arXiv:2607.17842v1 [cs.CV] 20 Jul 2026

CaT-GS: Efficient 3DGS Rendering for Large-Scale Scenes with Inter-frame Caching and Tile Scheduling

Tingjia Zhang[1*], Bo Chen[2], Shengzhong Liu[1†], Fan Wu[1], Guihai Chen[1]
[1]Shanghai Jiao Tong University,[2]University of Illinois Urbana-Champaign
[*]First Author [†]Corresponding Author

Abstract

Recent breakthroughs in 3D Gaussian Splatting (3DGS) have advanced neural rendering with high fidelity and efficiency. However, its performance degrades severely in large-scale scenes due to the increasing computational workload of tile-based rasterization. Existing acceleration approaches either require costly scene re-training or focus only on the rasterization stage of the pipeline, overlooking general pipeline redundancy in real-time rendering. Through a comprehensive analysis, we identify three primary sources of redundancy and low GPU utilization: 1) redundant inter-frame pre-processing, 2) viewpoint redundancy, and 3) imbalanced tile load distribution. To address these issues, we propose CaT-GS, a novel and efficient 3DGS rendering pipeline. CaT-GS introduces a speculative multi-frame preprocessing method to eliminate redundant computations across consecutive frames, and an inter-frame caching mechanism to eliminate viewpoint redundant rendering stages. Furthermore, it redistributes the rasterization tasks with a dedicated CUDA kernel to mitigate tile load imbalance and boost GPU utilization. Extensive experiments on large-scale scenes demonstrate that CaT-GS achieves a speedup of up to 10× over the original 3DGS and up to 70% over previous state-of-the-art (SOTA) methods, establishing a new benchmark for high-fidelity, real-time rendering of large-scale scenes.

1. Introduction

Driven by the recent breakthrough of 3D Gaussian Splatting (3DGS) [13], 3D reconstruction and neural rendering have exhibited significant potential across various interactive applications, including autonomous driving [11, 20, 36], immersive streaming [4, 29, 33], and virtual tours [16, 31]. While 3D Gaussian Splatting (3DGS) represents a significant advance over Neural Radiance Fields (NeRF) [24] in both reconstruction quality and rendering speed, achieving high frame rates remains challenging for certain real-world applications, such as virtual world and large-scale city reconstruction. These scenarios typically involve larger and more complex scenes compared to standard datasets, which substantially increase computational overhead and rendering latency. Therefore, it is necessary to investigate methods for accelerating 3DGS rendering in large-scale scenes.

To enhance 3DGS performance, several existing works [21, 23] aim to reduce the computational overhead from excessive Gaussians through model compression or pruning. These methods primarily focus on improving the Gaussian model's representation efficiency to reduce storage and Gaussian count, rather than directly optimizing its rendering pipeline. Moreover, these methods typically require scene re-training to generate efficient representations; such requirement for re-training impairs their practicality in real-world deployment. Another line of research aims to minimize computational redundancy within the rendering pipeline by analyzing the contribution of each Gaussian to specific tiles during rasterization to reduce the unnecessary computations. Prior works [5, 22, 32] primarily optimize the inefficiency of redundant Gaussian in rasterization by designing fine-grained Gaussian-tile intersection and overlap determination.

However, these approaches overlook critical aspects of actual deployment pipelines in applications. First, a significant limitation is their disregard for **viewpoint coherence**. The reported high-speed rendering results are mostly measured in controlled settings using fixed camera pose sets from predefined datasets, the same poses as used in capturing the ground-truth image of the dataset. This setup is only suitable for assessing reconstruction quality of the rendering algorithm, but lacks fidelity for benchmarking rendering efficiency under continuous and arbitrary camera trajectories in interactive applications where significant redundancies arise in the pre-processing stage (*e.g.*, frustum culling and sorting). Secondly, previous methods fail to fully address the issue of **low GPU utilization** caused by load imbalance. While Flash-GS [5] improves compute-memory balance by overlapping Gaussian loading and computing, it fails to address the low GPU utilization issue caused by tile-

level load imbalance. Alternatively, ADR-GS [32] attempts to alleviate load imbalance by retraining the model with an additional constraint loss on tile-aware Gaussian distribution; however, this approach leads to a degradation in reconstruction quality. We conduct a thorough analysis of the rasterization step and identify multiple sources of redundancy in standard rendering pipelines.

1. **Inter-Frame Redundancy:** In a typical 3DGS deployment, scenes are rendered as sequential, continuous frames for interactive applications, as the viewing pose changes gradually. When the frame rate is high, consecutive frames contain very similar rasterization information derived from pre-processing. These nearly identical pre-processing computations across frames lead to significant inter-frame redundancy.
2. **Uneven Tile Distributions:** 3DGS achieves high rasterization efficiency through its tile-based approach, which partitions the image and renders multiple tiles concurrently to leverage the parallel architecture of modern GPUs. However, in large and complex scenes, the distribution of Gaussian numbers across tiles can be highly biased. This leads to a severe load imbalance among GPU units during rasterization, resulting in overall GPU underutilization.

We therefore propose CaT-GS, the first 3DGS rendering pipeline that optimizes the interactive rendering process for large-scale scenes. To this end, CaT-GS introduces novel speculative multi-frame pre-processing and an inter-frame caching mechanism, which collectively reshape the rendering pipeline to mitigate inter-frame redundancy. Furthermore, CaT-GS incorporates a dynamic load redistribution strategy and an optimized rendering kernel to comprehensively solve the tile imbalance problem. Through evaluations on models from a large-scale UAV city reconstruction dataset as well as standard benchmarks, CaT-GS achieves up to $10\times$ speedup over the vanilla 3DGS pipeline and up to 70% speedup against existing SOTA methods.

Our main contributions in this paper include:

- We design a speculative rasterization pre-processing method that enables batched pre-processing for sequential frames to avoid full recomputations.
- We develop an inter-frame caching mechanism to reuse viewpoint-invariant features in 3DGS rendering, thereby bypassing redundant stages from the original pipeline.
- We alleviate tile load imbalance in 3DGS rasterization by proposing a task-splitting strategy and a dedicated CUDA rendering kernel to improve the GPU utilization during rasterization.
- CaT-GS achieves up to $10\times$ speed up compared to vanilla 3DGS pipeline and up to 70% speed up against existing SOTA baselines.

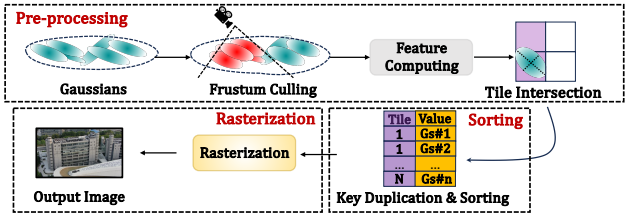


Figure 1. General 3DGS Rendering Pipeline, composed by pre-processing, sorting and rasterization stage.

2. Preliminaries and Background

2.1. 3DGS Rendering

The real-time rendering pipeline of 3D Gaussian Splatting (3DGS) can be decomposed into three main stages: 1) pre-processing, 2) sorting and 3) rasterization. This pipeline is meticulously designed to leverage parallel computing architectures, particularly modern GPUs.

