Efficient Parallel Execution through Hybrid Sorting Algorithms in 3D Gaussian Splatting

Seoyoon Kim
Electrical Engineering
KAIST
Daejeon, Republic of Korea
seoyoonkims@kaist.ac.kr

Junsoo Lim
Electrical Engineering
KAIST
Daejeon, Republic of Korea
junsoo0531@kaist.ac.kr

Hyunmin Choi
Electrical Engineering
KAIST
Daejeon, Republic of Korea
hchoi21@kaist.ac.kr

***Abstract*—This project addresses the efficiency bottlenecks in 3D Gaussian Splatting (3DGS), a cutting-edge technique for real-time radiance field rendering. Despite its advantages in performance and visual quality, 3DGS suffers from significant latency during depth sorting, which constitutes up to 30% of the rendering process. To overcome this, we propose a hybrid sorting mechanism combining Radix Sort and Bitonic Sort, optimized for both computational and memory efficiency. The solution incorporates bucket-based partitioning and early termination techniques to minimize unnecessary computations. Experimental results demonstrate up to 1.59%-8.09% performance improvement over the baseline, with increased adaptability to dataset-specific characteristics. This work highlights the potential of hybrid sorting strategies to enhance real-time rendering applications, particularly in memory-constrained environments such as mobile and virtual reality systems.**

I. Introduction

Neural network-based rendering methods, such as Neural Radiance Fields (NeRF) and 3D Gaussian Splatting, have advanced the detail and speed of rendering compared to traditional graphics methods by leveraging machine learning and parallel computing. Among these, 3D Gaussian Splatting, which represents radiance fields through anisotropic 3D Gaussians, has become a leading approach in real-time rendering.

Despite its strengths, 3D Gaussian Splatting faces efficiency and scalability bottlenecks, particularly in its data sorting stage. The rendering pipeline requires Gaussian depth-sorting for alpha blending, yet standard parallel sorting methods like radix sort significantly add to computational costs, with depth sorting alone comprising about 30 percent of rendering latency, according to NVIDIA NSight profiling. Since only certain gaussians are critical for the final image, an optimized approach could prioritize nearby gaussians to reduce sorting overhead and latency.

This project addresses sorting inefficiencies in 3D Gaussian Splatting by proposing a bucket-based hybrid sorting strategy that reduces redundant computations and maximizing memory efficiency. Our key contributions include selective depth sorting, bucket-based processing which together improve the performance and scalability of 3D Gaussian Splatting in real-time applications.

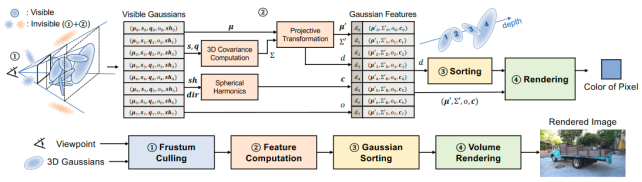

Fig. 1. Rendering pipeline for 3D Gaussian Splatting[1]

II. Background

A. 3D Gaussian Splatting

3D Gaussian splatting has become a leading technique in real-time radiance field rendering, offering high visual quality with efficient performance. Unlike Neural Radiance Fields (NeRF), which rely on complex multi-layer perceptrons (MLPs), 3D Gaussian splatting represents scenes with explicit anisotropic 3D Gaussians. This reduces computational demands, allowing direct manipulation of spatial attributes—ideal for real-time applications like virtual reality. The 3D Gaussian splatting pipeline includes frustum culling, feature computation, Gaussian sorting, and volume rendering. Frustum culling removes out-of-view Gaussians, while feature computation projects Gaussians onto the 2D plane, adjusting their spatial properties based on the camera view. Depth sorting then arranges Gaussians for correct front-to-back layering, crucial for accurate alpha blending during volume rendering, where ordered color accumulation produces the final image. This structure preserves both visual quality and differentiability, beneficial for optimization tasks. Depth sorting remains computationally heavy in CUDA based implementations, with parallel sorting algorithms like radix sort often becoming a bottleneck due to the high Gaussian count. Optimizing this stage is a key focus to further boost real-time rendering performance.

Profiling conducted with NVIDIA's NSight Systems on the 3D Gaussian splatting rendering process shows that Gaussian sorting consumes approximately 30% of the overall rendering latency, making it a substantial bottleneck in the pipeline. The current implementation performs a depth sort on all Gaussians before volume rendering, ensuring correct layering through sequential alpha blending. However, further analysis suggests

