Profiling and Optimizing Tensor Core Gaussian Splatting

Rahul Nadkarni (rn2592), Shika Rao (sr7463)

Abstract

We present a systematic profiling and optimization study of Tensor Core Gaussian Splatting (TC-GS) inference renderering. Through comprehensive analysis using Nsight Compute and nvprof, we identify critical bottlenecks: kernel launch overhead (132.95 μs/frame), memory allocation inefficiencies (327 μs/frame), tensor core underutilization (87.3%), and low GPU occupancy (12.5%). We implement four key optimizations: CUDA graph integration, static buffer reuse, restructured WMMA operations with improved instruction-level parallelism, and enhanced warp coherency. Our total optimized implementation achieves 336.24 FPS on MipNeRF-360 Bonsai, representing a 6.6% improvement over baseline TC-GS and a 342.1% improvement over traditional 3DGS, while maintaining identical quality (PSNR: 32.412 dB). The overall frame time is reduced from 16.7ms to 11.9ms, achieving a 1.40× speedup.

1 Introduction

3D Gaussian Splatting (3DGS) [1] achieves real-time rendering by representing scenes as collections of 3D Gaussians projected and blended to generate images. Traditional implementations rely on CUDA cores for operations like exponential evaluation and alpha blending. This becomes memory-bound at scale. Tensor Core Gaussian Splatting (TC-GS) [2] addresses this by reformulating computations as matrix multiplications accelerated via Tensor Cores, achieving substantial speedups during inference rendering. However, TC-GS still exhibits performance bottlenecks preventing peak performance. We conduct comprehensive profiling to identify and optimize these bottlenecks. There are papers like [3] which speed up Gaussian Splatting training, however TCGS and our work focuses only on rendering inference optimization.

Contributions: Our work makes three key contributions:

1. We perform systematic profiling of the TC-GS renderer at both system and kernel levels, quantifying the impact of various bottlenecks including kernel launch overhead, memory allocation patterns, tensor core utilization, and GPU occupancy.
2. We develop and implement four key optimizations addressing the identified bottlenecks: CUDA graph integration for reduced launch overhead, static buffer reuse for eliminated allocation costs, restructured WMMA operations with improved instruction-level parallelism, and enhanced warp coherency for better memory access patterns.
3. We demonstrate that our optimizations achieve a 6.6% performance improvement over baseline TC-GS and a 342.1% improvement over traditional 3DGS on the MipNeRF-360 Bonsai benchmark while maintaining identical rendering quality.

2 Related Work

3D Gaussian Splatting represents scenes using millions of 3D Gaussians with learned parameters (position, covariance, color, opacity) that are projected to 2D, sorted by depth, and alpha-blended. The original implementation achieves real-time performance through tile-based rasterization but relies on CUDA cores, creating computational bottlenecks. TC-GS converts fragment-level operations into batched matrix multiplications using Tensor Cores, which provide significantly higher throughput for matrix operations than CUDA cores. The Frag2Mat architecture batches Gaussian fragments into matrices processed via Tensor Core kernels, fundamentally changing computation from per-fragment scalar operations to batched matrix operations.

3 Methodology

3.1 3D Gaussian Representation

Each 3D Gaussian G_i in our scene representation is parameterized by: mean position $\mu \in R^3$, covariance $\Sigma = RSS^TR^T \in R^{3\times3}$ (where R is rotation from quaternion r and $S = \text{diag}(s_x, s_y, s_z)$), RGB color $(r, g, b) \in [0, 1]^3$, and opacity $\alpha \in [0, 1]$. The 3D Gaussian function is:

$$G(x; \mu, \Sigma) = \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right) \quad (1)$$

For rendering, we project each 3D Gaussian to 2D screen space through covariance transformation $\Sigma' = JW\Sigma W^TJ^T$, where W is the world transform and J is the projection Jacobian.

3.2 TC-GS Architecture

TC-GS differs fundamentally from traditional 3DGS in fragment processing. Traditional 3DGS iterates through sorted Gaussians per pixel, projecting each individually and evaluating opacity through exponentials. TC-GS instead collects all fragments affecting a tile, packs them into batch matrices, and launches Tensor Core kernels to compute all fragment opacities via matrix multiplication:

$$\alpha_{\text{batch}} = \text{WMMA}(F_{\text{pixel}}, F_{\text{gaussian}}, B_{\text{coef}}) \quad (2)$$

where F_{pixel} contains pixel coordinates, F_{gaussian} contains Gaussian parameters, and B_{coef} contains precomputed coefficients.