1) Pre-processing: In the pre-processing stage, 3D Gaussians are prepared for subsequent image-plane projection and rasterization. First, 3DGS applies **frustum culling** to discard the Gaussians outside the viewing volume and reduce computational load. Then the shape and color feature is estimated with 3D covariance matrix and SH functions. The 2D covariance matrix $\mathbf{\Sigma}'$ is computed from 3D covariance Σ , which formulates the ellipse shape on the splatted plate. Since 3DGS renders images in a tile-level parallelism, the screen is divided into fixed-size tiles (*e.g.*, 16×16 pixels). Each Gaussian is assigned to all tiles whose projected 2D bounding boxes intersect, which creates an initial list of Gaussians per tile.

2) Sorting: In the second stage, the associated Gaussians in each tile are sorted by their depth (typically the z-coordinate in the camera space). This ensures correct back-to-front blending during rasterization.

3) Rasterization: The final rasterization stage generates the output image. The render kernel is parallelized across pixelsto utilize the GPU capacity. For each pixel p, all Gaussians within its corresponding tile are loaded and processed in from near to far depth order. Each Gaussian computes an α value $\alpha_i(p) = \sigma_i g_i(p)$ to measure its contribution, where g_i is the contribution of the projected 2D Gaussian at pixel p:

$$g_i = e^q, q = -\frac{1}{2}(p - \mu_{i_{2D}})\mathbf{\Sigma}_{\mathbf{i_{2D}}}^{-\mathbf{1}}(\mathbf{p} - \mu_{\mathbf{i_{2D}}})^{\mathbf{T}}. \quad (1)$$

If $\alpha_i > \frac{1}{255}$, then the Gaussian is included in the alpha compositing of the pixel color C. The overall color of a pixel is obtained through alpha blending by color c_i and weight α_i:

$$C(p) = \sum_{i \in \mathcal{N}} c_i \alpha_i(p) \prod_{j=1}^{i-1}(1 - \alpha_j(p)). \quad (2)$$

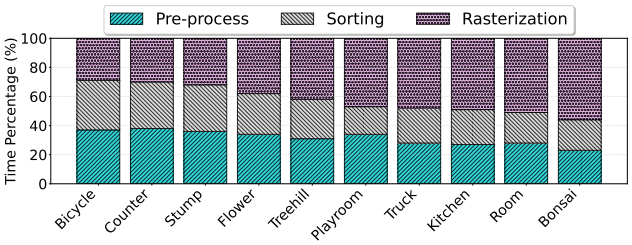


Figure 2. The ratio of rendering time spent on pre-processing, sorting, and rasterization in 3D Gaussian Splatting (3DGS). Note that this ratio depends on both the model and the viewpoint.

2.2. Accelerate 3DGS Rendering

Previous research has primarily focused on exploiting geometric redundancy to reduce computational overhead. Adr-GS [32] addresses the insufficiency of Gaussian tile intersection determination in vanilla 3DGS by designing a method with tighter axis-aligned bounding boxes (AABBs). This enables more precise intersection judgment and yields a shorter redundant rasterization list. Following this, Flash-GS [5] introduced an adaptive intersection judgment mechanism to further compress redundant tasks via opacity pruning. Collectively, these prior works have made a great step in decreasing intra-frame redundancy of 3DGS rendering.

However, while existing methods extensively explore intra-frame redundancy, few have considered inter-frame redundancy in consecutive rendering. The limitation is that these methods overlook this inter-frame redundancy, which represents a significant opportunity to reduce overhead in the sorting and pre-processing stages. Yet, as we profile several models from MipNeRF360 [2] at random example viewpoint in Figure 2, the sorting and pre-processing stage also accounts for a significant portion of the rendering time.

Alternatively, at the inter-frame level, we identify redundancy in the pre-processing and sorting stages. When the camera moves only slightly, the outcomes of these stages often remain largely unchanged. For instance, frustum culling, which identifies the visible part of the Gaussian model, typically produces similar results across consecutive frames when the rendering frame rate is high. Similarly, the depth-based sorting of Gaussians remains consistent when the camera viewpoint does not change drastically. This presents an opportunity to avoid repetitive computations by processing rendering frames in batches.

Second, we note remaining inefficiencies in the rasterization stage due to tile load imbalance. This issue arises from the non-uniform distribution of Gaussians among regions. Regions with high-frequency details tend to incur significantly higher rendering loads than others, as depicted in Figure 3. 10% tiles may contain over 50% compute load in some cases.This load imbalance is evident across various models, as shown by the CDF in the figure. Crucially, the heavy load tiles may cause delay in the parallel rendering.

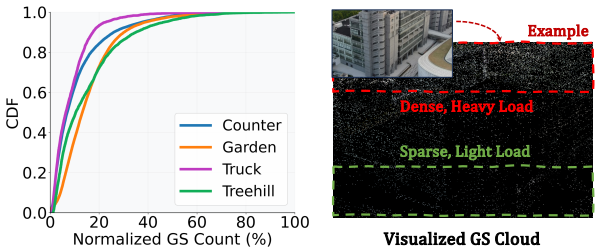


Figure 3. Unbalanced tile-load distribution across scenes. The left figure shows CDF on GS counts per-tile, right figure provides an example of imbalanced load.

3. CaT-GS Method

3.1. Overview

CaT-GS designs a novel pipeline for streamed rendering of 3D Gaussian Splatting (3DGS) in interactive applications, which eliminates inter-frame redundancy and improves GPU utilization. Unlike prior works, CaT-GS processes sequential frames in groups. The first frame in a group is designated as the *key frame*, and the subsequent ones as *sub-frames*, with distinct rendering pipelines applied to each type of frame.

As shown by the red line in Figure 4, the key frame pipeline begins with a preprocessing stage that generates rasterization information reusable across the entire frame group. This is achieved through our proposed viewpoint-speculative preprocessing (Section 3.2), which performs motion prediction based on viewpoint changes and applies a traced Gaussian intersection judgment to ensure the resulting Gaussian list covers all Gaussians needed throughout the frame group. In contrast, the sub-frame rendering pipeline skips several stages via an inter-frame caching strategy (Section 3.3). When viewpoint movement is small, sub-frames avoid repeated sorting and frustum culling by reusing information from the key frame, as they remain largely consistent under small viewpoint changes.

To further address the deficiency caused by tile load imbalance and enhance the GPU utilization during rasterization, we introduce a task-balancing method along with a high-performance rendering kernel (Section 3.4).

3.2. Speculative Multi-Frame Pre-Processing

The conventional pre-processing stage performs a thorough computation of rasterization information for every single frame. Alternatively, CaT-GS introduces a Speculative Multi-frame Pre-processing method. This approach generates rasterization information for a batch of sequential frames by leveraging motion prediction and fine-grained Gaussian intersection judgment. Furthermore, a motion-adaptive scheduling mechanism is incorporated to enhance robustness in extreme scenes and guarantee rendering quality.

