

[PLR] Optimizing Gaussian Splats for faster Rendering

Romain Guntz

rguntz@student.ethz.ch

Pavidiran Ramanathan

pramanathan@student.ethz.ch

Abstract—This project investigates methods for accelerating Gaussian Splatting, the rendering technique employed in the `gsplat` library, to enable RGB rendering of real-world 3D environments for reinforcement learning (RL) applications. Sorting was identified as the primary computational bottleneck. Although caching the results of the sorting step can yield up to a 9.7% speedup, the overhead introduced by the current RadixSort implementation offsets this gain, ultimately slowing down the process. In contrast, soft and hard pruning methods deliver a consistent 23% speedup with minimal quality degradation and are relatively simple to implement. Since pruning can be combined with other acceleration techniques, it offers promising potential for performance improvement. However, the current level of speedup is still insufficient for practical RL workloads. Achieving substantial performance gains will likely require a redesign of the `gsplat` system to support efficient caching. Further research is needed to explore and realize these improvements.

I. INTRODUCTION

Reinforcement Learning (RL) for autonomous navigation increasingly requires realistic and high-performance simulation environments. While current training setups often rely on depth and semantic information, the integration of photorealistic 3D scenes remains limited due to performance constraints. Providing agents with image-based observations of their environments can enhance decision-making and reduce the gap between simulation and real-world deployment, where agents typically rely on camera inputs. A recent advancement in 3D Gaussian Splatting offers real-time rendering of photorealistic scenes and has the potential to address this limitation. However, one major bottleneck remains: the sorting of Gaussians for tile-based rendering, which is still costly and limits achievable frame rates. Our project aims to further optimise the library `gsplat` [1] for RL-specific use cases, where agents often share similar viewpoints. These properties open up opportunities to reduce redundant computations and improve rendering efficiency. We explore whether caching, pre-sorting, or exploiting

the structure of RL environments helps accelerate `gsplat` beyond its current capabilities.

We further explore alternative techniques for reducing the number of Gaussians in a scene without any loss in quality. Two tools are introduced: a soft pruning tool, which can be applied during optimization, and a hard pruning tool, which can be applied after optimization.



Fig. 1. **Hard Pruning** of a 4.4 million Gaussian scene. The first image shows the baseline, while the second shows a version pruned by 50% using our technique. The results demonstrate improved quality, as the window that appeared blurred in the baseline becomes clearly visible.

Our Contributions

- Integrated existing optimization algorithms from Speedy Splat into the `gsplat` library to assess their performance impact.
- Investigated the impact of partial sorting and viewpoint-based reuse of data.
- Developed an easy to use pruning tool providing a speed-up of up to 23 % with minimal quality loss

II. RELATED WORK

Gaussian Splatting has rapidly emerged as an efficient alternative to volumetric neural radiance fields for real-time 3D scene rendering. Kerbl et al. [2] introduced *3D Gaussian Splatting for Real-Time Radiance Field Rendering*, a method that models the scene using a collection of 3D Gaussians, each described by a mean μ , a covariance matrix Σ , and an opacity value α . These parameters are optimized through a fully differentiable rasterization process that enables real-time rendering with high

visual fidelity. Following this foundation, several recent works have addressed limitations in rendering speed, optimization efficiency, and resource constraints. Hanson et al. [3] presented *Speedy-Splat*, which accelerates rendering by combining sparse pixel selection with sparse primitive filtering, significantly reducing computational overhead in dense scenes.

To further improve optimization efficiency, Lin et al. [4] proposed *RadSplat*, which integrates radiance field priors into Gaussian Splatting, achieving over 900 FPS while maintaining high-quality view synthesis. Similarly, Jiang et al. [5] introduced *3DGS-LM*, applying Levenberg-Marquardt optimization techniques to accelerate convergence during training, reducing fitting time while preserving rendering quality. Recent work by Zou et al. [6] introduced *Scaffold-GS*, which incorporates structured Gaussian splatting via hierarchical scene decomposition. Their approach achieves view-adaptive rendering by organizing splats into a scaffolded scene graph, improving both rendering efficiency and spatial consistency.

III. PRELIMINARIES

Gsplat Library

Gsplat is a highly optimized library for 3D rendering using Gaussian splats. Two key details underpin its performance: First, a dictionary is constructed to map each Gaussian to the tiles of the camera plane it influences, with each Gaussian processed in parallel. Second, instead of rasterizing the entire image in a single pass, the image is divided into 16×16 pixel tiles, which are rasterized independently and in parallel.

