

TC-GS: A Faster Gaussian Splatting Module Utilizing Tensor Cores

ZIMU LIAO, Shanghai Jiao Tong University, China and Shanghai Artificial Intelligence Laboratory, China

JIFENG DING, Shanghai Artificial Intelligence Laboratory, China and Fudan University, China

SIWEI CUI, Shanghai Artificial Intelligence Laboratory, China and Fudan University, China

RUIXUAN GONG, Beijing Institute of Technology, China

BONI HU, Shanghai Artificial Intelligence Laboratory, China and Northwestern Polytechnical University, China

YI WANG, Shanghai Artificial Intelligence Laboratory, China

HENGJIE LI*, Shanghai Artificial Intelligence Laboratory, China and Shanghai Innovation Institute, China

HUI WANG, Shanghai Artificial Intelligence Laboratory, China

XINGCHENG ZHANG, Shanghai Artificial Intelligence Laboratory, China

RONG FU*, Shanghai Artificial Intelligence Laboratory, China



Fig. 1. TC-GS: a Tensor Core-based acceleration module for 3D Gaussian Splatting that can be effortlessly applied to existing 3DGS rendering pipelines. Integrated into 3DGS and its variants, it achieves an extra 2× speedup while preserving rendering quality.

*Corresponding Authors

Authors' Contact Information: Zimu Liao, Shanghai Jiao Tong University, Shanghai, China and Shanghai Artificial Intelligence Laboratory, Shanghai, China; Jifeng Ding, Shanghai Artificial Intelligence Laboratory, Shanghai, China and Fudan University, Shanghai, China; Siwei Cui, Shanghai Artificial Intelligence Laboratory, Shanghai, China and Fudan University, Shanghai, China; Ruixuan Gong, Beijing Institute of Technology, Beijing, China; Boni Hu, Shanghai Artificial Intelligence Laboratory, Shanghai, China and Northwestern Polytechnical University, Xi'an, China; Yi Wang, Shanghai Artificial Intelligence Laboratory, Shanghai, China; Hengjie Li, Shanghai Artificial Intelligence Laboratory, Shanghai, China and Shanghai Innovation Institute, Shanghai, China, lihengjie@pjlab.org.cn; Hui Wang, Shanghai Artificial Intelligence Laboratory, Shanghai, China; Xingcheng Zhang, Shanghai Artificial Intelligence Laboratory, Shanghai, China; Rong Fu, Shanghai Artificial Intelligence Laboratory, Shanghai, China, furong@pjlab.org.cn.



This work is licensed under a Creative Commons Attribution 4.0 International License.

arXiv Preprint Version, 2025

© 2025 Copyright held by the owner/author(s).

3D Gaussian Splatting (3DGS) renders pixels by rasterizing Gaussian primitives, where conditional alpha-blending dominates the computational cost in the rendering pipeline. This paper proposes TC-GS, an algorithm-independent universal module that expands the applicability of Tensor Core (TCU) for 3DGS, leading to substantial speedups and seamless integration into existing 3DGS optimization frameworks. The key innovation lies in mapping alpha computation to matrix multiplication, fully utilizing otherwise idle TCUs in existing 3DGS implementations. TC-GS provides plug-and-play acceleration for existing top-tier acceleration algorithms and integrates seamlessly with rendering pipeline designs, such as Gaussian compression and redundancy elimination algorithms. Additionally, we introduce a global-to-local coordinate transformation to mitigate rounding errors from quadratic terms of pixel coordinates caused by Tensor Core half-precision computation. Extensive experiments demonstrate that our method maintains rendering quality while providing an additional 2.18× speedup over existing Gaussian acceleration algorithms, thereby achieving a total acceleration of up to 5.6×.

CCS Concepts: • Computing methodologies → Rasterization; Parallel algorithms.

Additional Key Words and Phrases: Gaussian Splatting, Tensor Cores, Real-Time Rendering

ACM Reference Format:

Zimu Liao, Jifeng Ding, Siwei Cui, Ruixuan Gong, Boni Hu, Yi Wang, Hengjie Li, Hui Wang, Xingcheng Zhang, and Rong Fu. 2025. TC-GS: A Faster Gaussian Splatting Module Utilizing Tensor Cores. In *Proceedings of arXiv Preprint Version*. ACM, New York, NY, USA, 13 pages.

1 Introduction

3D Gaussian Splatting (3DGS) represents a significant advancement in neural rendering, employing Gaussian primitives to achieve NeRF-comparable quality while reducing optimization time to tens of minutes [Durvasula et al. 2023; Mallick et al. 2024]. However, 3DGS rendering speed is heavily constrained by extensive model parameters and inefficient rendering pipelines[Jiang et al. 2024; Lu et al. 2024; Ren et al. 2024]. Further acceleration remains essential for deployment on resource-constrained devices, real-time large-scale scene rendering systems, and low-latency edge computing scenarios.

Despite numerous efforts to enhance computational efficiency [Feng et al. 2024; Hanson et al. 2024; Huang et al. 2025; Wang et al. 2024], they fail to tap into the most powerful computing units in modern GPUs. Moreover, existing acceleration modules are tightly coupled with core components, preventing modular reuse and inter-method adaptation. This coupling further limits the practical deployment and scalability of 3DGS-based solutions.

Recent research [Feng et al. 2024; Hanson et al. 2024; Huang et al. 2025; Wang et al. 2024] has identified the *conditional alpha-blending* process as the primary pipeline bottleneck. This process, which composites pixel colors with depth-sorted splatted Gaussian primitives, comprises three steps: (1) *Alpha computation*: computing the alpha value (opacity) of each splatted Gaussian fragment on the pixel; (2) *Culling*: eliminating fragments with low alpha values; and (3) *Blending*: compositing remaining fragments into final pixel colors.

Many approaches [Fan et al. 2024; Wang et al. 2024; Ye et al. 2025] employ early culling strategies to reduce the range of splatted Gaussians, thus minimizing redundant alpha-blending computations. Although they eliminate redundant computation, these methods require corresponding pipeline designs to mitigate load imbalance issues introduced by precise intersection [Feng et al. 2024] calculations or sacrifice certain precision[Wang et al. 2024]. Additionally, we observe that over 80% of Gaussian-pixel pairs that require culling remain in the pipeline. Consequently, the first two steps are the dominant part of the alpha-blending process.

Unlike previous efforts to remove redundant computing [Feng et al. 2024; Hanson et al. 2024; Huang et al. 2025; Wang et al. 2024], we focus on utilizing Tensor Core Units (TCUs) to accelerate alpha computation, which remains an unexplored area. In modern GPUs, TCUs are deployed to perform matrix multiply accumulate (MMA), i.e., $D = A \times B + C$ operations within a clock cycle, achieving high throughput on general matrix multiplication (GEMM) based computations including MLPs, CNNs, and transformers [Vaswani et al. 2017]. However, due to their specialized computation pattern, TCUs struggle to be applied to non-GEMM computations, leaving them idle in 3DGS.

To tackle this problem, this paper proposes TC-GS, a hardware-aligned redesign of alpha computation that unlocks Tensor Core acceleration for all 3DGS algorithms. TC-GS formulates a mapping from pixels and Gaussians into two matrices, where their matrix product yields logarithmic alpha values in batches. Implemented as a standalone module, TC-GS not only significantly improves rendering efficiency, but also enables seamless integration with existing 3DGS acceleration kernels, further enhancing overall performance.

In summary, the paper proposes the following contributions:

- A Detailed runtime analysis of the rendering pipeline showing that alpha computation in conditional alpha-blending is the performance bottleneck in 3DGS inference rendering.
- The first use of Tensor Cores to accelerate alpha-blending, expanding the Tensor Core applicability beyond GEMM-based computations.
- A global-to-local coordinate transformation ($G2L$, Section 4.3) that reduces the absolute magnitudes of quadratic coordinate terms, which are sensitive to Tensor Core half-precision computation, enabling lossless rendering quality while maintaining full acceleration benefits.

2 Related Works

2.1 Fast 3D Gaussian Splatting

3DGS [Kerbl et al. 2023] has emerged as a leading real-time rendering technique with photo-realistic visual quality, finding wide applications diverse domains including autonomous driving [Zhou et al. 2024], robotics [Zhu et al. 2024], and virtual reality [Jiang et al. 2024]. Current acceleration research explores two primary optimization strategies to enhance 3DGS performance.

The first strategy modifies the algorithm itself to achieve better efficiency [Lee et al. 2024; Mallick et al. 2024; Ye et al. 2025]. CompactGS [Lee et al. 2024], Mini-splatting [Fang and Wang 2024], PUP 3D-GS [Hanson et al. 2025], and gsplat [Ye et al. 2025] introduce more efficient radiance field representations and adaptive resolution primitive pruning algorithms, while Taming-3DGS [Mallick et al. 2024] presents mathematically equivalent but computationally efficient solutions for gradient computation and attribute updates, substantially accelerating training speed. Stochastic rasterization for sorting-free 3DGS [Kheradmand et al. 2025] is an another approach of improving the algorithm itself.

