussians, while cloth objects retain 55%. In our pruning-quality comparison, pruning changes PSNR by at most 0.51 dB and SSIM by at most 0.007 across train/test averages; LPIPS changes by at most 0.0064, and IoU changes by at most 0.7 points. This suggests that object-adaptive pruning preserves comparable image-space quality while reducing the number of Gaussians processed by the batched renderer. Since pruning is performed once per object and reused across subsequent batched runs, its preprocessing cost is amortized and does not affect per-run throughput.

Batched camera path. For single-instance execution and quality comparison, Boba follows the original PHYSTWIN rendering resolution of (848×480) . For BOBA-BATCHED, however, the goal is throughput-oriented rollout rendering and preview rather than high-resolution final display, so Boba renders each instance at a fixed moderate preview resolution of 640×480 , which is in line with practical interactive Gaussian-rendering setups such as GS-Playground [11]. This keeps per-instance rendering cost predictable and comparable across batch sizes. To make the batched outputs visually comparable across instances, Boba recenters each instance by subtracting its instance offset before rendering, so all instances appear in the same calibrated camera frame rather than being diagonally shifted in world space. Under the evaluated batched rendering configuration, BOBA-BATCHED closely matches PHYSTWIN, with case-mean SSIM/LPIPS of 0.964/0.019, indicating minimal visual difference from the corresponding PHYSTWIN outputs.

9.2 Visualization Optimization

This subsection details the post-render visualization optimization stage used in the BOBA-LOCAL and BOBA-DISTRIBUTED pipelines. BOBA-BATCHED uses an adapted, lighter-weight, batch-specialized variant of this post-render visualization path after batched rendering.

In the original PHYSTWIN implementation, after Gaussian rendering, the post-render visualization path composites the rendered RGBA frame with the background, applies shadow attenuation, and displays the final frame through CPU/OpenCV-style operations that require GPU-to-CPU transfers and CPU-side image processing. While straightforward to implement, this path adds overhead that matters for latency-sensitive interactive deployment and would be amplified in batched execution if repeated across many instances.

To reduce this overhead, Boba keeps the post-render visualization path on the GPU whenever possible. After Gaussian rendering, frame compositing is performed directly on GPU tensors using alpha blending, avoiding CPU-side image assembly and format conversion. Shadow compositing is likewise implemented on

the GPU with a parallel formulation that processes multiple lights concurrently. To further reduce cost, shadow masks are generated at a downsampled resolution, refined with lightweight morphological operations, and then upsampled to the final display resolution. The per-light masks are combined into a single attenuation map and applied to the composited frame using element-wise GPU operations.

At the final display stage, instead of copying the rendered frame back to the CPU for display, Boba transfers the final RGBA image directly from CUDA memory to an OpenGL pixel buffer object and displays it through a GPU-resident texture update. This keeps Gaussian rendering, frame compositing, shadow compositing, and final display within GPU memory, improving locality and reducing synchronization and data-transfer overhead along the interactive visualization path.

For BOBA-BATCHED, Boba uses a lighter-weight batch-specialized visualization path. After batched rendering, the per-instance RGBA frames are kept in a batched tensor layout and alpha-composited with the background in parallel. Because this path is used for throughput-oriented rollout rendering and preview rather than single-instance interactive display, it skips shadow-related computation, which does not affect the simulated state or Gaussian pose updates. The composited frames are then packed into a grid-style visualization output, allowing many rollout instances to be inspected together while avoiding repeated per-instance CPU/OpenCV display operations.

10 Communication Optimization

This section details the communication optimization used in BOBA-DISTRIBUTED. In BOBA-DISTRIBUTED, the server advances simulation and skinning, while the edge device performs rendering/visualization. Boba reduces communication by separating static data cached on both sides from per-frame dynamic updates. For the Gaussian rendering state, after each interaction step the server transmits only the dynamic pose attributes needed to render the updated digital twin: the updated Gaussian centers and orientations. Static template data, including the rest configuration, Gaussian appearance attributes, and other rendering metadata, are cached on both sides and are therefore not retransmitted.

