FlashGS: Efficient 3D Gaussian Splatting for Large-scale and High-resolution Rendering

Supplementary Material

Overview

This supplementary material further elaborates on the details of this work, providing extended analyses and data. These additions further validate the importance and completeness of our contributions. The content is organized as follows:

- Section 6 offers a comprehensive discussion of FlashGS, including its portability and limitations.
- Section 7 presents a more detailed illustration of the underlying significant redundancies in prior work.
- Section 8 illustrates the quality of rendered images, demonstrating that FlashGS can preserve image details.
- Section 9 provides a detailed description of the low-level implementation of our CUDA kernels, supporting our algorithmic design.
- Section 10 presents additional experimental data, further demonstrating the sources of our acceleration.
- Section **??** provides more tests on different aspects to show the performance impact of FlashGS.

6. Discussion and Limitation

FlashGS should be easy to integrate with other 3DGS approaches, further enhancing their performance. The design is independent of any specific 3DGS variant and functions as a standalone high-performance 3DGS renderer. For other algorithmic approaches, as long as they retain the original tile-based rasterization mechanism of 3DGS, our optimizations remain effective. For other acceleration techniques, such as distributed computing, our optimization strategies are orthogonal to theirs, enabling integration for additional performance improvements. As for some works that also focus on renderer optimization, our approach addresses the foundational challenges through an in-depth analysis, effectively resolving issues that have been inadequately addressed or even negligated.

The effectiveness of our approach is influenced by the characteristics of the original scene, specifically the size and distribution of the Gaussians representing the scene. When the Gaussians in a scene are small and evenly distributed, the performance improvement of our method relative to existing approaches may be diminished. For instance, in an extreme case where all Gaussians are evenly distributed across the screen and each Gaussian covers only a single tile, although performance issues are generally not encountered in such scenarios, the benefit of our method would be less pronounced.

7. Redundant Gaussian-tile Pair Analysis

We provide a clearer illustration of the redundancies mentioned in line 79, Section 1, emphasizing the critical role of our precise intersection algorithm described in Section 3.2.

We can roughly model the total redundant Gaussian-tile pairs as: $N = n_{\text{gs}} \cdot n_{\text{avg}}$, where n_{gs} is the number of Gaussians and n_{avg} is the average redundant intersections per Gaussian. For **large-scale** scenes, the number of Gaussians can be up to tens of millions [52]; for **high resolutions**, n_{avg}, which is the average number of false intersections per Gaussian, can increase. We will further explain these claims: For an ellipse with a major axis size a and minor axis size b. If the ellipse itself is axis-aligned, the wasted area of its axis-aligned bounding box (AABB) is $1 - \frac{\pi}{4}$, approximately 21%. Next, we consider the case where the angle between the ellipse's major axis and the horizontal axis is t. We start by analyzing the scenario with the maximum area waste, which requires solving the following constrained optimization problem: $x^2 + y^2 = a^2$, where $x \approx a \cos t$ and $y \approx a \sin t$. We maximize xy (the largest redundant area). Equivalently, we maximize $a^2 \sin t \cos t$. When $t = 45°$, $\sin t \cos t$ reaches its maximum value, giving $xy = \frac{a^2}{2}$. In this case, the wasted area becomes $\frac{a^2}{2} - \frac{\pi ab}{4}$ and the redundant ratio is $1 - \frac{\pi}{8} \cdot \frac{b}{a}$. Therefore, when the ellipse is very flat ($\frac{b}{a}$ is very small), the wasted area can be large, even approaching 100%.

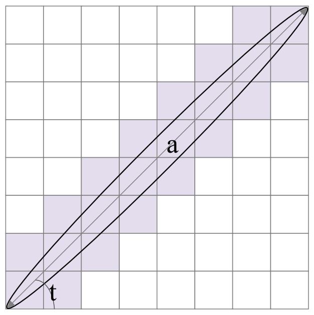

Figure 6. The redundancy of AABB: The colored regions represent the real intersection between the ellipse and the tiles. The valid intersections correspond to the three diagonal blocks in the matrix of tiles. In this example, the ellipse has a tilt angle $t = 45°$ with the horizontal axis and a major axis length of a.