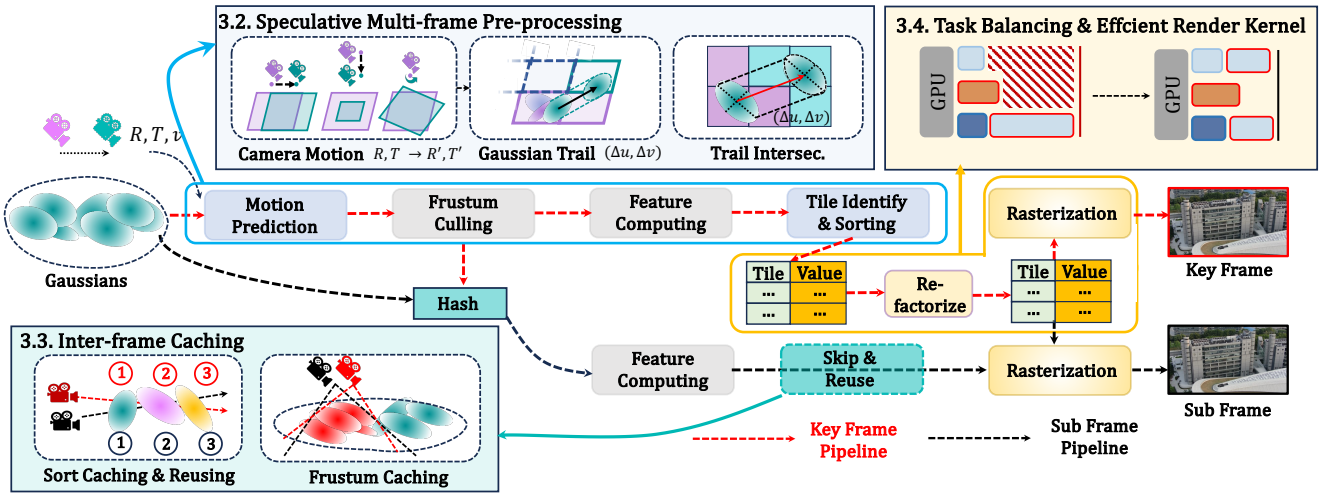


Figure 4. Overview of CaT-GS.

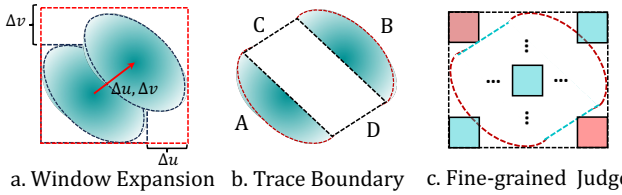
3.2.1. Motion Prediction

To extend the validity of pre-processing results across multiple adjacent frames, we begin by analyzing how frame contents change across frames, as this is determined by the input viewpoint's movement. We first establish a model to quantify the effect of viewpoint movement. We define three types of viewpoint motions that can occur in any instantaneous movement: *panning*, *zooming*, and *rotation*. Let the world coordinate be denoted by the translation vector $T = (T_x, T_y, T_z)$ and the orientation by a 3×3 rotation matrix $R = (R_1, R_2, R_3)$, and inner parameter matrix K.

By measuring the changes in the translation vector T and rotation vector R of the input camera over a short time interval, we can estimate the camera motion across a few frames. Within this short time interval, the motion can be considered nearly constant and unique. Suppose within the target time interval, the camera moves from position T to T's and R to R'. The affine transformation that maps a Gaussian 3D point at (x_g, y_g, z_g) to the pixel coordinates (u, v) to (u', v') can be derived from Equation 3. We can then derive a motion vector $(\Delta u, \Delta v) = (u', v') - (u, v)$ with respect to the coordinate change.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \frac{1}{z_c} K(R \cdot \begin{bmatrix} x_g \\ y_g \\ z_g \end{bmatrix} + T). \tag{3}$$

This motion vector $(\Delta u, \Delta v)$ represents the pixel-level trajectory of a Gaussian center across a sequence of frames. The region swept by the Gaussian along this trajectory is termed the Gaussian trace. Although the projected shape of the Gaussian changes slightly due to rotation, we treat it as constant since its variation is negligible.



a. Window Expansion b. Trace Boundary c. Fine-grained Judge

Figure 5. Steps to apply fine-grained intersection detection to a Gaussian trail.

3.2.2. Fine-grained Intersection Detection

Unlike previous methods that judge the intersection per Gaussian, CaT-GS determines tile intersection based on the entire swept area—the region encompassed by a Gaussian's motion along the vector $(\Delta u, \Delta v)$, as we call this the **Gaussian trail**. During rendering, each pixel iterates over all relevant Gaussians within the tile to compute its color. By predicting motion and judging intersection with the Gaussian trail, CaT-GS ensures that the rendered list includes all necessary Gaussians for a batch of frames.

To determine the intersection of a Gaussian trace, we first expand the bounding window by Δu and Δv, as illustrated in Figure 5a. Subsequently, we perform intersection tests on all tiles overlapped by this expanded rectangle. The boundary of a Gaussian trail comprises four segments: two half-ellipses centered at (u, v) and $(u + \Delta u, v + \Delta v)$, labeled A and B in Figure 5b, and two parallel lines C and D that connect the tangent points of the two ellipses along the motion vector. These four boundary segments can be mathematically estimated using 2D covariance Σ derived from Gaussian features. The two half-ellipses are implicitly defined by the 2D inverse covariance matrix Σ^{-1} of the Gaussian (as in Equation (4)) and a distribution clamp boundary parameter p. Specifically, each ellipse satisfies the Equation (5). From this implicit function, we can also derive the equations for lines C and D (as in Equation (6)) by computing

the tangent points (x_t, y_t) and $(-x_t, -y_t)$ with respect to the motion vector $(\Delta u, \Delta v)$.

$$\Sigma^{-1} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \tag{4}$$

$$ax^2 + bxy + cy^2 = 2p, \tag{5}$$

$$\begin{cases} X(t) = x_t + t \cdot \Delta u \\ Y(t) = y_t + t \cdot \Delta v \end{cases}, \quad t \in [0, 1]. \tag{6}$$

For each tile block in the expanded window, the intersection of two ellipses and lines is identified. The tile is valid for a Gaussian trace if: 1) The block intersects with any of the boundaries; 2) The block's center lies within the rectangle formed by the two ellipse centers.

3.2.3. Motion Adaptive Adjustment

A fundamental prerequisite for CaT-GS is the similarity between adjacent frames. When motion becomes too drastic, the speculative preprocessing would incorporate excessive redundant information; therefore, a boundary condition is automatically set to avoid such extreme cases. The displacement of a Gaussian point induced by camera motion generally depends on its projected depth, denoted as z_c in Equation (3). The closer a Gaussian is to the camera, the more significantly it moves in response to camera motion. Thus, to avoid interference from Gaussians that are too close to the camera, we assess the validity of the motion speculation by only considering Gaussians whose depth is beyond a threshold d.

We set the target rendering frame rate to F_{target}, which is higher than the frame rates in our test cases, and initialize the speculation window to W_{initial} frames—meaning one key frame followed by W_{initial} subsequent sub-frames. If any Gaussian deeper than a threshold d exhibits movement $M = \sqrt{\Delta u^2 + \Delta v^2}$ greater than the tile size l, the motion is considered too drastic. In such cases, the window size is reduced to $\lfloor W_{\text{initial}} \times \frac{l}{M} \rfloor$ to satisfy the constraint. Should the motion between two adjacent frames alone exceed the limit, resulting in $\lfloor W_{\text{initial}} \times \frac{l}{M} \rfloor = 0$, the speculation strategy is skipped for the current frame.

