# GauRast: Enhancing GPU Triangle Rasterizers to Accelerate 3D Gaussian Splatting

Sixu Li[1,2], Ben Keller[2], Yingyan (Celine) Lin[1], and Brucek Khailany[3]
[1]*School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA*
[2]*NVIDIA, Santa Clara, CA, USA*
[3]*NVIDIA, Austin, TX, USA*
{sli941, celine.lin}@gatech.edu, {benk, bkhailany}@nvidia.com

***Abstract*—3D intelligence leverages rich 3D features and stands as a promising frontier in AI, with 3D rendering fundamental to many downstream applications. 3D Gaussian Splatting (3DGS), an emerging high-quality 3D rendering method, requires significant computation, making real-time execution on existing GPU-equipped edge devices infeasible. Previous efforts to accelerate 3DGS rely on dedicated accelerators that require substantial integration overhead and hardware costs. This work proposes an acceleration strategy that leverages the similarities between the 3DGS pipeline and the highly optimized conventional graphics pipeline in modern GPUs. Instead of developing a dedicated accelerator, we enhance existing GPU rasterizer hardware to efficiently support 3DGS operations. Our results demonstrate a 23× increase in processing speed and a 24× reduction in energy consumption, with improvements yielding 6× faster end-to-end runtime for the original 3DGS algorithm and 4× for the latest efficiency-improved pipeline, achieving 24 FPS and 46 FPS respectively. These enhancements incur only a minimal area overhead of 0.2% relative to the entire SoC chip area, underscoring the practicality and efficiency of our approach for enabling 3DGS rendering on resource-constrained platforms.**

***Index Terms*—Hardware Architecture, Graphics Processors, 3D Gaussian Splatting**

## I. Introduction

3D intelligence is set to become the next major frontier in AI by leveraging rich 3D features to enhance understanding and interaction within complex environments. As Prof. Fei-Fei Li, co-founder of ImageNet, emphasized, *"...we need spatially intelligent AI that can model the world and reason about objects, places, and interactions in 3D space and time..."* [38]. This underscores the importance of 3D intelligent applications such as autonomous driving [39], robotics [32], and augmented/virtual reality (AR/VR) [4] shown in Fig. 1. A foundational task within 3D intelligence is 3D scene rendering, which provides essential features for downstream AI tasks that require spatial information.

Recently, 3D Gaussian Splatting (3DGS) [14] has emerged as the leading method for 3D scene rendering. As summarized in Table I, unlike previous techniques like triangle meshes [33] and neural radiance fields (NeRF) [20], 3DGS maps 3D scenes onto a set of Gaussian balls, offering superior rendering quality and automated scene reconstruction. Thanks to these advantages, 3DGS has been rapidly adopted in various 3D applications, including autonomous driving [13], [30], [42], robotics [1], [12], [19], and AR/VR [18], [31], [40].


Fig. 1. Representative examples of 3D intelligent applications, including autonomous driving, robotics, and augmented/virtual reality [11], [21], [27].

TABLE I
Comparison of Rendering Methodologies

| | Triangle Mesh [33] | NeRF [20] | 3D Gaussian [14] |
|---|---|---|---|
| Scene Reconstruction | Manual | Automatic | Automatic |
| Rendering Quality | Manually Decided | High | Very High |
| Rendering Speed on GPU [22] | Fast | Slow | Medium |

Despite its robust performance on high-powered ($\geq$ 200 W) GPUs [25], achieving real-time rendering of $\geq$ 30 frames per second (FPS), 3D Gaussian Splatting cannot achieve high framerates on edge computing platforms with GPUs, such as those with power limitations ($\leq$ 10 W) like the NVIDIA Jetson Orin NX [22]. These platforms are increasingly crucial due to the growing demand for 3D processing in mobile and embedded systems [11], [27]. Specifically, 3DGS achieves only 2-5 FPS on these platforms [22] with commonly used real-world, large-scale datasets [3], falling short of the performance requirement for most practical applications. This performance gap poses challenges for deploying advanced 3D intelligence in resource-constrained environments, highlighting the need for hardware acceleration of 3DGS.

Existing efforts to accelerate 3D Gaussian Splatting [16] and similar rendering pipelines [10], [17] typically focus on developing dedicated hardware accelerators. While these specialized units can provide performance benefits, they require exclusive hardware resources and often lead to increased system complexity and cost. Moreover, the integration of dedicated accelerators may not be feasible for all edge platforms, especially those with stringent area and power constraints. However, modern GPUs are already equipped with optimized fixed-function graphic acceleration hardware for triangle meshes. As these fixed-function units also target graphics rendering, their inherent capabilities present an opportunity to leverage


Fig. 2. Visualization of a 3D Gaussian representation: (a) Rendered RGB image depicting the scene with realistic color and detail; and (b) the corresponding Gaussian ball representation of the same scene, showing the underlying 3D structure before rendering. Both images are rendered using the 'Bonsai' scene from the NeRF-360 [3] dataset.

existing resources to accelerate 3DGS rendering without introducing significant additional overhead.

To mitigate the challenges associated with designing dedicated memory systems and software stacks for specialized accelerators [6], we propose an alternative solution: enhancing the existing graphics units—specifically, the rasterizer for triangle meshes—within GPUs to accelerate 3DGS rendering. By leveraging the capabilities of current GPU hardware, we aim to enhance performance without the need for additional dedicated accelerators. We begin with an in-depth profiling of the 3DGS rendering pipeline to identify time-consuming operations and analyze its similarities with operators in the triangle mesh rendering pipeline, which are already supported by the GPU's native graphic hardware units. This analysis reveals opportunities to adapt and enhance these units to efficiently process 3D Gaussian operations.

