PolySplat: Workload-Regime-Aware Rasterization for 3D Gaussian Splatting

Longzan Luo* **Bin Cui*†** **Xupeng Miao*‡**
Peking University
https://github.com/PKU-DAIR/PolySplat

Abstract

Rasterization dictates the interactive budget of 3D Gaussian Splatting (3DGS). However, the comparative speed of modern CUDA rasterizers is typically evaluated on a narrow canonical benchmark, masking severe regime-dependent performance reversals. This limited scope hides three critical kernel-level bottlenecks: warp lane underutilization, exposed global-memory latency during pixel shading, and terminal-tail penalties from hardware thread scheduling. In this paper, we present PolySplat, a workload-regime-aware 3DGS rasterizer designed to overcome these inefficiencies. PolySplat introduces a warp-saturating tile-key emitter with adaptive three-way dispatch, asynchronous shared-memory staging in the render kernel, and a persistent kernel architecture with centralized atomic dispatch to strictly bound tail latency. To rigorously validate our system, we introduce an extended 76-target benchmark that achieves comprehensive workload-regime coverage by stratifying targets across extreme Gaussian counts (up to 56.5M), high resolutions (up to 9K), and six diverse scene categories. Evaluated on this comprehensive suite, PolySplat achieves dataset-balanced geometric-mean speedups of 1.42x to 7.87x over state-of-the-art rasterizers at lossless visual quality. Notably, under sustained 60Hz interaction at extreme resolutions, PolySplat strictly meets 1-vsync display deadlines where existing renderers suffer from massive queue divergence and input-to-photon lag.

1 Introduction

3D Gaussian Splatting (3DGS) has firmly established itself as the standard representation for real-time, photorealistic novel-view synthesis (Kerbl et al., 2023; Chen and Wang, 2024; Wu et al., 2024; Ye et al., 2025; Feng et al., 2024; Wang et al., 2025). However, as its application scope expands—ranging from multi-megapixel head-mounted displays (Franke et al., 2025) and tens of millions of primitives in city-scale captures (Kerbl et al., 2024; Liu et al., 2024) to generative world models (Zhao et al., 2025)—the 3DGS rasterizer is being pushed far beyond its original design boundaries. In these advanced regimes, sustained 60 Hz interaction demands a strict 16.67 ms per-frame budget. A deviation of just a few milliseconds can shift a system from fluid responsiveness into a queue-divergent state, where input-to-photon latency balloons from milliseconds to seconds. As workloads diversify across extreme resolutions and primitive counts, a critical question arises: are current 3DGS rasterizers robust and consistent across diverse workload regimes, or do their performance characteristics vary significantly across regimes?

Our analysis reveals strong evidence for the latter. We demonstrate that simply holding a scene fixed while shifting its Gaussian count and rendering resolution can invert the wall-clock performance

*School of Computer Science & Beijing Key Laboratory of Software and Hardware Cooperative Artificial Intelligence Systems, Peking University, Beijing, China.
†Key Lab of High Confidence Software Technologies (MOE), Peking University, Beijing, China
‡Corresponding author.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

ordering of two leading open-source rasterizers, FlashGS (Feng et al., 2024) and gsplat (Ye et al., 2025). This reversal phenomenon concentrates within a specific joint workload regime—resolutions $\leq$ 2K combined with primitive counts ≥ 0.5 M. Yet, the community's reliance on the standard 13-scene benchmark inherently masks this vulnerability. By clustering heavily around intermediate resolutions (1.3K to 5.2K) and primitive counts (1.0 M to 6.1 M) while excluding structurally distinct environments like aerial or city-scale scenes, the canonical benchmark biases evaluation toward a limited region of the workload space, licensing cross-regime claims that fail to hold in practice.

To unmask these hidden inefficiencies, we construct a comprehensive 76-target benchmark stratifying 72 unique scenes across extreme Gaussian counts (up to 56.5 M), high resolutions (up to 9K), and six diverse scene categories. Evaluating existing CUDA rasterizers under this expanded stress test exposes three critical, kernel-level architectural bottlenecks previously obscured by canonical coverage: (i) **Warp lane underutilization:** The heavy-tailed distribution of per-Gaussian tile coverage leaves static, per-Gaussian thread mappings severely underutilized across the dominant 2–31 tile band. (ii) **Exposed global-memory latency:** Gather-style loads for per-Gaussian features directly expose pixel shading to high memory latency. (iii) **Hardware scheduling opacity:** Standard per-tile thread-block allocation leaves terminal-phase load balancing entirely to the hardware scheduler, allowing heavy tiles to serialize and stall the entire pipeline.

To break these bottlenecks, we present PolySplat, a forward-pass rasterizer intrinsically designed to be workload-regime-aware. The key insight is to decouple execution granularity from primitive granularity, allowing the renderer to dynamically match hardware utilization to workload characteristics. Specifically, PolySplat implements this philosophy at the CUDA level through three mechanisms:

- **Adaptive Tile-Key Emission:** We introduce a warp-lane saturating dispatcher that dynamically routes single-tile, fragmented multi-tile, and massive multi-tile Gaussians into three specialized execution paths based on a hardware-grounded $\max_t \geq 32$ predicate.
- **Asynchronous Shared-Memory Staging:** We decouple pixel throughput from gather latency by overlapping Gaussian-feature loads with shading execution directly inside the render kernel.
- **Centralized Persistent Scheduling:** By employing a persistent kernel architecture with an atomic global dispatch queue, we mathematically bound the terminal-phase tail latency to the cost of the single heaviest tile.

Evaluated across our stratified 76-target benchmark, PolySplat establishes a new state of the art. Crucially, unlike existing rasterizers that suffer from localized performance reversals, PolySplat's adaptive architecture ensures robust acceleration across the entire workload space. It achieves dataset-balanced geometric-mean speedups of $1.42\times$, $2.47\times$, $5.18\times$, and $7.87\times$ over FlashGS, gsplat, Flash3DGS (Wang et al., 2025), and the reference 3DGS rasterizer at lossless visual quality. More importantly, PolySplat transcends mere quantitative gains to deliver a qualitative leap in interactive usability: under a sustained 60 Hz input trajectory at 9216×6912 on a city-scale scene, PolySplat strictly meets every 1-vsync deadline, entirely preventing the multi-second queue divergence that cripples existing renderers in identical settings.

2 Related work

2.1 3D Gaussian Splatting rasterizers

Kerbl et al. (2023) established the foundational 3D Gaussian Splatting (3DGS) representation alongside a canonical tile-based forward-pass rasterizer. This reference CUDA kernel—which depth-sorts visible Gaussians and alpha-blends them per pixel—serves as the architectural baseline for most subsequent acceleration efforts. Follow-up works have largely preserved this tile-sorted alpha-blend interface while overhauling the underlying computational mechanics. We benchmark our system against four open-source implementations in this lineage: Ye et al. (2025) provide a highly modular CUDA library with PyTorch bindings tailored for training-time experimentation; Feng et al. (2024) achieve higher throughput via tighter tile assignment, kernel fusion, and memory-access tuning; and Wang et al. (2025) introduce axis-oriented rasterization to share arithmetic between rays, coupled with an early-sorting depth pass.

