FAMERS: An FPGA Accelerator for Memory-Efficient Edge-Rendered 3D Gaussian Splatting

Yuanfang Wang†, Yu Li, Jianli Chen, Jun Yu, Kun Wang*
State Key Lab of Integrated Chips & Systems, and School of Microelectronics, Fudan University, Shanghai, China
†yuanfangwang22@m.fudan.edu.cn, *kun.wang@ieee.org

***Abstract*—This paper introduces FAMERS, a tile-based hardware accelerator designed for efficient 3D Gaussian Splatting (3DGS) inference on edge-deployed Field Programmable Gate Arrays (FPGAs). 3DGS has emerged as a powerful technique for photorealistic image rendering, leveraging anisotropic Gaussians to balance computational efficiency and visual fidelity. However, the high memory and processing demands of 3DGS pose significant challenges for real-time applications on resource-constrained edge devices. To address these limitations, we present a novel architecture that optimizes both computational and memory overheads through model pruning and compression techniques, enabling high-quality rendering within the constrained memory and processing capabilities of edge platforms. Experimental results demonstrate that our implementation on the Xilinx XC7K325T FPGA achieves a 1.99× speedup and 13.46× energy efficiency compared to NVIDIA RTX 3060M Laptop GPU, underscoring the viability of our approach for real-time applications in virtual and augmented reality.**

***Index Terms*—FPGA, Hardware Accelerator, 3DGS**

I. Introduction

In recent years, the field of computer graphics has entered a transformative phase, catalyzed by the integration of machine learning techniques and innovative rendering methods. This evolution has led to the emergence of sophisticated algorithms capable of synthesizing photorealistic images from novel viewpoints with fidelity and efficiency. Among these advancements, neural representations such as Neural Radiance Fields (NeRF) [1] have significantly improved the ability to capture intricate geometric and photometric details of complex scenes.

However, the adoption of these advanced rendering techniques often incurs substantial computational and memory costs [2], which pose considerable challenges for real-time rendering applications, particularly in domains such as virtual reality (VR) and augmented reality (AR). Consequently, their applicability on embedded platforms remains limited. In contrast, the newly introduced 3D Gaussian Splatting (3DGS) [3] represents a notable breakthrough, striking a balance between rendering quality and computational efficiency. As depicted in Fig. 1, by employing anisotropic Gaussians as explicit rendering primitives, 3DGS facilitates rapid rendering, positioning it as an appealing candidate for deployment on edge devices.

The growing interest in 3DGS stems from its capability to deliver high-fidelity rendering with reduced computational overhead. Nevertheless, the substantial number of Gaussian primitives utilized presents challenges associated with memory-intensive operations, which can hinder its viability in edge computing environments. Recent works [4], [5], [6], [7] have focused on compressing the model size of Gaussians; however, the intermediate values of the Gaussian features still demand significant memory resources. This excessive memory requirement demands numerous off-chip data transfers, complicating real-time rendering processes.

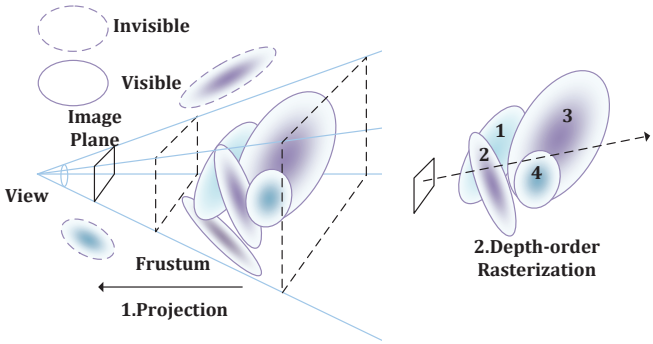


Fig. 1: 3DGS rendering steps.

Previous research [8] has proposed an ASIC design for a 3DGS accelerator, primarily concentrating on optimizing the Gaussian-tile intersection test. However, as 3DGS algorithms continue to evolve, the adaptability and flexibility of the underlying hardware become increasingly critical. In this context, Field Programmable Gate Arrays (FPGAs) emerge as more favorable platforms due to their inherent flexibility and reconfigurability when compared to ASICs [9].

To address these challenges, our work explores model pruning and compression strategies tailored for 3DGS. By strategically reducing the model's complexity, we aim to distill the capabilities of 3DGS into a more compact representation that preserves its rendering effectiveness while complying with the memory and computational constraints of FPGAs.

In summary, the main contributions are as follows:

Tile-based Hardware Accelerator for 3D Gaussian Splatting on Edge FPGAs. To the best of our knowledge, we are the first to propose an efficient edge FPGA accelerator for 3DGS inference, featuring an FPGA-specific architecture that reduces power consumption while maintaining necessary adaptability and flexibility.

Optimized Model Pruning and Clustering. We integrate pruning and clustering techniques within the 3DGS framework to minimize its memory requirements, ensuring conformity with edge FPGA specifications and enhancing rendering speed.

Tiling and Pipelined Execution Flow. We introduce a tiling method that allows for independent processing of each tile.

