VkSplat: High-Performance 3DGS Training in Vulkan Compute

Jingxiang Chen[1], Mohamed Ibrahim[1], and Yang Liu[1] †

[1]Huawei Canada

Abstract

We present VkSplat, a high-performance, cross-vendor 3D Gaussian Splatting (3DGS) training pipeline implemented fully in Vulkan compute, addressing performance and compatibility limitation of existing training pipelines. With various optimizations, we achieve 3.3× speed and 33% VRAM reduction over CUDA+PyTorch baseline, maintaining quality, and demonstrating compatibility across GPU vendors. To the best of our knowledge, this is the first fully-Vulkan-based 3DGS training pipeline that achieves state-of-the-art performance. Code: https://github.com/harry7557558/vksplat

CCS Concepts

• ***Computing methodologies*** → ***Rasterization;*** *Graphics processors; Reconstruction;* • ***Software and its engineering*** → ***Software performance;***

1. Introduction

Since its introduction [KKLD23], 3D Gaussian Splatting (3DGS) has been widely adopted for fast high-quality novel view synthesis. Despite its wide application, training time and memory usage remain barriers for practical use, especially for large scenes and applications requiring timely training. While novel algorithms have been introduced to improve 3DGS quality and capability over wide range of applications, focus on efficiency is often secondary. Existing 3DGS training implementations [KKLD23] [YLK*25] are often sub-optimal in performance, as well as CUDA ecosystem blocking deployment across hardware vendors.

To address these limitations, we present VkSplat, a high-performance, cross-vendor 3DGS training pipeline implemented fully in Vulkan compute. We introduce complete tile culling using a parallelizable scan-line formulation, per-Gaussian and tensor-based rasterization backward eliminating pixel-level atomic contention, and fully-fused backward projection and Adam optimizer in a single pass. Our end-to-end Vulkan implementation outperforms state-of-the-art CUDA pipelines, achieving 3.3× speed up over GSplat [YLK*25], 33% VRAM reduction, and identical image quality, as well as being compatible across GPU vendors.

2. Related works

3D Gaussian Splatting (3DGS) was introduced by [KKLD23], which involves representing a 3D scene using a set of Gaussian ellipsoids, typically trained through differentiable rendering using gradient descent. Each Gaussian in world-space is parameterized by mean μ interpreted as position, a covariance matrix Σ commonly parameterized by per-axis log scales and rotation quaternion, an opacity value typically in logit space, and color parameters as spherical harmonics (SH) coefficients typically with degree 3. Most of the existing works [KKLD23] [YLK*25] implement 3DGS training in CUDA and run on NVIDIA GPUs, typically using a tile-based rasterization pipeline consisting of projection, tile binning and sorting, rasterization, backward passes for rasterization and projection, optimizer, and often densification stages.

Training a 3DGS scene can take tens of minutes to hours depending on scene complexity and number of Gaussians. Training speed can be improved by reducing the number of false-positive intersections and therefore accelerate sorting and rasterization, like introduced in [RSP*24] [HTL*25] [LWC*25]. The rasterization backward step is often a bottleneck of 3DGS rasterization, and optimized implementations have been proposed by [MGK*24] [LWC*25]. Adam optimizer is also a large performance bottleneck with large number of Gaussians, which is addressed by [MGK*24].

In addition to training time, reliance on CUDA and PyTorch in existing implementations have limited 3DGS training to NVIDIA GPUs. Attempts have been made to use cross-vendor API like Vulkan for rasterization: [Par24a] [YH25] use the graphics pipeline for rasterization, but no end-to-end training was done. [MLH*25] claims to have a Slang+Vulkan differentiable rendering pipeline, but PyTorch is used for training. To the best of our knowledge, our work is the first end-to-end 3DGS training pipeline implemented entirely in Vulkan, free of any NVIDIA-specific extensions or dependencies, while delivering performance that surpasses CUDA-based baselines by a substantial margin.

† Chairman Eurographics Publications Board

DOI: 10.2312/egs.20261002

3. Pipeline overview

The design of VkSplat focuses on high performance, cross-vendor training, and fidelity consistent with academic baselines. For the GPGPU API, we choose Vulkan, a modern framework that offers high performance and support for mainstream GPU vendors. We based our optimizations on Slang-Gaussian-Rasterization [Kop24], a differentiable 3DGS renderer based on Slang shading language. A strength of Slang is its ability to target multiple backends. Although the original implementation was designed for CUDA and PyTorch, we successfully run it with a Vulkan backend and preserve the flexibility to support additional backends in the future.

