Architectural Modeling for 3DGS - EE367

Siddhant Gupta

Abstract—Recent advancements in sensing and display technologies have driven the rapid growth of extended reality (XR), requiring high-fidelity rendering techniques like novel view synthesis (NVS) for immersive experiences. Gaussian splatting, a new approach to NVS, has shown promise over Neural Radiance Fields (NeRF) by efficiently projecting 3D scenes to 2D views with improved inference speed and image quality. However, the computational workload remains a challenge for low-power XR devices. In this project, we analyze the performance bottlenecks in Gaussian splatting and evaluate the potential for accelerating its most time-consuming stage using a lightweight hardware accelerator. By mapping the bottleneck to a 16×16 parallel quantized processor, we explore the trade-offs between computational performance, performance, and signal-to-noise ratio (SNR) to enable high-quality, low-latency rendering for next-generation XR applications. While this is not a cycle accurate model, or simulating a specific hardware architecture, it is the first step towards devloping a suitable accelerator model for the application

Index Terms—Gaussian Splatting, 3D Rendering, Computer Architecture, Novel View Synthesis, Accelerator Design, Extended Reality,

1 Introduction

This Extended reality (XR) applications—including virtual reality (VR), augmented reality (AR), and mixed reality—are becoming increasingly prevalent due to rapid advances in sensing and display technologies. Modern XR systems often feature dual high-resolution displays (one per eye) paired with specialized optics to replicate human 3D perception. These systems also integrate multiple sensors (e.g., RGB cameras, time-of-flight, and LiDAR) to capture environmental depth and geometry, enabling immersive and realistic user experiences. While AR primarily overlays virtual elements onto the real world, another key XR use case is the rendering of pre-captured or synthetic 3D scenes. Achieving smooth, high-fidelity rendering in these scenarios demands substantial computational resources, which poses stringent constraints on power efficiency and latency—particularly in lightweight, wearable devices.

One of the core challenges in XR rendering pipelines is novel view synthesis (NVS), the process of generating new viewpoints of a scene from a limited set of input images. For several years, Neural Radiance Fields (NeRF) have been the state-of-the-art approach for NVS. However, despite ongoing efforts to optimize NeRF for real-time performance—such as dedicated hardware acceleration—NeRF-based pipelines often remain computationally heavy for resource-constrained platforms. More recently, a technique known as 3D Gaussian splatting has emerged, offering potentially superior performance and image quality. Instead of using dense voxel grids or implicit neural representations, 3D Gaussian splatting represents a scene as a set of Gaussian primitives. This approach exploits the sparsity inherent in many real-world scenes and can achieve high-quality rendering at reduced inference times compared to NeRF. Preliminary work indicates that Gaussian splatting may be especially well-suited for mobile and edge devices

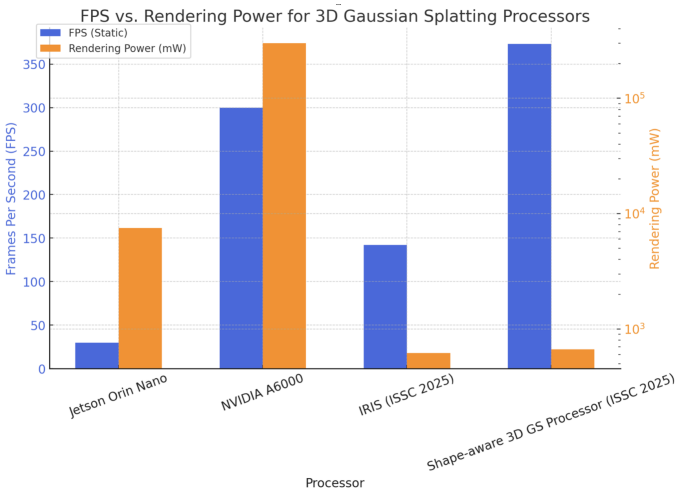

Fig. 1. Rendering performances (FPS) and power draws (mW) of Edge GPU, Server GPU, and two state of the art 3DGS accelerators running the same workload of 3DGS rendering inference

such as extended reality platforms due to the combination of rendering speed and quality of the projections [1].

Despite these promising results, the large memory footprint of 3D Gaussian splatting can still pose challenges for real-time or near-real-time XR applications, with scene representations sometimes exceeding tens of gigabytes. Some research efforts have begun to address this issue by applying quantization and pruning techniques, reducing memory usage by up to 25–27× while maintaining acceptable image quality [2]. Additionally, most existing implementations rely on general-purpose graphics processing units (GPUs) and CUDA, rather than exploring custom hardware acceleration. One of the few studies to investigate specialized hardware for Gaussian splatting proposed a co-processor design that offloads the volume rendering tasks from the GPU to a dedicated accelerator, effectively handling the inherent sparsity of the scene representation [3].