*Corresponding author.

978-3-9826741-0-0/DATE25/© 2025 EDAA

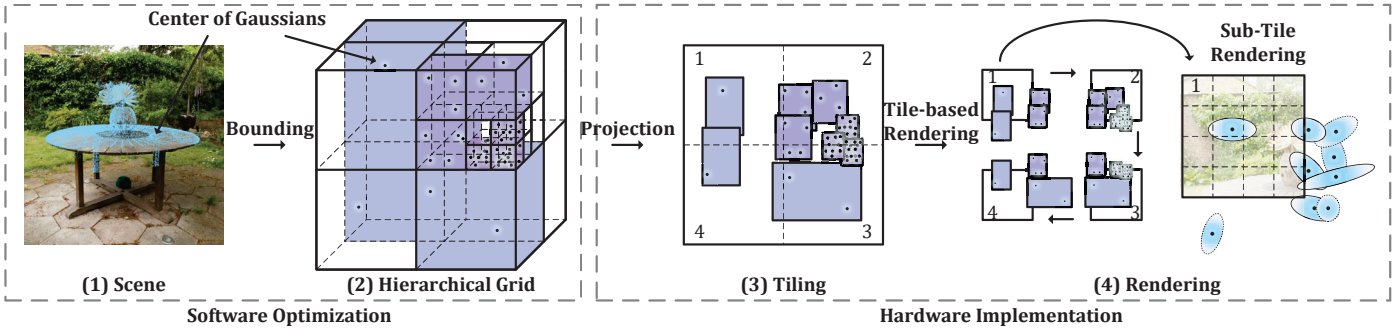
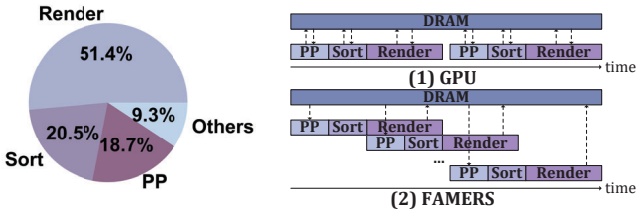


Fig. 2: Illustration of FAMERS rendering steps.



(a) (b)

Fig. 3: (a) Profiling for the NVIDIA RTX 3090 on scene Garden. (b) Execution diagram of GPU and FAMERS.

This pipelined execution strategy reduces the memory footprint during computation and facilitates the design of an architecture optimized for FPGA implementation.

Competitive Performance and Energy Efficiency. Our FPGA implementation achieves an average 1.99× speedup and 13.46× energy efficiency compared to the NVIDIA RTX 3060M Laptop GPU, and achieves 1.98× speedup over GSCore [8], demonstrating the effectiveness of our proposed approach on a Xilinx Kintex XC7K325T FPGA board.

II. Background and Motivation

A. Preliminaries of 3D Gaussian Splatting

3DGS models scene geometry using a collection of 3D Gaussians denoted as $\{G_i | i = 1, \ldots, N\}$. Each Gaussian G_i is defined by three parameters: its opacity $\alpha_i \in [0, 1]$, its center represented as $\boldsymbol{p}_i \in \mathbb{R}^{3\times1}$, and its covariance matrix in world space $\boldsymbol{\Sigma}_i \in \mathbb{R}^{3\times3}$. The mathematical formulation for each Gaussian is given by:

$$G_i(\boldsymbol{x}) = e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{p}_i)^T \boldsymbol{\Sigma}_i (\boldsymbol{x}-\boldsymbol{p}_i)}.$$

To render an image from this set of Gaussians, 3DGS employs an approximate depth sorting mechanism based on the distance from the Gaussian centers $\boldsymbol{p}_i$ to the image plane. Subsequently, the technique applies alpha blending according to the following expression: $\boldsymbol{c}(\boldsymbol{x}) = \sum_{i=1}^{N} w_i \cdot \boldsymbol{c}_i$, where the weight w_i is computed as: $w_i = T_i \cdot \alpha_i \cdot G_i^{2D}(\boldsymbol{x})$, and $T_i = \prod_{j=1}^{i-1}(1 - w_j)$. Here, $\boldsymbol{c}_i$ denotes the view-dependent color modeled by spherical harmonic (SH) coefficients, and G_i^{2D} represents the projected 2D Gaussian distribution of G_i through a local affine transformation.

In 3DGS, sparse point data derived from Structure-from-Motion (SfM) methods are initially modeled as Gaussian distributions. During the optimization process, these Gaussian representations are subjected to an adaptive density control strategy. This strategy allows the Gaussians to effectively learn and represent the scene.

B. Motivation

To render a 2D image from any viewpoint, the process of 3D Gaussian Splatting (3DGS) can be broadly divided into three distinct stages: preprocessing (PP), Gaussian sorting, and rasterization (volume rendering).