3.3 Profiling Infrastructure

We used NVIDIA Nsight Systems for system-level analysis (CPU-GPU synchronization, kernel launch overhead, memory allocation patterns) and NVIDIA Nsight Compute/nvprof for kernel-level analysis of `renderCUDA_TCGS` (memory/compute throughput, Tensor Core utilization, occupancy, register usage, shared memory conflicts).

3.4 Identified Bottlenecks

Through systematic profiling, we identified four major performance bottlenecks:

Kernel Launch Overhead: The baseline TC-GS implementation launches two kernels per frame: `transform_coefs` for preprocessing and `renderCUDA_TCGS` for rendering. Each launch incurs driver scheduling latency, stream synchronization overhead, and argument marshalling costs. Measured impact: 132.95 μs per frame (4.23% of total frame time), independent of scene complexity.

Memory Allocation Overhead: The `feature_encoded` buffer storing intermediate Gaussian feature encodings is dynamically allocated with `cudaMalloc` and freed with `cudaFree` every frame. This causes allocation latency, heap fragmentation, and cache pollution. Measured impact: 327 μs per frame (10.4% of total frame time).

Tensor Core Underutilization: Despite using Tensor Cores, we observed suboptimal utilization at 87.3% (target: 95%) with 7.66% stall cycles. Root causes include suboptimal WMMA instruction scheduling, memory loads not pipelined with computation, and insufficient instruction-level parallelism. Measured impact: WMMA core loop time of 3.9ms with potential for further reduction.

Register Pressure and Low Occupancy: The `renderCUDA_TCGS` kernel exhibits high register usage of 64 registers per thread, limiting achieved occupancy to 12.5% (theoretical maximum: 25%). Root cause: Overly strict early clobber constraints prevent compiler register optimization, limiting parallelism and reducing ability to hide memory latency.

4 System-Level Optimization Techniques

4.1 CUDA Graphs Integration

The baseline implementation requires launching two distinct kernels per frame, with each launch incurring overhead from driver scheduling, stream synchronization, and argument marshalling. For high frame rate applications (300+ FPS), the measured 132.95 μs overhead represents over 4% of frame time.

CUDA graphs capture GPU operation sequences into a reusable graph. Our implementation: (1) captures both kernel launches using `cudaStreamBeginCapture` during initialization; (2) instantiates with `cudaGraphInstantiate` for one-time optimization; (3) launches via `cudaGraphLaunch` each frame, bypassing driver overhead. Dynamic parameters are updated using `cudaGraphKernelNodeSetParams`. **Results:**

Launch overhead reduced from 132.95 μs to 2.3 μs per frame (57.8× speedup) through amortized driver overhead and reduced synchronization.

4.2 Static Feature Buffer Reuse

The baseline allocates and frees the `feature_encoded` buffer every frame, consuming 327 μs (10.4% of frame time) split 60% for allocation and 40% for deallocation. This causes allocation latency, heap fragmentation, cache pollution, and memory manager contention.

We implemented persistent buffer management with dynamic resizing. Initialization allocates capacity based on scene statistics with 20% headroom (600K capacity for 500K Gaussian scenes, 31.2 MB). Each frame reuses the buffer; geometric growth (1.5× resize) occurs only when capacity exceeds 20% threshold. **Results:** Eliminated 327 μs allocation overhead per frame. Reallocation occurs in ¡0.1% of frames, with one-time 2-3 ms initialization cost amortized over thousands of frames.

4.3 Restructured WMMA Operations

Profiling revealed the WMMA core loop suffered from repeated memory loads and insufficient instruction-level parallelism. The Pixel Matrix was loaded from global memory inside the main Gaussian processing loop, creating serialization bottlenecks. With millions of Gaussians per frame, this caused substantial inefficiency. Initial profiling showed Tensor Core utilization at only 87.3% with 7.66% stall cycles.

We restructured WMMA operations through three techniques: (1) **Hoisted Pixel Matrix Loading:** Load pixel data once before the loop and store in registers, eliminating $N - 1$ redundant loads and saving 256 MB memory traffic per frame. (2) **Pipelined Batch Processing:** Process Gaussians in batches of 64 with software pipelining, overlapping memory fetches with WMMA computation to hide 300-400 cycle latency. (3) **Improved Instruction Scheduling:** Manually reorder instructions and use `__pipeline_memcpy_async` to pipeline loads from global to shared memory. **Results:** Tensor Core utilization increased from 87.3% to 92.0%, stall cycles reduced from 7.66% to 3.12%, WMMA loop time decreased from 3.9ms to 3.2ms (1.22× speedup).