3.3. Inter-frame Caching

By leveraging a speculative pre-processing stage that generates rasterization information for a batch of frames, we can mitigate redundant computation. In vanilla 3DGS, the preprocessing stage contains three steps: 1) frustum culling, 2) feature computation, and 3) sorting and list generation. Only feature computation needs to be performed per frame, as the color, shape, and opacity of the Gaussians are highly viewpoint-dependent. For other steps, CaT-GS employs a caching mechanism to avoid redundant computations.

3.3.1. Frustum Caching

For a continuous sequence of frames, frustum culling only needs to be performed once, as the speculative preprocessing stage has already identified the required subset of Gaussians of the model. Therefore, in subsequent frames, frustum culling can be omitted entirely. To implement this caching mechanism, CaT-GS calculates and stores the hashing indices of the visible Gaussians from the first frame in the batch. For the following frames, CaT-GS no longer needs to iterate through the entire set of Gaussians and compute their affine transformations to determine visibility. Instead, it directly utilizes the pre-culled set. This optimization yields greater performance benefits in large scenes, where the culled set typically represents only a small portion of the entire model.

3.3.2. Sort Caching

Regarding the sorting step, an intuitive fact is that the depth order of Gaussians remains mostly consistent across frames within a short time, as camera position and rotation change very little under high frame rates. This insight led to the design of our sort caching mechanism. In CaT-GS, the first frame in a batch performs sorting and list generation. After rasterization, this list is cached and reused for subsequent frames, as it already contains all the necessary rasterization information for the entire batch. Consequently, later frames skip the sorting stage and rasterize directly using the cached list. Since reusing the list for multiple frames unavoidably introduces redundancy, as each tile contains unneeded points from the current viewpoint, the fine-grained intersection and motion-adaptive scheduling control the redundancy to an acceptable level, as discussed in Section 5.4. More importantly, sort caching also eliminates the need for repetitive tile intersection judgment and key-duplication in the preprocessing stage for subsequent frames, which significantly accelerates the overall rendering process.

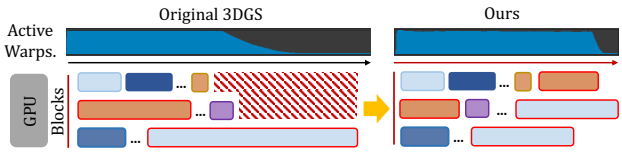


Figure 6. Task redistribution and efficient kernel improve rasterization performance. Active warps row shows an example result profiled from Nsight Compute.

3.4. Load-Aware Task Splitting

In this section, we focus on performance bottleneck caused by tile load imbalance. As analyzed in Section 2, the distribution of the Gaussian model leads to an uneven computational workload of each rasterization tile in parallel rendering of the GPU, resulting in low utilization.

Stalls in Parallel Rendering This issue stems from the fundamental execution strategy of the GPU. During rendering, each GPU unit—known as a Streaming Multiprocessor (SM)—concurrently launches multiple tile rasterization tasks (referred to as block launches). Each launched block is executed entirely on the same SM that launched it, meaning that the rendering task for a given tile is confined to a single SM. To fully utilize computational resources, all SMs on the GPU immediately launch a new block as soon as one is completed. This process can be simplified as shown in Figure 6, which illustrates 3 parallel compute units launching render blocks. However, since a small number of tasks often accounts for the majority of the computational overhead, this execution strategy can lead to stalls. Larger tasks in the distribution may become stuck on a single SM, delaying overall completion.

Tile-level Splitting To prevent large task blocks from stalling on a single Streaming Multiprocessor (SM), an intuitive approach is to split heavy tile tasks into smaller ones, allowing the GPU to schedule them across multiple SMs. In other words, for regions with high computational loads, we assign more SMs to collaboratively perform the computations, thereby accelerating the rendering process. However, this splitting cannot be done arbitrarily due to the order dependency in alpha blending. As described by Equation (2), the process originally requires ordered computation, which cannot be parallelized at the pixel level. To tackle such problem, we refactor the rasterization step in 3DGS. Specifically, to obtain color C of a pixel by blending N ordered points (splats) overlapping the pixel, we split the rendering task into k subdivisions. These correspond to subsets of the total list of N points, denoted as $N_1, N_2...N_k$, The alpha blending process can then be performed in a batched manner, which can be formulated as follows:

$$C = \sum_{i=1}^{k} (\sum_{j=1}^{|N_i|} c_j \alpha_j T_j) R_i, \tag{7}$$

$$R_i = \prod_{m=1}^{i-1} (1 - A_m), A_m = \sum_{j=1}^{|N_m|} \alpha_j T_j. \tag{8}$$

In this formulation, each tile block now processes one of the k subdivisions of size N/k instead of handling the entire original task. Therefore, for tiles with a dense Gaussian distribution, our model splits the original rendering task into smaller subtasks that can be scheduled efficiently by the GPU. These subtasks can be allocated to multiple SMs for parallel rendering, thereby preventing the workload from stalling on a single SM and improving overall utilization.

Split Render Kernel To implement in rasterization kernel, we obtain a load-aware splitting strategy. For a render list generated in the previous stage, the total computational load can be determined by its length L. The average load per tile is then $l = L/t$, where t is the number of tiles. We identify any tile containing more than $2l$ Gaussians to rasterize as a large tile, and subsequently split it into smaller tiles targeting a size of l. During rasterization, CaT-GS launches thread blocks equivalent to the total number of split tasks. Each task is assigned to a tile index t and a slice index k. A thread block computes a portion of the expression $\sum_{j=1}^{|N_k|} c_j \alpha_j T_j$ which corresponds to the part of the original task it is responsible for. After completing this partial computation, each block stores its slice color C_k and slice radiance residual A_k for the final image composition.

4. Implementation

We implemented the CaT-GS framework with ∼3K CUDA and Python code. The render pipeline is built via Pytorch [26] framework, and the rasterization kernel is compiled via Pytorch CUDA extension. To collect a sequential trace of rendering, we use the SIBR_Viewers renderer to collect user interactive trace while viewing the Gaussian models. The block size of rendering is set to 16×16. The target frame rate F_{target} is set to 120, threshold d to 0.4 and initial window size W_{initial} to 4 frames.

5. Evaluation

5.1. Experiment Setup

Test Platform We tested our method on the platform of a Nvidia Geforce RTX 5090 GPU and an AMD Ryzen 9 9950X CPU with Ubuntu 24.04 OS.

Datasets To comprehensively evaluate our rendering method, we employ both the standard public 3DGS datasets and the self-collected UAV city reconstruction dataset, which features a larger real-world scene scale. This dataset will be open-sourced in the future. The standard 3DGS datasets include Tanks & Temples [15] dataset, MipNeRF-360 [2] dataset, and Deep Blending [10] dataset. MipNeRF-360 dataset includes four indoor and five outdoor scenes, each featuring a complex central object or area with a detailed background. The Tanks & Temples and Deep Blending datasets include 4 extra scenes.