Under these circumstances, we further consider the tile-based intersection issue: when $t = 45°$, meaning the AABB is a square, the ellipse at least intersects with three diagonal

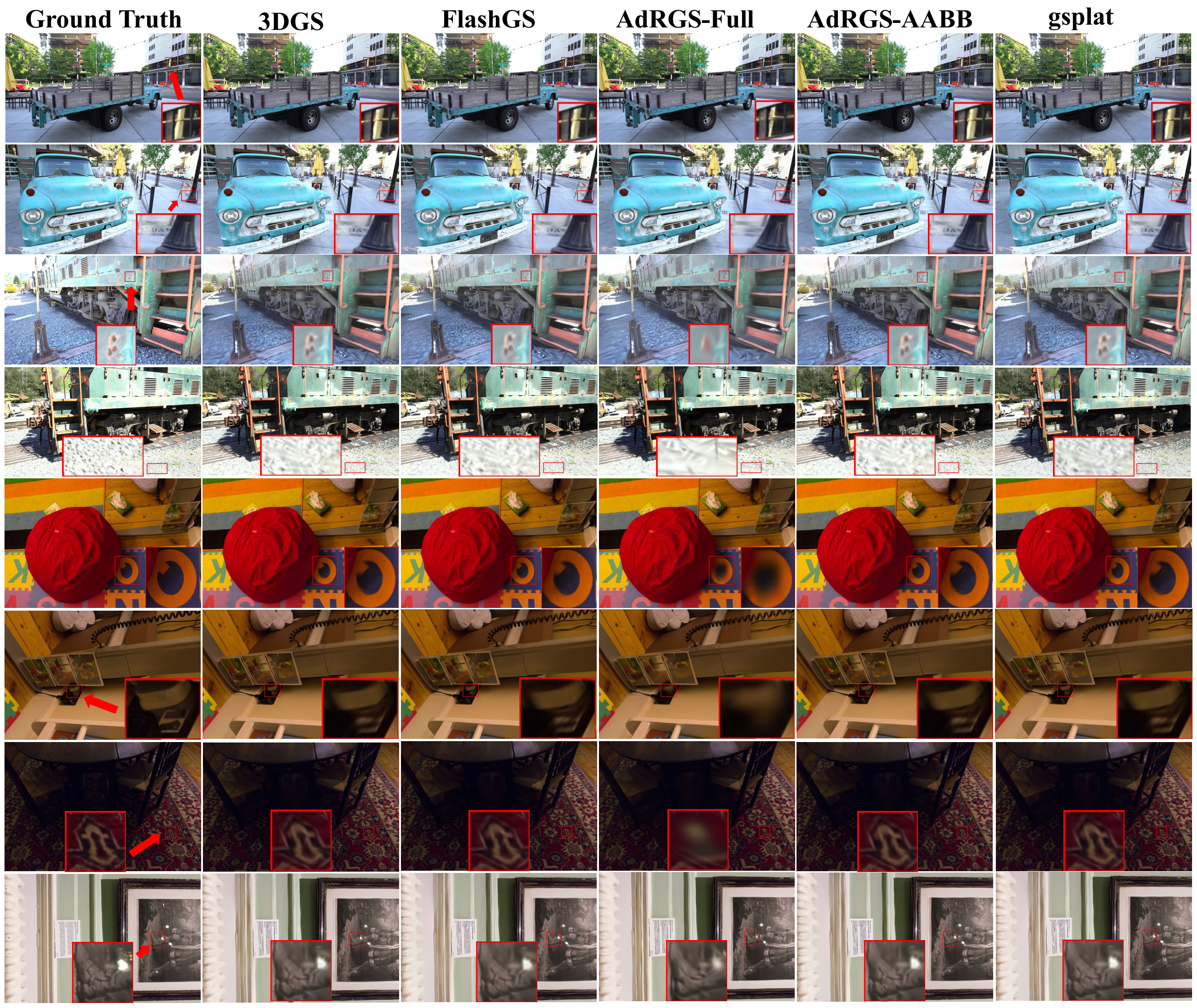

Figure 7. We show comparisons of different methods and the corresponding ground truth images. We observe that all methods, except AdRGS-Full (with smearing artifacts), effectively preserve the details of the original image. The scenes are Truck and Train from Tanks & Temple [24], Playroom and DrJohnson in DeepBlending [17]. The differences in quality are highlighted by arrows and insets.

tiles along, as depicted in Figure 6. The actual intersection area within the AABB is $\frac{3n-2}{n^2}$, where n^2 is the total number of tiles covered by this square AABB. As n increases, this ratio decreases. In other words, the AABB of the ellipse can occupy more tiles, when the relative size between the ellipse and the tile increases. This corresponds to **high-resolution** scenarios, since with a fixed tile size, higher resolutions will introduce more tiles across the screen (finer tile grid).