At each interaction step, after simulation and skinning, the server obtains the updated dynamic Gaussian pose state, consisting of Gaussian centers and orientations. To reduce communication time, Boba avoids sending this full `float32` pose state at every step and instead uses an adaptive snapshot/residual policy. The first step is sent as an absolute `float32` snapshot. For later steps, the server maintains a transport state that mirrors the Gaussian pose state the edge device would reconstruct. Before sending a candidate `float16` residual, the server locally simulates the edge-side reconstruction by adding the quantized residual to this transport state, and computes the reconstruction error against the current true Gaussian pose state. If the predicted reconstruction error exceeds a configured threshold, or if a maximum snapshot interval is reached, Boba sends a new absolute `float32` snapshot for that step; otherwise, it sends the compact `float16`

residual and updates the transport state to the predicted reconstruction. Thus, most interaction steps transmit fewer bytes through residual updates, while occasional snapshots reset accumulated quantization error when needed. This policy changes only the encoded representation transmitted to the edge device; the server always advances simulation and skinning using the full-precision state.

11 Additional Experiment Details

11.1 Hardware and Network Configuration

We use an NVIDIA Jetson Orin [24] as the edge device. Direct evaluation on commercial standalone XR headsets is challenging because many headsets are closed platforms, limiting the low-level pipeline modifications and fine-grained timing and power measurements required for our evaluation. An unconstrained Orin provides more compute and power headroom than typical standalone XR devices. We therefore configure the Orin to approximate an XR-class compute and power budget while retaining precise control over hardware resources. Specifically, we configure the Orin to use 8 CPU cores, 3 GPU TPCs (6 SMs, 768 CUDA cores), a maximum CPU frequency of 2201 MHz, a maximum GPU frequency of 1300 MHz, and a memory-controller frequency of 2133 MHz.

The server/workstation system uses an Intel i9 CPU [10] and either an NVIDIA RTX 4090 GPU [22] or an NVIDIA RTX PRO 6000 Blackwell SE GPU [23], depending on the experiment. The RTX 4090 is used for the server/workstation measurements reported in the main paper, including BOBA-DISTRIBUTED and the RTX 4090 single-instance and batched results. We include RTX 6000 Blackwell only as an additional supplementary comparison to characterize Boba’s throughput on a recent high-end workstation-class GPU. Table 3 summarizes the GPU configurations used in our evaluation.

Table 3: GPU configurations used in our evaluation.

GPU	Arch.	CUDA Cores	Memory (GB)	Max GPU MHz
Jetson Orin	Ampere	768	64	1300
NVIDIA GeForce RTX 4090	Ada	16384	24	3105
NVIDIA RTX PRO 6000 Blackwell SE	Blackwell	24064	96	2430

Note: For Jetson Orin, the CUDA core count and maximum GPU frequency reflect the XR-proxy configuration used in our evaluation, not the full Orin hardware capability.

In BOBA-DISTRIBUTED, the Orin connects over Wi-Fi to the same local network as the server, while the server connects over Ethernet. Communication between the two devices uses TCP sockets. In this local edge-server testbed, we measure an average bandwidth of 670 Mbps and an average round-trip time (RTT) of 3 ms. This low RTT reflects a local edge-server deployment rather than a wide-area or cloud deployment.

Power Measurement. We measure device power on the Jetson Orin using the *tegrastats* utility [1], sampled at 10 ms intervals. *Tegrastats* reports power consumption across multiple Jetson power rails, including CPU+CV, GPU+SoC, DDR, and SYS [25]. Here, CV refers to computer-vision accelerator components, SoC refers to on-chip system logic, DDR refers to DRAM, and SYS covers system-level components such as display, storage, and I/O. We sum the reported rails to obtain total device power and average over the execution window of each experiment. Because some rails aggregate multiple subsystems and may include background/platform activity, we report *incremental device power*, defined as workload-induced device power above the idle baseline. For brevity, we refer to this metric as *incremental power* throughout the paper.

For each evaluated Orin-side configuration, we first measure idle power by leaving the Jetson Orin powered on under the same configuration for more than 10 minutes before launching the experiment. During this idle window, no experiment workload is running. We average the reported total device power over this idle window and subtract it from the average total device power measured during execution:

$$P_{\text{inc}} = \overline{P}_{\text{exec}} - \overline{P}_{\text{idle}}.$$

This subtracts the static platform power of the configured Jetson setup and better isolates execution-related power differences.

For BOBA-DISTRIBUTED, the reported incremental power includes edge-side rendering/visualization, communication, and reconstruction of the dynamic Gaussian pose state from received snapshots or residual updates. Although the Jetson platform does not expose isolated power measurements for the Wi-Fi module, the reported incremental power includes any communication-related device power captured by the monitored Jetson rails during the full distributed workload. For BOBA-LOCAL, incremental power reflects the full on-device workload, including simulation, skinning, and rendering/visualization.