Building on these insights, we propose and develop an enhanced rasterizer, GauRast, for GPUs that enables the efficient execution of the dominant operator in the 3DGS rendering pipeline while preserving the original capabilities for standard triangle mesh rendering. This design ensures compatibility with existing GPU architectures and minimizes disruptions to conventional workflows. By enhancing existing hardware rather than introducing entirely new components, our approach strikes a balance between performance improvements and resource utilization. Our contributions are as follows:

- We advocate an alternative direction for accelerating 3DGS, the leading algorithm in neural rendering, by enhancing existing GPU units originally dedicated to traditional workloads.
- We detail comprehensive profiling and analysis of the 3DGS rendering pipeline [14] to identify the critical operators requiring acceleration, and demonstrate that the bottleneck operator in the 3DGS rendering pipeline shares similarities with the triangle rendering pipeline, which is already well-accelerated by existing GPU's triangle rasterizer.
- We propose an enhanced rasterizer, named GauRast, to support the dominant operator in the 3DGS rendering pipeline while maintaining its functionality for standard triangle mesh rendering tasks.
- We present a GauRast hardware prototype that achieves a 23× speedup and a 24× improvement in energy efficiency on the target SoC for the dominant operator with the original 3DGS rendering pipeline. This leads to a 6× end-to-end speedup in the original 3DGS rendering pipeline [14] and a 4× end-to-end speedup in the latest efficiency-optimized 3DGS rendering pipeline [9], achieving 24 FPS and 46 FPS, respectively. Notably, GauRast incurs an area overhead of only 21% for the enhanced graphics units, corresponding to 0.2% of the total SoC area.

Taken together, our proposals demonstrate a promising approach to enabling real-time 3DGS rendering on future edge devices.

## II. 3DGS and Its Rendering Bottleneck

### A. 3D Gaussian Splatting Algorithm

**3D Gaussian Scene Representation.** The state-of-the-art (SOTA) method for 3D scene rendering employs a 3D Gaussian-based representation [14] as illustrated in Fig. 2. This approach strikes a balance between high rendering quality and efficient rendering speed, especially on standard desktop-level GPUs. In this representation, objects are modeled as collections of elliptical 3D Gaussian balls. Each Gaussian ball is characterized by a 3D Gaussian probability density function and an associated color vector. The rendering process distributes colors over the regions covered by the Gaussians' probability densities, collectively forming the complete scene.

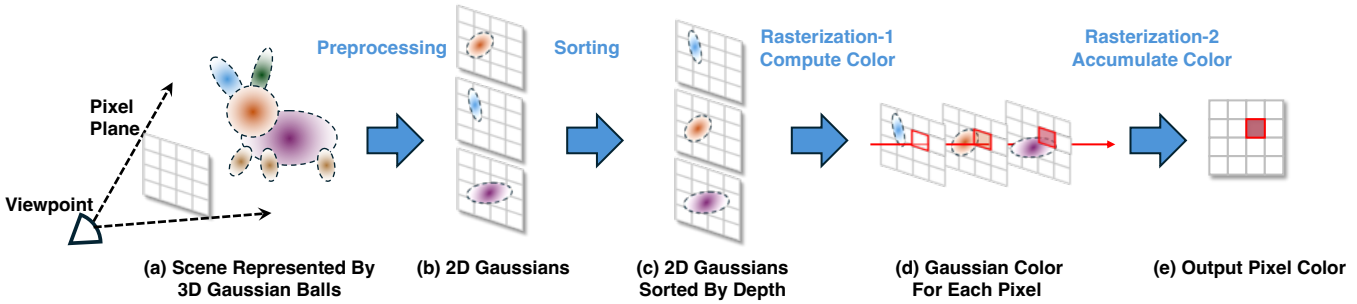


Fig. 3. Overview of the 3DGS pipeline [14]: (a) *Scene representation*: The scene is depicted as 3D Gaussian balls, viewed from a specific viewpoint and projected onto a 2D pixel plane. (b) *Preprocessing*: These 3D Gaussians are projected onto the 2D plane, resulting in 2D Gaussian representations. (c) *Sorting*: 2D Gaussians are ordered by depth to ensure the correct rendering sequence and handle occlusion properly. (d) *Initial rasterization*: Colors for each pixel are calculated based on the Gaussians covering that pixel. (e) *Color accumulation*: Colors are accumulated to produce the final pixel color output.

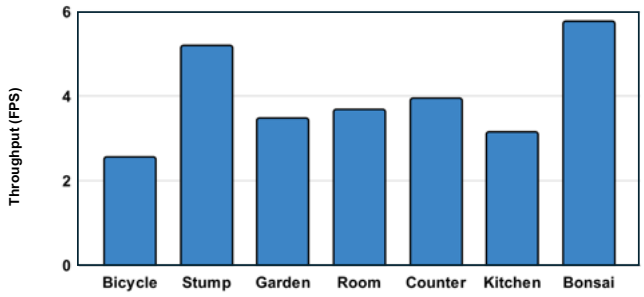


Fig. 4. Throughput achieved by the 3DGS rendering pipeline [14] across all seven scenes from the large-scale, real-world NeRF-360 dataset [3], as measured on the NVIDIA Jetson Orin NX [22] with a 10W power limit.