The second category focuses on improving computational efficiency of the original algorithm through intelligent task scheduling and CUDA kernel optimization [Durvasula et al. 2023; Feng et al. 2024; Gui et al. 2024; Hanson et al. 2024; Wang et al. 2024; Ye et al. 2025]. Balanced-3DGS [Gui et al. 2024] optimizes the forward computation of rendering CUDA kernels within warps while minimizing thread idle time and maximizing resource utilization by evenly distributing tasks across computational blocks. Several works, including AdRGaussian [Wang et al. 2024], FlashGS [Feng et al. 2024], and SpeedySplat [Hanson et al. 2024], aim to reduce the number of pixels processed per Gaussian by designing more precise Gaussian-tile bounding boxes, with FlashGS [Feng et al. 2024] and SpeedySplat [Hanson et al. 2024] further computing exact Gaussian-tile intersections. DISTWAR [Durvasula et al. 2023]

Table 1. Tensor Core & CUDA Core Comparison among various GPUs. * stands for using new sparse feature

GPU	Peak FP16	Peak FP16 Tensor Core
H100 (PCIe)	96 TFLOPS	800 TFLOPS 1600 TFLOPS*
A100	78 TFLOPS	312 TFLOPS 624 TFLOPS*
V100	31.4 TFLOPS	125 TFLOPS
RTX 4090	82.6 TFLOPS	330.3 TFLOPS 660.6 TFLOPS*

enhances processing efficiency through sophisticated thread management strategies, fully leveraging warp-level reductions in SM sub-cores and intelligently distributing atomic computations between SM and L2 atomic units based on memory access patterns. While these methods significantly accelerate training and rendering speed, tight coupling between algorithmic modules hinders transferability and reusability. Moreover, they fail to leverage modern hardware acceleration features such as Tensor Cores, further limiting the practical deployment and scalability of 3DGS solutions.

2.2 Tensor Cores for Non-GEMM Based Computations

Tensor Cores Units are specialized hardware units developed by NVIDIA to accelerate Deep Learning applications [Choquette et al. 2021]. Each Tensor Core can perform a matrix multiply accumulate operation $D = A \times B + C$ on small matrices within a GPU cycle, where A and B must be in half precision format while the accumulators, C and D, can be either single or half precision. Tensor Core can achieve higher throughput than normal CUDA Cores: for instance on the NVIDIA A100 GPU, CUDA Cores has maximum 78 TFLOPS fp16 performance, whereas Tensor Cores can achieve 312 TFLOPS which is 4× times faster than CUDA Cores.

Despite this theoretical advantage, it is not a trivial task to exploit Tensor Cores in arbitrary applications since Tensor Cores only support the matrix-multiply-accumulate (MMA) instruction rather than the full range of CUDA instructions set. This restriction requires algorithms to be reformulated in terms of small matrix operations, but the reward is exceptionally fast execution. Consequently, the strict requirements yet remarkable potential speed of Tensor Core operations have attracted researchers to explore adapting non-machine learning algorithms to GPU Tensor Cores, leveraging their performance potential. Previous works have successfully accelerated primitives such as reduction, scan and breadth-first search (BFS) using Tensor Cores [Dakkak et al. 2019a,b; Niu and Casas 2025]. Recently, ConvStencil [Liu et al. 2022] has managed to adapt stencil computation to TCUs, while other works have explored TCU-based implementations in applications such as HighQR [Leng et al. 2024] and TCUDB [Hu et al. 2022]. However, to our best knowledge, no prior work has explored how to exploit Tensor Cores directly within a 3D Gaussian Splatting rendering pipeline.

3 Preliminaries and Observations

3.1 3DGS Rendering Pipeline

In this paper, we focus on the official implementation of 3DGS, which is the mainstream pipeline widely used in most state-of-the-art acceleration methods.

A 3DGS model consists of a set of 3D Gaussians primitives $\mathcal{G} = \{G_1, G_2, \dots, G_p\}$. Each Gaussian $G : \mathbb{R}^3 \rightarrow \mathbb{R}$ is parameterized by its mean $\mu \in \mathbb{R}^3$, semi-definite covariance $\Sigma \in \mathbb{R}^{3 \times 3}$, opacity $o \in (0, 1]$, and color $c \in [0, 1]^3$ and maps any 3D position $x \in \mathbb{R}^3$ to its density:

$$G(x) = o e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)}. \quad (1)$$

The inference procedure of 3DGS can be divided into the following 3 stages:

Preprocess. The preprocess stage projects 3D Gaussian G into 2D planes in parallel according to the viewing transform W of a given camera:

$$\mu' = W\mu \in \mathbb{R}^2, \quad \Sigma' = JW\Sigma^T J^T \in \mathbb{R}^{2 \times 2}, \quad (2)$$

where $J = \frac{\partial \mu'}{\partial \mu}$ is the Jacobian of projection W . $\mu' = (\mu'_x, \mu'_y)$ and Σ' is the mean and the covariance of projected Gaussian G' . The complexity of the preprocessing stage is $\mathcal{E}_p = O(P)$, where P is the number of Gaussians.

Sorting. The screen space \mathcal{S} are divided into disjoint tiles \mathcal{T} with equal sizes. For each projected Gaussian, this stage identifies all the tiles it covers. Then, for each tile $\mathcal{T} \in \mathcal{S}$, the overlapped Gaussians are sorted by depth. A tile \mathcal{T} coupled with a covering Gaussian G' is denoted as a **splat** (\mathcal{T}, G') . The total number of splats given by

$$N = \sum_{j=1}^P \sum_{\mathcal{T} \in \mathcal{S}} (\mathcal{T} \cap G'_j \neq \emptyset) = \sum_{\mathcal{T} \in \mathcal{S}} n(\mathcal{T}). \quad (3)$$

where $\mathbf{1}$ is the indicator function that checks whether the Gaussian covers the tile. The process of determining coverage depends on the specific algorithm used. The complexity of the sorting stage is $\mathcal{E}_s = O(sN)$, where $s = 64$ is the coefficient in radix-sort.

ALGORITHM 1: Conditional Alpha-blending

```

Input: Tile  $\mathcal{T}$ , Sorted Projected Gaussians
 $\mathcal{G}'(\mathcal{T}) = (G'_1, \dots, G'_{n(\mathcal{T})})$ 
Output: Pixel colors  $C_{\mathcal{T}} = (C_1, \dots, C_{|\mathcal{T}|})$ 
foreach pixel  $p \in \mathcal{T}$  in parallel do
     $T \leftarrow 1$ ; // Initialize transmissivity
     $C \leftarrow (0, 0, 0)$ ; // Initialize RGB
    foreach Gaussian  $G' \in \mathcal{G}'(\mathcal{T})$  do
         $\alpha \leftarrow o \exp(-\frac{1}{2}(\mu' - p)^T \Sigma'^{-1}(\mu' - p))$ 
        if  $\alpha < \frac{1}{255}$  then
            continue; // Cull invisible Gaussians
        end
        if  $T - \alpha T < 0.0001$  then
            break; // Terminate blending
        end
         $C \leftarrow C + c\alpha T$ ; // Composite colors
         $T \leftarrow T - \alpha T$ ; // Update transmissivity
    end
end

```

Conditional Alpha-blending. This stage renders the final color of each pixel in a tile $\mathcal{T} = (p_1, p_2, \dots, p_{|\mathcal{T}|})$ in parallel as Algorithm 1. Initially, for each projected overlapping Gaussian, the renderer will perform **alpha computation** to get the local opacity α at each pixel $p = (p_x, p_y) \in \mathbb{R}^2$:

$$\alpha_{i,j} = o_j \exp\left(-\frac{1}{2}(\mu'_j - p_i)^T (\Sigma'_j)^{-1} (\mu'_j - p_i)\right). \quad (4)$$

Next, the renderer performs **culling** of Gaussians at each pixel if the local opacity is below a threshold:

$$\alpha_{i,j} < \frac{1}{255}, \quad (5)$$

The culling process is only a conditional judgment.

Otherwise, the surviving Gaussians will perform **blending** and contribute the pixel's color:

$$C_i = \sum_{j=1}^{n(\mathcal{T})} c_j \alpha_{i,j} T_{i,j}; \quad T_{i,j} = \prod_{k=1}^{j-1} (1 - \alpha_{i,k}). \quad (6)$$

where $T_{i,j}$ denotes the transmissivity and $n(\mathcal{T})$ is the number of splats of the tile \mathcal{T} . If $T_{i,j} < 0.0001$, the renderer will **terminate** the blending process for that pixel and output the color. A pixel p rendered by a Gaussian G is referred to as a **fragment**, (p, G) . The complexity of the alpha-blending stage is $\mathcal{E}_a = O(|\mathcal{T}|N)$.

Thus, the total complexity of the Gaussian rasterization pipeline can be summarized as $\mathcal{E} = \mathcal{E}_p + \mathcal{E}_s + \mathcal{E}_a$, where $\mathcal{E}_p, \mathcal{E}_s, \mathcal{E}_a$ represent the complexities of preprocess, sorting, and alpha-blending stages, respectively.