4. Key technical contributions

In addition to introducing a cross-vendor, minimal-dependency Vulkan-based 3DGS training pipeline, numerous optimizations are made to push performance exceeding CUDA baselines while maintaining fidelity. Notable contributions are summarized below.

4.1. Complete tile culling using scan-line intersection

In tile-based rendering, a list of Gaussian-tile intersections is generated and traversed. Existing implementations produce false-positive intersections: [KKLD23] determines intersections based on distance between tile and Gaussian center with a conservative radius; [RSP*24] uses tiles overlapping with tight bounding boxes of Gaussians, which still produces false positives; [RWFL25] introduces a Compact Box approach, which produces false negatives. False-positive intersections drop performance in sorting and rasterization, and false-negative intersections lead to rendering inaccuracy. [RSP*24] culls false positive intersections in a separate pass, but this introduces computation time and VRAM usage overhead.

In our implementation, we compute number of the intersecting tiles per Gaussian in the projection-forward pass, and use a separate pass to fill the buffer with tile-depth pairs. Inspired by [HTL*25] [LWC*25], we compute exact intersections using a scan-line approach. Given a Gaussian represented as an ellipse containing opacity above threshold and an interval for one axis, we compute closed-form interval on other axis that intersect the ellipse, allowing efficient intersection counting and traversal, as shown in Figure 1.

4.2. Rasterization backward with adaptive scheduling

In existing implementations [KKLD23] [YLK*25], rasterization backward launches threads per pixel, which atomically adds gradient to each Gaussian from each pixel, resulting in high contention from atomics and subgroup reduction. To address this, we switch between two rasterization backward implementations.

Our first implementation was inspired by [MGK*24], which parallelizes threads over Gaussians instead of pixels to reduce atomic contention. In our implementation, instead of using Gaussian batches fixed to warp size, we launch one thread block per tile and dynamically adjust Gaussian batch size based on number of Gaussians binned to the tile. For a tile with P pixels and N Gaussians, with Gaussian batch size S, the latency is approximately proportional to the product of number of batches $(P+S)/P$

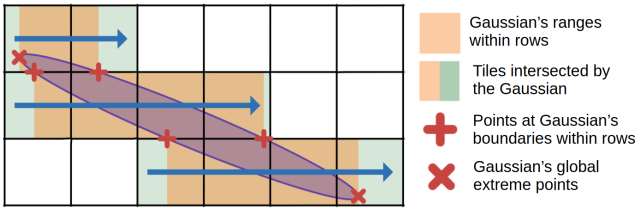

Figure 1: *Given a screen-space Gaussian ellipse, we first pick the shorter dimension (vertical in the case). For each row (or column) of tiles, we first find the ellipse's range of coordinates within the row, which is bounded by either the boundary or global extreme points of the ellipse. We implement the closed-form solution as a highly optimized branchless function. By rounding down the minimum coordinate and rounding up the maximum coordinate toward tile boundary, we are able to efficiently count or iterate through tiles overlapped by the ellipse without false positive or negative.*

and number of Gaussians traversed per batch $(N+S)/S$, minimized by $S=\sqrt{NP}$. We choose S by rounding up $\sqrt{NP}$ to a multiple of subgroup size, capped to 128 that balances performance and hardware occupancy, which empirically produces the lowest latency.

Our second implementation aims to improve thread utilization and minimize divergence. For a Gaussian batch, we first run a forward pass parallelized over pixels, compute transmittance and its derivative with respect to Gaussian parameters for each Gaussian-pixel pair, and store them in shared memory. Then, we run a backward pass paralellized over Gaussians, fetch pre-computed transmittance and derivatives from shared memory, and compute accumulated gradient. We parameterize projected Gaussians following [LDC*25], which simplifies opacity computation into matrix multiplication and allows efficient parallel computation.

In our benchmark, we notice the fastest implementation depends on training configuration. For example, on NVIDIA RTX 3090, the second implementation is 20%-30% faster than the first on Mip-NeRF 360 bicycle scene, but the first one is faster on garden scene. To automatically find the fastest implementation, we use a Thompson sampling scheduler, which stores a latency belief of each implementation as distributions, randomly selects an implementation with higher probabilities for faster ones, and updates belief based on measured latency. This improves performance over using a fixed implementation when a faster implementation is available, with minimal overhead when the fixed implementation is already faster.