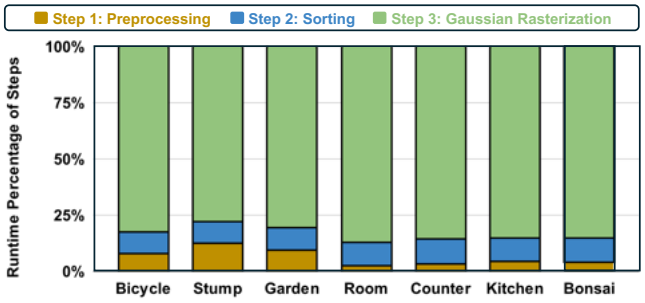


Fig. 5. Runtime breakdown of the 3DGS rendering pipeline [14] across all seven scenes from the large-scale, real-world NeRF-360 dataset [3] as measured on the NVIDIA Jetson Orin NX [22] with a power limit of 10W.

Rendering an image from a specific viewpoint using this representation (described in Fig. 3) involves three main steps:

**Step 1: Preprocessing.** The first stage of 3DGS rendering, depicted in Fig. 3(a), is preprocessing, which serves three purposes: projecting all 3D Gaussians onto a 2D plane according to the specified viewing position and angle, converting the color vector of each Gaussian to an RGB representation based on the viewing parameters, and computing the depth of each 3D Gaussian relative to the given viewpoint. This step transforms the 3D representation into a set of 2D Gaussians, each characterized by an opacity function $o$, an assigned RGB color $c$, a depth value $d$, a covariance matrix $\Sigma$ representing the 2D Gaussian probability distribution, and a center point $\mu$.

**Step 2: Sorting.** The second stage, as illustrated in Fig. 3(b), involves sorting the projected 2D Gaussians. Due to the potential overlap of Gaussians, each pixel may be influenced by multiple Gaussians. The rendering order impacts their visibility since Gaussians rendered earlier can obscure those rendered later. To maintain correct occlusion relationships and ensure visual consistency, the 2D Gaussians are sorted by their depth values. This depth-based sorting ensures that Gaussians nearer to the viewing position are rendered first, preserving proper occlusion handling.

**Step 3: Gaussian Rasterization.** The final step, Gaussian rasterization, applies the color of each Gaussian to the pixels it covers, based on opacity and Gaussian probability, and follows the depth order determined in Step 2. As shown in Fig. 3(c), for each Gaussian applied to a specific pixel, the density $\alpha_{P,i}$ is computed using the formula:

$$\alpha_{P,i} = o_i e^{-\frac{1}{2}(P-\mu_i)^T {\Sigma_i}^{-1}(P-\mu_i)}$$

where $P$ represents the pixel coordinate, $i$ is the index of the Gaussian, and $o_i$ is the opacity of the Gaussian. This density reflects the contribution of the Gaussian to the pixel. Once the density $\alpha_{P,i}$ is calculated, the RGB color contribution for that pixel is accumulated as follows, as shown in Fig. 3(d):

$$\mathbb{C}_P = \sum_{i=1}^{n} T_{P,i}\alpha_{P,i}c_i$$

The term $T_{P,i} = \prod_{j=1}^{i-1}(1-\alpha_{P,j})$ represents the accumulated density of all previously applied Gaussians at the pixel, accounting for the occlusion effects from preceding Gaussians. This ensures that only visible portions of overlapping Gaussians contribute to the final rendered color in Fig. 3(e).

### B. Locating the Bottleneck of 3DGS Rendering

To identify bottlenecks in rendering 3D Gaussians on edge SoCs, we conducted detailed profiling to analyze the runtime distribution across various rendering steps. Profiling was performed on the widely used 3D rendering dataset NeRF-360 [3], which includes seven real-world multi-object scenes. Kernel runtimes for the different rendering steps were measured using NVIDIA Nsight Systems [28] on the NVIDIA Jetson Orin NX [22] operating under a 10W power limit typical for edge devices [8], [41]. The average runtime for rendering all viewpoints in each scene is summarized in Fig. 4. The results indicate that the edge SoC achieves only 2-5 FPS, necessitating significant speedups before it can be used in real-world applications. The detailed runtime breakdown for each rendering step is shown in Fig. 5. Step 3, the Gaussian rasterization stage, dominates the overall runtime, accounting for over 80% of the total rendering time across all scenes, thereby representing the primary performance bottleneck.

## III. Leveraging the GPU Rasterizer for 3DGS

### A. Opportunities in GPU's Rasterizer

The computational demands of dominant Gaussian rasterization in the 3DGS rendering pipeline have motivated various acceleration efforts [16]. Current research primarily emphasizes the development of dedicated hardware accelerators for emerging 3D rendering pipelines such as 3DGS [16] and NeRF [10], [17]. Although these accelerators offer performance improvements, they can introduce significant trade-offs, including increased complexity in memory systems and software stacks [6] and the need for exclusive hardware resources. Unlike these dedicated approaches, our strategy focuses on enhancing existing GPU hardware to leverage the built-in capabilities of modern GPUs. This approach enhances 3DGS performance while preserving compatibility with conventional GPU operations and the original programming interface, minimizing disruptions to established workflows and fully utilizing the GPU's existing data path.