3.2 Observation and Motivation

To better understand the bottlenecks in current rendering methods, we conduct a detailed time breakdown analysis of the three stages mentioned above. Specifically, we profile 3DGS, AdR-Gaussian, and Speedy-Splat on an NVIDIA A800 using multiple datasets. While recent methods incorporate early culling strategies with more precise intersection checks to reduce the number of splats N , thereby decreasing the time spent on sorting and blending, the alpha blending stage still remains the dominant contributor to the overall rendering time, as shown in the bottom-left corner of Figure 3.

Motivated by this observation, we further analyze the core kernel `renderCUDA`, which implements the alpha blending stage, in order to uncover deeper inefficiencies and guide targeted improvements.

The alpha-blending stage will blend all fragments into pixel colors. These fragments are categorized into 3 types as Figure 2 shows:

- **Culled fragments:** Fragments that are discarded because $\alpha < \frac{1}{255}$. As shown in Figure 2, the second fragment is culled.
- **Blended fragments:** Fragments that contribute to the final pixel colors. As shown in Figure 2, the first, third, and fourth fragments are blended.
- **Skipped fragments:** Fragments that are ignored when $T < 0.0001$, causing the renderer to terminate the blending process for the corresponding pixel. As shown in Figure 2, the fifth fragment and the ones that follow are skipped. There is **no** computation on skipped fragments.

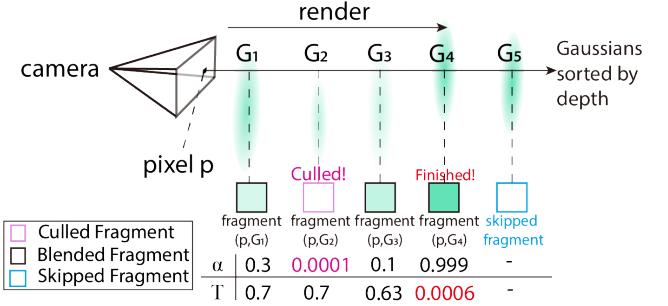


Fig. 2. a) Three types of fragments on a pixel. b) If the Gaussian only covers a small portion of tile, a large amount culled fragments are generated.

Table 2. The alpha computation and culling steps are run on both blended and culled fragments, while the blending step is only performed on blended fragments. There is no computation on skipped fragments.

Fragments	Alpha Computation	Culling	Blending
Blended	✓	✓	✓
Culled	✓	✓	
Skipped			

To model the computation for each type of fragments, we denote $k_\alpha, k_{\text{cull}}, k_{\text{blend}}$ as the average computation cost for alpha computation, culling, and blending. Similarly, let $f_{\text{blend}}, f_{\text{cull}}, f_{\text{skip}}$ represent the number of blended, culled and skipped fragments. According to Table 2, the total computation amount is:

$$C = k_{\text{blend}} f_{\text{blend}} + (k_{\text{cull}} + k_\alpha) (f_{\text{blend}} + f_{\text{cull}}). \quad (7)$$

To identify the bottleneck within the alpha-blending stage, we collect and categorize the generated fragments on various scenes using 3DGS, AdR-Gaussian and Speedy-Splat. As shown on the right side of Figure 3, culled and skipped fragments account for a large proportion. Since skipped fragments do not incur any computational cost, we can preliminarily conclude that the bottleneck lies in the alpha computation and culling steps.

4 Method

This section introduces how TC-GS utilizes the Tensor Cores to accelerate the alpha-computation and culling process, which are the bottlenecks of rendering. The fundamental components of TC-GS include *EarlyCull* for culling optimization, *Frag2Mat* for matrix transformation and *G2L* for coordinate transformation. The detailed design is shown in Figure 4.

4.1 EarlyCull: Reducing Exponential Computation

First, TC-GS adopts *EarlyCull*, an early-pruning strategy: it identifies invisible Gaussians that can be culled before the alpha-computation stage, greatly reducing the number of exponential instructions exp executed and improving throughput.

We begin by expressing Equation (4) as the following form:

$$\alpha_{i,j} = e^{\beta_{i,j}}, \quad (8)$$

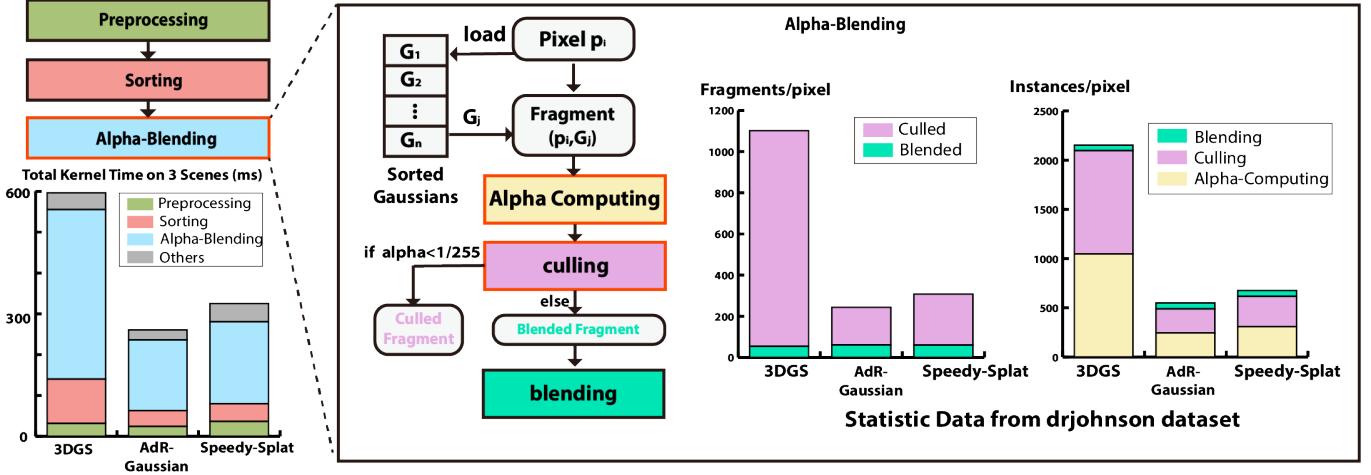


Fig. 3. Analysis of 3DGS rendering bottlenecks: The left side shows the time distribution of preprocessing, sorting, and alpha-blending for 3DGS, AdR-Gaussian, and Speedy-Splat, with alpha-blending dominating. The right side details alpha computation, culling, and blending, identifying culled fragments as the primary bottleneck due to redundant alpha computations, while skipped fragments incur no cost.

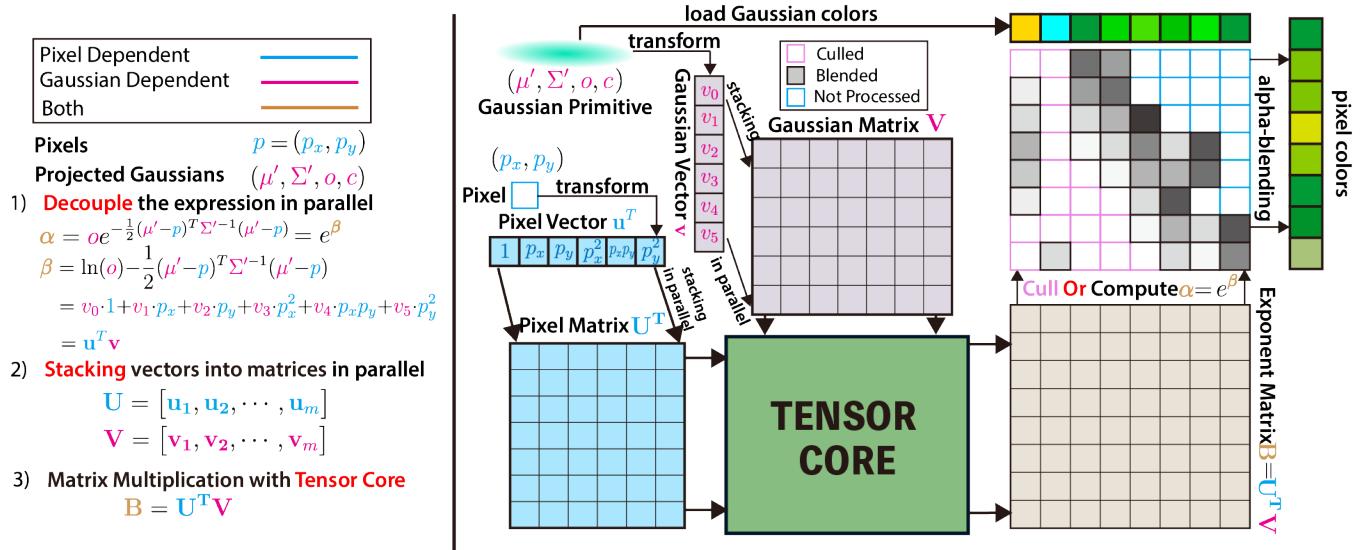


Fig. 4. Design of *Frag2Mat*: the alpha computation is reformulated as matrix multiplication to fully leverage Tensor Cores for accelerating alpha calculation.

where the exponent $\beta_{i,j}$ satisfies

$$\beta_{i,j} = \ln(o_j) - \frac{1}{2}(\mu'_j - p_i)^T (\Sigma'_j)^{-1}(\mu'_j - p_i). \quad (9)$$

Then the culling condition, Equation (5), can be converted into

$$\beta_{i,j} < -\ln(255). \quad (10)$$

EarlyCull allows us to filter out non-contributing Gaussians in advance, reducing the amount of expensive exp instructions issued by alpha computation. Since prior experiments show that culled fragments account for a large portion of the workload, this approach yields a substantial speed-up for the entire pipeline.