4.3. Fused projection backward and optimizer

Most existing 3DGS training implementations use Adam optimizer from PyTorch, which is not fused by default and incur unnecessary memory footprint. The largest Gaussian attribute, SH coefficients, involves separate learning rates for degree 0 and the rest of parameters, and concatenating large tensors in existing implementations [KKLD23] [YLK*25] unnecessarily increase memory footprint. [MGK*24] addresses this issue by avoiding concatenation, but the gradients of Gaussian parameters are still stored. Storing log and logit mapping of scales and opacities also introduces memory footprint. In VkSplat, we fuse projection backward and Adam optimizer into a single kernel and transform value and gradient of

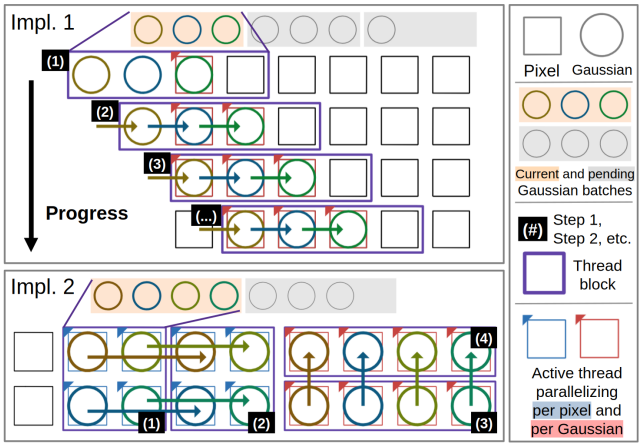

Figure 2: *Two rasterization backward implementations. The first implementation is similar to [MGK*24], except using one thread block per tile in a single pass, and dynamically adjusting Gaussian batch size based on number of Gaussians binned to the tile. The second implementation first performs a forward pass with per-pixel parallelization and stores transmittance and its derivative in shared memory, then performs gradient accumulation with per-Gaussian parallelization, utilizing all threads with minimum divergence.*

scales and opacities to log and logit space within the optimizer, which completely eliminates aforementioned memory footprint.

4.4. 32-bit tile-depth pair sorting

Sorting implementations in Vulkan [Par24b] are often slower than CUDA's heavily engineered built-in sorting. To address this limitation, we use 32-bit sorting keys that involve tile ID in higher bits and depth in lower bits. We map depth using $z \to (2z+1)/(z+1)$, which transforms positive real values into values between 1 and 2 (exclusive), in a way that only the lower 23 mantissa bits are different when represented in FP32. Instead of following popular CUDA implementations that use 64-bit sorting keys with tile ID in higher 32 bits and depth in lower 32 bits, we use a 32-bit sorting key with tile ID in minimum number of higher bits, and place higher bits of floating point mantissa in lower bits. On images with resolution up to 1080p with 16×16 tile size, this leads to at most 14 bits for tile ID and at least 18 bits for depth. We did not notice any difference in quality metrics compared to using 64-bit sorting.

4.5. Fully-fused loss gradient evaluation

3DGS training conventionally use a weighted sum of L_1 and SSIM losses. Existing 3DGS implementations [KKLD23] [YLK*25] explicitly evaluate losses in PyTorch and use fused-ssim [MGK*24] for SSIM loss, which operates on channel-first memory layout that requires conversion from channel-last 3DGS renders. In our implementation, we use a fully-fused kernel that directly computes the gradient of weighted sum of L_1 and SSIM losses in a single pass, without intermediate reduction or memory layout conversion. Also, we store the reference image in $4\times$UINT8 RGBA pixels, without conversion into FP32 that introduce additional memory overhead. Support for alpha masking is also fused with near-zero overhead.