To assess sequential rendering performance, we collected a series of user traces consisting of camera parameters for each model. Specifically, we gathered 10 trajectories per model, each spanning 2000 consecutive frames. These traces were recorded via interaction with the Gaussian models from the aforementioned datasets using the SIBR_Viewer. The camera movements encompass a full range of motion types—including panning, zooming, and rotation—at varying speeds. All traces were captured at 120 FPS, and the resolution is set to 1920×1080.

Evaluation Metrics As a render method, we evaluate CaT-GS on both render efficiency and quality. We use the average frame rate (*i.e.*, FPS) as the efficiency metric. To eval-

Table 1. Average FPS across all traces on each model mentioned in Section 5.1. The best result is marked as grey cell color. The improvement against the previous SOTA result is marked by ↑ after our result. Model size indicates the number of Gaussians in the model.

	Truck	Train	Counter	Garden	Playroom	UAV-1	UAV-2	UAV-3	UAV-4	UAV-5
Model Size	2,541,226	1,026,508	1,222,956	4,232,564	2,546,116	6,895,821	7,189,662	5,926,371	8,324,723	7,396,621
3DGS	63.4	89.4	122.3	92.6	68.1	25.2	23.2	54.3	47.2	36.1
ADR-GS	425.2	482.1	498.1	204.1	593.2	116.8	98.3	214.2	132.1	111.1
Flash-GS	478.3	528.3	606.3	201.1	634.2	129.2	113.1	241.1	167.3	132.3
Ours-w/o	494.4	550.2	626.1	242.4	654.8	155.3	133.2	249.6	174.2	150.1
Ours	736.5 (↑ 54%)	892.2(↑ 68%)	907.5 (↑ 49%)	295.5 (↑ 46%)	922.3 (↑ 45%)	241.5 (↑ 83%)	202.5(↑ 78%)	378.5(↑ 64%)	272.1 (↑ 62%)	217.3 (↑ 65%)

Table 2. Render quality on video-captured datasets. The ∗ after ADR-GS means the model is retrained due to the need for the method. The other baselines use the same model trained with 3DGS.

	Truck		Train		UAV-1		UAV-2		UAV-3		UAV-4		UAV-5	
	PSNR	SSIM	PSNR	SSIM	PSNR	SSIM	PSNR	SSIM	PSNR	SSIM	PSNR	SSIM	PSNR	SSIM
3DGS	28.21	0.858	25.82	0.872	30.15	0.935	29.89	0.889	29.53	0.909	30.37	0.943	28.34	0.872
ADR-GS*	26.50	0.824	23.92	0.768	29.12	0.883	29.31	0.903	27.34	0.878	28.69	0.872	27.26	0.831
FlashGS	28.19	0.856	25.78	0.869	30.09	0.931	29.86	0.886	29.50	0.905	30.32	0.940	28.30	0.869
Ours-Key	28.19	0.856	25.78	0.869	30.09	0.931	29.86	0.886	29.50	0.905	30.32	0.940	28.30	0.869
Ours-Sub	28.16	0.852	25.74	0.867	30.05	0.926	29.83	0.884	29.46	0.903	30.30	0.939	28.29	0.868

uate render quality, we employ two standard image quality metrics: Peak Signal-to-Noise Ratio (PSNR), Structural Similarity Index Measure (SSIM).

5.1.1. Baselines

We compare CaT-GS with the baselines below:

- **3DGS [13]:** We compare to the original 3DGS render method from the released source code of the 3D Gaussian splatting work.
- **ADR-GS [32]:** We compare to the previous ADR-GS baseline, as we re-trained the model with released source code.
- **Flash-GS [5]:** We compare to previous SOTA method in software 3DGS rendering optimization. The baseline is also implemented with the released source code.

5.2. Render Efficiency Improvement

To evaluate the render speed, we compare our model against our baselines in average FPS across the viewpoint trace. The result is shown in Table 1. CaT-GS reaches the highest average fps due to the redundancy cut down and balanced task. All results of CaT-GS across large scenes in UAV dataset reach over 200Fps, while other baselines fail to ensure 120Fps. Notably, CaT-GS-w/o represents our model without the inter-frame caching strategy, which also represents the worst case for CaT-GS. It can be seen that CaT-GS-w/o still reaches over 120FPS across large scale scenes due to efficiency improvement by tile task scheduling. Compared to the vanilla 3DGS, CaT-GS reaches up to $10\times$ speed up. From scene prospective, for the shown scenes from the standard datasets, which generally have Gaussian numbers between $1-4$ million, the improvement of CaT-GS against previous SOTA baseline is around 50%. For larger scenes in the UAV dataset, which have Gaussian number over 7 million, the improvement is generally over 60% and can go up to over 80%. CaT-GS generally exhibits a larger advantage as Gaussian model scales up, since a more complex model also involves heavier pre-processing and sorting stages.

5.3. Render Quality

To evaluate the rendering quality of our method, we built our testing pipeline upon the original 3DGS evaluation framework. Specifically, we captured the original camera trajectory for the source views of each model, aligning each rendered frame with its corresponding ground truth image. We utilize datasets extracted from videos, as it offers pose-continuous ground truth frames. For "truck" and "train" from Tanks & Temples dataset, the original ground truth was recorded at 30 FPS, we interpolated the frame rate to 120 FPS. The same 120 FPS interpolation was applied to UAV dataset as well.

To validate the speculative pre-processing stage, we assessed the rendering quality of both key-frames and sub-frames independently. This was done by aligning the timing of key-frames and sub-frames with the timestamps of available ground truth images, enabling a frame-accurate quality evaluation. The results are summarized in Table 2. ADR-GS's render quality degrades to original 3DGS due to its retrained model's lower accuracy. Flash-GS maintains comparable quality with only minimal metric degradation, which is attributable to its opacity-aware Gaussian culling. The key-frame rendering of CaT-GS adopts the same strategy, resulting in the same render quality to Flash-GS. For CaT-GS 's sub-frames, a slightly larger degradation occurs due to potential mispositioning of several Gaussians in the speculative process. Overall, the rendering quality degradation of CaT-GS is negligible compared to previous baseline models, demonstrating the effectiveness of the speculative design.

5.4. Ablation Study

Technique Ablation We evaluate the impact of the two major mechanisms in our method: the inter-frame caching and the load-aware task splitting. For a detailed analysis, we select one model from each standard dataset and three models from the UAV dataset. The results are presented in Table 3. The variant labeled "Ours-w/cache" denotes our system without inter-frame caching. As the results show, this mechanism improves overall performance by up to 80%. Similarly, "Ours-w/split" denotes the method without load-aware task splitting. The results indicate that this optimization to the rasterization stage contributes an overall performance improvement of over 10%.

Table 3. Impact of major mechanism in CaT-GS.