As illustrated in Fig. 3(a), we employed the Nsight System [10] and Nsight Compute [11] to profile the wall time of 3DGS rendering. Our analysis indicates that rasterization is the most time-intensive stage, accounting for over 50% of the total rendering duration. The prolonged execution time is primarily derived from the substantial number of intersections between the projected Gaussians (G_i^{2D}) and the pixels, which typically outnumber the Gaussians by two orders of magnitude. Consequently, a direct and effective strategy to reduce the rendering time associated with 3DGS is to minimize the number of Gaussians present in the scene. Further examination of the rendering process indicates that only a small subset of the Gaussians significantly contributes to rasterization, suggesting substantial potential for reducing their quantity.

In addition to rendering time, it is also necessary to consider the impact on memory consumption. The GPU implementation requires the storage and sorting of the entire array of projected Gaussians (G_i^{2D}), along with their pixel intersections, in a depth-ordered format. This approach presents two significant challenges. Firstly, the memory footprint of this array can occupy a considerable portion of the VRAM, often exceeding 2 GiB when accounting for the model size. This results in a significant amount of off-chip access overhead and low computation utilization in the preprocessing and sorting stages for edge device implementations. Consequently, minimizing memory access between the FPGA and off-chip memory is a crucial aspect of the design of our accelerator. Secondly, as illustrated in Fig. 3(b), the data dependencies between the preprocessing, sorting, and rasterization stages result in a serial execution model in GPU implementations, which highlights a promising optimization opportunity for FPGA architectures.

III. Algorithm Optimization

A. Model Compression

As discussed in Section II-B, the substantial memory footprint of the 3DGS model poses a significant challenge for accelerating rendering processes. To address this issue, we

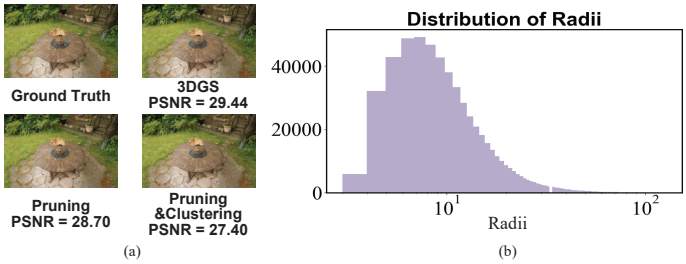


Fig. 4: (a) Ground Truth and rendered image for 3DGS and different compression methods. (b) Distribution of maximum Gaussians' radii of all views.

propose two strategies for model compression. The first strategy involves reducing the number of Gaussians utilized in the rendering process. The second approach focuses on compressing the color features associated with these Gaussians.

Several recent methods have been introduced to effectively reduce the size of the Gaussian model [4], [5], [6]. A commonality among these approaches is their emphasis on compressing the color features of the Gaussians. This is primarily due to the fact that generating the color representation for each Gaussian requires 45 spherical harmonic parameters, which can consume a substantial amount of memory.

To mitigate these challenges, we adopt the technique proposed in [7], which entails a trade-off in the quality of background reconstruction to achieve a reduction in the number of Gaussians to approximately 10% of the original count. Furthermore, we incorporate the sensitivity-aware k-means clustering method from [5] to consolidate similar color features into a single color representation, utilizing a codebook for efficient storage. As shown in Fig. 4(a), this combined approach not only enhances memory efficiency but also preserves critical rendering quality within acceptable limits.

B. Memory Overhead Reduction

As discussed in Section II-B, high memory consumption poses a significant challenge in Gaussian processing. Many operations within this context are redundant. For each Gaussian, it is necessary to compute its depth and the associated intersected tiles, which are subsequently stored in an array sorted by tile and depth. By leveraging the sorted order of Gaussians, we can optimize memory usage. Specifically, by pre-sorting Gaussians based on their center coordinates, we can render specific tiles using only a relevant subset of Gaussians, thus eliminating the need to retain the entire Gaussian-tile pair.

A naive approach entails partitioning the image plane into small tiles and performing rendering for each tile. However, this method still necessitates computation for all Gaussian-tile pairs, leading to excessive memory usage.

To identify the intersected tile in advance using the Gaussian center, the radius of the Gaussian must be below a predetermined threshold. However, during rendering, some Gaussians may possess larger radii, often resulting in visual artifacts and projections across large areas. To address these issues and facilitate Gaussian partitioning, we impose additional constraints during the training phase. Following the importance calculation method proposed by [7], [12], we compute blending

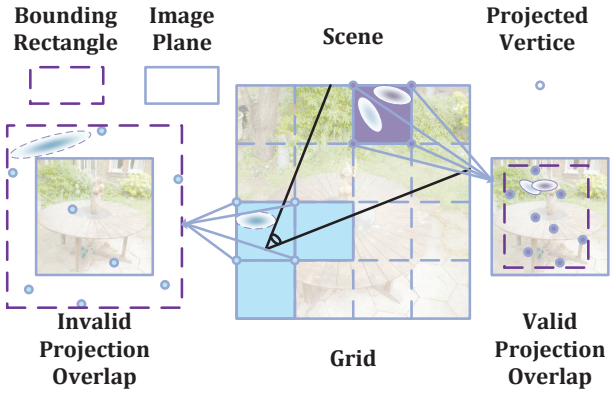


Fig. 5: Intersection estimation method.

weights w_{ij} for each Gaussian G_i to derive its importance score $I_i = \sum_{j=1}^{K} w_{ij}$, where K denotes the total number of pixels projected from each Gaussian. When high values of I_i are identified during training, we apply the splitting strategy outlined in [3] during optimization. By dividing larger Gaussians that contribute to artifacts into smaller components, we effectively mitigate visual disturbances and enhance the Gaussians' focus on pertinent scene details, thereby indirectly optimizing our tiling methodology.