We focus the power evaluation on the interactive single-instance setting relevant to XR deployment, where the primary comparison is between BOBA-LOCAL on the edge device and BOBA-DISTRIBUTED. In this setting, edge-device power is the most relevant metric because it directly affects the usability of untethered XR systems. We therefore do not report server-side power in this section. Server-side energy efficiency is also important, but it is outside the scope of the interactive XR power evaluation.

11.2 Additional Dataset Cases

The main paper uses the original PhysTwin benchmark to enable direct comparison under matched scenes, trajectories, and metrics. To evaluate whether the same trends hold beyond the main benchmark, we additionally evaluate Boba on nine later-released PhysTwin cases that include new digital twins and more deformation-heavy actions.

Table 4 shows that Boba’s latency improvements also hold on these nine later-released cases. BOBA-LOCAL reduces average Orin latency from 413.8 ms to

Table 4: Additional quantitative results on nine later-released PHYSTWIN cases. E2E latency reports average end-to-end latency on these cases. Tracking error is omitted because these released cases do not include ground-truth 3D tracking trajectories.

Task	Runtime	Reconstruction & Resimulation					Future Prediction				
Method	E2E Lat. ms↓	CD↓	IoU↑	PSNR↑	SSIM↑	LPIPS↓	CD↓	IoU↑	PSNR↑	SSIM↑	LPIPS↓
PHYSTWIN	413.8	0.004	94.7	27.8	0.956	0.037	0.007	85.5	23.5	0.940	0.058
BOBA-LOCAL	31.4	0.005	96.6	27.7	0.958	0.042	0.008	91.2	24.2	0.947	0.055
BOBA-DISTRIBUTED	23.9	0.005	96.8	27.8	0.958	0.043	0.008	91.3	24.3	0.947	0.055

31.4 ms, corresponding to a $13.2\times$ speedup over PHYSTWIN. BOBA-DISTRIBUTED further reduces latency to 23.9 ms, yielding a $17.3\times$ speedup over PHYSTWIN and a $1.3\times$ improvement over BOBA-LOCAL. These trends are consistent with the main-paper results: Boba reduces single-instance latency by over $10\times$, and the distributed configuration provides an additional latency reduction over local edge execution. On the quality side, BOBA-LOCAL and BOBA-DISTRIBUTED maintain comparable reconstruction/resimulation and future-prediction quality, suggesting that the latency improvements observed in the main benchmark carry over to these additional cases without materially degrading quality.

Tracking error is omitted for these additional cases because the released data do not include the ground-truth 3D tracking trajectories required to compute it. We therefore report Chamfer distance and image-space metrics only.

11.3 Physics-Only FEM Baseline

To further contextualize Boba’s physics computation stage, we also compare against a physics-only batched FEM baseline implemented with NVIDIA Newton’s VBD solver [33]. This comparison addresses the setting where only the physical simulation module is considered, rather than the full Gaussian digital-twin pipeline. We use a mesh size matched to Boba’s compact simulation model and evaluate several solver budgets, denoted by VBD iterations/substeps. Larger solver budgets improve physical behavior but increase runtime.

At batch size $N = 256$, BOBA-BATCHED is $3.2\times$ faster than Newton with a (4/4) solver budget and $93.8\times$ faster than Newton with a (16/32) solver budget. This comparison serves as a physics-only GPU baseline: Newton provides a mature batched FEM reference, but it does not include Gaussian appearance reconstruction, skinning to Gaussian primitives, or rendering/visualization, and is therefore not an end-to-end Gaussian digital-twin baseline comparable to Boba.