4.2 *Frag2Mat*: Fitting the MMA Operation

While Tensor Cores specialize in matrix multiply-accumulate (MMA) operations, the original alpha-computation involves per-pixel-per-Gaussian evaluations that exhibit non-coalesced memory access patterns and fragmented arithmetic operations – characteristics mismatched with matrix-oriented hardware architectures. To align with Tensor Cores’ computational paradigm and fully exploit their high throughput, we restructure the alpha-computation into batched matrix operations through algebraic reformulation.

We propose *Frag2Mat*, a computational restructuring that transforms per-pixel-per-Gaussian exponent calculations into unified

matrix operations. By expressing the exponent as a linear combination of pixel coordinates, Gaussian parameters and their quadratic terms, we decouple multivariate dependencies into two independent components: a pixel vector encoding coordinate polynomials for its pixel, and a Gaussian vector composed of parameters from its Gaussian. The final exponent values are batch-calculated through standard matrix multiplication of two matrices formed by stacking pixel vectors and Gaussian vectors respectively. LC The core idea of TC-GS is decoupling the computation of exponents $\beta_{i,j}$ into two separated terms: one that depends only on the pixel index and another that depends only on the Gaussian. Notably, we observe that the exponent can be expressed as a quadratic function of the pixel p . Based on this observation, TC-GS transforms Equation (9) into the following standard from:

$$\beta = v_0 + v_1 p_x + v_2 p_y + v_3 p_x^2 + v_4 p_x p_y + v_5 p_y^2, \quad (11)$$

where

$$\begin{aligned} v_0 &= \ln(o) - \frac{1}{2} \mu'^T \Sigma'^{-1} \mu', \\ v_1 &= \sigma_{11} \mu'_x + \sigma_{12} \mu'_y, \quad v_2 = \sigma_{12} \mu'_x + \sigma_{22} \mu'_y, \\ v_3 &= -\frac{1}{2} \sigma_{11}, \quad v_4 = -\sigma_{12}, \quad v_5 = -\frac{1}{2} \sigma_{22} \end{aligned} \quad (12)$$

with the notation that symmetric $\Sigma'^{-1} = \begin{bmatrix} \sigma_{11} & \sigma_{12} \\ \sigma_{21} & \sigma_{22} \end{bmatrix}$ with $\sigma_{12} = \sigma_{21}$.

The Equation (11) is the expanded dot product of two vectors:

$$\beta_{i,j} = \mathbf{u}_i^T \mathbf{v}_j, \quad (13)$$

where

$$\mathbf{u} = [1 \quad p_x \quad p_y \quad p_x^2 \quad p_x p_y \quad p_y^2]^T \quad (14)$$

and

$$\mathbf{v} = [v_0 \quad v_1 \quad v_2 \quad v_3 \quad v_4 \quad v_5]^T. \quad (15)$$

Here \mathbf{u} , the pixel vector solely dependent on pixel properties, which will be computed only *once* for each pixel. Similarly, each Gaussian vector \mathbf{v} can be only computed once. Therefore, the expression of the exponent is decoupled into the dot product of two individual vectors. To align the computation to the Tensor Cores, TC-GS stacks the vectors into two independent matrices:

$$\mathbf{U} = [\mathbf{u}_1 \quad \mathbf{u}_2 \quad \cdots \quad \mathbf{u}_m] \in \mathbb{R}^{6 \times m}, \quad (16)$$

$$\mathbf{V} = [\mathbf{v}_1 \quad \mathbf{v}_2 \quad \cdots \quad \mathbf{v}_n] \in \mathbb{R}^{6 \times n}, \quad (17)$$

where m is the the number of pixels within a tile, and n is the number of Gaussians whose projection is overlapped with the tile. Then TC-GS computes the exponent matrix $\mathbf{B} \in \mathbb{R}^{m \times n}$ as matrix multiplication:

$$\mathbf{B} = \begin{bmatrix} \beta_{1,1} & \cdots & \beta_{1,n} \\ \vdots & \ddots & \vdots \\ \beta_{m,1} & \cdots & \beta_{m,n} \end{bmatrix} = \begin{bmatrix} \mathbf{u}_1^T \mathbf{v}_1 & \cdots & \mathbf{u}_1^T \mathbf{v}_n \\ \vdots & \ddots & \vdots \\ \mathbf{u}_m^T \mathbf{v}_1 & \cdots & \mathbf{u}_m^T \mathbf{v}_n \end{bmatrix} = \mathbf{U}^T \mathbf{V}. \quad (18)$$

Therefore, we can leverage Tensor Cores to compute β for m pixels and n Gaussians simultaneously. As Tensor Cores natively support half-precision matrix operations, additional design is required to ensure numerical stability throughout the alpha computation.

4.3 G2L: Fitting the Half Precision

Although we adopted Tensor Cores for alpha computation and culling. The challenges remain as Tensor Cores do not support common single-precision floating-point operations. Instead, they only take FP16 or TF32 as the input of the MMA operation, resulting in a higher machine epsilon of $\epsilon_H = 9.77 \times 10^{-4}$ compared to FP32's $\epsilon_F = 1.19 \times 10^{-7}$. The rounding error of a floating number r is $|r| \epsilon_H$. Therefore, the error of computation on Tensor Cores is more sensitive at the absolute value of the input data.

We notice that the range of the pixel p is $[0, w] \times [0, h]$, where w, h are the width and height of the screen space, respectively. When loading the pixel vector \mathbf{u} in Equation (14) into Tensor Cores, its absolute values of quadratic terms $p_x^2, p_x p_y$ and p_y^2 can exceed a million, contributing catastrophic rounding errors to β . Moreover, overflow will occur if the FP16 range $[-6.55 \times 10^5, 6.55 \times 10^5]$ is used.

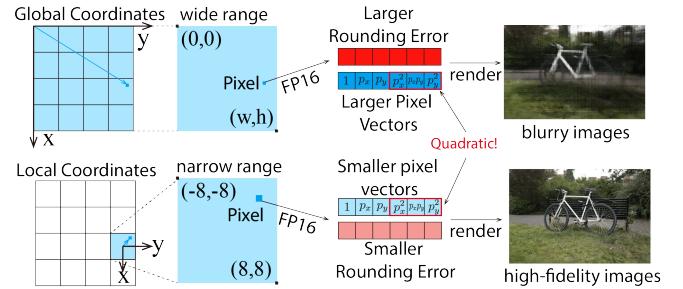


Fig. 5. The quadratic term of pixel coordinates significantly contributes the rounding error, resulting in blurry images. Local coordinates constrain the value of Δp into $[-8, 8]^2$, which can reduce the upper bound of the rounding error.

To tackle this problem, TC-GS has designed *G2L*, a coordinate mapping, which significantly reduces the absolute value of terms in pixel vectors. Since all pixels p within a tile \mathcal{T} will be rendered in parallel, *G2L* can map the coordinates of pixels into the local coordinate:

$$\Delta p = p - p(\mathcal{T}), \quad (19)$$

Since each tile contains 16×16 pixels, then the range of pixel in LC is $\Delta p \in [-8, 8]^2$, and the maximum element of \mathbf{u}_Δ will not exceed 64. This conversion does not only avoid the overflow in FP16, but also significantly reduces the rounding error.

Respectively, the means of Gaussians in the local coordinate is

$$\Delta \mu' = \mu' - p(\mathcal{T}). \quad (20)$$

Notice that the relative position between μ' and p stays invariant:

$$\Delta \mu' - \Delta p = \mu' - p. \quad (21)$$

Therefore, *Frag2Mat* can be also performed on local coordinates.

As the proof in the appendices goes, *G2L* reduces the quadratic rounding error $O(h^2 + hw + w^2) \epsilon_H$ into linear $O(h+w) \epsilon_H$, and achieve better rendering quality in FP16 as Section 5.3 demonstrates.

Table 3. Comparison With/Without TC-GS Module Across Datasets.