Hierarchical Grid. To determine the necessary Gaussian features for computation, we analyze the maximum radii of each Gaussian across all views, as illustrated in Fig. 4(b). Our analysis reveals that over 95% of Gaussians have maximum radii less than 32. As radius increases, the quantity of Gaussians decreases. Accordingly, we categorize Gaussians into intervals based on their radii: [0, 16], [16, 32], [32, 64], [64, 128], and [128, $+\infty$]. We retain Gaussians with radii greater than 128, as these may significantly cover the image plane. For Gaussians within the lower intervals, we adopt a grid-based strategy tailored for each interval, capitalizing on the sparse distribution of low-radius Gaussians.

To accommodate the varying distribution of Gaussians across different spatial regions, we employ a multi-resolution grid instead of a uniform grid. The scene space is recursively divided into eight equal parts, with each division halving the side length of the previous grid. The frequency of recalculation for each Gaussian is directly proportional to the area generated from grid projection. Therefore, we cease further division and record grid information if the product of the squared side length and the number of Gaussians exceeds a predefined hyperparameter; otherwise, we continue subdividing until reaching a minimum grid size. Grids devoid of Gaussians are excluded from memory, further reducing overhead. Consequently, the resulting grid structure resembles that depicted in Fig. 2(2). As shown in Fig. 2(3) and (4), during rendering, we first partition the entire image plane into smaller tiles, computing and storing only the Gaussians that contribute to each tile on the chip.

Intersection Estimate. As illustrated in Fig. 5, we utilize a bounding rectangle defined by the minimum and maximum coordinates of eight projected vertices to estimate the locations of Gaussians within the grid on the image plane. During rendering, we initially check if any vertex depths exceed a

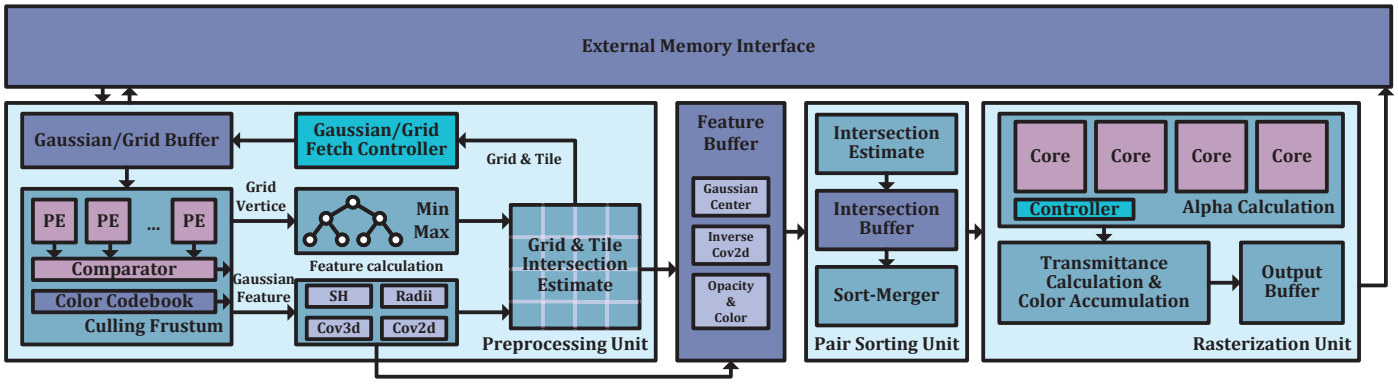


Fig. 6: Overall micro-architecture of FAMERS.

preset threshold of 0.2 (as specified in the 3DGS algorithm). We then adjust the bounding rectangle by expanding its sides based on the corresponding maximum radii. If the adjusted bounding rectangle intersects with the rendering tile, we consider all Gaussians within the grid as potential contributors. However, when a grid is positioned too close to the view center, the area of bounding rectangle may become too large, inadvertently including many unnecessary Gaussians. Consequently, for grids whose bounding rectangles encompass the entire image plane, we first identify all Gaussians within those grids.

IV. MICRO-ARCHITECTURE

A. Overall Architecture

As illustrated in Fig. 6, FAMERS consists of three primary units: the Preprocessing Unit, the Pair Sorting Unit, and the Rasterization Unit. The Preprocessing Unit computes the Gaussian intersections with tiles and extracts the corresponding Gaussian features. The Pair Sorting Unit organizes the Gaussian and tile pairs based on depth order, while the Rasterization Unit rasterizes the final image.