Beyond these direct comparators, performance can also be improved by compressing the per-tile workload prior to rendering, as demonstrated by the pruning and bounds-tightening techniques of Hanson et al. (2025). Other efforts abandon standard GPU software pathways entirely: Lee et al. (2024) introduce dedicated 3DGS hardware units to mobile rendering pipelines, Li et al. (2025) extend GPU triangle rasterizers with minimal hardware support for 3DGS, and Hou et al. (2025) and Govindarajan et al. (2025) eliminate per-frame depth sorting by fundamentally altering the rendering equation (using order-independent and ray-traced variants, respectively). To ensure our wall-clock performance and visual quality metrics remain strictly comparable to the leading reference implementations, we deliberately retain the canonical sort-then-blend architecture. Notably, our forward-pass warp-atomic design shares conceptual DNA with the backward-pass atomic scheduling proposed by Durvasula et al. (2024), representing the closest systems-level cousin to our approach.

Related concerns about reducing processing cost, allocating runtime budgets, and limiting data movement also arise in data-intensive systems (Wan and Han, 2024; Han and Chun, 2025; Cao et al., 2025).

2.2 Benchmarks for novel-view synthesis

The evaluation of 3DGS acceleration—including the four reference rasterizers discussed above and Hanson et al. (2025)—has overwhelmingly relied on the 13-scene canonical suite from Kerbl et al. (2023). This set amalgamates scenes from Mip-NeRF 360 (Barron et al., 2022), Tanks & Temples `train`/`truck` (Knapitsch et al., 2017), and Deep Blending `drjohnson`/`playroom` (Hedman et al., 2018). However, as detailed in Section 3.2, this suite samples a critically narrow band of the (Gaussian-count, resolution) workload space and exclusively features indoor or outdoor-handheld captures. This localized clustering risks overfitting architectural designs to a specific performance bubble.

Fortunately, the broader novel-view synthesis community has produced a rich corpus of captures spanning highly diverse regimes. These include forward-facing scenes from LLFF (Mildenhall et al., 2019) and Nerfstudio (Tancik et al., 2023); high-resolution domains from Zip-NeRF (Barron et al., 2023) and VR-NeRF's Eyeful Tower (Xu et al., 2023); aerial drone footage from Mega-NeRF (Turki et al., 2022); city-scale environments from Hierarchical-3DGS (Kerbl et al., 2024) and UrbanScene3D (Lin et al., 2022); multi-scene collections like DL3DV (Ling et al., 2024); and underwater domains from SeaThru-NeRF (Levy et al., 2023). Our 76-target benchmark (Section 3.2) actively curates from this wider corpus, systematically stratifying targets by Gaussian count, rendering resolution, and scene category. By enforcing bucket-occupancy minimums across these axes, our benchmark ensures that cross-rasterizer performance claims are rigorously validated regime-by-regime, rather than being obscured by a globally pooled average.

3 Workload-regime analysis and stratified benchmark

The forward cost of 3DGS is primarily governed by Gaussian count and image resolution, which together define a *workload regime*. We show below that rasterizer rankings can flip across this space, and construct a stratified benchmark that exposes the dependence.

3.1 Empirical observation: same-scene reversals across Gaussian count and resolution

We use Tanks & Temples' `train`, `truck`, and `church` scenes as a concrete case study. For each scene, we extract the point cloud (PLY) at two different training iterations (7k and 30k) to create a natural shift in Gaussian count. We then render each PLY at two different resolutions, yielding a 2×2 workload grid per scene (12 measurement cells in total). Under identical hardware conditions and a uniform 20-camera sampling protocol (detailed in Section 6), we measure the end-to-end (E2E) wall-clock time for both the FlashGS and gsplat rasterizers cell-by-cell.

As shown in Table 1, gsplat only outperforms FlashGS in the specific corner where low resolution coincides with high Gaussian count. In all other configurations, FlashGS matches or exceeds gsplat. The inversion appears only when low resolution and high Gaussian count co-occur, showing that rasterizer rankings depend on the joint workload regime rather than on either axis alone. Consequently, comparisons based on a limited set of workloads may not generalize beyond their coverage.

Table 1: FlashGS/gsplat E2E time ratios on Tanks & Temples scenes, averaged over 20 cameras. Ratio < 1 means FlashGS is faster; bold marks the low-res/30k regime where gsplat overtakes FlashGS.

Scene	High-res 7k	High-res 30k	Low-res 7k	Low-res 30k
`train`	0.703	0.818	0.922	**1.079**
`truck`	0.716	0.758	0.999	**1.066**
`church`	0.654	0.737	0.997	**1.109**

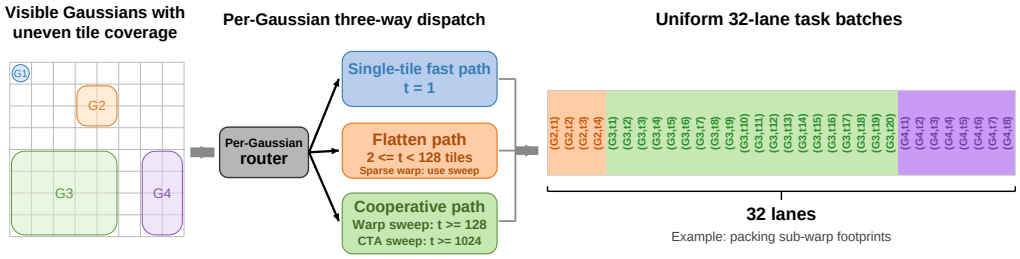

Figure 1: Adaptive three-way dispatch (Section 4). The measured implementation flattens footprints below 128 tiles, sweeps larger footprints per warp, and uses all CTA warps from 1024 tiles. Single-tile work bypasses flatten; sparse flatten groups also use a warp sweep. The right panel illustrates packing sub-warp footprints into 32-lane task batches.

3.2 Benchmark coverage and extended design

The widely used 13-scene benchmark occupies a relatively narrow region of the workload space, both in terms of Gaussian count and rendering resolution, and is limited to indoor and outdoor-handheld scenes. As a result, performance comparisons derived from this benchmark primarily reflect behavior within this restricted regime.

To enable systematic evaluation, we construct an extended benchmark comprising 76 measurement targets across 72 scenes, stratified along three axes: Gaussian count (from below 0.5M to above 50M), resolution (from sub-1K to 9K), and scene type (extended with aerial, underwater, driving, and city-scale environments). This allows performance to be analyzed within specific regimes rather than collapsed into a single global mean.

Across this extended benchmark, the reversal observed in Section 3.1 extends well beyond a three-scene curiosity: gsplat overtakes FlashGS on 9 of 63 newly added targets while no reversals occur on the canonical 13, and every reversal falls in regime $\mathcal{J}$ defined as long edge ≤ 2 K and Gaussian count ≥ 0.5 M. The canonical suite therefore misses a localized but systematic failure mode, motivating workload-regime-aware rasterization.

4 Warp-lane saturating tile-key emission

The preprocess stage emits one $(\text{tile_id}, \text{depth}) \mapsto \text{Gaussian_id}$ pair for each tile passing the intersection test within a visible Gaussian's projected bounding box (bbox). Across representative scenes, tile coverage concentrates well below the 32-lane warp width: the median Gaussian covers only 3–5 tiles, 7.2%–20.9% touch a single tile, and 77.8%–84.8% fall in the 2–31 range. Since the conventional one-warp-per-Gaussian schedule activates only $\min(t, 32)$ lanes per Gaussian, this dominant sub-warp population drives average lane utilization below one-third, severely bottlenecking the entire stage.