Modern GPUs have dedicated units optimized for triangle mesh rasterization [23]. This process involves iterating over

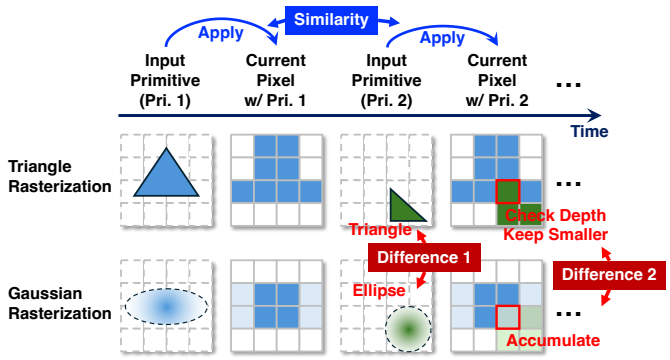


Fig. 6. Comparison of triangle and Gaussian rasterization. Both techniques sequentially apply primitives to pixels with key differences in primitive type (Difference 1) and reduction algorithm (Difference 2).

primitives (here referring to triangles) and mapping their parameters to display pixels. The pipeline is highly parallelized, relying on fixed-function accelerators for rapid computation [2], [5], [37]. Our approach begins with an investigation of the similarities and differences between Gaussian rasterization and triangle rasterization, the latter of which is efficiently supported by these optimized GPU hardware units.

### B. Contrasting Gaussian and Triangle Rasterization Methods

3DGS rasterization shares key foundational elements with traditional triangle mesh rasterization. As illustrated in Fig. 6, both processes iterate over primitives—triangles in conventional rendering and Gaussians in 3DGS—and map their respective parameters onto pixels. This similarity in the basic dataflow and computation pattern provides an opportunity to adapt existing GPU rasterizers to support 3D Gaussian operations.

However, there are also differences between the two rasterization processes. First, in triangle rasterization, the algorithm involves determining if a pixel lies within a triangular boundary. In contrast, 3DGS requires calculating Gaussian probability distributions to assess the coverage. This distinction necessitates modifications to the detection function to accommodate elliptical shapes rather than triangles. Second, for triangle mesh rasterization, the reduction method relies on a minimum-depth selection to determine which primitive's parameters should be applied to each pixel, whereas 3D Gaussian Splatting aggregates color contributions from multiple overlapping Gaussians. This approach increases the complexity of the reduction function, as overlapping Gaussians contribute to pixel color based on their calculated densities and occlusion effects. Effectively addressing these differences is essential to adapting current rasterization hardware for 3DGS, ensuring compatibility and preserving performance.

### C. Enhancing the Triangle Rasterizer for 3DGS

Building on our identified similarities and differences between Gaussian and triangle rasterization, we propose enhancing the existing triangle rasterizer to support 3DGS. A comparison of the resource requirements for triangle and Gaussian rasterization is summarized in Table II. As shown in the table, both rasterization processes have identical input and output parameter sizes and follow a similar procedure: initializing pixel storage and then applying primitives to each pixel. Owing to this shared I/O width and access pattern, the existing memory interface can be directly reused from the existing triangle rasterizer.

TABLE II
COMPUTATIONAL PRIMITIVES FOR RASTERIZATION

| Subtask (Operator) | **Triangle Rasterization [33]** | **Gaussian Rasterization [14]** |
|---|---|---|
| Input | Vertices' Coordinates (9 FP Numbers) | $\Sigma$, $o$, $\mu$, $c$ (9 FP Numbers) |
| 1 | Coordinate Shift (ADD, MUL) | Coordinate Shift (ADD, MUL) |
| 2 | Intersection Detection (ADD, MUL, DIV) | Gaussian Probability Computation (ADD, MUL, EXP) |
| 3 | UV Weight Computation (ADD, MUL) | Color Weight Computation (ADD, MUL) |
| 4 | Min-Depth Color Hold (ADD, MUL) | Color Accumulation (ADD, MUL) |
| Output | UV Weight, Depth (3 FP Numbers) | Accumulated Color (3 FP Numbers) |

Regarding computational resources, as highlighted in Table II, both processes primarily require multipliers and adders for their core tasks. This similarity allows us to introduce a reconfigurable datapath capable of supporting both triangle and Gaussian primitives with the same hardware resources. However, each primitive type has specific resource requirements: triangle rasterization requires a divider, while Gaussian rasterization necessitates an exponentiation unit. To address these specialized needs, we propose adding dedicated hardware units for these distinct operations, enabling seamless support for both types of rasterization.

## IV. PROPOSED GAURAST HARDWARE

### A. Overview of the GPU architecture with GauRast

Building on the analysis in Section III-C, we present the hardware design of GauRast, an enhanced rasterizer that extends existing hardware capabilities to efficiently support 3DGS. Unlike previous approaches that rely on dedicated accelerators, GauRast utilizes the built-in GPU rasterization hardware, specifically the triangle rasterizer, to execute 3DGS rendering tasks. This approach ensures seamless integration with existing GPU workflows, maintaining compatibility with conventional rendering while introducing minimal overhead.