FAMERS is designed to operate in a pipelined manner, thereby facilitating the transfer of intermediate values between distributed buffers without requiring a write-back to DRAM, which significantly reduces memory overhead. The data flow within FAMERS progresses sequentially from the Preprocessing Unit to the Pair Sorting Unit and finally to the Rasterization Unit. Initially, the Preprocessing Unit processes the grids to determine the order of rendered Gaussians before fetching the Gaussians from DRAM to initiate image rendering.

B. Preprocessing Unit

The Preprocessing Unit is responsible for determining Gaussian intersections with tiles and calculating their features. It comprises a Culling Frustum Engine, a Feature Calculation Engine, and a Grid and Tile Intersection Engine. As described in Section III-B, the grids must be projected to identify intersected tiles. Both grids and Gaussians utilize the Culling Frustum Engine and the Grid and Tile Intersection Engine. The Gaussian/Grid Fetch Controller manages the initial grid processing and the subsequent grid-tile intersection calculations, controlling the reading of Gaussians from DRAM.

The Culling Frustum Engine consists of eight Processing Elements (PEs) and a comparator designed to eliminate center points that fall outside the frustum. Each PE features a Multiply-Accumulate (MAC) array and a normalization unit responsible for computing depth and Normalized Device Coordinates (NDC) of the centers. By utilizing the view matrix for depth computation and the projection matrix for NDC calculation, we enhance efficiency by substituting the third row of the projection matrix with the view matrix. This revised matrix is then used to multiply the coordinate vector. The color codebook is used to store the values associated with the clustered color feature.

For grid input, the resulting NDC coordinates are sent to a binary search tree to determine the minimum and maximum values along the x and y axes. Each node in the binary search tree contains a comparator and two swap registers. For Gaussian input, the features associated with each Gaussian are processed by the Feature Calculation Engine. This engine comprises a substantial MAC array capable of handling significant matrix operations, specifically for matrices smaller than 4×4. The output is then directed to the Feature Buffer in the Rasterization Unit for subsequent rasterization.

The Grid and Tile Intersection Engine receives the bounding box of the grid and identifies the intersecting tiles based on the rectangle defined by the minimum and maximum x and y values. The results of the Grid and Tile Intersection are forwarded to the Gaussian Fetch Controller, which directs the reading of the relevant Gaussians. The identified Gaussian and tile pairs are then transmitted to the Pair Sorting Unit to establish the Rasterization order.

C. Pair Sorting Unit

The Pair Sorting unit is a dedicated hardware implementation designed for efficient sorting operations. Unlike GSCore [8], which utilizes a bucket sorting method, our approach circumvents the substantial memory overhead and unstable time complexity associated with bucket sorting, both of which are incompatible with a pipelined architecture. Instead, we adopt the sort-merger from Spaghetti [13], which operates in a pipelined manner and completes the sorting of an array of size N in $2N + S$ cycles, where S represents the number of stages in the sort-merger and is defined as $S = \lceil \log_2 N \rceil$.

Initially, we identify the intersecting sub-tiles of Gaussians

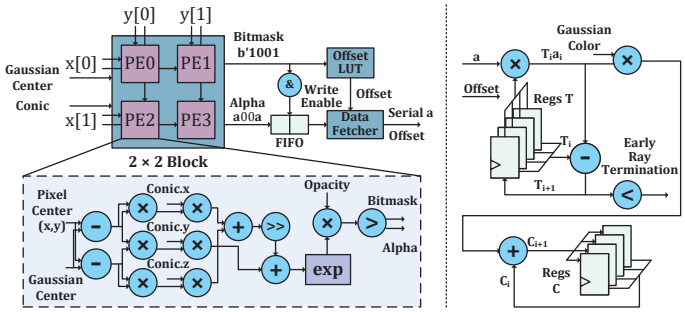


Fig. 7: (a) Details of Alpha Calculation core. (b) Details of Transmittance Calculation and Color Accumulation.

and collect the relevant data into a designated buffer. To enhance sorting efficiency, we implement two sort-mergers alongside a ping-pong buffer, facilitating seamless switching between depth queues. This design ensures that the rasterization process receives input data without delay. Subsequently, the corresponding depth values and Gaussian indices are transmitted to the Rasterization Unit for further processing.

D. Rasterization Unit

The rasterization unit comprises two primary components: the alpha calculation engine and the transmittance calculation and color accumulation engine. During the intersection judgment phase, we approximate the Gaussian distribution as a square with a side length of 3σ, where σ denotes the variance of the Gaussian. As a result, a substantial portion of the square may fall outside the Gaussian, leading to alpha values lower than 0.004 at these pixels. Such low-alpha pixels are considered negligible for volume rendering. Consequently, we have designed the alpha calculation engine and the transmittance calculation and color accumulation engine as distinct units.