8. Visual Comparison

We present and compare the rendering results produced by FlashGS and previous methods, as shown in the Figure 7. FlashGS demonstrates efficient rendering while maintaining high image quality, preserving details effectively, similar to vanilla 3DGS, AdR-AABB, and gsplat. However, for AdRGS-Full, one of the current state-of-the-art rending methods, its load-balancing performance benefits only manifest after retraining the model. During retraining, AdRGS-Full reduces the number of Gaussians in regions with excessive overlaps, while increasing Gaussians in surrounding areas to compensate for the quality loss. Despite these adjustments, we still observe a noticeable loss of detail. We attribute this to the loss of information inherent in the original scene. The distribution and details of objects in the scene naturally result in uneven Gaussian distributions. AdRGS-

Full's approach to balancing the Gaussian distribution by interfering with the scene reconstruction leads to detail loss. This presents as smearing artifacts in the rendered images. For instance, in Truck, AdRGS-Full alters the shape of the pillar and loses the letters on the sign and the texture details on the lamp base. In the Train examples, it removes the spots on the metal surface and the gravel patterns on the ground. In Playroom, it blurs the contours of the letter Q and the details of the white plug. In DrJohnson, AdRGS-Full loses the carpet patterns and the fine folds in the boots.

9. Low-level Implementation and Optimization

We elaborate on the implementation and optimization details discussed in Section 4.1. These optimizations primarily encompass parallelism and memory enhancements, such as efficient task partitioning to improve data reuse, assembly-level implementations to enhance instruction efficiency, reducing thread divergence to minimize resource idling, and various memory optimizations.

Thread Level Workload Partition We avoid splitting the task of one tile into multiple warps, considering the task dispatch granularity. In the original implementation, each thread only processed one pixel, and each 16×16 tile is divided into 8 warps. The obvious disadvantage of this method is that it requires reading the same memory address 8 times, and even using shared memory cannot eliminate this overhead. Therefore, we assign the entire tile to one warp, so that each thread handles multiple pixels. In addition to reducing memory load operations, the benefit also includes reducing the floating-point operations. Since the pixels are arranged in a regular two-dimensional array, for each tile, each column shares the same x-coordinate and each row shares the same y-coordinate. When calculating the distance between the pixels and the center of an ellipse, $(x - x_0)^2$ and $(y - y_0)^2$ are common subexpressions. By assigning multiple pixels to each thread, we expose the opportunities for common subexpression elimination (CSE) to further improve computational performance by reducing repetitive floating-point operations.

Assembly Level Optimizations We also utilize assembly-level optimizations to better leverage efficient instructions supported by the GPUs. In Gaussian computations, an important operation is to evaluate the exponential function. Using `__expf` built-in function still generates additional floating-point operations. Due to the fact that the special function unit (SFU) directly supported by the hardware has a base of 2, this function needs to be multiplied by the constant $log_2 e$ to replace the base when generating the SASS instructions. Firstly, we fold this constant with the other constants before. However, there are still 3 additional instructions in SASS for handling particularly small values, as SFU does not support outputting non-normalized floating-point numbers. Specifically, for values ranging from -252 to -126, calculate $t = 2^{0.5x}$ to output a normalized floating-point number, then calculate t^2 to obtain a relatively accurate non-normalized floating-point number to approximate 2^x. But when we put it in the background of the task, we will find that such a small value is not worth considering at all, because when the opacity is so close to zero, it is already no different from zero in the following alpha blending stage. Therefore, we avoid this overhead by using the PTX instruction `ex2.approx.ftz.f32` to directly use the SASS instruction `mufu.ex2`, which tells the compiler not to consider non-normalized floating-point number.

Warp Divergence Control The original implementation will determine whether the opacity is less than $1/255$ after calculating the Gaussian function, and if so, skip the alpha blending stage. Since we have already eliminated the vast majority of such situations before, this judgment becomes meaningless except leading to warp divergence. Warp divergence seriously affects computational efficiency, so we have removed this judgment.