As shown on the left side of Fig. 7(a), the GauRast hardware builds upon the Graphics Processing Cluster (GPC) of modern GPU architectures [24], [26]. Each GPC contains multiple Streaming Multiprocessors (SMs) for general-purpose computations and specialized fixed-function units for graphics processing. Within this architecture, we implemented an enhanced rasterizer capable of handling both triangle and Gaussian rasterization, extending the functionality of the original rasterizer that only supports triangle rasterization. As shown in Fig. 7(a), our modifications are confined to the rasterizer within the GPC, leaving the SMs untouched and preserving the overall GPU structure. The enhanced rasterizer is optimized

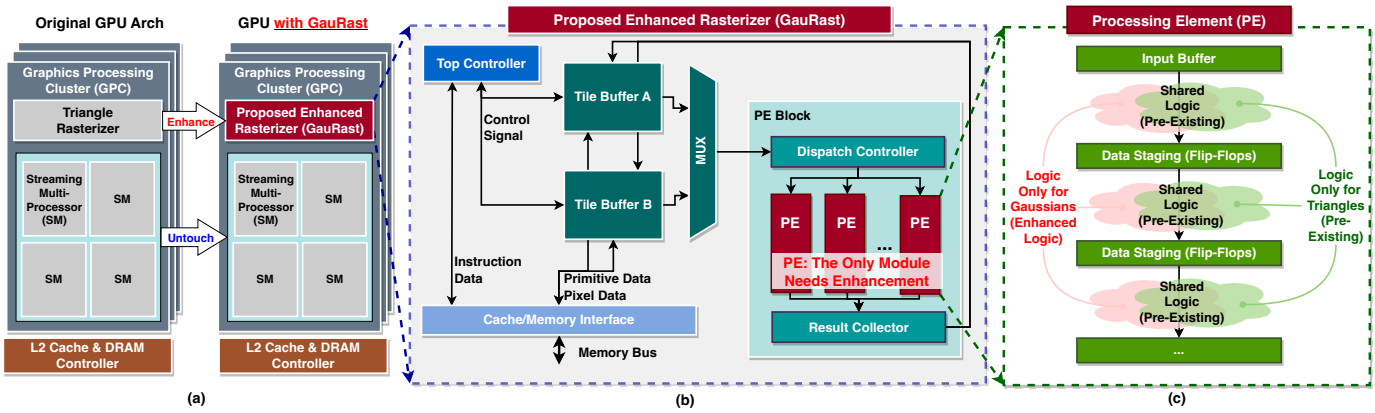


Fig. 7. Overview of the proposed hardware architecture: (a) Block diagram of the GPU's Graphics Processing Cluster (GPC) with proposed enhancements. (b) Detailed view of the enhanced rasterizer. (c) Internal structure of each PE, displaying dedicated and shared logic paths for Gaussian and triangle primitives.

to process Gaussian splatting, incorporating new logic paths alongside the existing hardware inside of each Processing Element (PE) as shown in Fig. 7(c), thus enabling seamless switching between traditional triangle rendering and Gaussian rasterization.

### B. Proposed GauRast Hardware

The enhanced rasterizer consists of several key components, as shown in Fig. 7(b):

**Tile Buffers (A and B).** These ping-pong buffers store primitives (either Gaussians or triangles) and pixel data, enabling efficient data management during the rendering process. By alternating between Tile Buffer A and Tile Buffer B, GauRast minimizes memory latency and allows concurrent data access for both Gaussian and triangle primitives.

**PE Block.** The core computation of the rasterization process is handled by the PE Block, which contains multiple PEs. Each PE is tasked with computations required for either Gaussian or triangle primitives. The PE Block operates with a high degree of parallelism, leveraging the intrinsic parallel structure of the rendering task for acceleration.

**Processing Element (PE).** Each PE supports both Gaussian and triangle rasterization and is equipped with a combination of shared and dedicated logic units, as shown in Fig. 7(c). The shared logic handles common operations, while dedicated logic paths manage primitive-specific computations, such as evaluating Gaussian probability distributions for 3DGS rendering and performing depth division for triangle meshes. Importantly, the existing triangle rasterizer includes both shared and triangle-specific logic; thus, only Gaussian-specific logic is added. Compared to a standard triangle rasterizer, our design requires minimal additional hardware in each PE: two adders, one multiplier, and one exponentiation unit for Gaussian support, while reusing nine adders and nine multipliers from the triangle rasterizer. The PE is the only module requiring additional logic to facilitate 3DGS rendering.

The GauRast hardware optimizes performance and resource utilization by reusing existing memory interfaces, such as Tile Buffers and Controllers, and incorporates minimal additional logic for 3DGS tasks. This design ensures that the enhanced rasterizer remains compatible with the existing triangle rendering pipeline, thus minimizing hardware complexity.

### C. CUDA-Collaborative Scheduling

In triangle mesh rasterization, CUDA cores and the hardware rasterizer collaborate to complete the rendering process. Specifically, non-dominant tasks such as 3D-to-2D projection and color querying are handled by the CUDA cores, while the rasterization is managed by the hardware rasterizer [23]. Similarly, we adopt a hybrid scheduling strategy to enhance efficiency, assigning non-dominant operations like preprocessing and sorting to the CUDA cores, while the enhanced rasterizer handles the dominant rasterization workload which is the primary bottleneck in 3DGS rendering.