However in the latest computer architecture and VLSI

• *Siddhant Gupta is with the Department of Electrical and Computer Engineering, Stanford University, Stanford, CA, 94305.*
E-mail: siddg [at] stanford [dot] edu.

conferences, new accelerators have been proposed for 3DGS with massive improvements to efficiency. Figure 1 compares the power usage and rendering frames per second (fps) between two latest works presented in ISSCC 2025, an edge computing GPU, and a high performance data center grade GPU. As the chart shown in figure 1, custom accelerators can attain far higher performance (in fps) relative to their power draw. Building on these insights, this project aims to explore whether 3D Gaussian splatting can be efficiently mapped to a lightweight, low-power hardware accelerator.

Specifically, we target a CGRA (Course Grain Reconfigurable Array) style SoC [4], which is a more general-purpose SoC. By identifying the primary computational bottleneck in the Gaussian splatting pipeline and simulating parallelization strategies on a modest array of processing elements (e.g., a 16×16 array), we seek to determine what the critical bound to performance is for the algorithm. First, we ensure that we have a code base that can render Gaussians without the use of CUDA or any other SIMD/GPU libraries - this involves substituting a c rasterizer into the codebase for the 3D Gaussian splatting codebase developed by the original authors. The codebases used and their authors are credited in the acknowledgments.

Next, the C-based code has some profiling injected to diagnose the bottlenecks in the performance. After analyzing the bottleneck: the tiled rendering portion of the algorithm, we simulate what the performance and accuracy would be if this portion of code was run on 16x16 (256) parallel processing units at reduced precision. The precisions attempted in this work are the 16 bit floating point representation known as bfloat16, and the 8 bit floating points E5M2 and E4M3. Lastly, we analyze the resulting simulated runtimes and SNRs and provide an analysis of the results, as well as ideas for future work.

2 Related Work

As 3DGS is a relatively new method of rendering, most publications on this matter contain algorithmic improvements and utilize the CUDA GPU processing library for implementations. Early hardware works proposed offloading some portion of the algorithm off of the GPU, but the bulk of the rasterization was still done using CUDA on a GPU [3].

During the ISSCC 2025 conference, two more accelerators were proposed targeting 3DGS rendering applications. These accelerators both are standalone systems that do not depend on a GPU coprocessor using a CUDA framework. One work, IRIS, has a large portion of the chip area focused on a dedicated hardware unit to extract coarse grain surfaces from a scene and store associated gaussians by surface. This accelerator seeks to exploit spacial locality in the storage of gaussians in memory to reduce the computations significantly and has three very large cluster processing units to rasterize the pixels [5].

Another work, titled ”1.78mJ/Frame 373fps 3D GS Processor Based on Shape-Aware Hybrid Architecture Using Earlier Computation Skipping and Gaussian Cache Scheduler”, was also proposed in ISSCC 2025 and uses 90% less chip area than IRIS. This work also uses spatial locality in the processing of gaussians to reduce computations, but uses a different architecture to process the pixels. While IRIS had three large clusters, this work uses a hybrid array of processing elements (between pure rasterization and interpolation elements) [6].

While both of these mentioned ISSCC works had excellent results in terms of power usage for rendering throughput, they are highly specialized SoCs for this task. In this work, we seek to do the first step of architecture exploration to target a more general SoC/accelerator: something like a CGRA. A CGRA is a course grain reconfigurable array, with memory and compute tiles where the dataflow between the tiles and the operations of the tile are runtime reconfigurable [4]. The system also has a CPU to co-process with the CGRA.

3 Proposed Method

This work differs from most 3DGS related works as it seeks to conduct intitial analysis and exploration for an accelerator rather than improvements to the algorithm or proposing a new accelerator. Specifically, the target is a more generic computing paradigm than the accelerators in the mentioned ISSCC 2025 works. In this case, we analyze how 16 by 16 paralellism on quantized compute units affect the bottleneck of the algorithm and the resulting accuracy from quantization.

Most architectural performance analysis is done using a cycle-accurate model, where the simulated hardware exactly matches the functionality and features of the proposed accelerator, and accurately models the compute and memory-related cycles for a workload. However, for this stage of the process and timeframe for this project, discussions in this work are limited to simulated performance on a desktop CPU. The details of this implementation follow in this section.