Memory Access Optimization In the preprocessing, some memory access issues remain, particularly with the SH coefficients, which consist of 48 floats per Gaussian and are non-reusable. These massive uncoalesced and scalar accesses to global memory put significant pressure on the L1 cache bandwidth. We mitigate this problem through several memory access optimizations. For non-reusable shs, we reduced the number of memory access instructions by using vectorized loads. For reusable parameters (such as projection matrices and transformation matrices), we leveraged constant memory to reduce latency. Specifically, we directly pass these data as parameters to the kernel instead of using pointers to their memory locations.

Memory Management 3DGS employs a dynamic memory allocation strategy, which can result in non-negligible overhead, especially when handling a large number of memory allocation requests. *We adopt a more static strategy.* We avoid the performance reduction of frequent memory allocation, which could be caused by the overhead of system calls and memory fragmentation. We extract the dynamic memory allocation operations and related preprocessing operations to the initial stage. For example, a unified preprocessing can be done in advance for computations involving different viewpoints of the same scenario.

10. Detailed Evaluation Results

We further analyze the sources of our acceleration by presenting specific data and metrics. The following experiments highlight the performance of FlashGS compared to the vanilla Gaussian implementation on representative test cases, including frames with the highest and lowest speedup achieved by FlashGS.

Runtime Breakdown Analysis In this section, we com-

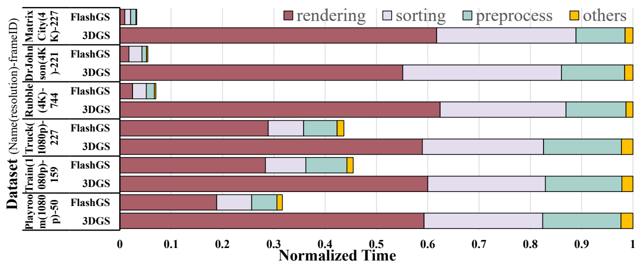

Figure 8. Rasterization runtime breakdown on 6 representative frames from different datasets, normalized to 3DGS.

pare and analyze the runtime breakdown of FlashGS to 3DGS, aiming to reveal the source of our excellent performance. Figure 8 shows the rasterization time and the breakdown for 6 representative frames shows max or min speedup. This demonstrates that we have accelerated all stages, including preprocess, sort, and render. In FlashGS, these stages respectively account for average 19.6%, 29.1%, and 47.6% of the total time, whereas in 3DGS, they account for 13.2%, 25.4%, and 59.6%. The optimizations in the preprocess and render stages have already been discussed earlier, while the speedup in the sorting stage is primarily due to the reduction in the number of key-value pairs to be sorted after applying our precise intersection algorithm. We will give a more detailed analysis below.

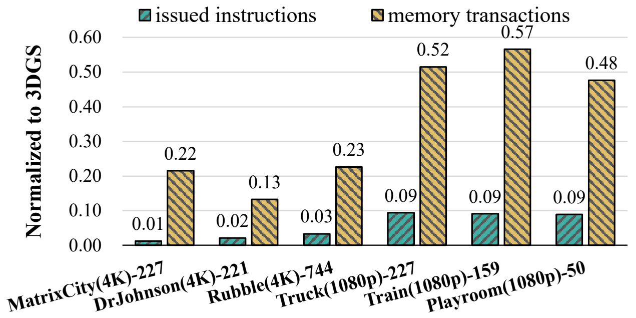

Figure 9. The number of instructions issued in rendering and the memory transactions in preprocessing. All results are normalized to 3DGS.

Profiling Results We further demonstrate our optimizations in the rendering and preprocessing stages through profiling results of memory and compute units, as shown in the figures. Figure 9 shows that we reduce the issued instructions in the rendering stage, alleviating the computation-bound problem. This proves the effectiveness of our precise intersection algorithm and the optimizations for low-latency rendering. The total issued instructions is significantly reduced by one to two orders of magnitude. For memory access transactions in preprocessing, we also reduced the number of global memory accesses by 43%-87% compared to 3DGS. Figure 10 further shows the reason for

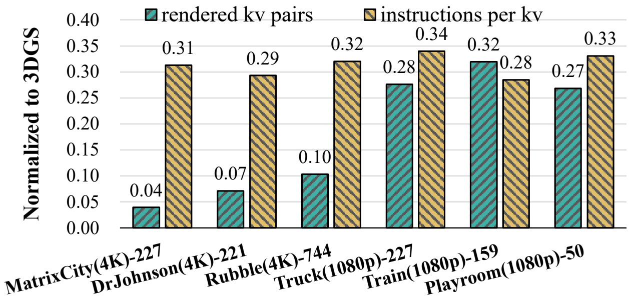

Figure 10. Number of rendered Gaussian-tile (kv) pairs and instructions issued per pair of FlashGS (Normalized to 3DGS).