## V. Evaluation

### A. Evaluation Setup

**Dataset & Baselines.** To evaluate the processing speedup and efficiency improvement achieved by our proposed GauRast, we conducted experiments using the widely adopted NeRF-360 dataset [3], a large-scale, real-world dataset using both the original 3DGS algorithm [14] and its latest efficiency-optimized variant [9]. For baselines, we consider both the NVIDIA Jetson Orin NX SoC [22], a representative edge GPU, and the only previously published accelerator proposal for 3DGS, GScore [16].

**Hardware Implementation.** We implemented a prototype of the enhanced rasterizer with 16 PEs using C++ and synthesized the design into RTL using Siemens Catapult [34]. This RTL was then synthesized, placed, and routed using Synopsys Fusion Compiler [36], targeting a 28 nm CMOS technology node (typical corner, 0.9 V, 1 GHz clock), following the methodology described in [15]. Fig. 8 shows the layout and area breakdown of the prototype. The typical power consumption of the prototype is 1.7 W as reported by Synopsys PrimePower [35] based on post-layout netlists and RTL simulation activities using randomly sampled test images from the target dataset. To ensure correct implementation, we validated the functional accuracy of both triangle and Gaussian rasterization against the software implementations, confirming that the RTL implementation's rendering output

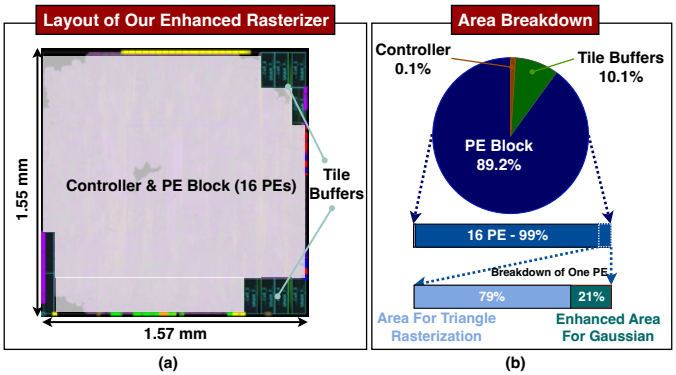


Fig. 8. Layout and area breakdown of the proposed enhanced rasterizer.

for both triangle rasterization [7] and 3DGS rendering [14] matches perfectly without any loss in rendering quality.

**Simulator Setup.** To match the effective area of the triangle rasterizer units in the baseline SoC [22], we scaled up our GauRast design to include 15 instances of the 16-PE rasterizer module, totaling 300 PEs. Consequently, the enhanced portion of the hardware occupies approximately 0.2% of the total area of the baseline Jetson Orin NX SoC [22]. We further developed a cycle-accurate simulator for fast evaluation of this scaled-up design. The simulator's runtime and power outputs were validated against the aforementioned RTL simulation results to ensure accuracy and reliability.

### *B. Comparison Against the Baseline SoC*

Fig. 9 illustrates the speedup and efficiency improvements achieved by GauRast in Gaussian rasterization compared to CUDA implementations across all scenes in the benchmark dataset, for both the original algorithm [14] and its latest efficiency-optimized version [9]. The results show that our enhanced rasterizer achieves an average runtime reduction of 23× and an average energy efficiency improvement of 24× for the original algorithm. For the efficiency-optimized algorithm, it achieves a 20× runtime reduction and a 22× improvement in energy efficiency. Additionally, as shown in Fig. 10, replacing CUDA-based rasterization with our proposed enhanced rasterizer leads to substantial end-to-end performance improvements. GauRast achieves a 6× reduction in runtime for the original 3D Gaussian Splatting algorithm and a 4× reduction for the optimized algorithm, yielding average frame rates of 24 FPS and 46 FPS, respectively.

### *C. Comparison Against SOTA 3DGS Accelerator*

SOTA work GSCore [16] on 3DGS rendering acceleration achieved a 20× speedup on an edge SoC [29] for Gaussian rasterization, utilizing a dedicated area of $3.95\,\mathrm{mm}^2$ with FP16 precision. If we re-implement GauRast to perform FP16 operations, our design achieves equivalent performance to GSCore but requires only $0.16\,\mathrm{mm}^2$ area, achieving a 24.7× improvement in area efficiency. This efficiency gain is attributed to the reuse of existing resources built in the triangle rasterizer. These results highlight the effectiveness of our enhancement approach in efficiently leveraging existing GPU components.

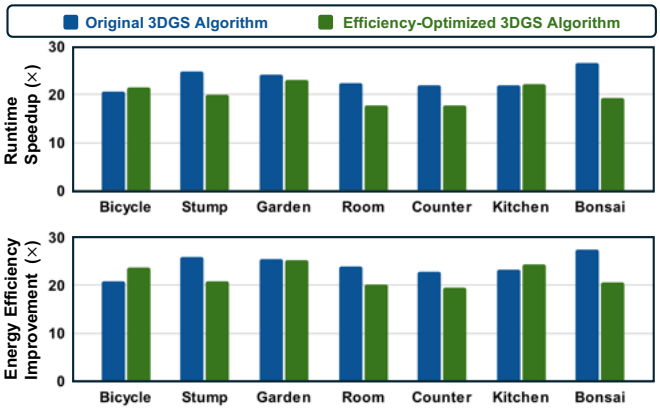


Fig. 9. Speedup and energy efficiency improvements achieved by GauRast in Gaussian rasterization, compared to CUDA implementations on the NVIDIA Jetson Orin NX [22]. Evaluations were conducted using both the original 3DGS algorithm [14] and its latest efficiency-optimized version [9], across all seven scenes of the large-scale, real-world NeRF-360 dataset [3].