Method	Tanks & Temple				Deep-Blending				Mip-NeRF360			
	FPS↑	PSNR↑	SSIM↑	LPIPS↓	FPS↑	PSNR↑	SSIM↑	LPIPS↓	FPS↑	PSNR↑	SSIM↑	LPIPS↓
3DGS	161.548	23.687	0.851	0.169	131.373	29.803	0.907	0.238	128.88	26.546	0.785	0.25
3DGS(+TC-GS)	323.423(2.13×)	23.682	0.851	0.169	286.819(2.185×)	29.803	0.906	0.236	258.809(2.01×)	26.544	0.785	0.25
FlashGS	326.835	23.706	0.851	0.169	563.590	29.789	0.907	0.239	399.772	26.551	0.786	0.251
FlashGS(+TC-GS)	539.387(1.65×)	23.684	0.851	0.169	735.777(1.305×)	29.806	0.906	0.236	479.338(1.199×)	26.508	0.785	0.25
Speedy-Splat	268.602	23.687	0.852	0.17	315.822	29.803	0.907	0.238	264.23	26.546	0.785	0.25
Speedy-Splat(+TC-GS)	521.284(1.94×)	23.684	0.852	0.169	579.988(1.84×)	29.802	0.906	0.236	465.308(1.76×)	26.544	0.785	0.25
AdR-Gaussian	278.65	23.635	0.852	0.17	324.18	29.814	0.907	0.238	261.339	26.52	0.785	0.25
AdR-Gaussian(+TC-GS)	545.154(1.955×)	23.632	0.851	0.169	612.063(1.89×)	29.815	0.906	0.236	467.310(1.78×)	26.519	0.785	0.25

5 Experiment

5.1 Experiment Setup

Implementation. Current computation patterns which Tensor Cores support do not include length-6 MMA operations. A simple way to align the required length-8 MMA operation is zero padding. In our implementation, we replace v_0 in \mathbf{v} with $\frac{v_0}{3}$ and duplicate it 3 times. The constant term 1 in \mathbf{u} are duplicated respectively. This approach can reduce the rounding error by reducing the absolute value.

datasets&metrics. To prove that our method is universal and efficient, we conduct experiments on the same datasets as those used in 3DGS[Kerbl et al. 2023]. Specifically, the datasets contain all scenes from Mip-NeRF360[Barron et al. 2022], two outdoor scenes (Truck & Train) from Tanks & Temple[Knapsitsch et al. 2017], two indoor scenes (Drjohnson & playroom) in Deep-Blending[Hedman et al. 2018]. For each scene, CUDA events are inserted at the start and end of the forward rendering procedure to measure FPS values, and the reported results represent the average metrics of the test datasets. All experiments are conducted on an NVIDIA A800 GPU (80GB SXM) equipped with 432 Tensor Cores, achieving a peak 624 TFLOPS FP16 performance.

5.2 Comparisons

TC-GS demonstrates significant performance improvements in all tested methods while keeping rendering quality. The qualitative and quantitative comparison between four baseline rendering pipelines and their results after integrating the TC-GS module are shown in Table 3.

Rendering speed. Across all datasets, incorporating the TC-GS module roughly **doubles** frame throughput except for FlashGS. The state-of-the-art performance is attained by FlashGS integrated with TC-GS, which achieves a **3.3–5.6×** speedup compared to the original 3DGS, while delivering rendering speeds of **479.3–735.8** FPS.

By comparing the performance of alpha-blending, as shown in Figure 4, the rendering pipeline gains a 2-4× performance improvement. Combined with the previous analysis, this further emphasizes the importance of optimizing the rendering pipeline.

While AdR-Gaussian and Speedy-Splat already employ advanced optimizations to reduce redundant computation, TC-GS further pushed their performance boundaries, delivering significant speedup of **1.87×** and **1.84×** respectively. As for FlashGS, while its existing

Table 4. Comparison of alpha-blending time across methods and datasets.

Method	Scene	with TC-GS time(ms)	original time(ms)
3DGS	drjohnson	1.348(3.49×)	4.705
	train	1.217(3.57×)	4.348
	flowers	0.880(3.68×)	3.236
FlashGS	drjohnson	0.540(2.18×)	1.179
	train	0.609(2.03×)	1.234
	flowers	0.625(2.38×)	1.487
AdR-Gaussian	drjohnson	0.406(4.76×)	1.931
	train	0.518(4.23×)	2.191
	flowers	0.469(4.39×)	2.061
Speedy-Splat	drjohnson	0.510(3.86×)	1.969
	train	0.672(3.46×)	2.326
	flowers	0.592(3.52×)	2.083

pipeline rendering optimizations yield a more modest 1.38× average acceleration with TC-GS integration, we emphasize that the pipeline optimizations and TC-GS’s approach operate orthogonally.

Image Quality. Table 3 also demonstrates the rendered images quality metrics across all methods. We observe negligible differences in PSNR, SSIM, and LPIPS metrics between each method’s original implementation and its TC-GS-enhanced counterpart. The minimal variations observed likely originate from hardware-level instruction set disparities and floating-point precision conversion artifacts in the rasterization pipeline.

5.3 Ablation Study

To determine the source of the acceleration, we conducted an ablation study on *EarlyCull* and *Frag2Mat* with *G2L* across various renderers and datasets. As Table 5 shows, both methods can accelerate the rendering speed. *EarlyCull* alone achieves 1.06 – 1.51× speedup, but it accelerates slightly when *Frag2Mat* is enabled. *Frag2Mat* alone achieves 1.97 – 2.24× speedup, and it still accelerates significantly with *EarlyCull* enabled. Therefore, *Frag2Mat* with *G2L* contributes the main speedup.

To clearly illustrate the impact of *G2L* on rendering quality and performance, we ran the following comparisons shown in Table 6 and the rendering images are shown in Figure 6. The ablation experiment shows that all Tensor Core variants boosted FPS. However, direct TF32 computation already introduced noticeable image-quality degradation, and FP16 alone dropped PSNR to just 8 dB. In contrast,

Table 5. Rendering FPS across settings, renderers, and datasets. To evaluate the effect of each component in TC-GS, we conducted an ablation study on EarlyCull and Frag2Mat with G2L.

EarlyCull	Frag2Mat with G2L	3DGS		SpeedySplat	
		truck	drjohnson	truck	drjohnson
✓		169.834	108.123	366.457	349.673
	✓	210.110	163.967	406.434	371.605
✓	✓	351.965	242.993	732.008	688.217
✓	✓	361.839	253.770	727.647	701.254

combining FP16 Tensor Core optimization with Local Coordinates method preserved nearly same image quality while delivering the highest FPS. This demonstrates that our G2L can simultaneously maximize performance and maintain rendering fidelity.

Table 6. Ablation study on applying TC-GS on original 3DGS.

Frag2Mat	G2L	Precision	drjohnson		truck	
			PSNR↑	FPS↑	PSNR↑	FPS↑
✓		FP32	29.48	108.123	25.44	169.834
✓		FP16	8.85	253.79	6.24	356.619
✓		TF32	20.02	181.429	14.27	290.047
✓	✓	FP16	29.46	253.77	25.44	361.839



Fig. 6. Ablation study on G2L when applying TC-GS on original 3DGS.

5.4 Discussion

5.4.1 Limitations. While our method achieves significant acceleration in rendering computations, further optimization could be attained by adopting pipeline techniques similar to those proposed in FlashGS. However, unlike FlashGS’s thread-level pipelining, a warp-level pipelining approach is required. Notably, post-optimization profiling reveals an increased proportion of preprocess overhead in the forward procedure. This preprocess phase inherently involves numerous matrix-based operations such as view transformations, providing opportunities for optimization using Tensor Cores.

5.4.2 Training. Although TC-GS is primarily designed to optimize real-time rendering, our implementation serves as a plug-and-play accelerator for both inference and training, requiring no modifications to the original 3DGS model or the training process.

6 Conclusion

In this paper, we propose TC-GS, a high-performance rendering module for accelerating 3D Gaussian Splatting. TC-GS optimizes the rendering pipeline by decomposing alpha-blending into two components, i.e., Gaussian-related and pixel-related matrices. Through reformulating alpha-blending as matrix multiplications, TC-GS achieves a significant speedup of $2.03\times$ to $4.76\times$ for alpha blending on Tensor Cores. When integrated into state-of-the-art pipelines, TC-GS attains approximately a $2\times$ throughput improvement; even when compared against highly optimized, software-pipelined methods, it delivers a $1.38\times$ speed-up. These gains are realized without any degradation in image quality, demonstrating TC-GS’s superior performance, scalability, and broad applicability for future real-time neural rendering systems.

Acknowledgments

This work is supported by the National Key R&D Program of China (No. 2022ZD0160201), Shanghai Artificial Intelligence Laboratory, and Shanghai Municipal Science and Technology Major Project.