To enable tile-level parallelization, the Gaussians must first be sorted by camera (to render each view independently), then by tile (to group all Gaussians affecting the same tile), and finally by depth (to render closer Gaussians before those farther away). To accomplish this, the authors encode each Gaussian with a camera code, a tile code, and a depth code, processing each Gaussian in a separate CUDA thread.

```
cid_enc | (tile_id << 32) | depth_id_enc
```

To find the tiles affected by a Gaussian, the authors create a bounding box using the minimum and maximum horizontal and vertical tiles touched. This can be efficiently computed using two nested loops: one over the x-tiles and one over the y-tiles. This is implemented in the first pass of the function called `isect_tiles`. Furthermore, tile

assignment is a necessary process that depends not only on the viewing direction but also on the camera’s position, and therefore cannot be avoided or cached.

A radix sort is then applied during the second pass of `isect_tiles` to sort the list of Gaussians by camera, tile, and depth. Among all the steps in the rendering pipeline, radix sorting is the most computationally expensive. This challenge motivated the idea of precomputing the depth of the Gaussians, which could lead to faster sorting.

Speedy-Splat

In *Speedy-Splat*, the authors identified two main computational bottlenecks: the large number of Gaussians in the scene and the number assigned to each pixel, or more generally, each tile. To address the first, they introduced efficient pruning techniques: soft pruning during optimization and hard pruning after. To address the second, they proposed improved bounding strategies to reduce the number of tiles each Gaussian intersects. These algorithms, called Snugbox and Acutile, significantly improved culling efficiency.

IV. METHOD

A. Caching Method

To improve rendering performance without modifying `isect_tiles`, we pre-sorted its inputs at the `rendering.py` level. This module connects 3D Gaussian properties (means, colors, etc.) to the final 2D image. Since inputs are initially unordered, we leveraged the observation that in RL settings, agents typically look horizontally and operate at similar heights. Agents facing the same direction also tend to see similar Gaussian sequences. Based on this, we discretized horizontal view directions into 5-degree bins and pre-sorted Gaussians for each. These were cached in memory, with only depth-sorted means stored on disk. Other tensors were reconstructed from these means during the first render.

B. Accurate Tiling Method

Another approach we tested was implementing the Speedy Splatts AccuTile method in `gsplat`. While the original Speedy Splatts paper integrated this extension into the 3DGS codebase, we evaluated how the algorithm performs within the `gsplat` rendering environment. The authors introduced the Snugbox algorithm, which computes a more precise rectangular bounding box around each Gaussian, and the AccuTile algorithm, which determines the exact tiles affected by the Gaussian.

C. Pruning Method

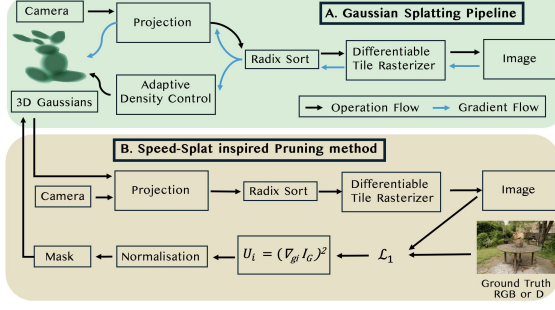


Fig. 2. GS-Prune Pipeline

Two observations can be made from the original splatting pipeline: First, the inference time is correlated with the number of Gaussians. Second, the number of Gaussians in the final output is often excessive. Consequently, a simple yet effective strategy is to reduce the number of Gaussians. To this end, we developed two pruning approaches inspired by Speedy Splat’s pruning method: *soft pruning* and *hard pruning*. Soft pruning is applied during optimization at regular intervals, while hard pruning is performed after the optimization process is complete.

The pruning approach focuses on removing Gaussians that have minimal influence on the scene. This is achieved by squaring the gradients of each Gaussian’s parameters and summing them. Only Gaussians with cumulative gradients below a certain threshold are retained, indicating that they have little influence on the loss between the rendered image and the ground truth.

V. EXPERIMENTS / EVALUATIONS

A. Caching Method

First, we measured the rendering stage to identify performance bottlenecks. The CUDA kernel *isect_tiles* has the largest impact, as it assigns Gaussians to tiles and sorts them by depth. It is called three times per frame. In the first pass, the number of tiles intersected by each Gaussian is computed, taking approximately 0.0784 ms. In the second pass, Gaussian depths and tile indices are packed into a 64-bit tensor, *isect_ids*, while a separate array, *flatten_ids*, maps these entries to their corresponding Gaussian IDs. This step takes about 0.932 ms. The third and most time-consuming step performs sorting using RadixSort on *flatten_ids* and *isect_ids*, which lasts around 4.471 ms. Finally, an offset table is generated to index intersections per (camera, tile) pair. In total, the three calls to

isect_tiles take approximately 5.63 ms, allowing for an estimated rendering speed of roughly 200 FPS.