4.4 Warp Coherency Optimization

The `culling_and_blending` function exhibited warp divergence (threads processing independent pixels with divergent execution paths), non-coalesced memory access (random pixel assignments causing scattered memory accesses), and inefficient early termination (per-thread exit wasting cycles until all 32 threads complete). Profiling showed this stage consuming 6.1ms per frame with 73.4% warp execution efficiency.

We implemented three warp-level optimizations: (1) **Dual-Pixel Processing:** Each thread processes two adjacent pixels, reducing divergence as threads follow similar paths. (2) **Coalesced Shared Memory Access:** Restructured addressing so thread t accesses memory at base + t, creating coalesced 128-byte transactions and eliminating bank conflicts. (3) **Warp-Level Early Exit:** Use `__ballot_sync()` to collect opacity status into a bitmask; when all bits set, the entire warp exits without divergence. Additional optimizations include vector loads and prefetching. **Results:** Culling/blending time reduced from 6.1ms to 4.4ms (1.38× speedup), warp efficiency improved from 73.4% to 91.2%, memory throughput increased from 412 GB/s to 548 GB/s, bank conflicts reduced from 8.2% to 0.3%.

5 Computation-Level Optimization Techniques

5.1 Inference-Time Gaussian Culling

To accelerate rendering, we apply a lightweight, inference-only culling strategy that removes Gaussians with negligible contribution to the final image. We first discard Gaussians with opacity below a fixed threshold. On top of this, we optionally apply three conservative geometric filters: (i) screen-space radius culling, which removes Gaussians whose projected footprint falls below a minimum pixel size; (ii) view-frustum culling, which removes Gaussians whose spatial extent lies entirely outside the camera frustum; and (iii) distance-based level-of-detail (LOD) culling, which removes distant Gaussians that are both low-opacity and sub-pixel in projection.

6 Experiments

6.1 Experimental Setup

All experiments were conducted on an NVIDIA RTX 8000 GPU, featuring 4608 CUDA cores, 576 Tensor Cores, 48 GB GDDR6 memory, and 672 GB/s memory bandwidth. The software environment consisted of CUDA Toolkit 11.8, PyTorch 2.0.1, and Python 3.10.

We evaluated our implementation on the MipNeRF-360 Bonsai scene. This scene contains complex geometry with fine details, approximately 500K Gaussians after training, and is rendered at 1920×1080 resolution. Our evaluation metrics include Frames Per Second (FPS) as the primary performance metric, frame time measuring average time per frame in milliseconds, and Peak Signal-to-Noise Ratio (PSNR) comparing rendered output to ground truth.

We compared four implementations: (1) **3DGS Baseline** — the original CUDA core implementation; (2) **TC-GS Baseline** — the Tensor Core implementation without optimizations; (3) **TC-GS Optimized (System)** — TC-GS with all system-level optimizations as described in Section 4; (4) **TC-GS Optimized (System + Computation)** — TC-GS Optimized with additional inference-time Gaussian culling from Section 5.

7 Results

7.1 Performance- System Optimizations

Table 1 presents a comprehensive performance comparison. Our implementation achieves 330.69 FPS, representing a 4.85% improvement over baseline TC-GS and a 334.9% improvement over traditional 3DGS. The render time is reduced from 3.2ms in baseline TC-GS to 3.1ms in our version, and from 13.0ms in the original 3DGS implementation. Critically, all three implementations maintain identical PSNR of 32.412 dB, demonstrating that our system optimizations preserve rendering quality while improving performance.

7.2 Speedup Analysis - System Optimizations

7.2.1 Per-Component Analysis

Table 2 provides a detailed breakdown of the timing improvements for each optimization component. The most dramatic improvements come from eliminating overhead: CUDA graphs provide a 57× speedup on kernel launches, reducing this component from 132.95 μs to 2.3 μs per frame. Static buffer reuse completely eliminates the 327 μs allocation overhead, effectively providing infinite speedup for this component. Together, these optimizations save 457.65 μs per frame.

Kernel optimizations provide meaningful but more modest improvements. WMMA restructuring saves 700 μs per frame through improved instruction-level parallelism and hoisted loads, achieving a 1.22× speedup on this component. Warp coherency improvements save 1700 μs per frame through dual-pixel processing, coalesced memory access, and warp-level early exit, achieving a 1.38× speedup on the culling and blending stage.

The overall frame time is reduced from 16.7ms in the baseline to 11.9ms in our optimized implementation, representing a 1.40× combined speedup. This demonstrates that addressing multiple bottlenecks through complementary optimizations yields cumulative performance benefits.