References

- Jonathan T Barron, Ben Mildenhall, Dor Verbin, Pratul P Srinivasan, and Peter Hedman. 2022. Mip-nerf 360: Unbounded anti-aliased neural radiance fields. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 5470–5479.
- Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. NVIDIA A100 Tensor Core GPU: Performance and Innovation. *IEEE Micro* 41, 2 (2021), 29–35. doi:10.1109/MM.2021.3061394
- Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. 2019a. Accelerating reduction and scan using tensor core units. In *Proceedings of the ACM International Conference on Supercomputing*. 46–57.
- Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. 2019b. Accelerating reduction and scan using tensor core units. In *Proceedings of the ACM International Conference on Supercomputing (Phoenix, Arizona) (ICS’19)*. Association for Computing Machinery, New York, NY, USA, 46–57. doi:10.1145/3330345.3331057
- Sankeerth Durvasula, Adrian Zhao, Fan Chen, Ruofan Liang, Pawan Kumar Sanjaya, and Nandita Vijaykumar. 2023. Distwar: Fast differentiable rendering on raster-based rendering pipelines. *arXiv preprint arXiv:2401.05345* (2023).
- Zhiwen Fan, Wenyang Cong, Kairun Wen, Kevin Wang, Jian Zhang, Xinghao Ding, Danfei Xu, Boris Ivanovic, Marco Pavone, Georgios Pavlakos, et al. 2024. InstantSplat: Unbounded sparse-view pose-free gaussian splatting in 40 seconds. *arXiv preprint arXiv:2403.20309* 2, 3 (2024), 4.
- Guangchi Fang and Bing Wang. 2024. Mini-splatting: Representing scenes with a constrained number of gaussians. In *European Conference on Computer Vision*. Springer, 165–181.
- Guofeng Feng, Siyan Chen, Rong Fu, Zimu Liao, Yi Wang, Tao Liu, Zhilin Pei, Hengjie Li, Xingcheng Zhang, and Bo Dai. 2024. Flashgs: Efficient 3d gaussian splatting for large-scale and high-resolution rendering. *arXiv preprint arXiv:2408.07967* (2024).
- Hao Gui, Lin Hu, Rui Chen, Mingxiao Huang, Yuxin Yin, Jia Yang, Yong Wu, Chen Liu, Zhongxu Sun, Xueyang Zhang, et al. 2024. Balanced 3DGs: Gaussian-wise parallelism rendering with fine-grained tiling. *arXiv preprint arXiv:2412.17378* (2024).
- Alex Hanson, Allen Tu, Geng Lin, Vasu Singla, Matthias Zwicker, and Tom Goldstein. 2024. Speedy-Splat: Fast 3D Gaussian Splatting with Sparse Pixels and Sparse Primitives. *arXiv preprint arXiv:2412.00578* (2024).
- Alex Hanson, Allen Tu, Vasu Singla, Mayukha Jayawardhana, Matthias Zwicker, and Tom Goldstein. 2025. Pup 3d-gs: Principled uncertainty pruning for 3d gaussian splatting. In *Proceedings of the Computer Vision and Pattern Recognition Conference*. 5949–5958.
- Peter Hedman, Julien Philip, True Price, Jan-Michael Frahm, George Drettakis, and Gabriel Brostow. 2018. Deep blending for free-viewpoint image-based rendering. *ACM Transactions on Graphics (ToG)* 37, 6 (2018), 1–15.
- Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. 2022. TCUDB: Accelerating database with tensor processors. In *Proceedings of the 2022 International Conference on Management of Data*. 1360–1374.
- Xiaotong Huang, He Zhu, Zihan Liu, Weikai Lin, Xiaohong Liu, Zhezhi He, Jingwen Leng, Minyi Guo, and Yu Feng. 2025. SeeLe: A Unified Acceleration Framework for Real-Time Gaussian Splatting. *arXiv preprint arXiv:2503.05168* (2025).
- Ying Jiang, Chang Yu, Tianyi Xie, Xuan Li, Yutao Feng, Huamin Wang, Minchen Li, Henry Lau, Feng Gao, Yin Yang, et al. 2024. Vr-gs: A physical dynamics-aware

- interactive gaussian splatting system in virtual reality. In *ACM SIGGRAPH 2024 Conference Papers*. 1–1.
- Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 2023. 3d gaussian splatting for real-time radiance field rendering. *ACM Trans. Graph.* 42, 4 (2023), 139–1.
- Shakiba Kheradmand, Delio Vicini, George Kopanas, Dmitry Lagun, Kwang Moo Yi, Mark Matthews, and Andrea Tagliasacchi. 2025. StochasticSplats: Stochastic Rasterization for Sorting-Free 3D Gaussian Splatting. *arXiv preprint arXiv:2503.24366* (2025).
- Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. 2017. Tanks and temples: Benchmarking large-scale scene reconstruction. *ACM Transactions on Graphics (ToG)* 36, 4 (2017), 1–13.
- Joo Chan Lee, Daniel Rho, Xiangyu Sun, Jong Hwan Ko, and Eunbyung Park. 2024. Compact 3d gaussian representation for radiance field. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 21719–21728.
- Yuhan Leng, Gaoyuan Zou, Hansheng Wang, Panruo Wu, and Shaoshuai Zhang. 2024. High Performance Householder QR Factorization on Emerging GPU Architectures Using Tensor Cores. *IEEE Transactions on Parallel and Distributed Systems* (2024).
- Xiaoyan Liu, Yi Liu, Hailong Yang, Jianjin Liao, Mingzhen Li, Zhongzhi Luan, and Depei Qian. 2022. Toward accelerated stencil computation by adapting tensor core unit on gpu. In *Proceedings of the 36th ACM International Conference on Supercomputing*. 1–12.
- Tao Lu, Mulin Yu, Linning Xu, Yuanbo Xiangli, Limin Wang, Dahua Lin, and Bo Dai. 2024. Scaffold-gs: Structured 3d gaussians for view-adaptive rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 20654–20664.
- Saswat Subhajyoti Mallick, Rahul Goel, Bernhard Kerbl, Markus Steinberger, Francisco Vicente Carrasco, and Fernando De La Torre. 2024. Taming 3dgs: High-quality radiance fields with limited resources. In *SIGGRAPH Asia 2024 Conference Papers*. 1–11.
- Yuyao Niu and Marc Casas. 2025. BerryBees: Breadth first search by bit-tensor-cores. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 339–354.
- Kerui Ren, Lihai Jiang, Tao Lu, Mulin Yu, Linning Xu, Zhangkai Ni, and Bo Dai. 2024. Octree-gs: Towards consistent real-time rendering with lod-structured 3d gaussians. *arXiv preprint arXiv:2403.17898* (2024).
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
- Xinze Wang, Ran Yi, and Lizhuang Ma. 2024. AdR-Gaussian: Accelerating Gaussian Splatting with Adaptive Radius. In *SIGGRAPH Asia 2024 Conference Papers*. 1–10.
- Vickie Ye, Ruilong Li, Justin Kerr, Matias Turkulainen, Brent Yi, Zhuoyang Pan, Otto Seiskari, Jianbo Ye, Jeffrey Hu, Matthew Tancik, et al. 2025. gsplat: An open-source library for Gaussian splatting. *Journal of Machine Learning Research* 26, 34 (2025), 1–17.
- Xiaoyu Zhou, Zhiwei Lin, Xiaojun Shan, Yongtao Wang, Deqing Sun, and Ming-Hsuan Yang. 2024. Drivinggaussian: Composite gaussian splatting for surrounding dynamic autonomous driving scenes. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 21634–21643.
- Siting Zhu, Guangming Wang, Xin Kong, Dezhi Kong, and Hesheng Wang. 2024. 3d gaussian splatting in robotics: A survey. *arXiv preprint arXiv:2410.12262* (2024).

A Derivation and Proof of the Upper Bound of Rounding Error on TC-GS

A.1 Floating-Point Numbers

In computing, floating-point numbers arithmetic is arithmetic on subset of real numbers, which can be represented with a finite amount of memory. A floating-point number r is formed by a significand s , base b and exponent e :

$$r = s \cdot b^e. \quad (22)$$

In this paper, we only consider the binary formats, whose base b is 2.

In most modern computing systems, a floating-point number is consists of 3 parts:

- **Sign bit:** Represents the sign of the number. A sign bit of 0 indicates a positive number, and 1 indicates a negative number.
- **Exponent:** Represents the scale of the number. The exponent is stored with an offset.
- **Significand:** Represents the mantissa of the floating-point number. The number of bits of significand is the precision.

For example, if the significand has m bits, the real value of the floating-point number is:

$$(-1)^{\text{sign}}(1.b_{m-1}b_{m-2}\dots b_0)_2 \times 2^{\text{exponent} - \text{offset}}, \quad (23)$$

where $b_{m-1}b_{m-2}\dots b_0$ is the significand bits. Table 7 shows 3 floating-point formats in this paper. FP32 is also known as single-precision floating-point format, and FP16 is called half-precision floating-point format.

Format	Sign	Exponents	Significand	Machine Epsilon
FP16	1bit	5bits	10bits	$\epsilon_H = 9.77 \times 10^{-4}$
TF32	1bit	8bits	10bits	$\epsilon_H = 9.77 \times 10^{-4}$
FP32	1bit	8bits	23bits	$\epsilon_F = 1.19 \times 10^{-7}$

Table 7. 3 binary floating-point formats in this paper.