AvgFPS.	Garden	Truck	Playroom	UAV-1	UAV-2	UAV-3
Ours-Full	295.5	736.5	922.3	241.5	202.5	378.5
Ours-w/cache ↓	242.4	494.4	654.8	155.3	133.2	249.6
Ours-w/split↓	279.4	672.3	840.6	210.3	178.4	337.6

Stage Acceleration Analysis We further break down the rendering time to better understand the improvements. The modification made by CaT-GS to the rendering pipeline involves four parts in 3D Gaussian splatting (3DGS), namely pre-processing, sorting, tile identification, and rasterization. We compare the average acceleration of each stage against the original 3DGS implementation. The results are shown in Table 4. In the pre-processing stage, Flash-GS improves upon the original 3DGS by fusing feature computation and key duplication into a single kernel. The key-frame pre-processing in CaT-GS is slightly slower than Flash-GS due to the additional computation involved in speculative pre-processing. In contrast, the sub-frame processing in CaT-GS only requires computing the features of the specific Gaussian group selected during frustum culling in the key-frame, achieving a speed-up of 6.8× compared to the original 3DGS. As for the sorting stage, all methods utilize the *RadixSort* method. The main improvement comes from the shorter rasterization list resulting from more fine-grained intersection checks. The key-frame processing in CaT-GS is slightly slower due to a longer list that includes extra Gaussians for subsequent frames. However, since sub-frame processing does not require sorting, the average speed-up for this stage reaches up to 7.2× when averaged across CaT-GS. In the rasterization stage, though CaT-GS computes slightly more redundant points, it still achieves up to 30% acceleration compared to the previous SOTA baseline, and an average 5× speed up compared to original 3DGS.

Redundancy Analysis To further evaluate the rendering performance, we analyze the render list generated during pre-processing to assess the impact of the additional computation in our speculative multi-frame pre-processing. As shown in Table 5, the average render list length, which represents the number of Gaussian-tile intersections, is significantly reduced in Flash-GS. This reduction is achieved through more precise intersection tests with ellipses, which substantially decreases the computational load. In contrast, CaT-GS's key-frame speculative pre-processing accounts for Gaussian motion across sub-frames, inevitably introducing some redundancy into the render list. However, through fine-grained intersection checks and motion-adaptive scheduling, our design successfully limits this redundancy to an acceptable level, which only increases by around 10%.

Table 4. Stage acceleration compared to original 3DGS.

Speedup ↑	Pre-processing	Sorting	Tile Identify	Rasterization
3DGS	1.0×	1.0×	1.0×	1.0×
FlashGS	3.2×	3.3×	1.7×	4.2×
Ours-key	3.1×	2.9×	1.3×	5.3×
Ours-sub	6.8×	0*	0*	5.3×

Table 5. Render list length across models.

List Len.($\times 10^3$)	Garden	Truck	Playroom	UAV-1	UAV-2	UAV-3
3DGS	18,593	18,969	22,550	37,821	52,336	34,532
FlashGS	6,682	4,124	3,818	10,766	12,101	8,416
Ours-key	7,032	4,289	4,182	11,222	12,886	8,628
Ours-sub	7,032	4,289	4,182	11,222	12,886	8,628

6. Related Works

3DGS [13] has marked a significant advance in 3D reconstruction and neural renderin, and has become commonly used in interactive applications [7, 27, 28]. Subsequently, model compression techniques [1, 8, 9, 17, 21, 30, 35] have been proposed to optimize 3DGS models by reducing their size and the number of Gaussians. Other research efforts focus on enhancing the rendering efficiency of 3DGS. Some approaches address this from a software perspective [5, 12, 14, 25, 32, 34], while others target hardware optimizations [6, 18]. Several studies further improve 3D reconstruction and rendering by exploiting frame redundancy [3, 6, 19]. Notably, Lumina [6] is a hardware-oriented optimization work for 3DGS that specifically addresses redundancy in the sorting stage during rendering.

7. Conclusion

This paper presents CaT-GS, a novel rendering pipeline that significantly accelerates 3D Gaussian Splatting for large-scale scenes. By introducing a speculative multi-frame pre-processing mechanism with inter-frame caching, CaT-GS effectively eliminates redundant computations across consecutive frames. It further addresses tile-level load imbalance through a dynamic task-splitting strategy and an optimized CUDA kernel. Extensive experiments demonstrate that CaT-GS achieves a speedup of up to 10× over vanilla 3DGS and up to 70% over prior SOTA methods.

Acknowledgement

This work was supported in part by National Key R&D Program of China (No. 2023YFB4502400), in part by China NSF grant No. 2472278, 62441236, 62372296, 62432007, 2332014, 2332013, and U25A6024, in part by Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM103), in part by Alibaba Group through Alibaba Innovation Research Program, in part by Tencent Rhino Bird Key Research Project, and in part by Shanghai QiYuan Innovation Foundation.

References

[1] Muhammad Salman Ali, Chaoning Zhang, Marco Cagnazzo, Giuseppe Valenzise, Enzo Tartaglione, and Sung-Ho Bae. Compression in 3d gaussian splatting: A survey of methods, trends, and future directions. *arXiv preprint arXiv:2502.19457*, 2025. 8

[2] Jonathan T Barron, Ben Mildenhall, Dor Verbin, Pratul P Srinivasan, and Peter 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. 3, 6

[3] Xingyu Chen, Xinmin Fang, Shuting Zhang, Xinyu Zhang, Liang He, and Zhengxiong Li. You only render once: Enhancing energy and computation efficiency of mobile virtual reality. In *Proceedings of the 23rd Annual International Conference on Mobile Systems, Applications and Services*, pages 263–276, 2025. 8

[4] Zihao Chen, Longhao Zou, and Xiaofeng Tao. 3dgv: 3d gaussian splatting-based holographic video streaming over wireless networks. In *2025 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pages 1–6. IEEE, 2025. 1

[5] Guofeng Feng, Siyan Chen, Rong Fu, Zimu Liao, Yi Wang, Tao Liu, Boni Hu, Linning Xu, Zhilin Pei, Hengjie Li, et al. Flashgs: Efficient 3d gaussian splatting for large-scale and high-resolution rendering. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 26652–26662, 2025. 1, 3, 7, 8

[6] Yu Feng, Weikai Lin, Yuge Cheng, Zihan Liu, Jingwen Leng, Minyi Guo, Chen Chen, Shixuan Sun, and Yuhao Zhu. Lumina: Real-time neural rendering by exploiting computational redundancy. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, pages 1925–1939, 2025. 8

[7] Huasong Han, Kaixuan Zhou, Xiaoxiao Long, Yusen Wang, and Chunxia Xiao. Ggs: Generalizable gaussian splatting for lane switching in autonomous driving. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 3329–3337, 2025. 8

[8] Alex Hanson, Allen Tu, Geng Lin, Vasu Singla, Matthias Zwicker, and Tom Goldstein. Speedy-splat: Fast 3d gaussian splatting with sparse pixels and sparse primitives. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 21537–21546, 2025. 8

[9] Alex Hanson, Allen Tu, Vasu Singla, Mayuka Jayawardhana, Matthias Zwicker, and Tom Goldstein. Pup 3d-gs: Principled uncertainty pruning for 3d gaussian splatting. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 5949–5958, 2025. 8

[10] Peter Hedman, Julien Philip, True Price, Jan-Michael Frahm, George Drettakis, and Gabriel Brostow. Deep blending for free-viewpoint image-based rendering. *ACM Transactions on Graphics (ToG)*, 37(6):1–15, 2018. 6

[11] Georg Hess, Carl Lindström, Maryam Fatemi, Christoffer Petersson, and Lennart Svensson. Splatad: Real-time lidar and camera rendering with 3d gaussian splatting for autonomous driving. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 11982–11992, 2025. 1