11.4 MPC Rollout Accuracy

We further evaluate whether Boba preserves planning accuracy in the MPC rollout setting. Fig. 9 reports rollout target CD for MPC planning. Boba matches the accuracy of the full PHYSTWIN simulator while requiring substantially less planning time. Under the same planning-time budget, PHYSTWIN evaluates fewer rollouts and therefore produces higher target error. The full-PHYSTWIN point

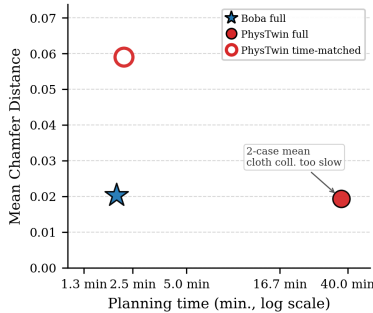


Fig. 9: MPC rollout accuracy. Boba matches the rollout target CD of full PHYSTWIN while using substantially less planning time. Under the same planning-time budget, PHYSTWIN evaluates fewer rollouts and yields higher target error.

averages the two non-collision cases; the cloth-collision case was prohibitively slow to evaluate with full PHYSTWIN. These rollout results suggest that Boba’s coarsened simulation is not the dominant source of the remaining real-world execution gap in these MPC experiments.

11.5 Additional Ablation Studies

Single-Instance Optimization Breakdown This breakdown summarizes the step-by-step impact of each optimization on single-instance end-to-end latency on both Jetson Orin and RTX 4090. We include both platforms to show that the latency reductions are not specific to a single hardware configuration or bottleneck profile, but remain effective across both an edge platform and a high-performance desktop GPU. We report both *incremental speedup*, relative to the previous stage, and *cumulative speedup*, relative to the PHYSTWIN baseline, with speedup computed from latency ratios.

Because the ablation is cumulative, the incremental speedup of each optimization is order-dependent: later optimizations are measured after earlier stages have already reduced the workload or shifted the bottleneck. Thus, small incremental gains should be interpreted as marginal contributions within this cumulative optimization stack, rather than as standalone measures of each optimization’s full potential impact.

Fig. 10 first shows the Jetson Orin breakdown for the full ablation sequence, with all configurations executed entirely on the edge device.

Starting from PHYSTWIN (330.9 ms, 3.0 FPS), spatiotemporal coarsening reduces latency to 127.8 ms ($2.59\times$ incremental speedup, 61.4% latency reduction). Rest-state guided collision pruning further improves latency to 115.2 ms ($1.11\times$ incremental speedup), and locality-aware node ordering gives a smaller but consistent gain to 114.2 ms ($1.01\times$ incremental speedup). Together, this **spring-mass optimization stack** reduces latency to 114.2 ms, corresponding to a **2.90 $\times$** cumulative speedup (65.5% latency reduction) over PHYSTWIN.

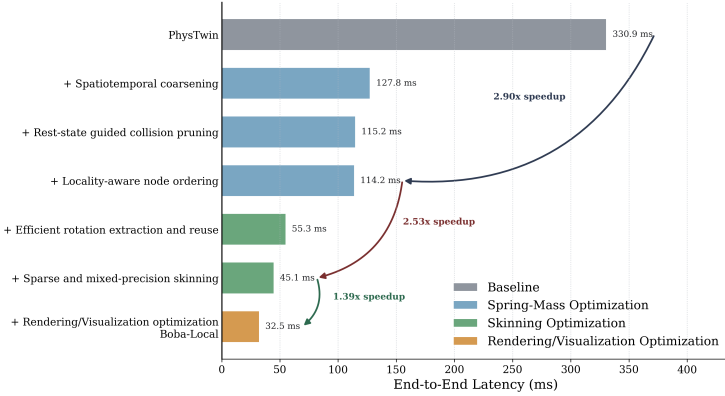


Fig. 10: Optimization breakdown on Jetson Orin. Step-by-step end-to-end latency breakdown for single-instance execution on Jetson Orin, from PHYSTWIN through intermediate ablation configurations to BOBA-LOCAL. The figure reports the impact of each optimization stage, including the spring-mass optimization stack, skinning optimization stack, and GPU-based rendering/visualization optimization. We annotate both incremental speedup relative to the previous stage and cumulative speedup relative to the PHYSTWIN baseline. Overall, BOBA-LOCAL reduces end-to-end latency from 330.9 ms to 32.5 ms, achieving a $10.18\times$ cumulative speedup and increasing performance from 3.0 FPS to 30.8 FPS.

Efficient rotation extraction and reuse then delivers the largest second-stage improvement, reducing latency to 55.3 ms ($2.07\times$ incremental speedup), and sparse mixed-precision skinning further lowers it to 45.1 ms ($1.23\times$ incremental speedup). Together, this **skinning optimization stack** reduces latency from 114.2 ms to 45.1 ms, an additional $2.53\times$ speedup (60.5% reduction relative to the pre-skinning stage).