A.2 Machine Epsilon

Rounding is a procedure for representation of a real number r ($r \neq 0$) into a floating-point format as $R(r)$, where R is the rounding function. The **relative error** ϵ of the rounding procedure is defined as:

$$\epsilon(r) = \frac{|R(r) - r|}{|r|}. \quad (24)$$

For a fixed floating-point system and a rounding procedure, **machine epsilon** ϵ_{mach} is an upper bound of relative error if no underflow or overflow occurs:

$$|R(r) - r| \leq \epsilon_{\text{mach}} |r|. \quad (25)$$

PROPOSITION A.1. If ϵ_{mach} is a machine epsilon of a floating-point format, the rounded value of a real number r satisfies:

$$R(r) = r + O(\epsilon_{\text{mach}}). \quad (26)$$

PROOF. Since the ϵ_{mach} does not depend on r , then Inequality (25) shows $|R(r) - r| = O(\epsilon_{\text{mach}})$. Therefore, $R(r) - r = O(\epsilon_{\text{mach}})$.

PROPOSITION A.2. If a binary floating-point format has m significand bits, $\varepsilon_{mach} = 2^{-m}$ is its machine epsilon.

PROOF. Considering $|r| \in [2^e, 2^{e+1}]$ is a real number which does not overflow or underflow in this format, it will lies between two adjacent floating-point numbers, r_b and r_u :

$$r_b \leq |r| < r_u. \quad (27)$$

Suppose $x_b = (1.b_{m-1}b_{m-2}\dots b_0)_2 \times 2^e$, then the adjacent r_u is:

$$r_u = (1.b_{m-1}b_{m-2}\dots b_0)_2 + (0.00\cdots 1)_2 \times 2^e = r_b + 2^{e-m}. \quad (28)$$

Since $R(r)$ is either r_u or r_b ,

$$|R(r) - r| \leq |r_u - r_b| = 2^{e-m} = 2^e \cdot 2^{-m} \leq 2^{-m}|r|. \quad (29)$$

Therefore, 2^{-m} is a machine epsilon of the floating-point format.

From Proposition A.2, we can conclude that the machine epsilon is only depend on the number of significand bits.

In this paper, we represent the machine epsilon of TF32 and FP16 as $\varepsilon_H = 9.77 \times 10^{-4}$, and the machine epsilon of FP32 as $\varepsilon_F = 1.19 \times 10^{-7}$. Meanwhile, the rounding process of FP16 and TF32 is denoted as R_H , while the rounding process of FP32 is represented as R_F .

Furthermore, if the format is fixed, the rounding error is only depend on the absolute value $|r|$. The larger absolute value contribute the larger rounding error. The core idea of G2L in Section 4.3 is reducing the absolute value during the computing process.

A.3 Rounding Error of Dot Product with Tensor Cores

Matrix multiplication can be interpreted as the dot product of vectors at each element, as discussed in Section 4.2. Therefore, to analyze the rounding error in matrix multiplication, it suffices to examine the error in the dot product:

$$\beta = \mathbf{u}^T \mathbf{v} = \sum_{i=1}^n u_i v_i, \quad (30)$$

where n is the length of the vectors.

The dot product can be decomposed as a multiplication step and a accumulation step. On Tensor Cores, the multiplication step are performed on half-precision format (FP16 or TF32) and the accumulation step can be executed on FP32 format.

We omit the detailed analysis on terms of small ε_H^2 and ε_F and represent them as $O(\varepsilon_H^2)$ and $O(\varepsilon_F)$.

PROPOSITION A.3. On Tensor Cores, the absolute error of $\beta = \mathbf{u}^T \mathbf{v}$ satisfies

$$|\hat{\beta} - \beta| \leq 2\varepsilon_H \sum_{i=1}^n |u_i v_i| + O(\varepsilon_H^2 + \varepsilon_F), \quad (31)$$

where $\hat{\beta}$ is the computed value of β with Tensor Cores.

PROOF. At first, we consider product of u_i and v_i . Before multiplication, these two real number are stored in half-precision format as $R_H(u_i)$ and $R_H(v_i)$ satisfying

$$|R_H(u_i)| \leq |R_H(u_i) - u_i| + |u_i| \leq |u_i|\varepsilon_H + |u_i| = (1 + \varepsilon_H)|u_i| \quad (32)$$

and

$$|R_H(v_i)| \leq (1 + \varepsilon_H)|v_i|. \quad (33)$$

Therefore,

$$|R_H(u_i)R_H(v_i)| \leq (1 + \varepsilon_H)^2(|u_i||v_i|) = (1 + 2\varepsilon_H)|u_i v_i| + O(\varepsilon_H^2). \quad (34)$$

The value product are stored in FP32 format for next accumulation step on Tensor Cores:

$$\begin{aligned} |R_F(R_H(u_i)R_H(v_i))| &\leq (1 + \varepsilon_F)|R_H(u_i)R_H(v_i)| \\ &\leq (1 + \varepsilon_F)((1 + 2\varepsilon_H)|u_i v_i| + O(\varepsilon_H^2)) \\ &\leq (1 + 2\varepsilon_H)|u_i v_i| + O(\varepsilon_H^2) + O(\varepsilon_F). \end{aligned} \quad (35)$$

Notice that the rounding function does not change the sign of the number. Therefore, relative error of the product is:

$$\begin{aligned} \varepsilon(u_i v_i) &= \frac{|u_i v_i - R_F(R_H(u_i)R_H(v_i))|}{|u_i v_i|} \\ &= \frac{||u_i v_i| - |R_F(R_H(u_i)R_H(v_i))||}{|u_i v_i|} \\ &\leq 2\varepsilon_H + O(\varepsilon_H^2) + O(\varepsilon_F). \end{aligned} \quad (36)$$

To simply the proof, we represent a_i as the accumulator on Tensor Cores:

$$a_i = R_F(R_H(u_i)R_H(v_i)) = u_i v_i (1 \pm \varepsilon(u_i v_i)) \quad (37)$$

We assume that the addition is performed from left to right. Since the sum of each two values are stored in FP32 as a new accumulator, the computed cumulative sum s_i of $\{a_i\}_{i=1}^n$ is:

$$s_1 = R_F(a_1) = a_1, \quad s_i = R_F(s_{i-1} + a_i). \quad (38)$$

Therefore, the computed value of β on Tensor Cores is:

$$\hat{\beta} = R_F(R_F(\dots R_F(R_F(a_1 + a_2) + a_3) \dots) + a_n) = s_n. \quad (39)$$

Next, we will prove that $s_i = \sum_{j=1}^i a_j + O(\varepsilon_F)$ inductively:

- (1) It is obvious that $s_1 = a_1 = a_1 + O(\varepsilon_F)$;
- (2) For a positive integer $i < n$, we assume that $s_i = \sum_{j=1}^i a_j + O(\varepsilon_F)$. According to Proposition A.1,

$$s_{i+1} = R_F(s_i + a_{i+1}) = s_i + a_{i+1} + O(\varepsilon_F) = \sum_{j=1}^{i+1} a_j + O(\varepsilon_F). \quad (40)$$

According to Equation (37),

$$\hat{\beta} = s_n = \sum_{i=1}^n a_i + O(\varepsilon_H) = \sum_{i=1}^n (u_i v_i \pm \varepsilon(u_i v_i)) + O(\varepsilon_F). \quad (41)$$

To sum up, the absolute error of β satisfies

$$\begin{aligned} |\hat{\beta} - \beta| &= \left| \sum_{i=1}^n u_i v_i (1 \pm \varepsilon(u_i v_i)) + O(\varepsilon_F) - \sum_{i=1}^n u_i v_i \right| \\ &= \left| \sum_{i=1}^n \pm u_i v_i \varepsilon(u_i v_i) + O(\varepsilon_F) \right| \\ &= \left| \sum_{i=1}^n \pm u_i v_i (2\varepsilon_H + O(\varepsilon_H^2) + O(\varepsilon_F)) + O(\varepsilon_F) \right| \\ &= \left| \sum_{i=1}^n \pm 2u_i v_i \varepsilon_H + O(\varepsilon_H^2) + O(\varepsilon_F) \right| \\ &\leq 2\varepsilon_H \sum_{i=1}^n |u_i v_i| + O(\varepsilon_H^2 + \varepsilon_F). \end{aligned} \quad (42)$$

The proof of Proposition A.3 is completed here.

A.4 Absolute Error of β on Global Coordinates

This section discusses the absolute of exponent β on global coordinates.

The absolute error of β is dependent on v , and v is depend on the parameters of projected Gaussian G' . Therefore, we discusses the range of Gaussians' parameters at first.

We represent the projected Gaussian as $G' = (\mu', \Sigma', o, c)$ with $\mu' = (\mu'_x, \mu'_y)$ and symmetric $\Sigma'^{-1} = \begin{bmatrix} \sigma_{11} & \sigma_{12} \\ \sigma_{21} & \sigma_{22} \end{bmatrix}$ satisfying $\sigma_{12} = \sigma_{21}$.

The alpha value at pixel $p = (p_x, p_y)$ is $\alpha = e^\beta = e^{u^T v}$ as Section 4.2 discussed.

The width and height of rendered image are represented as w and h respectively.

Means μ' : To simplify the analysis, we only consider Gaussians whose center within the screen space. Therefore, $\mu' \in [0, w] \times [0, h]$.