[12] Qiqi Hou, Randall Rauwendaal, Zifeng Li, Hoang Le, Farzad Farhadzadeh, Fatih Porikli, Alexei Bourd, and Amir Said. Sort-free gaussian splatting via weighted sum rendering. *arXiv preprint arXiv:2410.18931*, 2024. 8

[13] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Trans. Graph.*, 42(4):139–1, 2023. 1, 7, 8

[14] Shakiba Kheradmand, Delio Vicini, George Kopanas, Dmitry Lagun, Kwang Moo Yi, Mark Matthews, and Andrea Tagliasacchi. Stochasticsplats: Stochastic rasterization for sorting-free 3d gaussian splatting. *arXiv preprint arXiv:2503.24366*, 2025. 8

[15] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. Tanks and temples: Benchmarking large-scale scene reconstruction. *ACM Transactions on Graphics*, 36(4), 2017. 6

[16] Ohyang Kwon and Jeongmin Yu. Realistic and interactive virtual museum representation using 3d gaussian splatting. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, pages 185–192, 2025. 1

[17] Yongjae Lee, Zhaoliang Zhang, and Deliang Fan. Safeguardgs: 3d gaussian primitive pruning while avoiding catastrophic scene destruction. *arXiv preprint arXiv:2405.17793*, 2024. 8

[18] Sixu Li, Ben Keller, Yingyan Celine Lin, and Brucek Khailany. Gaurast: Enhancing gpu triangle rasterizers to accelerate 3d gaussian splatting. *arXiv preprint arXiv:2503.16681*, 2025. 8

[19] Yang Li, Jinglu Wang, Lei Chu, Xiao Li, Shiu-hong Kao, Ying-Cong Chen, and Yan Lu. Streamgs: Online generalizable gaussian splatting reconstruction for unposed image streams. *arXiv preprint arXiv:2503.06235*, 2025. 8

[20] Pei Liu, Qingtian Ning, Xinyan Lu, Haipeng Liu, Weiliang Ma, Dangen She, Peng Jia, Xianpeng Lang, and Jun Ma. Omnireason: A temporal-guided vision-language-action framework for autonomous driving. *arXiv preprint arXiv:2509.00789*, 2025. 1

[21] Xiangrui Liu, Xinju Wu, Pingping Zhang, Shiqi Wang, Zhu Li, and Sam Kwong. Compgs: Efficient 3d scene representation via compressed gaussian splatting. In *Proceedings of the 32nd ACM International Conference on Multimedia*, pages 2936–2944, 2024. 1, 8

[22] Tao Lu, Mulin Yu, Linning Xu, Yuanbo Xiangli, Limin Wang, Dahua Lin, and Bo Dai. Scaffold-gs: Structured 3d gaussians for view-adaptive rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 20654–20664, 2024. 1

[23] Saswat Subhajyoti Mallick, Rahul Goel, Bernhard Kerbl, Markus Steinberger, Francisco Vicente Carrasco, and Fernando De La Torre. Taming 3dgs: High-quality radiance fields with limited resources. In *SIGGRAPH Asia 2024 Conference Papers*, pages 1–11, 2024. 1

[24] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1):99–106, 2021. 1

[25] Michael Niemeyer, Fabian Manhardt, Marie-Julie Rakotosaona, Michael Oechsle, Daniel Duckworth, Rama Gosula, Keisuke Tateno, John Bates, Dominik Kaeser, and Federico Tombari. Radsplat: Radiance field-informed gaussian splatting for robust real-time rendering with 900+ fps. In *2025 International Conference on 3D Vision (3DV)*, pages 134–144. IEEE, 2025. 8

[26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019. 6

[27] Zhiyin Qian, Shaofei Wang, Marko Mihajlovic, Andreas Geiger, and Siyu Tang. 3dgs-avatar: Animatable avatars via deformable 3d gaussian splatting. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 5020–5030, 2024. 8

[28] Yulin Shen, Boyu Li, Jiayang Huang, David Yip, and Zeyu Wang. Gaussianshopvr: Facilitating immersive 3d authoring using gaussian splatting in vr. In *Proceedings of the 38th Annual ACM Symposium on User Interface Software and Technology*, pages 1–14, 2025. 8

[29] Yuang Shi. 3d gaussian-based immersive media streaming in networked extended reality. In *Proceedings of the 16th ACM Multimedia Systems Conference*, pages 356–360, 2025. 1

[30] Xiangyu Sun, Joo Chan Lee, Daniel Rho, Jong Hwan Ko, Usman Ali, and Eunbyung Park. F-3dgs: Factorized coordinates and representations for 3d gaussian splatting. In *Proceedings of the 32nd ACM International Conference on Multimedia*, pages 7957–7965, 2024. 8

[31] Yuan-Chun Sun. Streaming 3dgs virtual worlds in 6dof over next-generation networks. In *Proceedings of the 33rd ACM International Conference on Multimedia*, pages 13561–13565, 2025. 1

[32] Xinzhe Wang, Ran Yi, and Lizhuang Ma. Adr-gaussian: Accelerating gaussian splatting with adaptive radius. In *SIGGRAPH Asia 2024 Conference Papers*, pages 1–10, 2024. 1, 2, 3, 7, 8

[33] Nan Wu, Weikai Lin, Ruizhi Cheng, Bo Chen, Yuhao Zhu, Klara Nahrstedt, and Bo Han. Advancing immersive content delivery with dynamic 3d gaussian splatting. In *Proceedings of the 26th International Workshop on Mobile Computing Systems and Applications*, pages 109–114, 2025. 1

[34] Vickie Ye, Ruilong Li, Justin Kerr, Matias Turkulainen, Brent Yi, Zhuoyang Pan, Otto Seiskari, Jianbo Ye, Jeffrey Hu, Matthew Tancik, et al. gsplat: An open-source library for gaussian splatting. *Journal of Machine Learning Research*, 26(34):1–17, 2025. 8

[35] Zhaoliang Zhang, Tianchen Song, Yongjae Lee, Li Yang, Cheng Peng, Rama Chellappa, and Deliang Fan. Lp-3dgs: Learning to prune 3d gaussian splatting. *Advances in Neural Information Processing Systems*, 37:122434–122457, 2024. 8

[36] Xiaoyu Zhou, Zhiwei Lin, Xiaojun Shan, Yongtao Wang, Deqing Sun, and Ming-Hsuan Yang. Drivinggaussian: Composite gaussian splatting for surrounding dynamic autonomous driving scenes. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 21634–21643, 2024. 1

CaT-GS: Efficient 3DGS Rendering for Large-Scale Scenes with Inter-frame Caching and Tile Scheduling

Supplementary Material

Algorithm 1: Identify Tile Ranges

Data: Render length L, point list $point_list_keys$, ranges array $ranges$, task counter $task_count$, tile number $tile_number$
Result: Updated $ranges$ array, render label $label$.