3.1 Adapting Code and Profiling

The original codebase for the rendering of 3DGS of 3DGS uses CUDA kernels. In order to profile and experiment with quantization, it is important to have a clearer serial version of the code. A member of AnySyn3D provides a C++ version of the differentiable rasterizer submodule used in the original work, which can be used (with modifications) in the full top-level codebase (credited in Acknowledgements). Changes needed to be made for compatibility as the release versions of the two codebases are different. For example, the c rasterizer does not support or return a depth map or perform any anti-aliasing, so these need to be removed from the top-level code base.

The code changes needed for the bulk of this work are in the forward.c file, which defines the CPU preprocessing and rasterization for the forward pass of 3DGS rendering. First, the code is refactored such that the processing of single pixel is moved to a separate function. The loop is rewritten for pixel-parallelism at the tile level using calls to the pixel processing function. A couple of timing macros are defined using the C time.h library to print the elapsed time between the start and end calls for a given block. Macros were added to log the execution time of rendering one image, and finer grain rendering time for the individual stages: preprocessing, rasterization/rendering, and rendering per pixel.

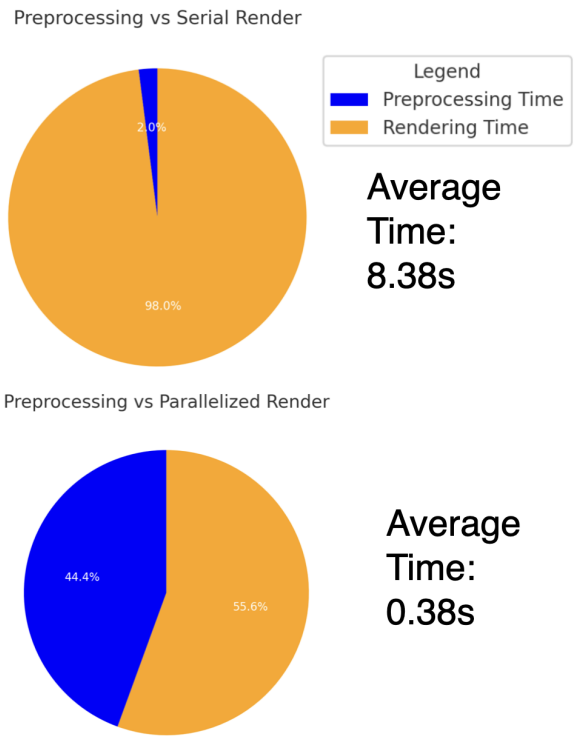

Fig. 2. Simulated runtime on an ARM64 Macbook CPU for rendering one image, with and without parallelization. Note that this is not a cycle accurate model and does not account for overhead at the system level.

3.2 Quantizing Rasterization

While most implementations of 3DGS at an algorithmic level use double precision (64 bit) or full precision (32) bit floating point representation for data values, this contributes to high computation energy, data storage cost, and high system memory usage. A common method to designing an efficient accelerator involves quantifying operations and reducing the size of data, where possible. Since the original CGRA supports operations up to 16 bit precision, this work attempts both 8-bit and 16-bit quantization schemes for the differentiable rasterization block.

Quantization in hardware is done such that both intermediate results, and mathematical operations happen at reduced precision. However, for modeling accuracy impacts, it is common and sufficient to quantize results before and after each mathematical operation rather than to define bit-accurate reduced-precision operations in C/C++.

In order to implement quantization in this work, custom quantized data types are defined for bfloat16, E5M2, and E4M3. There is also a function to convert to and from float for each of these data types. This conversion is done for each data member within the rasterization code at each mathematical operation to ensure the quantization model is as close as possible to if it were done using a reduced precision ALU.

The evaluation metrics (PSNR, SSIM, LPIPs) are calculated for each image and averaged for the baseline code, and the various quantizations. The results are described in Section 4.

3.2.1 8-bit Quantization

Two common quantizations for 8-bit floating point in our project are the E5M2 and E4M3 representations. In both cases, the most significant bit is used as the sign bit, while the remaining bits are divided between the exponent and the mantissa. The E5M2 format allocates 5 bits for the exponent (with a bias of 15) and 2 bits for the mantissa, providing a wide dynamic range that can represent very large or very small values; however, this comes at the cost of coarse fractional precision. Conversely, the E4M3 format uses 4 exponent bits (with a bias of 7) and 3 mantissa bits, yielding better precision in the fractional part but with a more limited range. In our codebase, we implemented conversion routines that extract the IEEE 754 32-bit representation of a float and then remap the sign, exponent, and mantissa to the appropriate 8-bit layout for each format. Special cases such as zero, subnormals, and infinities are carefully handled, ensuring that our compressed representations maintain acceptable accuracy for intermediate values like alpha or conic parameters in the rendering pipeline.