Opacity o : The fragment with alpha value below $\frac{1}{255}$ will be culled. Since $\alpha \leq o$ since Σ'^{-1} is semi-definite, $o \in [\frac{1}{255}, 1]$.

Inverse Covariance Σ'^{-1} : The covariance Σ' identifies the shape of the ellipse on the screen space. If λ_1 and λ_2 are the two eigenvalues of Σ' , then length of two semi-axes of the ellipse are λ_1 and λ_2 .

We assume the length of semi-axis is larger than 0.5, otherwise the Gaussian will not be rendered:

$$\lambda_1, \lambda_2 \geq 0.5. \quad (43)$$

Then the eigenvalues of the inverse covariance Σ'^{-1} satisfies:

$$0 < \lambda_1^{-1} + \lambda_2^{-1} \leq 4, \quad 0 < \lambda_1^{-1} \lambda_2^{-1} \leq 4. \quad (44)$$

Consider the characteristic polynomial of Σ'^{-1} :

$$f(\lambda^{-1}) = |\lambda^{-1}I - \Sigma'| = \lambda^{-2} - (\sigma_{11} + \sigma_{22})\lambda^{-1} + \sigma_{11}\sigma_{22} - \sigma_{12}^2. \quad (45)$$

These two eigenvalues satisfy $f(\lambda^{-1}) = 0$. Applying the Vieta's Formulas,

$$0 < \sigma_{11} + \sigma_{22} \leq 4, \quad 0 < \sigma_{11}\sigma_{22} - \sigma_{12}^2 \leq 4. \quad (46)$$

Then,

$$\sigma_{12}^2 < \sigma_{11}\sigma_{22} \leq \left(\frac{\sigma_{11} + \sigma_{22}}{2} \right)^2 \leq 4. \quad (47)$$

Therefore,

$$0 < \sigma_{11} < 4, \quad -2 \leq \sigma_{12} \leq 2, \quad 0 < \sigma_{22} < 4. \quad (48)$$

According to Proposition A.3, the absolute error of β on *Frag2Mat* is:

$$|\hat{\beta} - \beta| \leq 2\varepsilon_H \sum_{i=0}^5 |u_i v_i| + O(\varepsilon_H^2 + \varepsilon_F). \quad (49)$$

Applying the global coordinate, the coordinate of pixel satisfies $p = (p_x, p_y) \in [0, w] \times [0, h]$.

Then,

$$|u_0 v_0| = \left| \ln(o) - \frac{1}{2} \sigma_{11} \mu_x^2 - \sigma_{12} \mu_x \mu_y - \sigma_{22} \mu_y^2 \right| = O(w^2 + wh + h^2), \quad (50)$$

$$|u_1 v_1| = |p_x (\sigma_{11} \mu_x + \sigma_{12} \mu_y)| = O(w^2 + wh), \quad (51)$$

$$|u_2 v_2| = |p_y (\sigma_{12} \mu_x + \sigma_{22} \mu_y)| = O(wh + h^2), \quad (52)$$

$$|u_3 v_3| = \left| p_x^2 \left(-\frac{1}{2} \sigma_{11} \right) \right| = O(w^2), \quad (53)$$

$$|u_4 v_4| = |p_x p_y (-\sigma_{12})| = O(wh), \quad (54)$$

$$|u_5 v_5| = \left| p_y^2 \left(-\frac{1}{2} \sigma_{22} \right) \right| = O(h^2). \quad (55)$$

Therefore, the absolute error of β satisfies quadratic

$$|\hat{\beta} - \beta| = O(w^2 + wh + h^2) \varepsilon_H + O(\varepsilon_H^2 + \varepsilon_F). \quad (56)$$

A.5 Absolute Error of β on Local Coordinates

This section discusses the absolute of exponent β on local coordinates.

At first, we discuss the range of Gaussians' parameters as Section A.4. In this section, $p(\mathcal{T})$ is the center of tile \mathcal{T} , and the size of each tile is 16×16 .

Means $\Delta\mu'$: Since $\mu', p(\mathcal{T}) \in [0, w] \times [0, h]$, $\Delta\mu' = \mu' - p(\mathcal{T}) \in [-w, w] \times [-h, h]$.

Opacity o : The opacity is the same as Gaussians on global coordinates. $o \in [\frac{1}{255}, 1]$.

Inverse Covariance Σ'^{-1} : The covariance is the same as Gaussians on global coordinates: $0 < \sigma_{11}, \sigma_{12} < 4, -2 \leq \sigma_{12} \leq 2$.

All pixels on local coordinates is within a tile, then $\Delta p = (\Delta p_x, \Delta p_y) \in [-8, 8]^2$.

Therefore,

$$|u_1 v_1| = |\Delta p_x (\sigma_{11} \Delta \mu_x + \sigma_{12} \Delta \mu_y)| = O(w + h), \quad (57)$$

$$|u_2 v_2| = |\Delta p_y (\sigma_{12} \Delta \mu_x + \sigma_{22} \Delta \mu_y)| = O(w + h), \quad (58)$$

$$|u_3 v_3| = \left| \Delta p_x^2 \left(-\frac{1}{2} \sigma_{11} \right) \right| = O(1), \quad (59)$$

$$|u_4 v_4| = |\Delta p_x \Delta p_y (-\sigma_{12})| = O(1), \quad (60)$$

$$|u_5 v_5| = \left| \Delta p_y^2 \left(-\frac{1}{2} \sigma_{22} \right) \right| = O(1). \quad (61)$$

According to Proposition A.3,

$$|\hat{\beta} - \beta| \leq (2|u_0 v_0| + O(w + h)) \varepsilon_H + O(\varepsilon_H^2 + \varepsilon_F). \quad (62)$$

We only care the precision of fragments which are not culled, then $-\ln(255) \leq \beta < 0$. Since $\beta = \sum_{i=0}^5 u_i v_i$, then $u_0 v_0 = \beta - \sum_{i=1}^5 u_i v_i$. Therefore,

$$|u_0 v_0| \leq |\beta| + \sum_{i=1}^5 |u_i v_i| = O(w + h). \quad (63)$$

In conclusion, the absolute error of β satisfies

$$|\hat{\beta} - \beta| \leq O(w + h) \varepsilon_H + O(\varepsilon_H^2 + \varepsilon_F). \quad (64)$$

B Notations

To clearly present the various parameters and their definitions involved in this paper, the following parameter table lists each symbol

along with its corresponding meaning, facilitating readers' understanding of the subsequent analysis and computational processes.

Notation	Meaning	Notation	Meaning
\mathcal{G}	Set of all 3D Gaussian	G	3D Gaussian
P	Number of 3D Gaussians	μ	Mean of 3D Gaussian
Σ	Covariance of Covariance	o	Opacity of 3D Gaussian
c	Color of 3D Gaussian	W	Matrix projecting 3D position into 2D plane
J	Jacobian of projection	$\mu' = (\mu'_x, \mu'_y)$	Mean of projected Gaussian
$\Sigma' = \begin{bmatrix} \sigma_{11} & \sigma_{12} \\ \sigma_{21} & \sigma_{22} \end{bmatrix}^{-1}$	Covariance of projected Gaussian	G'	Projected Gaussian
\mathcal{G}'	Set of projected Gaussians	$p = (p_x, p_y)$	Pixel
\mathcal{T}	Tile, containing pixels	\mathcal{S}	Screen space, containing all tiles
C	Rendered pixel color	α	Alpha value of fragments.
T	Transmissivity of fragments.	$n(\mathcal{T})$	Number of splats on tile \mathcal{T}
N	Number of splats.	\mathcal{E}	Complexity of the pipeline in 3DGS
w	Width of the screen	h	Height of the screen
\mathcal{E}_p	Complexity of preprocessing	\mathcal{E}_s	Complexity of sorting
\mathcal{E}_a	Complexity of alpha-blending	k_α	Computation amount in alpha computation
k_{cull}	Computation amount in culling	k_{blend}	Computation amount in blending
f_{cull}	Number of culled fragments	f_{blend}	Number of blended (shaded) fragments.
f_{skip}	Number of skipped fragments	C	Total computation amount in alpha-blending.
\mathbf{u}	Pixel vector	\mathbf{v}	Gaussian vector
(p_x, p_y)	Coordinates of pixel p	β	$\ln(\alpha)$, the exponent
\mathbf{U}	Pixel matrix	\mathbf{V}	Gaussian matrix
\mathbf{B}	Exponent matrix	$p(\mathcal{T})$	Center of tile \mathcal{T}
$\Delta p = (\Delta p_x, \Delta p_y)$.	Pixel on local coordinates	$\Delta \mu' = (\Delta \mu'_x, \Delta \mu'_y)$	Projected mean on local coordinates
ϵ_H	Machine epsilon of FP16 or TF32	ϵ_F	Machine epsilon of FP32
R_H	Rounding function of FP16 or TF32	R_F	Rounding function of FP32
$\hat{\beta}$	Computed value of β with Tensor Cores	λ_1, λ_2	Two eigenvalues of Σ'

Table 8. Notations in this paper.