Design principle: three-way adaptive dispatch. We saturate all 32 lanes by flattening per-warp tile lists into a 1D task queue with one $(\text{Gaussian}, \text{tile})$ pair per lane (Figure 1). Single-tile Gaussians bypass the prefix sum, while large footprints use spatial sweeps that expose work within one Gaussian. The flatten path handles moderate footprints below 128 tiles, including the dominant 2–31 tile band, while footprints of at least 1024 tiles are distributed across the CTA.

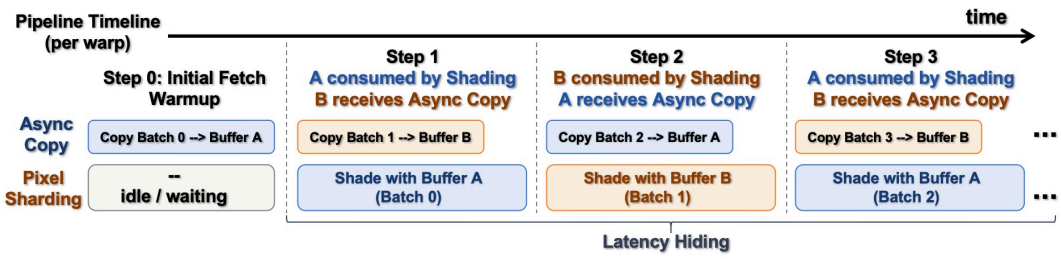

Figure 2: Async double-buffer pipeline timeline per warp. Step 0 prefetches the first feature batch. Thereafter, the warp shades the current shared-memory batch while the next batch is asynchronously copied into the alternate buffer. In steady state, feature-load latency is fully covered by the previous batch's shading work.

Single-tile fast path. When a Gaussian's bbox covers a single tile, each lane independently performs one ellipse intersection test; on a hit, it claims a key offset via a single atomic increment and writes the key-value pair directly. Because there is no inter-lane dependency, processing one Gaussian per lane naturally saturates the warp. Routing the single-tile workload through this fast path completely bypasses the overhead of the cooperative scheduling that follows.

Multi-tile cooperative scheduling. Let t_g denote the number of tiles covered by Gaussian g's bbox. Initial classification uses each Gaussian's own footprint, followed by the sparse-warp rule below:

$$\text{path}(g) = \begin{cases} \text{single-tile fast path} & t_g = 1, \\ \text{flatten (cross-Gaussian task tiling)} & 2 \le t_g < 128, \\ \text{fallback (cooperative spatial sweep)} & t_g \ge 128. \end{cases}$$

A warp-level ballot forms a bitmask $\mathcal{M}$ of eligible flatten work. If at most two lanes remain in $\mathcal{M}$, their work also uses the warp sweep to avoid the prefix-sum overhead. Otherwise, a large footprint in the same input warp does not force these moderate footprints onto the sweep path.

Fallback path. A warp sweeps an 8×4 spatial window across a Gaussian's bbox. From 1024 tiles, the four warps in a 128-thread CTA take disjoint tile rows. The CTA holds up to 32 such giant footprints; additional ones fall back to their own warp. These paths retain the same intersection predicate and shading arithmetic.

Flatten path. An inclusive warp prefix sum of local tile counts $\tilde{t}_\ell$ gives cumulative counts C_ℓ, with zero counts outside $\mathcal{M}$ and offsets $p_\ell = C_\ell - \tilde{t}_\ell$. Each batch assigns task $q = 32b + \ell$ to lane ℓ when $q < T$, where $T = \sum_\ell \tilde{t}_\ell$. A six-step shuffle-based binary lifting search finds the first owner with $C_{\text{owner}} > q$. Reciprocal division followed by an integer correction maps the residual $q - p_{\text{owner}}$ to its exact row and column.

5 Render kernel

This section unfolds two designs around the rendering kernel: Section 5.1 strips Gaussian feature loading off the critical path of pixel shading via asynchronous shared-memory loading, and on this basis fixes the block size to one warp; Section 5.2 eliminates block-level tail latency via a persistent kernel plus a centralized atomic counter. Both designs act inside the rendering kernel; quantitative comparisons are concentrated in Section 6.

5.1 Memory overlap via asynchronous shared-memory staging

Pixel shading requires 40 bytes of feature data per Gaussian, accessed via gather-style reads that inject hundreds of cycles of global memory latency. To remove these irregular reads from the critical path, we decouple memory I/O from compute via a producer–consumer pipeline (Figure 2). We partition Gaussians into batches of size $B = 32$ to perfectly align with the GPU's warp execution model. This batch size is carefully chosen: $B < 32$ would underutilize memory bandwidth and increase instruction overhead, while $B > 32$ would linearly inflate the shared memory footprint and degrade

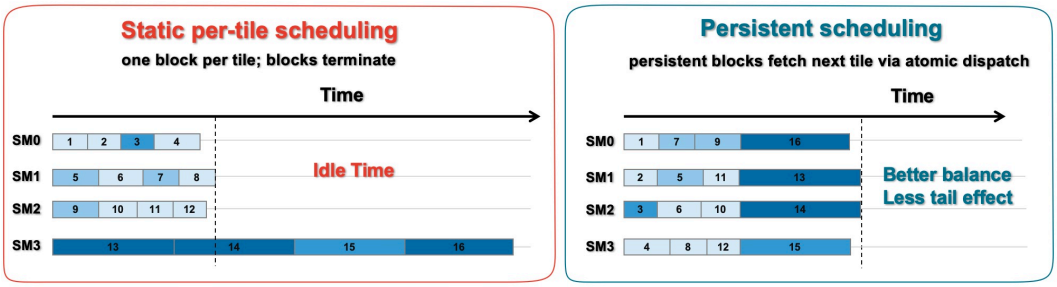

Figure 3: Persistent vs. static per-tile scheduling (Section 5.2). *Left*: one-block-per-tile dispatch leaves a terminal long tail—once light tiles drain, the remaining hot tiles serialize on a few SMs while the rest sit idle. *Right*: our persistent kernel keeps blocks resident and pulls work via a centralized atomic counter.

Streaming Multiprocessor (SM) occupancy. Using double buffering, the hardware asynchronously fetches the next batch into shared memory while the kernel processes the current one, seamlessly overlapping data movement with shading.

To maximize the overlap window, the measured configuration uses a single warp (32 threads) per block and warp-level synchronization between batches. A warp maps to a 16×16 tile (256 pixels), processing eight pixels per thread. Each 32-Gaussian batch supplies 8192 pixel-Gaussian interactions while the next 1280-byte feature batch is prefetched. The persistent grid is sized using CUDA's occupancy query for the compiled kernel, and the benefit of this overlap is measured in Table 4.