3.2.2 16-bit Quantization

For 16-bit quantization, we employ a bfloat16 representation—a format that dedicates 1 bit to the sign, 8 bits to the exponent, and 7 bits to the mantissa. This format is particularly well-suited for applications where preserving the dynamic range is critical, as its exponent field mirrors that of a full 32-bit float, while the truncated mantissa reduces precision. In our implementation, converting a 32-bit float to bfloat16 is performed using a simple truncation approach, whereby the lower 16 bits of the 32-bit representation are discarded. Similarly, to convert back, the bfloat16 value is shifted to reconstruct a 32-bit float. This approach results in a significant reduction in memory usage and bandwidth, which is especially beneficial in high-throughput or real-time rendering scenarios, while still retaining the essential dynamic range necessary for robust performance.

3.3 Simulating Parallelism & Bottleneck Analysis

The initial runtime (without parallelism) was heavily dominated by the rasterization step compared to the preprocessing step. About 95 percent of the runtime was in the pixel rasterization - the images are 1024 by 1024, and which is about a million individual pixels to be rasterized.

In order to simulate the effects of using parellel processing, the code was refactored to have inner loops of 16 by 16 dimensions calling the single-pixel compute function. Since the processor cannot actually process all 256s fully parallel, this work tracks the maximum pixel processing time for a given 16 by 16 tile, and uses that to represent the time for that tile. Then, when computing the final run time of rasterization, the representative time for each tile is summed. The resulting figure and analysis is presented in the Experimental Results and Analysis section.

4 Experimental Results and Analysis

Due to very poor runtime and unrecognizable image reconstructions, the E4M3 and E5M2 representations are excluded from the results. Resulting images were primarily black

Baseline, full precision	Quantized, bfloat16
Playroom Dataset SSIM : 0.8103411 PSNR : 24.4170532 LPIPS: 0.4125682	Playroom Dataset SSIM : 0.7620919 PSNR : 22.2413616 LPIPS: 0.4173917
Flowers Dataset SSIM : 0.4580899 PSNR : 18.8644238 LPIPS: 0.4646910	Flowers Dataset SSIM : 0.43133744 PSNR : 16.6820946 LPIPS: 0.54300000

Fig. 3. PSNR, SSIM, and LPIPS average over two datasets with and without quantization

with a few colorful stripes running in arbitrary directions. The range of exponents and mantissa are not sufficient to perform 3DGS rasterization at 8 bit precision with either floating point scheme, though E5M2 seemed to work better than E4M3.

However, the bfloat16 quantization appears to be viable. When bfloat16 quantization is enabled, the theoretical benefit is that the memory used for representing each of the gaussians can be reduced by half and computation can be done on the 16 bit processing elements. In terms of image quality, each of the images looks largely similar to the baseline representations without quantization. Quantitatively, there is about a 10% drop in PSNR with this quanitzation. with the SSIM dropping 5% on the playroom dataset and the LPIPS dropping about 20% on the flowers dataset, all else being about the same.

Qualitatively, most of the image looks about the same between the bfloat16 and the baseline, with the quantized image appearing a little sharper. However, there are bands of artifacts towards the right side of the image. Even though the bfloat16 representation is much better than the 8-bit floating ones, these artifacts are likely caused by the bfloat16 range's limitations as gaussian transparencies' contributions stretch the exponent range, but the coordinates and distances stretch the mantissa range.

Sample images are included in Figure 4 that demonstrate the quality of the reconsuutrction along with the artifacts. In future works, this can likely be addressed by performing mixed precision math, using a scaled integer representation for the distances and floating for the other components.

From a performance standpoint, the runtime of the code between fully serial and the model for parallelized goes from 8.38 to about 0.38 seconds. However, as previously mentioned, this timing result is not an accurate model of timing for the accelerator. For example, by using 16 by 16 parallelism for a tile, theoretically the improvement in the rendering time should be almost 256x but the actual improvement is an order of magnitude lower than this.

This is due to the course grain simulation being compiled and run on a desktop grade CPU, with overheads such as context switching, and loading memory. However, even this improvement suggests that utilizing 16 by 16 parallelism can be viable for achieving a target FPS of over 30 - as on the CPU it was able to bring the runtime down to be similar to that of the pre-processing stage. Of course, 0.38 seconds per frame is not enough to meet a reasonable FPS for an extended reality application but a cycle accurate hardware model, that has low-level optimizations, and accurately models latency for memory overhead and computations will provide a much more helpful metric for further optimizations.