Finally, GPU-based rendering/visualization optimization further reduces latency to 32.5 ms ($1.39\times$ incremental speedup), yielding 30.8 FPS and an overall $10.18\times$ cumulative speedup (90.2% end-to-end latency reduction) over the baseline.

Fig. 11 repeats the same single-instance optimization breakdown on RTX 4090.

Starting from PHYSTWIN on RTX 4090 (59.5 ms, 16.8 FPS), spatiotemporal coarsening reduces latency to 20.2 ms ($2.95\times$ incremental speedup, 66.1% latency reduction). Rest-state guided collision pruning further improves latency to 17.8 ms ($1.13\times$ incremental speedup), and locality-aware node ordering gives a smaller additional gain to 17.6 ms ($1.01\times$ incremental speedup). Together, this **spring-mass optimization stack** reduces latency from 59.5 ms to 17.6 ms, corresponding to a $3.38\times$ cumulative speedup (70.4% latency reduction) over the baseline.

Efficient rotation extraction and reuse then reduces latency to 15.0 ms ($1.17\times$ incremental speedup), and sparse mixed-precision skinning further lowers it to 14.7 ms ($1.02\times$ incremental speedup). Together, this **skinning optimization**

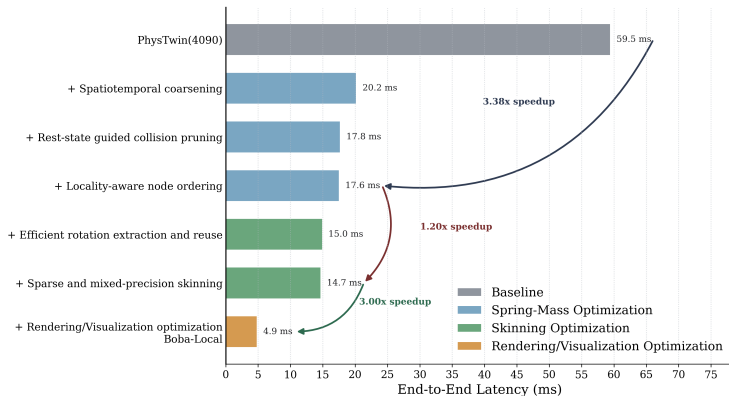


Fig. 11: Optimization breakdown on RTX 4090. Same single-instance ablation breakdown as Fig. 10, evaluated entirely on RTX 4090, including the spring-mass optimization stack, skinning optimization stack, and GPU-based rendering/visualization optimization. Overall, BOBA-LOCAL reduces end-to-end latency from 59.5 ms to 4.9 ms, achieving a $12.14\times$ cumulative speedup and increasing performance from 16.8 FPS to 204.5 FPS.

stack reduces latency from 17.6 ms to 14.7 ms, an additional $1.20\times$ speedup (16.5% reduction relative to the pre-skinning stage).

Finally, GPU-based rendering/visualization optimization reduces latency to 4.9 ms ($3.00\times$ incremental speedup), yielding 204.5 FPS and an overall $12.14\times$ cumulative speedup (91.8% end-to-end latency reduction) over the RTX 4090 baseline.

Compared with Orin, the overall trend on RTX 4090 is similar: both platform breakdowns show over $10\times$ cumulative gains, demonstrating that Boba’s optimization stack transfers across both edge and desktop GPUs. However, the marginal contribution of each stage differs across platforms. On Orin, the skinning optimization stack provides a large gain because the original SVD-based rotation extraction path incurs temporary materialization, scratch-space allocation, and excess memory traffic, which are especially costly on the lower-bandwidth edge platform. On RTX 4090, these overheads are less dominant due to substantially higher compute throughput and memory bandwidth, so the skinning-stage speedup is more modest and the remaining bottleneck shifts to rendering/visualization.

Overall, the gains are complementary: simulation-side optimizations provide strong early reductions on both platforms, skinning optimizations remove memory-heavy deformation overheads that are especially costly on Orin, and rendering/visualization optimization removes the remaining post-deformation bottleneck, enabling over $10\times$ speedup on both platforms.

Quality Preservation Across Ablations We also evaluate whether the cumulative optimization stack changes rendering quality in the future-prediction/test setting. Fig. 12 reports image-space quality across the same ablation sequence used in

the latency breakdown, using the same evaluation protocol for all cumulative variants. Across all variants from PHYSTWIN to BOBA-LOCAL, SSIM remains between 0.949 and 0.953, while PSNR remains between 23.21 and 23.27 dB. This indicates that the large latency reductions in the single-instance optimization stack do not introduce measurable rendering-quality degradation.