5.2 Tail elimination via persistent kernel with centralized tile dispatch

The spatial workload distribution in 3DGS is inherently heavy-tailed: highly occluded regions concentrate rendering operations into a few "hot" tiles, while sky and background tiles evaluate almost instantly. Our empirical measurements indicate that the top 1% of tiles can account for 5%–8% of total $(\text{tile}, \text{Gaussian})$ interactions, with the heaviest tile requiring up to $21\times$ the computation of the median tile. Standard per-tile block scheduling—assigning one thread block per tile—suffers from two critical bottlenecks under this distribution. First, it incurs an independent block dispatch and state-initialization overhead that scales linearly with the tile count. Second, and more severely, it exposes the execution to hardware scheduling opacity: if the hardware queuing mechanism arbitrarily dispatches the remaining heavy tiles to the same SM near the end of the kernel execution, they will process serially. This strands other SMs in an idle state and artificially inflates the terminal idle time (Figure 3, *left*), severely bottlenecking the total wall-clock time.

Decoupled scheduling via persistent thread blocks. To bypass the hardware scheduler's blind spots, we decouple the block lifecycle from the tile workload (Figure 3, *right*). Adapting the persistent-thread paradigm proven on irregular GPU workloads such as ray traversal and general task management (Aila and Laine, 2009; Tzeng et al., 2010), we launch a persistent kernel with a statically bounded number of blocks. Instead of terminating after processing a single tile, each resident block enters a continuous fetch-and-execute loop (Section 5.1), claiming new tiles from a centralized global task queue. This architectural shift amortizes the block launch and cold-start overheads across multiple tiles.

Occupancy-driven grid sizing. The launch configuration is entirely agnostic to the scene's spatial resolution or tile count T. The grid size is strictly clamped to $\text{SMs} \times K$, where K is the maximum number of concurrent blocks an SM can accommodate given the stringent register and shared-memory constraints discussed in Section 5.1. This ensures maximum theoretical occupancy from the onset while preventing the launch overhead associated with massive, scene-dependent grid dimensions.

Warp-coordinated dynamic fetching. The persistent blocks initialize by processing the first grid-sized batch of tiles. Upon completion of a tile, a block dynamically requests its next assignment via an atomic increment on a global counter (initialized to the grid size). To minimize synchronization

overhead, this atomic fetch is executed solely by lane 0 of the warp, and the retrieved tile index is subsequently broadcast to the remaining lanes via hardware warp-synchronous intrinsics.

Strict bound on terminal imbalance. Let w_k denote the computational work of tile k. In naive per-tile scheduling, the makespan is highly sensitive to the stochastic placement of heavy tiles by the hardware scheduler. By employing a centralized atomic queue, our persistent kernel guarantees a dynamic work-stealing behavior: any idle SM immediately pulls the next available tile. Consequently, the terminal imbalance (the time other SMs wait for the final SM to finish) is strictly bounded by the processing time of the single heaviest tile, $\max_k w_k$, rather than the arbitrary clustering of multiple heavy tiles. This mathematical tightening of the tail latency drastically accelerates the kernel's completion phase.

6 Evaluation

This section addresses four primary questions: (i) how PolySplat compares to existing rasterizers in terms of forward-pass speed on the 76-target extended benchmark detailed in Section 3, (ii) under which workload regimes these performance gains are sustained, (iii) which pipeline stages drive these improvements, and (iv) which individual component designs from Sections 4 and 5 are responsible for these stage-level gains. All headline results strictly adhere to a same-PLY, same-camera protocol, ensuring that each renderer processes identical model files and camera parameters across all targets.

6.1 Experimental setup

Renderers. We benchmark PolySplat against four reference renderers: FlashGS, the gsplat rasterizer, the official INRIA 3DGS rasterizer, and Flash3DGS. To ensure a fair comparison, all systems render images at identical resolutions using the exact same input models and camera parameters.

Benchmark. Our evaluation suite comprises 72 unique scenes, yielding 76 distinct PLY and camera-list targets grouped into 12 dataset categories. This collection provides 1518 camera-level samples, spanning model scales from approximately 0.1 M to 56.5 M Gaussians, and image resolutions with long edges ranging from 960 to 9288 pixels. The scenes represent a highly diverse array of environments, including indoor, outdoor, aerial, underwater, driving, and city-scale settings. Headline wall-clock measurements are conducted on a single NVIDIA H100 PCIe 80 GB GPU; Section 6.3 reports results on additional GPUs.

6.2 Main results

Overall performance. Figure 4 visualizes per-dataset forward-pass wall-clock for PolySplat and the four baselines. PolySplat achieves speedups of $1.42\times$, $2.47\times$, $5.18\times$, and $7.87\times$ over FlashGS, gsplat, Flash3DGS, and 3DGS, respectively.

Robust acceleration without performance regressions. As established in Section 3.2, gsplat overtakes FlashGS on $9/63$ newly added targets in the low-resolution, high-primitive regime $\mathcal{J}$. PolySplat is faster than FlashGS on all 76 targets. Across the 12 dataset categories in Figure 4, mean speedups range from $1.21\times$ to $1.64\times$.

Stage-level breakdown. Figure 5 breaks down forward-pass GPU time into preprocessing, sorting, and rendering. For PolySplat, these stages account for 27.94%, 25.54%, and 46.52% of total stage time, respectively, under dataset-balanced aggregation.

Rendering quality vs. reference. We evaluate rendering quality against ground-truth images on the same 76-target benchmark, comparing PolySplat with the INRIA 3DGS reference. Across the 12 datasets, PolySplat achieves dataset-balanced PSNR/SSIM of 24.03 dB/0.7430, compared with 24.04 dB/0.7431 for 3DGS. These results show that PolySplat maintains rendering quality comparable to the reference while accelerating the forward pass.

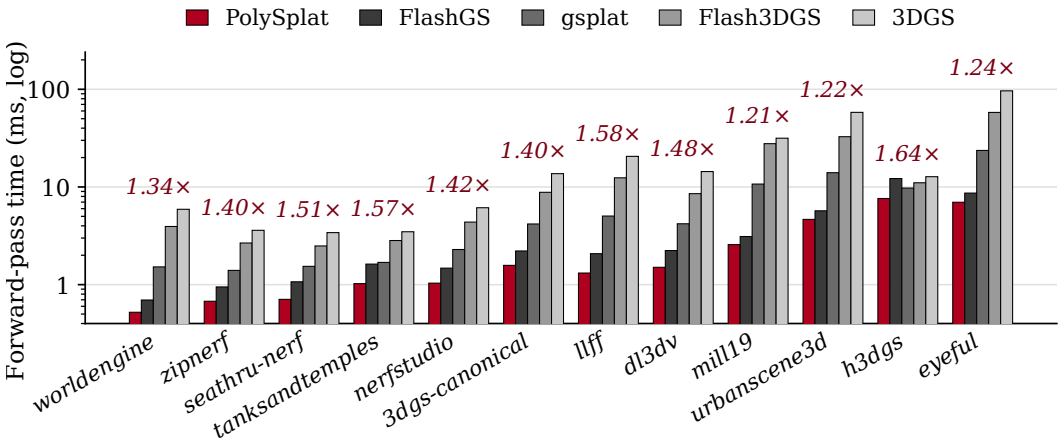

Figure 4: Per-dataset forward-pass time on H100 (ms, log scale), 76 targets. Red labels average target-level FlashGS/PolySplat ratios within each dataset.