5 Conclusion

In this work, we conducted an architectural analysis for a specialized accelerator targeting 3D Gaussian Splatting, a promising yet computationally heavy method for rendering 3D scenes from arbitrary camera viewpoints. By very roughly modeling a CGRA SoC composed of a CPU and a 16×16 array of parallel processing elements, we focused on balancing the two primary stages of the pipeline—preprocessing and rendering.

Our results on a 1K image confirm that rendering is the main bottleneck on CPU-based implementations, prompting us to explore reduced-precision parallelization to see if it is feasible to map this to the actual CGRA with 16 bit precision. Specifically, we evaluated bfloat16, E4M3, and E5M2 quantization methods; while the latter two suffered from exponent saturation and unacceptable image degradation, bfloat16 maintained a reasonable trade-off between accuracy and performance, incurring a modest PSNR drop of about 2 dB. There was some artifacting, that we are reasonably show can be mitigated with mixed-precision math and is also a topic for future exploration.

Furthermore, simulating tile-based (16×16) parallel rendering showed that the pipeline could be better balanced, potentially reducing overall runtime. Although these findings are based on a rough CPU-based runtime model, they provide valuable insight into how specialized accelerators might be structured for more efficient 3D Gaussian Splatting. Future efforts will involve developing a cycle-accurate model to capture memory transactions, concurrency, and interconnect overhead. Specifically, the cycle count for one pixels rendering as a function of the number of gaussions on a CGRA PE will be measured. Then, a more accurate model

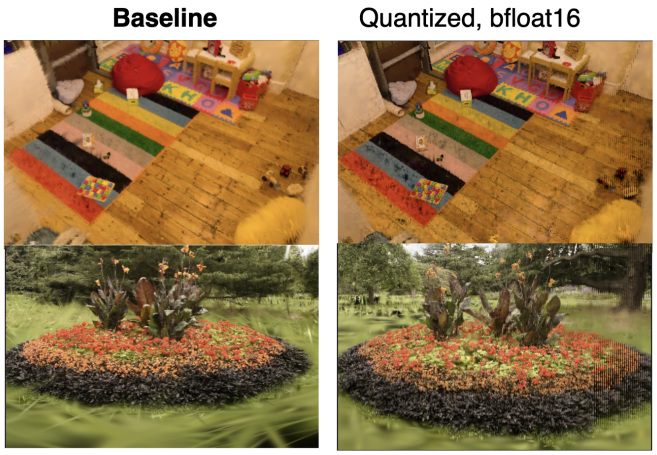

Fig. 4. Sample images from the Playroom and Flowers datasets used in the original Gaussian Splatting paper with and without bfloat16 quantization

for rendering throughput and latency can be developed from there on.

ACKNOWLEDGMENTS

The author would like to thank Brian Chao and Gordon Wetzstein for their support and mentorship in this course project. Additional acknowledgments are warranted for the author of the c-diff-rasterization codebase (the C++ implementation of the rasterizer) whose work is not affiliated with a publication (https://github.com/MrSecant/diff-gaussian-rasterization).

REFERENCES

[1] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis, "3d gaussian splatting for real-time radiance field rendering," 2023. [Online]. Available: https://arxiv.org/abs/2308.04079

[2] P. Papantonakis, G. Kopanas, B. Kerbl, A. Lanvin, and G. Drettakis, "Reducing the memory footprint of 3d gaussian splatting," *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 7, no. 1, p. 1–17, May 2024. [Online]. Available: http://dx.doi.org/10.1145/3651282

[3] W. Lizhou, H. Zhu, S. He, J. Zheng, C. Chen, and X. Zeng, "Gauspu: 3d gaussian splatting processor for real-time slam systems," 11 2024, pp. 1562–1573.

[4] T. Kong, K. Koul, P. Raina, M. Horowitz, and C. Torng, "Hardware abstractions and hardware mechanisms to support multi-task execution on coarse-grained reconfigurable arrays," 2023. [Online]. Available: https://arxiv.org/abs/2301.00861

[5] S. Song, S. Kim, W. Park, J. Park, S. An, G. Park, M. Kim, and H.-J. Yoo, "Iris: A 8.55mj/frame spatial computing soc for interactable rendering and surface-aware modeling with 3d gaussian splatting," 02 2025, pp. 1–3.

[6] X. Feng, H. Wang, C. Tang, T. Wu, H. Yang, and Y. Liu, "1.78mj/frame 373fps 3d gs processor based on shape-aware hybrid architecture using earlier computation skipping and gaussian cache scheduler," in *2025 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 68, 2025, pp. 1–3.

Siddhant Gupta Siddhant Gupta is a Masters student in Electrical Engineering at Stanford University, supervised by Professor Priyanka Raina. His interests are at the intersection of compute architecture and extended reality systems.