Operation	Time (ms)
Total Tile Kernel Execution Time	5.63
Isect tiles kernel launch first pass	0.0784
Isect tiles kernel launch second pass	0.932
Radix Sort and Offset array	4.471

TABLE I
PERFORMANCE METRICS FOR CUDA OPERATIONS

Caching tensors

Based on the caching strategy mentioned in the Method section, we tested whether caching preprocessed tensors could speed up rendering. While this approach theoretically avoids redundant computations during the *isect_tiles* stage, practical implementation showed that transferring cached data to the GPU introduced a significant overhead of 21.58 ms per frame. This outweighed any performance gain. Attempts to mitigate this by moving certain steps to CUDA did not yield improvements.

RadixSort with partially sorted tensors

In addition, we wanted to investigate whether having partially sorted tensors would result in a speedup for RadixSort. In our case, the Gaussians would be presorted by depth, while the tile assignments would remain random. RadixSort would then only need to sort by tile. To test this in isolation, we created a small script that generated the *isect_ids* and *flatten_ids* tensors with random entries, except for the depth values, which were strictly increasing. The maximum number of unique tiles was set to 4000. Since each tile is 16 by 16 pixels, assuming a square resolution of 1024 × 1024 pixels, this results in approximately 4096 tiles. We then ran RadixSort twice: once with the ordered depth values and once after shuffling the entries. We observed a speedup of 9.7% (averaged over 100 runs).

Sorting Method	Average Sort Time (ms)
Random order	4.4083
Presorted by depth	4.01825
Speedup (random / sorted)	1.09707×

TABLE II
COMPARISON OF SORTING TIMES

B. Accurate Tiling Method

The AccuTile method did reduce the number of tiles a Gaussian affected. But the speedup did not compensate for the overhead generated by more accurate bounding boxes as shown in Tab. III.

Method	Time (ms)
Snugbx-AccuTile	6.543
Original Code	6.190

TABLE III
RENDERING TIME COMPARISON

C. Pruning Method

To evaluate the effectiveness of pruning we tested different configurations. In the small garden scene we first tested a range of hard pruning ratios. In Fig. 3, we see that for small pruning ratios the quality metric Peak Signal-to-Noise Ratio (PSNR) decreases only slightly while the rendering time goes down linearly. In this scene we determine the optimal pruning ratio to be 0.5, leading to a 23 % decrease in rendering time.

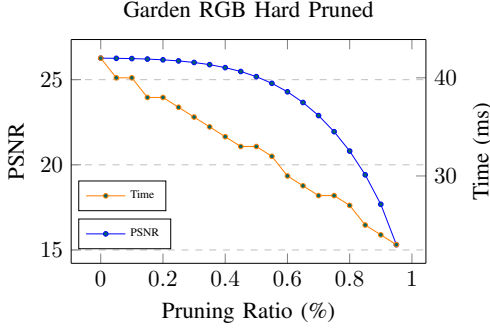


Fig. 3. PSNR and time versus pruning ratio for Garden RGB Hard Pruned.

A similar result can be observed in Fig. 4 with the park scene. The PSNR does not decrease at all in that case. Note that the park scene has sparse training data and the initial PSNR is already low. When inspecting the splat with a viewer, degradation becomes visible at higher pruning ratios.

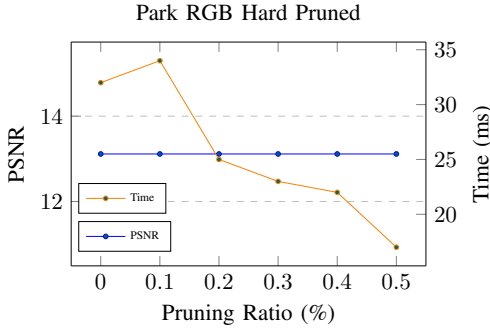


Fig. 4. PSNR and time versus pruning ratio for Park RGB Hard Pruned.

For depth hard pruning in Fig. 5, we observe that the PSNR decreases after a pruning ratio of 0.3.

When inspecting this splat, the degradation appears less noticeable than in the RGB hard pruning case, despite its higher PSNR score.

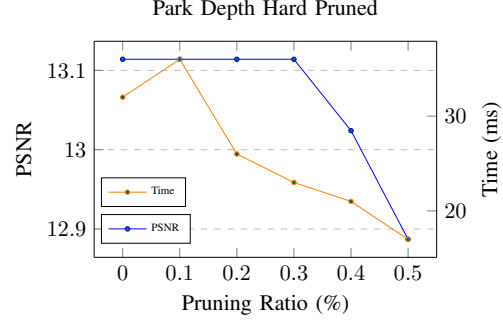


Fig. 5. PSNR and time versus pruning ratio for Park Depth Hard Pruned.