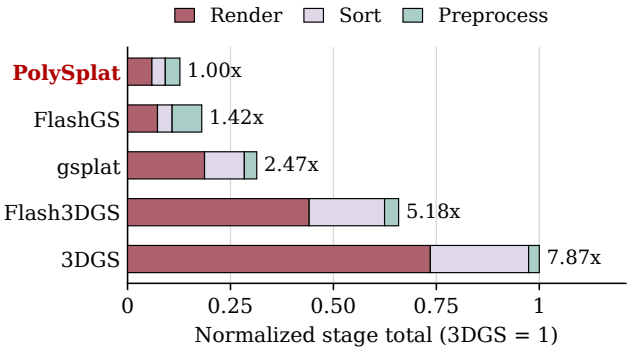

Figure 5: Dataset-balanced stage composition (preprocess / sort / render). Bar length is total time normalized by 3DGS; right labels report total time ratios relative to PolySplat.

Metric	PolySplat	gsplat
Deadline-miss @ 1 vsync (%)	**0.00**	99.72
Throughput μ_{eff} (fps)	**60.00**	28.13
Time above 100 ms HCI (%)	**0.00**	96.94

Table 2: Per-frame deadline behaviour over the 360-frame Mill19 `Rubble` trajectory at 9216 × 6912 (case study of Section 7). Deadline-miss ratio uses a 1-vsync firm deadline; the 100 ms time-above-threshold reflects the "instantaneous" HCI boundary of Card et al. (1991).

6.3 Cross-hardware portability

Table 3 summarizes speedups on H100, H20, A800, RTX 4090 D, and A40, using dataset-balanced geometric means over 76 targets. The render kernel uses `cp.async` (compute capability 8.0+); Hopper preprocessing additionally uses TMA, while pre-Hopper devices use direct loads. The default path uses no thread-block clusters, DSMEM, or WGMMA. H100 is built for `sm_90a`; H20, A800, RTX 4090 D, and A40 builds target `sm_90`, `sm_80`, `sm_89`, and `sm_86`.

Table 3: Dataset-balanced geometric-mean speedup of PolySplat over the four baselines on a range of NVIDIA SKUs spanning data-center and consumer tiers.

GPU	vs. FlashGS	vs. gsplat	vs. Flash3DGS	vs. 3DGS
H100 80 GB HBM3	1.42×	2.47×	5.18×	7.87×
H20	1.36×	2.39×	4.99×	7.49×
A800	1.34×	2.34×	4.79×	7.39×
RTX 4090 D	1.24×	2.21×	4.54×	6.80×
A40	1.18×	2.09×	4.39×	6.61×

Shared-sort bottleneck. Profiling on Garden shows sort shares of 23.61%, 22.03%, 28.21%, 46.90%, and 49.80% on H100, H20, A800, RTX 4090 D, and A40, with bandwidth utilization of

Table 4: Per-component ablation, paired against PolySplat on the 76-target benchmark; only the counterfactual under test changes between rows. "Dataset-bal. Δ" is the arithmetic mean over the 12 dataset buckets, "Target-bal. Δ" over the 76 targets, and "slower" counts targets where the counterfactual is strictly slower than PolySplat.

Design question	Counterfactual	Dataset-bal. Δ (%)	Target-bal. Δ (%)	Slower targets
Tile-key dispatch (§4)	one Gaussian per warp	$+29.52$	$+31.05$	$76/76$
	force flatten path	$+20.94$	$+15.27$	$76/76$
Shared-memory staging (§5.1)	disable staging/batching	$+50.17$	$+70.43$	$76/76$
Persistent scheduling (§5.2)	static per-tile blocks	$+2.32$	$+2.63$	$70/76$

81.82%, 80.31%, 90.72%, 93.91%, and 95.73%. These results are consistent with a tighter Amdahl limit on lower-bandwidth devices. The dataset-balanced stage composition is reported in Figure 5.

6.4 Component ablations

Table 4 changes one design choice at a time in the fused implementation. Modes rotate within each camera's 50 measurement rounds after 10 warmup calls. Preprocess TMA, sorting, tile-range lookup, pixel arithmetic, and the early-exit cadence are shared except for the factor under test. Forced fallback and flatten also route giant footprints through the forced path, while retaining the single-tile fast path. Disabling render staging retains persistent scheduling; static dispatch retains staging.

Forcing one warp to sweep each multi-tile Gaussian raises E2E time by 29.52% under dataset-balanced aggregation; force-flatten raises it by 20.94%. Both are slower on all 76 targets. Disabling render staging has the largest measured penalty, 50.17%. Static per-tile blocks cost 2.32% on average and are slower on $70/76$ targets, indicating a workload-dependent scheduling benefit. A separate 9K Rubble control uses the 20 canonical sampled cameras with doubled intrinsics and resolution: force-flatten increases mean time from 8.0312 to 11.2187 ms ($+39.69\%$). This control is distinct from the continuous trajectory in Section 7.

7 Case study: sustained 60 Hz rendering on a 9K city-scale trajectory

Target applications of 3DGS—including immersive VR (Franke et al., 2025), embodied agent simulation, and generative world model rollouts (Zhao et al., 2025)—demand strict adherence to a 60 Hz motion-to-photon budget during continuous camera movement. We compare PolySplat against gsplat under these constraints using a 6.0 s, 360-frame continuous flight through Mill19 `Rubble` (3.19 M Gaussians). The trajectory is rendered at 9216×6912 and replayed under the strict-order 1-vsync (16.67 ms) queue protocol defined by Liu (2000).

Table 2 details system responsiveness. Because gsplat's render time (35.37 ms median, 38.00 ms max) consistently violates the 16.67 ms budget, its input queue rapidly diverges: 99.72% of vsync deadlines are missed, throughput collapses to 28.13 fps, and latency peaks at 3.22 s. The user experiences latency above the critical 100 ms HCI threshold for 96.94% of the trajectory, and gsplat requires 6.78 s just to drain the queue after input ceases. Conversely, PolySplat limits render times to 11.58 ms per frame, successfully sustaining 60.00 fps end-to-end without a single vsync or HCI threshold violation. This two-orders-of-magnitude gap in deadline-miss rate, rather than the $\sim 3\times$ wall-clock gap alone, is what governs whether a rasterizer is deployable in the latency-critical 3DGS pipelines listed above.

8 Conclusion

We presented PolySplat, a workload-regime-aware forward rasterizer for 3D Gaussian Splatting. Warp-saturating emission, asynchronous render staging, and persistent tile scheduling address the bottlenecks exposed by the stratified benchmark. On H100, PolySplat achieves a $1.42\times$ dataset-balanced geometric-mean speedup over FlashGS with wins on all 76 targets. Paired ablations quantify each component's contribution, and the measured 9K trajectory remains within one vsync under the

specified 60 Hz queue replay. These results support evaluation by workload regime rather than a single canonical average.

Acknowledgments and Disclosure of Funding

This work is supported by the National Natural Science Foundation of China (Grant Nos. U23B2048 and U22B2037), the Beijing Municipal Natural Science Foundation (Grant No. L2603008), and the Fundamental Research Funds for the Central Universities, Peking University.

References

B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis. 3D Gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42(4), 2023.