7.2.2 Overall Analysis

WMMA optimizations increased Tensor Core utilization from 87.3% to 92%. The 700 μs improvement allows rendering 4.4% more Gaussians or supporting higher resolution. The remaining 8% gap stems from memory bandwidth constraints, control flow overhead, and warp synchronization requirements.

GPU occupancy remains at 12.5% due to register pressure (64 registers per thread), limiting memory latency hiding. System-level optimizations (CUDA graphs, static buffers) provide constant overhead reduction regardless of scene complexity, while kernel-level optimizations scale with scene complexity. Simple scenes benefit primarily from reduced system overhead; complex scenes see additional gains from improved kernel efficiency.

Table 1: Performance comparison on MipNeRF-360 Bonsai scene

Metric	3DGS	TC-GS	Ours	Improvement
FPS	76.04	315.38	330.69	+4.85%
Render Time (ms)	13.0	3.2	3.1	-3.125%
PSNR (dB)	32.412	32.412	32.412	0%

Table 2: Detailed pipeline timing breakdown

Stage	Before	After	Speedup	Technique
Launch (μs)	132.95	2.3	57×	CUDA Graphs
Alloc (μs)	327.0	0.0	∞	Static Buffer
WMMA (ms)	3.9	3.2	1.22×	ILP + Hoisting
Cull/Blend (ms)	6.1	4.4	1.38×	Warp Coherency
Frame (ms)	**16.7**	**11.9**	**1.40×**	**Combined**

Table 3: PSNR, FPS, and rendering time for different culling methods

Method	PSNR	FPS	Time (s)
Baseline	32.41	330.69	0.0031
Opacity Culling	32.41	336.24	0.0030
Opacity + LOD Culling	32.39	340.24	0.0030
Opacity + Radius Culling	31.64	345.45	0.0029
Opacity + Frustum Culling	17.02	539.19	0.0019

7.3 Performance and Speedup Analysis- System + Computation Optimizations

We evaluate the impact of various inference-time Gaussian culling techniques on top of the system-level optimizations described above. Table 3 reports PSNR, FPS, and per-frame time for each of the methods described in Section 5.

8 Conclusion

We presented a comprehensive profiling and optimization study of Tensor Core based Gaussian Splatting renderers. Through systematic analysis using Nsight Compute and nvprof, we identified four critical performance bottlenecks: kernel launch overhead (132.95 μs per frame), memory allocation inefficiencies (327 μs per frame), tensor core underutilization (87.3%), and low GPU occupancy (12.5%).

We developed and implemented targeted system-level optimizations addressing each bottleneck. CUDA graph integration reduced launch overhead by 57×, static buffer reuse eliminated 327 μs of allocation cost per frame, restructured WMMA operations increased tensor core utilization from 87.3% to 92%, and improved warp coherency reduced culling and blending time by 1.38×. The combined optimizations achieve a 1.40× overall speedup, reducing frame time from 16.7ms to 11.9ms. In addition, opacity culling gives us 6.6% improvement over TC-GS in FPS while keeping PSNR constant. The rest of the culling methods give us FPS improvements but lead to a drop in PSNR.

Our systems-level optimized implementation achieves 330.69 FPS on the MipNeRF-360 Bonsai scene, representing a 4.85% improvement over baseline TC-GS and a 334.9% improvement over traditional 3DGS, while maintaining identical rendering quality (PSNR: 32.412 dB). Our systems-level optimization combined with opacity culling gives us 6.6% improvement over TC-GS while maintaining quality. These results demonstrate that careful profiling and targeted optimization can extract significant performance gains even from already-optimized Tensor Core implementations.

8.1 Future Work

Future work could explore kernel splitting or alternative data layouts to improve occupancy beyond 12.5%. Multi-GPU strategies could enable

real-time rendering of larger scenes. Mixed precision (FP16/INT8) could further improve Tensor Core utilization. Evaluating optimizations across diverse scenes would establish robustness and identify scene-dependent characteristics.

References

[1] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering, 2023.

[2] Zimu Liao, Jifeng Ding, Siwei Cui, Ruixuan Gong, Boni Hu, Yi Wang, Hengjie Li, XIngcheng Zhang, Hui Wang, and Rong Fu. Tc-gs: A faster gaussian splatting module utilizing tensor cores, 2025.

[3] Hexu Zhao, Haoyang Weng, Daohan Lu, Ang Li, Jinyang Li, Aurojit Panda, and Saining Xie. On scaling up 3d gaussian splatting training, 2025.