For soft pruning of the garden scene in Fig. 6, we observe that at small pruning ratios, the PSNR even increases slightly while the render time continues to decrease. This suggests that soft pruning may improve quality in some scenes.

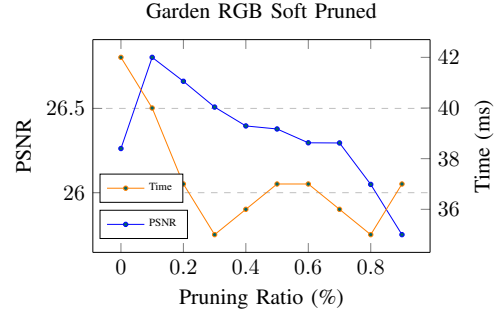


Fig. 6. PSNR and time versus pruning ratio for Garden RGB Soft Pruned.

VI. CONCLUSION

Even when pre-caching the means, the achieved speedup is minimal and does not outweigh the overhead introduced by the additional operations in the current RadixSort implementation. In contrast, pruning methods deliver a solid speedup of about 23% with negligible loss in quality. Since pruning can be combined with other acceleration techniques, it holds significant promise. However, the current speedup remains insufficient for reinforcement learning applications. Achieving substantial acceleration will require a redesign of gsplat to support effective caching. Further research is necessary to realize these improvements.

REFERENCES

- [1] Vivian Ye, Ruoshi Li, Jackson Kerr, Mikko Turkulainen, Benyuan Yi, Zhen Pan, and Angjoo Kanazawa. gsplat: An open-source library for gaussian splatting. *Journal of Machine Learning Research*, 26(34):1–17, 2025.
- [2] Bernhard Kerbl et al. 3d gaussian splatting for real-time radiance field rendering. In *SIG-GRAPH*, 2023.
- [3] Andrew Hanson et al. Speedysplat: Fast 3d gaussian splatting with sparse pixels and sparse primitives. 2024.
- [4] Michael Niemeyer et al. Radsplat: Radiance field-informed gaussian splatting for robust real-time rendering with 900+ fps. 2024.
- [5] Lukas Höllein, Aljaž Božič, Michael Zollhöfer, and Matthias Nießner. 3dgs-lm: Faster gaussian-splatting optimization with levenberg-marquardt, 2024. URL: <https://arxiv.org/abs/2409.12892>.
- [6] 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 (CVPR)*, pages 20654–20664, 2024.

APPENDIX



Fig. 7. Garden Scene Baseline



Fig. 8. Garden Scene 30% Pruned



Fig. 9. Garden Scene 50% Pruned

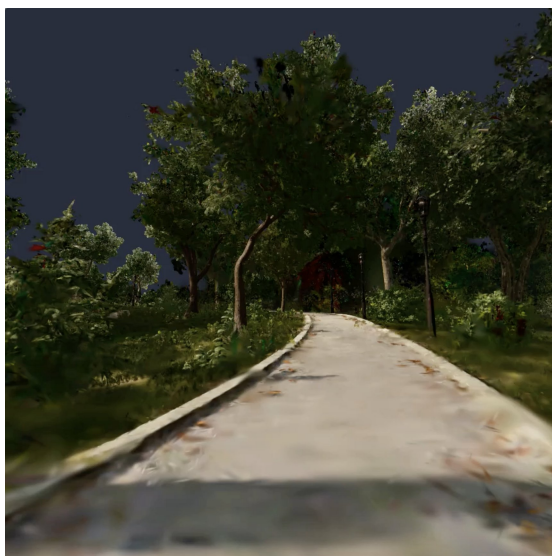


Fig. 10. Park Scene Baseline

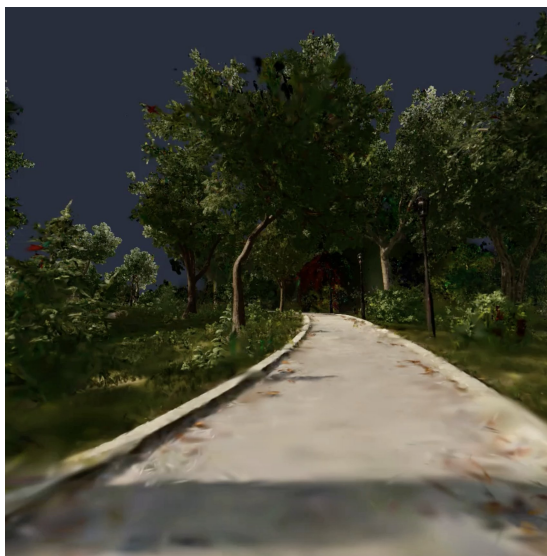


Fig. 11. Park Scene 30 % Hard Pruned