$idx \leftarrow global\ thread\ index$;
$k \leftarrow gaussian\ counts\ threshold$;
$key \leftarrow point_list_keys[idx]$;
$currtile \leftarrow key \gg 32$;
$prevtile \leftarrow 0$;
if $idx = 0$ **then**
  $ranges[currtile].x \leftarrow 0$;
  $task_count \leftarrow tile_number$;
end
else
  $prevtile \leftarrow point_list_keys[idx-1] \gg 32$;
  if $currtile \neq prevtile$ **then**
    $ranges[prevtile].y \leftarrow idx$;
    $ranges[currtile].x \leftarrow idx$;
  end
end
if $idx = L - 1$ **then**
  $ranges[currtile].y \leftarrow L$;
end
Synchronize threads, split the task.;
if $currtile \neq prevtile$ ***and*** $idx \neq 0$ **then**
  $start \leftarrow ranges[currtile].x$;
  $end \leftarrow ranges[currtile].y$;
  $task_size \leftarrow end - start$;
  if $end - start > k$ **then**
    $split_num \leftarrow \lfloor task_size/k \rfloor$;
    $ranges[currtile].y \leftarrow start + (task_size - split_num * k)$;
    for $i \leftarrow 1$ ***to*** $split_num$ **do**
      $offset \leftarrow \text{atomicAdd}(task_count, 1)$;
      Write additional task range to $ranges[offset]$.
      Write tile information $currtile$ and split information i to $label[offset]$.
    end
  end
end

7.1. Detailed Implementation

7.1.1. Dataset Details

The UAV City Dataset is designed for benchmarking city reconstruction tasks using UAV-captured data. The dataset is constructed by flying a UAV around urban areas and extracting frames from recorded videos. It comprises 16 large-scale **real-world** scenes, each covering one to two city blocks, with a typical area exceeding 20,000 m². For each scene, the processed 3DGS dataset includes approximately 800 images along with the original video clip recorded at 30 FPS.

Since real-world scenes contain more complex geometries and textures—such as trees and grasslands—this dataset offers a more realistic simulation of practical applications compared to previous graphically modeled 3D city datasets like UrbanScene3D. This dataset will be open-sourced in recent months.

7.1.2. Task Splitting Implementation

We provide a detailed explanation of our design in Section 3.4 of the paper. The task splitting strategy is implemented by modifying the Tile-Identify step of 3DGS. As outlined in Algorithm 1, after generating the tile range for rasterization, we check whether the number of Gaussians in each tile exceeds a threshold k. If so, the tile task is split with respect to k, generating new subtasks. To identify the tile and the number of splits during rasterization, additional task information of tile index and partition number is stored in a $label$ array. The total task number is reformed to $task_count$.

During the rasterization stage, the kernel launches $task_count$ blocks. Blocks with an index below the original tile count proceed with rasterization as usual. For the additional blocks corresponding to split tasks, each one retrieves its respective parameters from the $label$ array. Based on the tile index and split index provided by the $label$ array, the output is written to the corresponding pixel and slice of the image tensor and radiance map to store the values of R and A as defined in Equation (6) of the paper.

A redundancy introduced by our task-splitting kernel involves the early-termination mechanism of rasterization. In standard 3DGS, this mechanism halts alpha blending once the accumulated radiance becomes sufficiently small, which can be intuitively understood as the light being absorbed by preceding Gaussians along the path. Consequently, in our split rendering approach, the latter portions of a split task may become unnecessary. To mitigate this, we employ two

key adjustments: first, the early-termination threshold is set higher for these additional task blocks; second, during task splitting, we allocate a larger size to the latter tasks, the first sub-task is set to a size of $task_size - split_num * k$ rather than k. Generally, this approach introduces some additional computational overhead compared to the original method. Nevertheless, the overall acceleration achieved still results in performance improvement.

7.2. Speculative Pre-processing Details

We will further demonstrate the effectiveness and robustness of our speculative multi-frame pre-processing design in Section 3.3. Figure 7 shows an example of the last frame in a cached batch, which illustrates the impact of our approach. The first column under w/o-cache shows the result without inter-frame caching—reflecting the original 3DGS pipeline. The second column presents the result of applying inter-frame caching directly without speculative pre-processing. As shown, stride-like visual artifacts appear due to missing Gaussians in certain tiles caused by viewpoint movement. The third column displays the result of CaT-GS, where the speculative pre-processing stage successfully incorporates all necessary points across the frame batch, as the render result remains consistent with the original pipeline.

Table 6. The impact of speculation window size.

UAV-1	FlashGS	None	W-2	W-4	W-8	W-16
Render Len.$\times 10^3$	10,766	10,766	10,982	11,222	15,482	25,934
Max Latency(ms)	7.7	6.3	6.4	6.6	9.1	14.1
UAV-2	FlashGS	None	W-2	W-4	W-8	W-16
Render Len.$\times 10^3$	12,101	12,101	12,382	12,886	15,926	24,621
Max Latency(ms)	8.6	7.1	7.2	7.5	10.9	13.4

We also conducted an ablation study on the speculation window size to evaluate its impact on performance and robustness. As an example, Table 6 presents the corresponding results. Rows labeled W-2 to W-16 show the outcomes when the speculation window size is set from 2 to 16. It can be observed that the rendering length increases nonlinearly as the window size grows, which is attributed to the quadratic decrease in image tile overlap across frames. Although the rendering quality remains consistent even with a window size of 16, the maximum latency increases significantly if the window size is set too large, and this will impact the rendering smoothness. Therefore, we set the initial window size to 4 and designed a Motion Adaptive Adjustment mechanism to dynamically control the window size. A more fine-grained adaptive window strategy could be developed in the future as a potential improvement to our work.

Figure 7. The speculative process includes the needed Gaussian in subsequent frames and avoids fault effects.

7.3. Compatibility With Pruning Method

In this section, we discuss the adaptability of render pipeline optimization and model pruning. Although model pruning suffers from certain deficiencies that limit its practicality, as outlined in Section 1, we demonstrate that it can be effectively combined with our rendering optimization method to yield situational advantages. Even in these scenarios, our approach maintains superior performance over previous baselines. To evaluate this, we applied the Compact-3DGS method to retrain and prune models on the UAV dataset. As shown in Table 7, UAV1-3 represents the original large-scale model, while UAV1-3P denotes the pruned versions. The results indicate that our method still retains an advantage over the baselines. Furthermore, the combination of model pruning and our method exhibits promising potential for achieving smooth rendering in larger-scale scenes.

Table 7. Render speed across original models and pruned models.

AvgFPS.	UAV-1	UAV-2	UAV-3	UAV-1P	UAV-2P	UAV-3P
Size$\times 10^3$	6,895	7,189	5,926	2,671	3,269	2,241
3DGS	25.2	23.2	54.3	49.2	44.6	73.2
Flash-GS	129.2	113.1	241.1	296.6	278.2	436.9
Ours	241.5	202.5	378.5	489.4	448.3	746.2

7.4. Visual Effects

Finally, we present visual comparisons between our method and the baseline approaches. The figure below shows rendering examples from random viewpoints. It can be observed that our results are largely consistent with those of Flash-GS, which is expected as our method also employs opacity-aware culling in the pre-processing stage. In contrast, when compared with 3DGS, our approach and FlashGS exhibit minor distortions in detailed image regions, often manifesting as color drift in uniform areas. We attribute these artifacts to the opacity-aware culling strategy and consider this issue a potential direction for future optimization.

3DGS	FlashGS	Ours-key	Ours-sub

3DGS
FlashGS
Ours-key
Ours-sub