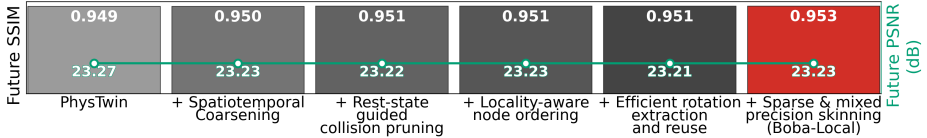


Fig. 12: Quality preservation across cumulative ablations. Future-prediction image-space quality remains stable across the same cumulative ablation sequence used in the latency breakdown. From PHYSTWIN to BOBA-LOCAL, SSIM stays between 0.949 and 0.953, and PSNR stays between 23.21 and 23.27 dB, indicating that the optimization stack improves efficiency without measurable rendering-quality degradation.

11.6 Additional Batching Throughput Analysis

To further characterize the batching scalability of BOBA-BATCHED, we report aggregate throughput and supported batch sizes across workloads on two GPU platforms in Table 5. We use the same throughput convention as in the main paper: throughput denotes aggregate FPS across all batched instances for the full executed pipeline, summarized across evaluated workloads.

Table 5: Batching throughput and capacity across GPU platforms. Average throughput is the average of per-workload peak aggregate FPS, and maximum throughput is the highest per-workload aggregate FPS. Average maximum batch size is averaged across workloads, while maximum batch size is the largest feasible batch size observed across workloads.

GPU	Mem. (GB)	Avg. Throughput	Max Throughput	Avg. Max Batch Size	Max Batch Size
RTX 4090	24	3310	8477	368.3	688
RTX PRO 6000 Blackwell SE	96	3530	10283	1389	2000

On RTX 4090, BOBA-BATCHED achieves 3,310 FPS average aggregate throughput, with the highest per-workload throughput reaching 8,477 FPS. Its average maximum supported batch size is 368.3 instances, and its maximum supported batch size is 688 instances across workloads.

We also report the same analysis on RTX PRO 6000 Blackwell SE. Due to its larger memory capacity, RTX PRO 6000 Blackwell SE supports larger batch sizes in all workloads. Compared with RTX 4090, the average maximum

supported batch size increases from 368.3 to 1,389 instances, a $3.77\times$ increase, which is close to the $4.0\times$ increase in GPU memory capacity. The maximum supported batch capacity increases from 688 to 2,000 instances, a $2.91\times$ increase. In contrast, throughput improves more modestly: average aggregate throughput increases from 3,310 FPS to 3,530 FPS, a $1.07\times$ improvement, while the highest per-workload throughput increases from 8,477 FPS to 10,283 FPS, a $1.21\times$ improvement.

These results suggest that the larger-memory GPU provides substantially more batching headroom, with average batch capacity scaling nearly proportionally with memory capacity. Throughput, however, improves more modestly because memory capacity mainly determines how many instances can fit, while aggregate FPS also depends on how quickly the GPU executes the batched simulation, skinning, and rendering/visualization pipeline. In addition, although RTX PRO 6000 Blackwell SE has more SMs than RTX 4090, its lower clock frequency makes the rough SM-count-times-clock compute proxy only about $1.15\times$ higher. As a result, larger batch capacity does not directly translate into proportional throughput gains.

Overall, these results show that BOBA-BATCHED maintains strong batching scalability across GPU platforms. Moving to a higher-memory workstation-class GPU substantially increases the supported batch-size range and provides additional, but workload-dependent, throughput gains.

12 Additional Application Demonstrations

Beyond the batched MPC rollout setting emphasized in the main paper, we include two additional demonstrations of how Boba can be used in downstream interactive and learning workloads: an immersive XR prototype and an RL training task.

12.1 Immersive XR Interactive Application

We further demonstrate Boba in an immersive XR prototype on a Quest 3 headset. The user manipulates a simulated Gaussian physical twin using the headset controllers while viewing the interaction in a reconstructed indoor scene. Fig. 13 shows representative first-person views from the prototype, including single-controller and two-controller manipulation, target-directed placement, and interaction with scene geometry. The full interaction is shown in the supplementary video.