Table 1: *Metrics of GSplat [YLK*25] and our VkSplat with uncertainty. Each implements default [KKLD23] and MCMC [KRS*24] densification strategies. Number of Gaussians is in millions.*

Metric	GSplat Default	VkSplat Default	GSplat MCMC	VkSplat MCMC
PSNR	29.[19-25]	29.2[0-7]	29.4[3-5]	29.[39-45]
SSIM	0.87[8-9]	0.87[8-9]	0.881	0.881
LPIPS	0.124	0.12[4-5]	0.1[29-30]	0.130
NumGS	3.0[6-8]	3.0[2-6]	1.00	1.00

4.6. Additional features

We implement initialization and two densification strategies, default [KKLD23] and MCMC [KRS*24], closely following [YLK*25] but in as few shader launches as possible. Degree 3 SH include 48 FP32 coefficients per Gaussian, which we split into 12 128-bit values in a column-based format aligned with subgroup size, offering improved memory coalescing. Likewise, we also fuse scale and opacity into a single 128-bit value.

5. Results

We evaluate VkSplat on 7 permissively released scenes from the Mip-NeRF 360 dataset [BMV*22] and report mean quantitative results across datasets. Following [YLK*25], we downscale images with the highest resolution and save in lossless 8-bit PNG format. We use image resolution, validation image selection, and training hyperparameters consistent with default of [YLK*25].

5.1. Quality

Since 3DGS training is stochastic, we trained each scene with each method 5 times and reported metrics (PSNR, SSIM, LPIPS evaluated consistent with [YLK*25], and number of Gaussians) with 90% confidence interval. Our implementation produces quality consistent with baseline, as shown in Table 1.

5.2. Resource usage

We benchmarked our implementation against the baseline [YLK*25] (commit `b60e917`) on an NVIDIA RTX 3090 GPU. We queried each stage of the 3DGS training and reported corresponding timing, averaged across the 7 scenes, presented in Table 2. As can be seen, we are over $3.3\times$ faster than the baseline, using around 33% less VRAM. We are also faster than the baseline in every single step of the pipeline. Our complete tile culling accelerates rasterization, added on top of our optimized rasterization backward. Fused projection backward and optimizer and efficient memory layout of SH coefficients speed up projection forward, projection backward, and optimizer by multiple times. With fused kernels, loss gradient evaluation and densification is also multiple times faster. GSplat has large time unaccounted by queries; profiling shows this is largely backward of SH tensor concatenation and small kernel launches managed by PyTorch.

Table 2: *Resource usage of VkSplat and baseline, including overall time/VRAM and timing breakdown of each step in seconds. In VkSplat, projection backward and optimizer are fused into one step.*

Metric	GSplat Default	VkSplat Default	GSplat MCMC	VkSplat MCMC
Total Time [s]	1384	412	995	285
Total VRAM [GiB]	4.56	3.01	1.37	0.93
Projection Fwd	94	19	40	8
Tiling/Sorting	42	25	41	27
Rasterization Fwd	69	25	71	25
Loss	103	32	110	32
Rasterization Bwd	246	130	268	120
Proj Bwd + Optim	61+398	131	33+172	46
Densify	31	5	61	3
Unaccounted	341	43	200	23

5.3. Cross-vendor compatibility

To demonstrate our capability to perform on GPUs without CUDA support, we trained bicycle scene with default densification on both NVIDIA RTX 3090 and AMD Radeon RX 7800 XT, on respectively Windows 11 and Ubuntu 24.04. For both GPUs, we produced consistent quality metrics, VRAM usage, and number of Gaussians. Training took 575 seconds on RTX 3090 and 1201 seconds on Radeon RX 7800 XT. We notice the step with the largest speed differences is loss computation (24s vs 303s) since transferring reference image from host to device appears to be magnitudes slower on AMD. Memory-bound tasks like projection backward/optimizer and projection forward are over twice as slow, but the gap is less for compute-bound tasks like rasterization backward. We believe the gap can be reduced with hardware-specific optimizations.

6. Discussion and limitations

While we mainly tested with Vulkan, due to the versability of Slang, this work can be extended to other backends like CUDA, Metal, DirectX, WebGPU, etc., offering compatibility across hardware and platforms. Since the same 3DGS pipeline components (tile culling, rasterization, optimizer) are shared with other splatting variants, our optimizations can be easily extended to other splatting pipelines and benefit wide range of applications. However, our implementation current lacks practical features for real-world datasets such as exposure correction, depth/normal supervision, batching, multi-GPU training, etc. that are implemented in mature 3DGS trainers [KKLD23] [YLK*25]. Also, we closely followed [YLK*25] in densification [KKLD23] [KRS*24] and currently do not support more efficient densification strategies [MGK*24] [LWC*25] . Additional features and more efficient densification can be implemented into VkSplat with engineering effort.