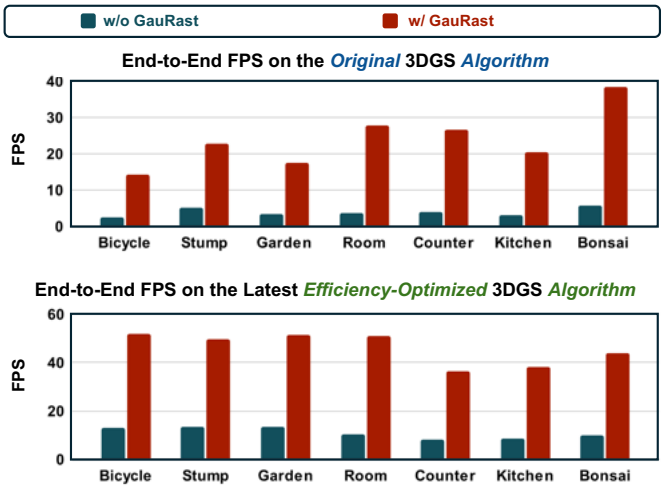


Fig. 10. End-to-end FPS comparison between the baseline SoC [22] and the baseline SoC enhanced with GauRast, using both the original 3DGS [14] and its latest efficiency-optimized version [9]. The evaluation was conducted across all seven scenes from the large-scale, real-world NeRF-360 dataset [3].

## VI. Conclusion

This work presents GauRast, an enhancement for existing GPU rasterizers designed to accelerate 3DGS rendering. Experiment results show that our enhanced rasterizer achieves a 6× speedup in end-to-end runtime for the original 3DGS and a 4× speedup for the latest efficiency-optimized pipeline on the NVIDIA Jetson Orin NX, with an area overhead of only 0.2%, demonstrating a promising path forward to enable real-time 3DGS rendering on edge devices.

## VII. Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable questions and constructive feedback. This work is supported by an internship at NVIDIA Research, the National Science Foundation (NSF) Computing and Communication Foundations (CCF) program (Award ID: 2312758), and the Department of Health and Human Services Advanced Research Projects Agency for Health (ARPA-H) under Award Number AY1AX000003.

## REFERENCES


[1] J. Abou-Chakra, K. Rana, F. Dayoub, and N. Suenderhauf, "Physically Embodied Gaussian Splatting: A Visually Learnt and Physically Grounded 3D Representation for Robotics," in *8th Annual Conference on Robot Learning*, 2024.

[2] H.-J. Ackermann, "Single chip hardware support for rasterization and texture mapping," *Computers & Graphics*, vol. 20, no. 4, pp. 503–514, 1996.

[3] J. T. Barron, B. Mildenhall, D. Verbin, P. P. Srinivasan, and P. Hedman, "Mip-nerf 360: Unbounded anti-aliased neural radiance fields," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 5470–5479.

[4] M. Q. Blog, "Tackling telepresence: 'spatial' delivers collaborative computing on oculus quest," https://www.meta.com/nl-nl/blog/quest/tackling-telepresence-spatial-delivers-collaborative-computing-on-oculus-quest, 2020.

[5] C.-H. Chen and C.-Y. Lee, "Reduce the memory bandwidth of 3D graphics hardware with a novel rasterizer," *Journal of Circuits, Systems, and Computers*, vol. 11, no. 04, pp. 377–391, 2002.

[6] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated {End-to-End} optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[7] Dmitry V. Sokolov, "TinyRenderer," 2022, https://github.com/ssloy/tinyrenderer, accessed 2022-05-20.

[8] M. Elgamal, D. Carmean, E. Ansari, O. Zed, R. Peri, S. Manne, U. Gupta, G.-Y. Wei, D. Brooks, G. Hills, and C.-J. Wu, "Carbon-efficient design optimization for computing systems," in *Proceedings of the 2nd Workshop on Sustainable Computer Systems*, ser. HotCarbon '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3604930.3605712

[9] G. Fang and B. Wang, "Mini-splatting: Representing scenes with a constrained number of gaussians," in *European Conference on Computer Vision*, 2024.

[10] Y. Feng, Z. Liu, J. Leng, M. Guo, and Y. Zhu, "Cicero: Addressing Algorithmic and Architectural Bottlenecks in Neural Rendering by Radiance Warping and Memory Optimizations," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 1293–1308.

[11] D. Graham-SmithMar, "What is the Metaverse?" https://www.techfinitive.com/explainers/what-is-the-metaverse/, (Accessed on 11/02/2024).

[12] J. Hu, X. Chen, B. Feng, G. Li, L. Yang, H. Bao, G. Zhang, and Z. Cui, "Cg-slam: Efficient dense rgb-d slam in a consistent uncertainty-aware 3D gaussian field," in *European Conference on Computer Vision*. Springer, 2025, pp. 93–112.

[13] N. Huang, X. Wei, W. Zheng, P. An, M. Lu, W. Zhan, M. Tomizuka, K. Keutzer, and S. Zhang, "S3Gaussian: Self-Supervised Street Gaussians for Autonomous Driving," *arXiv preprint arXiv:2405.20323*, 2024.

[14] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis, "3D Gaussian Splatting for Real-Time Radiance Field Rendering." *ACM Trans. Graph.*, vol. 42, no. 4, pp. 139–1, 2023.