This prototype uses a server-rendered streaming setup. The headset streams controller poses to a workstation over Wi-Fi, while the workstation runs the application loop, including simulation, skinning, rendering/visualization, scene composition, and stereo frame streaming back to the headset. This differs from the BOBA-DISTRIBUTED configuration evaluated earlier, where simulation and skinning are offloaded but rendering/visualization remains on the edge device. This server-rendered streaming setup was chosen for prototype integration: it allows us to reuse an existing ALVR-based [31] headset streaming path rather

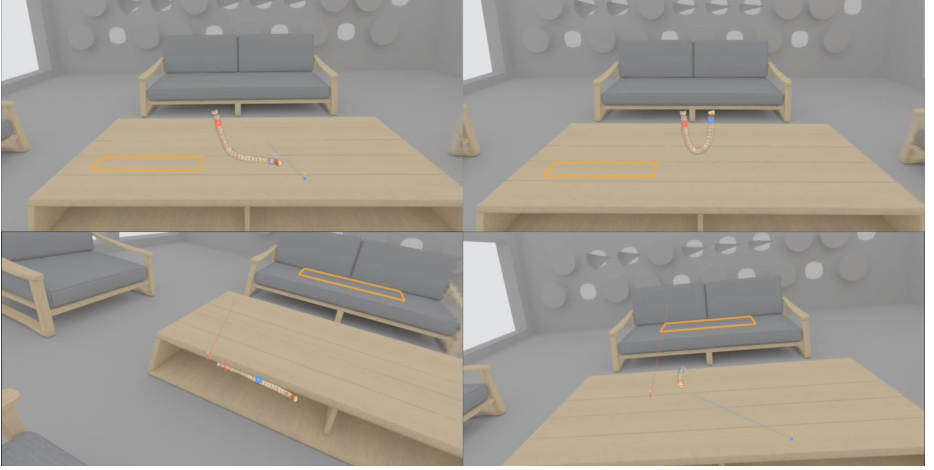


Fig. 13: Immersive XR interactive application. Representative first-person headset views from the XR prototype. The user manipulates a simulated Gaussian physical twin in a reconstructed indoor scene using Quest 3 controllers. The task requires application-level interaction logic beyond the core simulation pipeline, including single-controller and two-controller manipulation, target-directed placement, and contact-aware interaction with scene geometry such as the table, support surfaces, and scene boundaries.

than building and debugging a custom native Quest-side rendering/visualization backend. We therefore treat this XR prototype as an application demonstration of Boba’s core simulation and skinning pipeline, rather than as the optimized BOBA-DISTRIBUTED split evaluated earlier.

The prototype illustrates that Boba’s optimized physical digital-twin pipeline can be embedded in a complete XR interaction loop with controller input, target-directed manipulation, and scene-aware contact handling. The core simulation and skinning optimizations reduce the cost of updating the physical twin, leaving more application budget for interaction logic, scene composition, and stereo headset streaming/display.

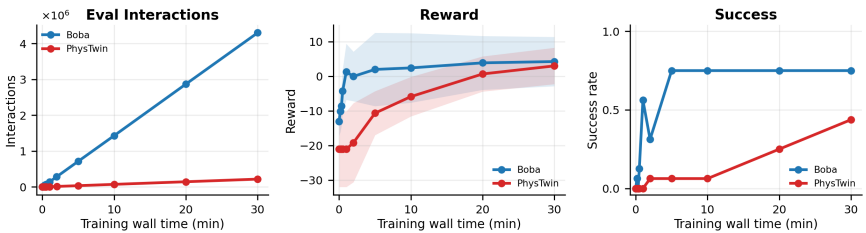


Fig. 14: RL training on a rope-reaching task. Under identical PPO settings, BOBA-BATCHED generates substantially more interactions per wall-clock time than PHYSTWIN and reaches higher reward and success earlier, indicating higher rollout throughput while preserving a useful learning signal for this task.

12.2 RL Training Application

Fig. 14 shows PPO training results on a single-lift rope-reaching task, where the policy controls the grasped rope end to move the free tip toward a target on the table plane. Both methods use the same observations, reward, success metric, hyperparameters, and evaluation targets; only the simulator differs. BOBA-BATCHED processes more interactions per wall-clock time and reaches higher reward and success earlier, suggesting that it provides a useful learning signal for this task while substantially increasing rollout throughput.