the significant instruction reduction in the rendering stage, which is dominant in the rasterization. In tile-based rendering, the total number of issued instructions is the product of the number of tiles and the instructions per tile. Therefore, this reduction mainly stems from two aspects. In Figure 10, the number of rendered key-value pairs benefit from our intersection optimization, reducing by 68%-96%. The instructions involved in each tile's computation also decrease by 67%-71%, as we further optimize the render operator at the instruction level. The reduction in the number of generated key-value pairs also benefits the sorting process, as it significantly decreases the size of the list to be sorted. Additionally, less memory is required to store key-value pairs.

11. FlashGS Performance in More Aspects

We will provide more experiments to demonstrate and discuss the performance impact of FlashGS, including tests on more datasets and platforms, as well as its effects on training and other strategies.

Rendering Performance on Mip-NeRF 360 Table 5 presents the FPS and speedup achieved by FlashGS compared to vanilla 3DGS on the Mip-NeRF 360 dataset for Bosai and Bicycle at different resolutions. The results demonstrate that FlashGS effectively eliminates redundant intersections, achieving a speedup ranging from 3.47× to 12.3×, thereby consistently enabling real-time rendering.

Table 5. Speedup across different scenes from Mip-NeRF 360 Dataset and resolutions (3DGS/FlashGS) on NVIDIA A800.

Bonsai				Bicycle			
Resolution	FPS↑	#Intersects↓	Speedup↑	Resolution	FPS↑	#Intersects↓	Speedup↑
3118×2078	67.12/458.5	13.1M/2.62M	6.83×	4946×3286	13.62/167.5	75.4M/9.82M	12.3×
1200×799	195.0/677.2	2.85M/0.849M	3.47×	1237×822	60.62/254.1	7.78M/2.72M	4.19×

Performance on Different GPU Table 6 presents the performance of FlashGS on two GPUs. Since the rendering step is heavily dependent on FP32 performance, the 3090

(35.6 TFLOPS) outperforms the A100 (19.5 TFLOPS). Although memory-bound tasks like sorting benefit from the A100's higher bandwidth, the rendering bottleneck results in an overall slower performance compared to the 3090. Nevertheless, FlashGS consistently achieves real-time rendering across different test datasets and platforms.

Table 6. Rendering speed of FlashGS on NVIDIA 3090 and A100).

AvgFPS ↑	Truck		Train		Playroom		DrJohnson		MatrixCity		Rubble
Resolution	1080p	4k	1080p	4k	1080p	4k	1080p	4k	1080p	4k	4608*3456
3090	**450.5**	**289.0**	**518.1**	**301.2**	**694.4**	**367.7**	**613.5**	**334.4**	**310.6**	**207.4**	**183.9**
A100	355.9	252.7	400.3	272.1	522.8	276.5	471.7	294.9	293.2	186.7	154.1

Impacts on Training Speed Table 7 shows the accelerated forward, backward, and loss computation in training GSDF. This demonstrates the effectiveness of our optimization, and its compatibility with optimized fused_ssim and atomic_add in Taming 3DGS [35].

Table 7. GSDF [49] training time with different optimizations.

method	Forward	Loss	Backward	Total
baseline	17.55ms	4.28ms	38.67ms	8m12s
+fused_ssim	17.55ms	0.62ms	37.53ms	7m59s
+FlashGS	10.68ms	0.86ms	25.52ms	6m22s
+atomic_opt	**10.44ms**	**0.77ms**	**17.05ms**	**4m3s**

Integrate with Level-of-Detail Strategies Some enhanced 3DGS representations incorporating multiple Levels of Detail (LOD) enable rendering each pixel with a small and manageable set of Gaussians. Our proposed software pipelining remains effective in accelerating these more balanced scenes. FlashGS can be easily integrated with Octree-GS [41], achieving a 1.6× speedup in the rasterization stage. In an extreme case (one tile with 10 Gaussians) we set, Nsight Compute measurements show that the baseline 3DGS and FlashGS exhibit latencies of $23k$ and $11k$ cycles, respectively. This demonstrates that our pipeline effectively reduces startup overhead and can enhance LOD-based methods.