G. Chen and W. Wang. A survey on 3D Gaussian splatting. *arXiv preprint arXiv:2401.03890*, 2024.

T. Wu, Y.-J. Yuan, L.-X. Zhang, J. Yang, Y.-P. Cao, L.-Q. Yan, and L. Gao. Recent advances in 3D Gaussian splatting. *Computational Visual Media*, 10(4):613–642, 2024.

V. Ye, R. Li, J. Kerr, M. Turkulainen, B. Yi, Z. Pan, O. Seiskari, J. Ye, J. Hu, M. Tancik, and A. Kanazawa. gsplat: An open-source library for Gaussian splatting. *Journal of Machine Learning Research*, 26(34):1–17, 2025.

G. Feng, S. Chen, R. Fu, Z. Liao, Y. Wang, T. Liu, Z. Pei, H. Li, X. Zhang, and B. Dai. FlashGS: Efficient 3D Gaussian splatting for large-scale and high-resolution rendering. *arXiv preprint arXiv:2408.07967*, 2024.

Z. Wang, G. He, D. Liu, L. Gao, S. X. Hu, C. Zhang, Z. Song, N. Lane, W. Luk, and H. Fan. Accelerating 3D Gaussian splatting with neural sorting and axis-oriented rasterization. *arXiv preprint arXiv:2506.07069*, 2025.

A. Hanson, A. Tu, G. Lin, V. Singla, M. Zwicker, and T. Goldstein. Speedy-Splat: Fast 3D Gaussian splatting with sparse pixels and sparse primitives. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2025.

J. Lee, S. Lee, J. Lee, J. Park, and J. Sim. GSCore: Efficient radiance field rendering via architectural support for 3D Gaussian splatting. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

S. Li, B. Keller, Y.-C. Lin, and B. Khailany. GauRast: Enhancing GPU triangle rasterizers to accelerate 3D Gaussian splatting. In *Proceedings of the 62nd ACM/IEEE Design Automation Conference (DAC)*, 2025.

S. Durvasula, A. Zhao, F. Chen, R. Liang, P. K. Sanjaya, and N. Vijaykumar. DISTWAR: Fast differentiable rendering on raster-based rendering pipelines. *arXiv preprint arXiv:2401.05345*, 2024.

Q. Hou, R. Rauwendaal, Z. Li, H. Le, F. Farhadzadeh, F. Porikli, A. Bourd, and A. Said. Sort-free Gaussian splatting via weighted sum rendering. In *International Conference on Learning Representations (ICLR)*, 2025.

S. Govindarajan, D. Rebain, K. M. Yi, and A. Tagliasacchi. Radiant foam: Real-time differentiable ray tracing. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2025.

L. Franke, L. Fink, and M. Stamminger. VR-Splatting: Foveated radiance field rendering via 3D Gaussian splatting and neural points. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 8(1):18, 2025.

Y. Liu, C. Luo, L. Fan, N. Wang, J. Peng, and Z. Zhang. CityGaussian: Real-time high-quality large-scale scene rendering with Gaussians. In *European Conference on Computer Vision (ECCV)*, pages 265–282, 2024.

G. Zhao, C. Ni, X. Wang, Z. Zhu, X. Zhang, Y. Wang, G. Huang, X. Chen, B. Wang, Y. Zhang, W. Mei, and X. Wang. DriveDreamer4D: World models are effective data machines for 4D driving scene representation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12015–12026, 2025.

J. T. Barron, B. Mildenhall, D. Verbin, P. P. Srinivasan, and P. Hedman. Mip-NeRF 360: Unbounded anti-aliased neural radiance fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5470–5479, 2022.

A. Knapitsch, J. Park, Q.-Y. Zhou, and V. Koltun. Tanks and Temples: Benchmarking large-scale scene reconstruction. *ACM Transactions on Graphics*, 36(4), 2017.

P. Hedman, J. Philip, T. Price, J.-M. Frahm, G. Drettakis, and G. Brostow. Deep blending for free-viewpoint image-based rendering. *ACM Transactions on Graphics*, 37(6), 2018.

B. Mildenhall, P. P. Srinivasan, R. Ortiz-Cayon, N. K. Kalantari, R. Ramamoorthi, R. Ng, and A. Kar. Local light field fusion: Practical view synthesis with prescriptive sampling guidelines. *ACM Transactions on Graphics*, 38(4), 2019.

M. Tancik, E. Weber, E. Ng, R. Li, B. Yi, T. Wang, A. Kristoffersen, J. Austin, K. Salahi, A. Ahuja, D. McAllister, and A. Kanazawa. Nerfstudio: A modular framework for neural radiance field development. In *ACM SIGGRAPH 2023 Conference Proceedings*, 2023.

J. T. Barron, B. Mildenhall, D. Verbin, P. P. Srinivasan, and P. Hedman. Zip-NeRF: Anti-aliased grid-based neural radiance fields. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2023.

D. Levy, A. Peleg, N. Pearl, D. Rosenbaum, D. Akkaynak, S. Korman, and T. Treibitz. SeaThru-NeRF: Neural radiance fields in scattering media. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023.

H. Turki, D. Ramanan, and M. Satyanarayanan. Mega-NeRF: Scalable construction of large-scale NeRFs for virtual fly-throughs. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022.

B. Kerbl, A. Meuleman, G. Kopanas, M. Wimmer, A. Lanvin, and G. Drettakis. A hierarchical 3D Gaussian representation for real-time rendering of very large scenes. *ACM Transactions on Graphics*, 43(4), 2024.

L. Xu, V. Agrawal, W. Laney, T. Garcia, A. Bansal, C. Kim, S. Rota Bulò, L. Porzi, P. Kontschieder, A. Božič, D. Lin, M. Zollhöfer, and C. Richardt. VR-NeRF: High-fidelity virtualized walkable spaces. In *SIGGRAPH Asia 2023 Conference Papers*, 2023.

L. Lin, Y. Liu, Y. Hu, X. Yan, K. Xie, and H. Huang. Capturing, reconstructing, and simulating: The UrbanScene3D dataset. In *European Conference on Computer Vision (ECCV)*, 2022.

L. Ling, Y. Sheng, Z. Tu, W. Zhao, C. Xin, K. Wan, L. Yu, Q. Guo, Z. Yu, Y. Lu, X. Li, X. Sun, R. Ashok, A. Mukherjee, H. Kang, X. Kong, G. Hua, T. Zhang, B. Benes, and A. Bera. DL3DV-10K: A large-scale scene dataset for deep learning-based 3D vision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024.

X. Wan and X. Han. Efficient top-k frequent itemset mining on massive data. *Data Science and Engineering*, 9(2):177–203, 2024. https://doi.org/10.1007/s41019-024-00241-2.

J. Han and S. Chun. Budget-constrained runtime allocation of linked data services in stream processing. *Data Science and Engineering*, 10(2):277–295, 2025. https://doi.org/10.1007/s41019-024-00277-4.

H. Cao, C. Huang, S. Liu, H. Hu, M. Zhao, X. Zhou, Y. Tu, and W. Qian. Aion: Live migration for in-memory databases with zero downtime and reduced redundant data transfer. *Data Science and Engineering*, 10(2):212–229, 2025. https://doi.org/10.1007/s41019-024-00276-5.

T. Aila and S. Laine. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of High Performance Graphics*, 2009.

S. Tzeng, A. Patney, and J. D. Owens. Task management for irregular-parallel workloads on the GPU. In *Proceedings of High Performance Graphics*, pages 29–37, 2010.