7. Conclusion

We presented VkSplat, an end-to-end 3DGS training pipeline implemented fully in Vulkan compute, which demonstrates careful GPU optimization yielding substantial performance gains, achieving 3.3× speedup, using 33% less VRAM, being cross-platform, while remaining high-fidelity. We publicly release our code for reproducibility and community benefit.

References

[BMV*22] BARRON J. T., MILDENHALL B., VERBIN D., SRINIVASAN P. P., HEDMAN P.: Mip-nerf 360: Unbounded anti-aliased neural radiance fields. *CVPR* (2022). 3

[HTL*25] HANSON A., TU A., LIN G., SINGLA V., ZWICKER M., GOLDSTEIN T.: Speedy-splat: Fast 3d gaussian splatting with sparse pixels and sparse primitives. In *Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR)* (June 2025), pp. 21537–21546. URL: https://speedysplat.github.io/. 1, 2

[KKLD23] KERBL B., KOPANAS G., LEIMKÜHLER T., DRETTAKIS G.: 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics 42*, 4 (July 2023). URL: https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/. 1, 2, 3, 4

[Kop24] KOPANAS G.: Slang.d gaussian splatting rasterizer, 2024. URL: https://github.com/google/slang-gaussian-rasterization. 2

[KRS*24] KHERADMAND S., REBAIN D., SHARMA G., SUN W., TSENG Y.-C., ISACK H., KAR A., TAGLIASACCHI A., YI K. M.: 3d gaussian splatting as markov chain monte carlo. In *Advances in Neural Information Processing Systems (NeurIPS)* (2024). Spotlight Presentation. 3, 4

[LDC*25] LIAO Z., DING J., CUI S., GONG R., HU B., WANG Y., LI H., ZHANG X., WANG H., FU R.: Tc-gs: A faster gaussian splatting module utilizing tensor cores, 2025. URL: https://arxiv.org/abs/2505.24796, arXiv:2505.24796. 2

[LWC*25] LIAO K., WANG H., CHEN Z., WANG L., TANG Y.: Litegs: A high-performance framework to train 3dgs in subminutes via system and algorithm codesign, 2025. URL: https://arxiv.org/abs/2503.01199, arXiv:2503.01199. 1, 2, 4

[MGK*24] MALLICK S. S., GOEL R., KERBL B., STEINBERGER M., CARRASCO F. V., DE LA TORRE F.: Taming 3dgs: High-quality radiance fields with limited resources. In *SIGGRAPH Asia 2024 Conference Papers* (New York, NY, USA, 2024), SA '24, Association for Computing Machinery. URL: https://doi.org/10.1145/3680528.3687694, doi:10.1145/3680528.3687694. 1, 2, 3, 4

[MLH*25] MÜLLER J. U., LANDSGESELL R. T., HOLLAND L. V., STOTKO P., KLEIN R.: Moment-based 3d gaussian splatting: Resolving volumetric occlusion with order-independent transmittance, 2025. URL: https://arxiv.org/abs/2512.11800, arXiv:2512.11800. 1

[Par24a] PARK J.: vkgs, 2024. URL: https://github.com/jaesung-cs/vkgs. 1

[Par24b] PARK J.: vulkan_radix_sort, 2024. URL: https://github.com/jaesung-cs/vulkan_radix_sort. 3

[RSP*24] RADL L., STEINER M., PARGER M., WEINRAUCH A., KERBL B., STEINBERGER M.: StopThePop: Sorted Gaussian Splatting for View-Consistent Real-time Rendering. *ACM Transactions on Graphics 4*, 43 (2024). 1, 2

[RWFL25] REN S., WEN T., FANG Y., LU B.: Fastgs: Training 3d gaussian splatting in 100 seconds. *arXiv preprint arXiv:2511.04283* (2025). 2

[YH25] YUAN Y., HE Q.: Efficient differentiable hardware rasterization for 3d gaussian splatting. *arXiv preprint arXiv:2505.18764* (2025). 1

[YLK*25] YE V., LI R., KERR J., TURKULAINEN M., YI B., PAN Z., SEISKARI O., YE J., HU J., TANCIK M., KANAZAWA A.: gsplat: An open-source library for gaussian splatting. *Journal of Machine Learning Research 26*, 34 (2025), 1–17. 1, 2, 3, 4