Fig. 7(a) presents the architecture of the alpha calculation core, which computes $G_i(\boldsymbol{x})$ of 2×2 pixels as detailed in Section II-A and verifies whether the calculated alpha value exceeds 0.004. Fig. 7(b) depicts the architecture of the transmittance calculation and color accumulation engine. This engine is responsible for computing transmittance and accumulating the product of transmittance and color contributions from the Gaussians. The design of this engine is akin to that of the NeRF accelerator [14], [15]. When the transmittance falls below a predetermined threshold, we implement early ray termination, disregarding any further alpha values for that pixel. Once rays are terminated early or all Gaussians have been processed, the rasterization unit advances to compute the color for the subsequent tile.

V. EXPERIMENTS

A. Experimental Settings

We evaluate the performance of our FAMERS accelerator on the Xilinx Kintex-7 325T FPGA board, implementing using Verilog in the Vivado Design Suite 2020.2. The resource utilization is summarized in Table I. We set the clock frequency to 200 MHz for performance comparisons. Our implementation

TABLE I: Comparison of the baseline GPU, ASIC accelerator and FAMERS.

Platform	Frequency (MHz)	Technology	Bitwidth	DDR Bandwidth	Area or Used Resources	Power (W)
RTX 3060M Laptop (GPU)	900	8nm	FP32	GDDR6 336GB/s	3840 CUDA Cores	96.76
GSCore[8] (ASIC)	1000	28nm	FP16	LPDDR4 51.2GB/s	$3.95mm^2$	0.87
FAMERS (XC7K325T FPGA)	200	28nm	INT16/INT8	DDR3 12.5GB/s	173522 LUTs 241175 FFs 708 DSPs 286 BRAMs	15.23

TABLE II: Evaluated workloads.

Dataset	Scene (resolution)	Type
Tanks & Temples [16]	Train (980×545)	Real World & Outdoor
	Truck (979×546)	
Mip-NeRF360 [17]	Bicycle (1237×822)	
	Flowers (1256×828)	
	Garden (1297×840)	
	Stump (1245×825)	
	Treehill (1267×832)	
	Room (1557×1038)	Real World & Indoor
	Counter (1558×1038)	
	Kitchen (1558×1039)	
	Bonsai (1559×1039)	
Deep Blending [18]	Playroom (1264×832)	
	Drjohnson (1332×876)	

TABLE III: PSNR↑ and LPIPS↓ of rendered image from different optimization methods.

Method	Metric	Tanks & Temples	MipNeRF360 Outdoor	MipNeRF360 Indoor	Deep Blending
Baseline	PSNR	23.36	24.53	30.73	29.43
	LPIPS	0.19	0.24	0.19	0.25
Pruning	PSNR	22.28	24.47	30.26	29.90
	LPIPS	0.21	0.25	0.19	0.26
Pruning & Clustering	PSNR	22.73	24.21	29.69	29.42
	LPIPS	0.22	0.27	0.20	0.28
Ours	PSNR	22.75	24.15	29.66	29.38
	LPIPS	0.22	0.28	0.21	0.28

quantizes the parameters related to the covariance matrix to a 16-bit fixed-point representation, while the SH coefficients and opacity are quantized to 8 bits, as the SH coefficients have a minimal impact on image quality.

Datasets and Benchmarks. As shown in Table II, we utilize the same datasets and adhere to the experimental settings specified in 3DGS [3], specifically: Mip-NeRF 360, Tanks & Temples, and Deep Blending. For the Mip-NeRF 360 outdoor scenes, we employ images at 1/4 the resolution of the ground truth. For indoor scenes in Mip-NeRF 360, we use images at 1/2 the resolution of the ground truth. In the cases of Tanks & Temples and Deep Blending scenes, we retain the original resolution of the ground truth images.

B. Image Quality

The image quality of the proposed FAMERS accelerator was evaluated using PSNR and LPIPS across various optimization methods. As shown in Table III, the baseline performance provided a reference for comparison. After applying pruning, there was a slight reduction in PSNR and a marginal increase in LPIPS, indicating minimal degradation in image quality. Introducing clustering with pruning led to further small declines in PSNR and slight increases in LPIPS, reflecting the expected trade-off between computational efficiency and image fidelity.

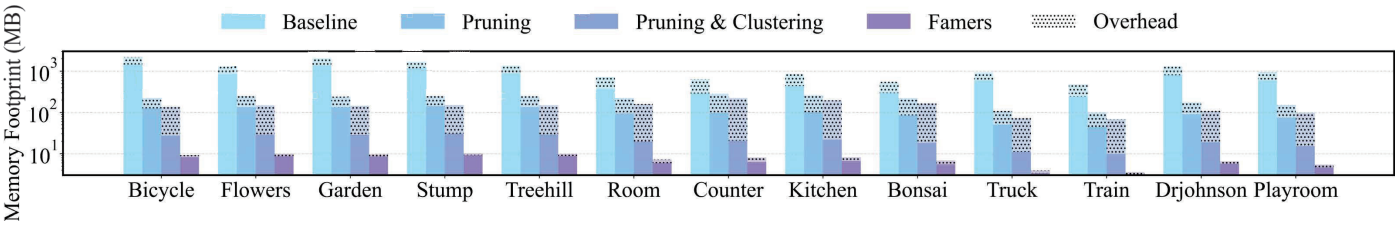


Fig. 8: Memory footprint of models and overhead introduced by computation.