J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

S. K. Card, G. G. Robertson, and J. D. Mackinlay. The information visualizer, an information workspace. In *Proceedings of CHI '91*, pages 181–186, 1991.

A Reproducibility

This appendix specifies the hardware, software versions, build flags, measurement protocol, and per-target manifest needed to reproduce Table 2,Table 3, and Table 4. The benchmark uses a strict same-PLY same-camera-list protocol (Section 6.1); each configuration entry below applies uniformly to all five compared systems unless explicitly noted otherwise.

A.1 Hardware

All wall-clock measurements use a single NVIDIA H100 PCIe 80 GB HBM3 GPU (compute capability 9.0, 132 streaming multiprocessors, $\sim$2.0 TB/s HBM3 memory bandwidth on the PCIe SKU, ECC enabled, persistence mode on). The GPU is selected by setting `CUDA_VISIBLE_DEVICES` to a single index so that no rasterization step ever observes more than one device; we do not measure or claim multi-GPU rasterization. Maximum graphics clock is $1{,}980$ MHz and maximum memory clock is $2{,}619$ MHz. The host machine is a dual-socket Intel Xeon Gold 6448Y (128 logical cores). Host CPU and DRAM characteristics have negligible impact on the reported numbers because timing brackets only the rasterization call under `cudaDeviceSynchronize`.

A.2 Software

Table 5 lists the version pin or commit hash of every component required to reproduce the measurements. Components above the rule are environment-level; components below the rule are the five rasterizers under comparison.

Table 5: Software versions and commit hashes . Renderer commits are pinned at the SHA observed at measurement time (May 2026). `gsplat` is shared by both gsplat and Flash3DGS as a backend dependency; the renderer commits below select which forward kernel is invoked.

Component	Version / commit
OS	Ubuntu 22.04.5 LTS (Linux 5.14.0-284.25.1.el9_2)
NVIDIA driver	580.65.06
CUDA toolkit	12.8 (`nvcc` 12.8.93)
Python	3.11.13
PyTorch	2.8.0+cu128
PolySplat	commit `0e6f8f7` on branch `exp/pre-opt-baseline`
FlashGS	commit `cdfc4e4`
Flash3DGS	commit `83b6524`
gsplat	PyPI release 1.4.0
3DGS	official INRIA reference rasterizer at the standard public release

A.3 Licenses for software components

Table 6 lists the license under which each component of Table 5 is distributed, as declared by the upstream release at the time of access (May 2026). PolySplat's source code will be released under the MIT License upon publication, matching FlashGS. The reference 3DGS rasterizer of Kerbl et al. (2023) is distributed under a non-commercial research-only license (the Inria/MPII "Gaussian-Splatting Research License"). Flash3DGS is, at the time of measurement, an author-provided pre-release of Wang et al. (2025); the upstream open-source release and its final license are forthcoming.

Table 6: Licenses of the software components of Table 5. PolySplat's planned license is MIT, matching FlashGS. The 3DGS baseline carries a non-commercial research-only license; our use is research-only.

Component	License
OS (Ubuntu 22.04)	various open-source (per Ubuntu release)
NVIDIA driver	NVIDIA proprietary EULA
CUDA toolkit 12.8	NVIDIA CUDA EULA
Python 3.11	PSF License
PyTorch 2.8.0	BSD-3
PolySplat	MIT (planned, upon publication)
FlashGS	MIT
Flash3DGS	pre-release; upstream license forthcoming
gsplat	Apache-2.0
3DGS	Inria/MPII Gaussian-Splatting Research License (non-commercial)

A.4 Build flags

All four CUDA rasterizers (PolySplat, FlashGS, Flash3DGS, 3DGS) are compiled against compute capability `sm_90` so that the emitted SASS is specialised for H100 rather than down-ranked to a shared-baseline ISA. Host-side flags follow the upstream FlashGS build recipe and are reused unchanged for all comparators that share that recipe (PolySplat, FlashGS): `-O3`, `-DNDEBUG`, `-std=c++17` for the host compiler, and `-O3`, `--use_fast_math`, `--extra-device-vectorization`, `-Xptxas="-O3"` for `nvcc`. The gsplat wheel is the unmodified PyPI 1.4.0 release; 3DGS uses the upstream INRIA build recipe with the same `sm_90` target.

A.5 Measurement protocol

Camera sampling. For each scene we list the cameras of the upstream training corpus, sort them by frame index, and select every 20th camera (stride-20 uniform sampling). When this would yield fewer than 20 cameras for small-trajectory scenes (e.g. certain *seathru-nerf* captures), we fall back to a denser stride that still draws cameras uniformly across the trajectory. The same per-target camera list is fed to all five systems.

Per-camera timing. For each camera we run 10 untimed warmup iterations followed by 50 measured iterations. Each iteration synchronises the device with `cudaDeviceSynchronize`, takes a host-side high-resolution timestamp, invokes a single rasterization call, synchronises the device again, and records the wall-clock interval. The reported per-camera mean is the arithmetic mean of these 50 samples.

PLY input. All five systems consume the 30,000-iteration training checkpoint of the upstream training pipeline—a single PLY file per target, identical across rasterizers. We do not re-fit, re-prune, or re-pack the SH coefficients between runs.

A.6 Compute resources

This subsection accounts for the GPU time and the peak GPU memory consumed by the experiments. All figures are single-GPU H100 PCIe 80 GB hours; no multi-GPU or multi-node experiment was run.

Per-call latency. A single rasterization call ranges from 0.63 ms (*worldengine_navtest*, PolySplat) to 96.4 ms (*eyeful*, 3DGS). The per-camera measurement is 60 such calls (10 warmup + 50 timed) under `cudaDeviceSynchronize`; the per-target run multiplies this by 20 stride-sampled cameras and the 5 rasterizers under comparison.

Profiling and exploratory compute. Beyond the reported and verification runs, the project consumed approximately 3 GPU-hours of Nsight Compute and `nvprof` captures used to validate the

per-stage attribution of Section 6.4 (instrumented kernels run roughly $50\times$–$100\times$ slower than untraced timing, so even a small target subset is significant in wall-clock terms). Kernel prototyping, regression testing of intermediate PolySplat commits, and ablation configurations that did not enter the final paper account for an estimated $\sim$60 GPU-hours of additional exploratory compute. We therefore put the full project footprint at $\sim$70–80 H100 PCIe 80 GB-hours.

Peak GPU memory. Per-target HBM3 utilisation is dominated by the PLY representation and per-tile auxiliary buffers. Across the 76 targets it ranges from $\sim$0.3 GB on *worldengine_navtest* (0.10 M Gaussians) to $\sim$28 GB on *h3dgs* `campus_full` (56.5 M Gaussians, the worst case in the benchmark); the median target uses 1–2 GB. All 76 targets fit within the 80 GB envelope of a single H100 PCIe SKU. The case study of Section 7, rendered at 9216×6912 on Mill19 `Rubble` (3.19 M Gaussians), peaks at $\sim$3.4 GB.

No training in scope. The reported numbers cover the inference-time forward pass only. All PLYs are the 30,000-iteration checkpoint of the upstream training pipeline of Kerbl et al. (2023) (Section A.5); any compute associated with that pipeline is upstream of this paper and is not included in the budget above.