[15] B. Khailany, E. Khmer, R. Venkatesan, J. Clemons, J. S. Emer, M. Fojtik, A. Klinefelter, M. Pellauer, N. Pinckney, Y. S. Shao *et al.*, "A modular digital vlsi flow for high-productivity soc design," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[16] J. Lee, S. Lee, J. Lee, J. Park, and J. Sim, "GSCore: Efficient Radiance Field Rendering via Architectural Support for 3D Gaussian Splatting," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 497–511.

[17] S. Li, C. Li, W. Zhu, B. Yu, Y. Zhao, C. Wan, H. You, H. Shi, and Y. Lin, "Instant-3D: Instant neural radiance field training towards on-device ar/vr 3D reconstruction," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

[18] S. Ma, Y. Weng, T. Shao, and K. Zhou, "3D gaussian blendshapes for head avatar animation," in *ACM SIGGRAPH 2024 Conference Papers*, 2024, pp. 1–10.

[19] H. Matsuki, R. Murai, P. H. Kelly, and A. J. Davison, "Gaussian splatting slam," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 18 039–18 048.

[20] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "Nerf: Representing scenes as neural radiance fields for view synthesis," *Communications of the ACM*, vol. 65, no. 1, pp. 99–106, 2021.

[21] NVIDIA, "Isaac Sim - Robotics Simulation and Synthetic Data — NVIDIA Developer," https://developer.nvidia.com/isaac, (Accessed on 11/02/2024).

[22] ——, "Jetson Orin for Next-Gen Robotics — NVIDIA," https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/, (Accessed on 04/02/2024).

[23] ——, "Life of a triangle - NVIDIA's logical pipeline," https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline, (Accessed on 11/02/2024).

[24] ——, "NVIDIA AMPERE GA102 GPU ARCHITECTURE," https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf, (Accessed on 11/02/2024).

[25] ——, "NVIDIA RTX A6000 For Powerful Visual Computing — NVIDIA," https://www.nvidia.com/en-us/design-visualization/rtx-a6000/, (Accessed on 11/02/2024).

[26] ——, "NVIDIA TESLA V100 GPU ARCHITECTURE," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, (Accessed on 11/02/2024).

[27] ——, "Recreate High-Fidelity Digital Twins with Neural Kernel Surface Reconstruction," https://developer.nvidia.com/blog/recreate-high-fidelity-digital-twins-with-neural-kernel-surface-reconstruction/, (Accessed on 11/02/2024).

[28] NVIDIA, "NVIDIA Nsight Systems," 2024, https://developer.nvidia.com/nsight-systems, accessed 2024-05-20.

[29] NVIDIA Inc., "Jeston Xavier NX Series Modules," 2022, https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/, accessed 2022-06-01.

[30] Z. Qi, J. Ma, J. Xu, Z. Zhou, L. Cheng, and G. Xiong, "GSPR: Multi-modal Place Recognition Using 3D Gaussian Splatting for Autonomous Driving," *arXiv preprint arXiv:2410.00299*, 2024.

[31] Z. Qian, S. Wang, M. Mihajlovic, A. Geiger, and S. Tang, "3dgs-avatar: Animatable avatars via deformable 3D gaussian splatting," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 5020–5030.

[32] W. Shen, G. Yang, A. Yu, J. Wong, L. P. Kaelbling, and P. Isola, "Distilled Feature Fields Enable Few-Shot Language-Guided Manipulation," in *7th Annual Conference on Robot Learning*, 2023.

[33] P. Shirley, M. Ashikhmin, and S. Marschner, *Fundamentals of Computer Graphics*. AK Peters/CRC Press, 2009.

[34] Siemens, "Catapult High-Level Synthesis and Verification," https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/, (Accessed on 11/02/2024).

[35] synopsys, "PrimePower," 2022, https://www.synopsys.com/implementation-and-signoff/signoff/primepower.html, accessed 2022-05-20.

[36] Synopsys, "Fusion Compiler: RTL-to-GDSII Design Solution," 2024, https://www.synopsys.com/implementation-and-signoff/physical-implementation/fusion-compiler.html, accessed 2024-05-20.

[37] X. Wang, F. Guo, and M. Zhu, "A more efficient triangle rasterization algorithm implemented in FPGA," in *2012 International Conference on Audio, Language and Image Processing*. IEEE, 2012, pp. 1108–1113.

[38] World Labs, "Hello, World Labs," 2024, https://www.worldlabs.ai/about, accessed 2024-11-01.

[39] Z. Xie, J. Zhang, W. Li, F. Zhang, and L. Zhang, "S-NeRF: Neural Radiance Fields for Street Views," in *The Eleventh International Conference on Learning Representations*, 2023.

[40] Y. Yuan, X. Li, Y. Huang, S. De Mello, K. Nagano, J. Kautz, and U. Iqbal, "Gavatar: Animatable 3D gaussian avatars with implicit mesh learning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 896–905.

[41] Y. Zhang, R. Wang, Y. Huo, W. Hua, and H. Bao, "Powernet: Learning-based real-time power-budget rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, no. 10, pp. 3486–3498, 2022.

[42] X. Zhou, Z. Lin, X. Shan, Y. Wang, D. Sun, and M.-H. Yang, "Drivinggaussian: Composite gaussian splatting for surrounding dynamic autonomous driving scenes," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 21 634–21 643.