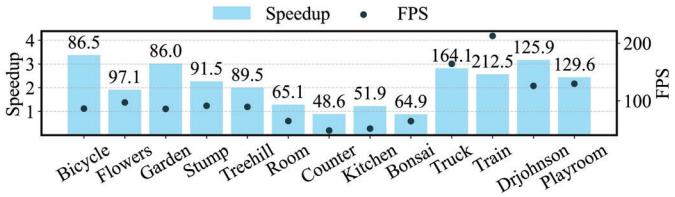


Fig. 9: Speedup of FAMERS over the baseline GPU.

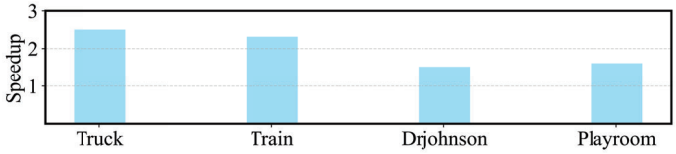


Fig. 10: Speedup of FAMERS over GSCore.

In addition to pruning and clustering, our proposed method incorporates a quantization step, which further enhances computational efficiency without significant impact on image quality. The results show that the PSNR and LPIPS of our method remain close to those of the pruning and clustering approach, with only minor differences. This confirms that the FAMERS accelerator is capable of optimizing performance while maintaining high visual fidelity, making it suitable for real-time applications in resource-constrained environments.

C. Speedup and Energy Efficiency

Fig. 9 and Fig. 11 present the speedup and energy efficiency achieved by FAMERS in comparison to the RTX 3060M Laptop GPU, which has 10× CUDA cores of NVIDIA Jetson Xavier NX and can achieve real-time rendering. FAMERS exhibits an average 1.99× speedup and a 13.46× energy efficiency improvement over the RTX 3060M. These advancements can be primarily attributed to an optimized pipelined execution flow and a reduced memory footprint, as illustrated in Fig. 8. However, FAMERS's performance is less optimal for the "counter" and "bonsai" scenes within the Mip-NeRF 360 indoor dataset. This reduction in performance can be attributed to the relatively large sizes of the rendered images, which require substantial computational resources, while the associated memory transfer demands remain comparatively low. Furthermore, Fig. 10 contrasts FAMERS's performance with that of GSCore [8], highlighting an average speedup of 1.98×. This enhancement can be largely attributed to the model compression techniques employed in FAMERS, as well as the reduction in the size of memory off-chip transfer.

D. Tile Length Exploration

The tile length directly regulates both the number of Gaussians and the rendered area within a single pipelined flow.

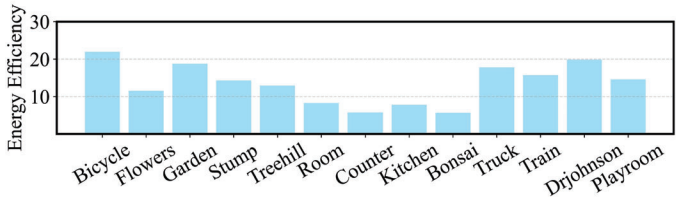


Fig. 11: Energy efficiency of FAMERS over the baseline GPU.

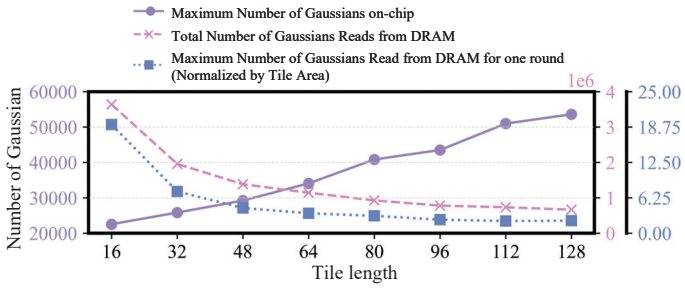


Fig. 12: Exploration of tile length.

Consequently, the tile length significantly impacts the on-chip buffer size and the size of off-chip memory transfers. As illustrated in Fig. 12, an increase in tile length necessitates a larger on-chip buffer, which in turn reduces the size of off-chip memory transfers. To optimize the trade-off between on-chip buffer requirements and off-chip memory transfer sizes, a tile length of 64 has been selected as the optimal configuration.

VI. Conclusion

In conclusion, this paper presents FAMERS, an FPGA accelerator designed for the efficient deployment of 3DGS for edge computing. By integrating model pruning and clustering techniques, FAMERS effectively addresses the computational and memory constraints of resource-limited environments, achieving a balance between visual fidelity and processing efficiency. Experimental results demonstrate improvements in both speed and energy efficiency over conventional GPU-based systems and ASIC accelerators, making this accelerator a viable solution for real-time applications.

Acknowledgments

The authors thank the anonymous reviewers for their suggestions. This work was financially supported in part by National Key Research and Development Program of China under Grant 2021YFA1003602, in part by Shanghai Pujiang Program under Grant 22PJD003, in part by Project of Shanghai EC under Grant 24KXZNA12. The computations in this research were performed using the CFFF platform of Fudan University.