B Limitations and scope

Cross-architecture scope. The headline measurements use a single NVIDIA H100 PCIe 80 GB GPU ; Table 3 additionally reports dataset-balanced geometric-mean speedups across a diverse set of GPUs spanning Hopper, Ampere, and Ada architectures, including H100, H20, A800, A40, and RTX 4090 D. Beyond the populated rows of Table 3 we make no claim about portability to earlier NVIDIA architectures or to non-NVIDIA accelerators. The persistent-kernel concurrency, register budget, and warp-aligned batch size of Section 5 are tuned against H100 SM resource limits and would have to be re-fit for execution units with materially different SIMT width or per-SM block capacity.

C Broader impacts

This appendix discusses the broader societal impacts of PolySplat in both directions. Because PolySplat is a forward-pass rasterizer that accelerates an existing rendering primitive without altering its inputs or outputs, its impact is mediated by the downstream applications it enables.

Positive impacts. The interactivity result of Section 7—sustained 60 Hz, 1-vsync-stable rendering of a 9216×6912 city-scale trajectory where prior rasterizers diverge—directly supports three classes of interactive applications motivated in the introduction: immersive VR and head-mounted display rendering (Franke et al., 2025), where queue divergence manifests as motion-to-photon lag and, in clinical or training settings, simulator sickness; embodied-agent simulation and generative world-model rollouts (Zhao et al., 2025), where the rasterizer is the inner-loop bottleneck and the 2–6$\times$ speedup translates linearly into reduced GPU-hours and energy per simulated trajectory; and city-scale digital twins or large-scene visualisation that today trade interactivity for fidelity. The same speedups reduce the per-frame energy draw of equivalent 3DGS workloads.

Potential negative or dual-use impacts. PolySplat is a forward rasterizer for a pre-trained 3DGS representation; it does not generate scene content or produce synthetic photographs from scratch, so the classical generative-AI risks (e.g. deepfake media) do not apply directly. Two indirect risks are worth naming. First, faster rasterization of high-resolution captured scenes lowers the cost of immersive replay of real locations, which has dual-use implications for surveillance, reconnaissance, or unconsented location reconstruction when the upstream capture itself is non-consensual—a risk inherent to 3DGS as a representation rather than specific to our acceleration. Second, large-scale interactive simulators built on 3DGS can be applied to beneficial ends (autonomous-driving safety, robotics training, education) or to harmful ones (combat/surveillance training); our work makes both classes of simulator cheaper without distinguishing between them.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes].

   Justification: The abstract and introduction state the paper's scope, the workload-regime motivation, the three system mechanisms, the 76-target benchmark, and the reported speedups; the corresponding evidence is presented in Sections 3–7.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes].

   Justification: Section B records the scope of the reported results—the headline measurements use a single NVIDIA H100 PCIe 80 GB GPU, with cross-hardware portability additionally reported on a consumer-tier RTX 4090 (Ada) and a data-center H20 in Table 3; beyond those populated rows we make no claim about portability to earlier NVIDIA architectures or to non-NVIDIA accelerators.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A].

   Justification: The paper does not present formal theorems or proofs. The scheduling bound in Section 5.2 is used as an informal systems argument rather than a formal theoretical result.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes].

   Justification: Reproducibility details are documented in Section A: the single-GPU H100 PCIe 80GB hardware, the software versions and commit hashes for all five compared systems, the same-PLY same-camera measurement protocol with 10 warmup + 50 measured iterations under cudaDeviceSynchronize bracketing, and the per-target manifest covering all 76 measurement targets.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes].

   Justification: The source code for PolySplat, together with the 76-target benchmark manifest—including the per-target camera lists, the stride-20 sampling indices, and the build flags described in Section A.4—is publicly available at `https://github.com/PKU-DAIR/PolySplat.git` under the MIT License. The experiments rely on publicly available datasets and baseline renderers pinned at the commit hashes listed in Table 5, so each external dependency is independently obtainable. Section A additionally documents the hardware, software, build flags, measurement protocol, and per-target manifest needed to re-run the benchmark end-to-end.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes].

Justification: Sections 3 and 6 describe the benchmark scale, target counts, sampling protocol, compared renderers, wall-clock metric, quality metric, and GPU used for evaluation.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No].

   Justification: The paper reports paired target counts and aggregate means, but it does not include error bars, confidence intervals, or a statistical significance test for the reported speedups.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes].

   Justification: Section A.1 specifies the compute worker (a single NVIDIA H100 PCIe 80 GB GPU on a dual-socket Intel Xeon Gold 6448Y host); Section A.6 reports the per-table GPU-time budget, the peak GPU-memory range across the 76 benchmark targets ($\sim$0.3–28 GB of HBM3) and the case-study workload ($\sim$3.4 GB at 9216×6912), and an estimate of the full project compute footprint ($\sim$70–80 single-GPU H100 PCIe 80 GB-hours, including verification re-runs and the exploratory profiling, kernel prototyping, and unselected ablation configurations that lie outside the reported tables).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes].

   Justification: The work is a systems and benchmarking study over 3DGS rasterization and does not introduce human-subject experiments, private data collection, or sensitive personal-data processing.

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes].

    Justification: Section C discusses both directions: it grounds the positive societal impacts in the case study of Section 7—sustained 1-vsync-stable interactive rendering enables immersive VR/HMD use (Franke et al., 2025), embodied-agent and world-model simulation (Zhao et al., 2025), and city-scale interactive visualisation, with a corresponding reduction in per-frame energy draw—and names two indirect or dual-use risks (cheaper immersive replay of real-world captures with surveillance / unconsented-reconstruction implications, and dual-use training simulators) that follow from making 3DGS rasterization faster.

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A].

    Justification: The paper does not release high-risk pretrained language models, image generators, or scraped datasets requiring special safeguards.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes].

Justification: Table 6 (Section A.3) lists the licenses of all software components used in our experiments, including the four renderer baselines (FlashGS, gsplat, Flash3DGS, and 3DGS) and the environment stack in Table 5. The original developers are credited through the corresponding citations in Table 5. Non-commercial or research-only restrictions associated with 3DGS, *tanksandtemples*, *h3dgs*, *eyeful*, *dl3dv*, and *urbanscene3d* are explicitly identified and fully respected; all such components are used solely for research purposes.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes].

    Justification: We introduce two new assets: (i) the PolySplat source code and (ii) the 76-target extended benchmark specification. The PolySplat repository documents the implementation, intended use, hardware and software requirements, known limitations, and license. The benchmark is documented through a per-target manifest specifying the datasets and scenes, camera lists, sampling indices, and build configurations required to reproduce each target, together with the applicable upstream data licenses. Because the benchmark is constructed from existing public datasets, we distribute the benchmark specification and manifest rather than redistributing upstream PLY files where their licenses do not permit redistribution. The PolySplat source code is publicly available at `https://github.com/PKU-DAIR/PolySplat.git` under the MIT License, and the benchmark manifest is released alongside the source code.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A].

    Justification: The paper does not involve crowdsourcing or human-subject experiments.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A].

    Justification: The paper does not involve crowdsourcing or human-subject experiments, so IRB approval is not applicable.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A].

    Justification: The core method and experiments do not use LLMs as an important, original, or non-standard component.