REFERENCES

[1] 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.

[2] T. Müller, A. Evans, C. Schied, and A. Keller, "Instant Neural Graphics Primitives with a Multiresolution Hash Encoding," ACM Trans. Graph., vol. 41, no. 4, pp. 1–15, Jul. 2022, doi: 10.1145/3528223.3530127.

[3] B. Kerbl, G. Kopanas, T. Leimkuehler, and G. Drettakis, "3D Gaussian Splatting for Real-Time Radiance Field Rendering," ACM Trans. Graph., vol. 42, no. 4, pp. 1–14, Aug. 2023, doi: 10.1145/3592433.

[4] J. C. Lee, D. Rho, X. Sun, J. H. Ko, and E. Park, "Compact 3D Gaussian Representation for Radiance Field," Feb. 15, 2024, arXiv: arXiv:2311.13681. Accessed: Apr. 01, 2024. [Online]. Available: http://arxiv.org/abs/2311.13681

[5] S. Niedermayr, J. Stumpfegger, and R. Westermann, "Compressed 3D Gaussian Splatting for Accelerated Novel View Synthesis," in 2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Seattle, WA, USA: IEEE, Jun. 2024, pp. 10349–10358. doi: 10.1109/CVPR52733.2024.00985.

[6] P. Papantonakis, G. Kopanas, B. Kerbl, A. Lanvin, and G. Drettakis, "Reducing the Memory Footprint of 3D Gaussian Splatting," Proc. ACM Comput. Graph. Interact. Tech., vol. 7, no. 1, pp. 1–17, May 2024, doi: 10.1145/3651282.

[7] G. Fang and B. Wang, "Mini-Splatting: Representing Scenes with a Constrained Number of Gaussians," Oct. 16, 2024, arXiv: arXiv:2403.14166. doi: 10.48550/arXiv.2403.14166.

[8] 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, La Jolla CA USA: ACM, Apr. 2024, pp. 497–511. doi: 10.1145/3620666.3651385.

[9] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He, "OPU: An FPGA-Based Overlay Processor for Convolutional Neural Networks," IEEE Trans. VLSI Syst., vol. 28, no. 1, pp. 35–47, Jan. 2020, doi: 10.1109/TVLSI.2019.2939726.

[10] "NVIDIA Nsight Systems," NVIDIA Developer. Accessed: Jan. 18, 2025. [Online]. Available: https://developer.nvidia.com/nsight-systems

[11] "Nsight Compute | NVIDIA Developer | NVIDIA Developer." Accessed: Jan. 18, 2025. [Online]. Available: https://developer.nvidia.com/nsight-compute

[12] M. Niemeyer et al., "RadSplat: Radiance Field-Informed Gaussian Splatting for Robust Real-Time Rendering with 900+ FPS," Mar. 20, 2024, arXiv: arXiv:2403.13806. doi: 10.48550/arXiv.2403.13806.

[13] R. Hojabr, A. Sedaghati, A. Sharifian, A. Khonsari, and A. Shriraman, "SPAGHETTI: Streaming Accelerators for Highly Sparse GEMM on FPGAs," in 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Seoul, Korea (South): IEEE, Feb. 2021, pp. 84–96. doi: 10.1109/HPCA51647.2021.00017.

[14] Y. Wang, Y. Li, H. Zhang, J. Yu, and K. Wang, "Moth: A Hardware Accelerator for Neural Radiance Field Inference on FPGA," in 2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Marina Del Rey, CA, USA: IEEE, May 2023, pp. 227–227. doi: 10.1109/FCCM57271.2023.00055.

[15] Z. Ma, Z. Li, Y. Wang, Y. Li, J. Yu, and K. Wang, "Booth-NeRF: An FPGA Accelerator for Instant-NGP Inference with Novel Booth-Multiplier," in 2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC), Incheon, Korea, Republic of: IEEE, Jan. 2024, pp. 527–532. doi: 10.1109/ASP-DAC58780.2024.10473990.

[16] A. Knapitsch, J. Park, Q.-Y. Zhou, and V. Koltun, "Tanks and temples: benchmarking large-scale scene reconstruction," ACM Trans. Graph., vol. 36, no. 4, pp. 1–13, Aug. 2017, doi: 10.1145/3072959.3073599.

[17] J. T. Barron, B. Mildenhall, D. Verbin, P. P. Srinivasan, and P. Hedman, "Mip-NeRF 360: Unbounded Anti-Aliased Neural Radiance Fields," in 2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), New Orleans, LA, USA: IEEE, Jun. 2022, pp. 5460–5469. doi: 10.1109/CVPR52688.2022.00539.

[18] P. Hedman, J. Philip, T. Price, J.-M. Frahm, G. Drettakis, and G. Brostow, "Deep blending for free-viewpoint image-based rendering," ACM Trans. Graph., vol. 37, no. 6, pp. 1–15, Dec. 2018, doi: 10.1145/3272127.3275084.