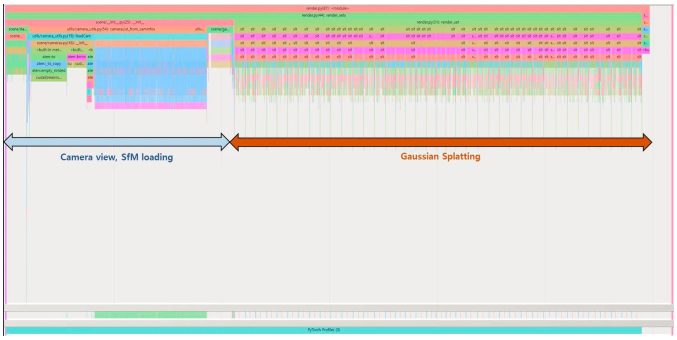

Fig. 2. Analysis of Gaussian Splatting through Nsight

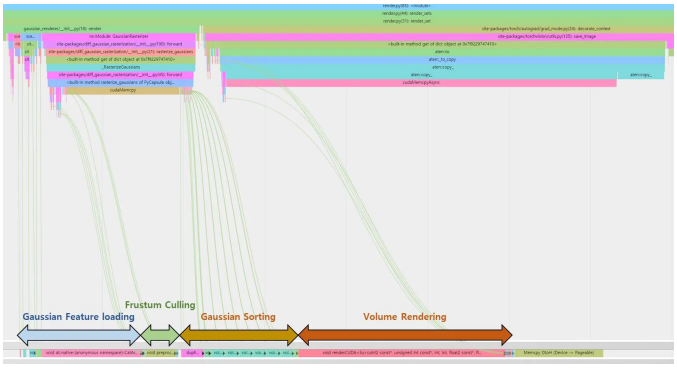

Fig. 3. Additional analysis on Gaussian Splatting pipeline through Nsight

that not all Gaussians equally impact the final rendered image, and thus a significant portion of Gaussian sorting may involve unnecessary computation. Through additional investigation, it was observed that Gaussians closer to the camera—those with lower depth—tend to have a higher priority in the rendering process, as they play a more substantial role in determining final pixel colors. This insight suggests that incorporating depth prioritization could allow for a more selective sorting mechanism, where only Gaussians most likely to influence visible layers are sorted and processed. By implementing such selective sorting, it is expected that overall sorting latency can be reduced substantially, resulting in higher frames per second (FPS) and improved real-time rendering performance.

B. Radix sort

Radix Sort is a non-comparative sorting algorithm that processes data by examining and sorting digits or characters one at a time, starting from the least significant digit (LSD) or the most significant digit (MSD). It is particularly effective for sorting integers or fixed-length strings, as it exploits the positional structure of the data. While Radix Sort offers a lower time complexity than many comparison-based algorithms in specific scenarios, it also presents notable inefficiencies and constraints, especially in memory-constrained environments or when full sorting is unnecessary. These limitations can be categorized as follows:

1) Inefficiency in Partial Sorting Scenarios: Radix Sort inherently processes every digit of the data, regardless of

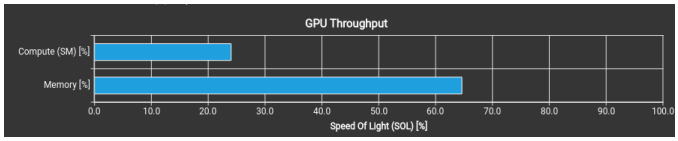

Fig. 4. Analysis of Radix Sort through NSight Compute

pre-existing order or the need for only partial results. This characteristic makes it less efficient for tasks that do not require complete sorting. For example, in applications such as selecting the k-th smallest element, algorithms like Quickselect are significantly more efficient as they focus exclusively on the desired subset of the data, avoiding unnecessary computations. In contrast, Radix Sort performs a full traversal of all digits, leading to redundant processing and increased resource utilization.

2) Memory Bandwidth Constraints: Radix Sort relies on auxiliary data structures (e.g., buckets) and frequent memory accesses to redistribute data across multiple passes. In systems where memory bandwidth is a critical bottleneck, this frequent data movement can severely degrade performance. Such environments, including embedded systems or hardware with limited on-chip memory, are particularly unsuitable for Radix Sort due to its intensive memory access patterns and substantial auxiliary space requirements. Using the Nsight Compute profiling tool on an RTX Titan, we observed that memory is more heavily utilized than compute. Additionally, despite the use of the optimized configurations provided by the CUB library, the algorithm's inherent characteristics prevent memory from being fully coalesced, resulting in lower utilization compared to the theoretical bandwidth.

3) Trade-offs in Radix Size and Memory Usage: Radix Sort's memory consumption is heavily influenced by the choice of radix size. A larger radix increases the number of buckets and memory usage, with memory consumption growing proportionally to O(r), where r is the number of buckets. Conversely, a smaller radix reduces bucket memory usage but requires more sorting rounds, potentially offsetting performance gains. This trade-off highlights the dependency of Radix Sort on the dataset size and the range of keys, making it most effective when the dataset is large, memory is sufficient, and the key range is relatively small.

4) Comparing Radix Sort and Bitonic Sort: In contrast to Radix Sort, Bitonic Sort is advantageous in memory-constrained environments due to its ability to perform in-place sorting, requiring minimal additional memory. In-place algorithms sort data directly within the input array, utilizing negligible extra space. This makes Bitonic Sort highly suitable for sorting large-scale data in parallel processing hardware environments. While Radix Sort often has lower time complexity, it demands significantly more memory, particularly for sorting at each digit position. Its memory usage grows with both the size of the dataset (N) and the range of keys (M).

To address these limitations, we propose a hybrid method that combines sorting techniques: first, we apply Radix Sort

	Rendered gaussians	Allocated gaussians	Skipped gaussians (%)
Lego	2197	2799	24%
T-rex	1147	1387	18%
BouncingBalls	1439	2733	44%
JumpingJacks	802	1039	22%
avg	1396	1989	27%

Fig. 5. Ratio of skipped gaussians for each dataset

to the upper n bits of the keys, dividing the data into multiple buckets. Then, the sorting of each bucket is completed independently and flexibly using Bitonic Sort.

For our target application in 3DGS, we apply Radix Sort to sort the tiles and an additional n bits, while the remaining data is sorted in parallel using Bitonic Sort for the required depths within each tile. This hybrid method leverages the strengths of both sorting algorithms, optimizing performance for the given dataset and hardware constraints.

III. Mechanism

Before explaining our proposed mechanism, we analyze the limitations of the existing 3D Gaussian Splatting kernel and discuss our approach to addressing these issues. The basic 3D Gaussian Splatting volume rendering kernel processes Gaussians in a tile-wise manner. Within a tile, the attributes of multiple Gaussians are sequentially accumulated, and once the transmittance exceeds a certain threshold, the rendering is considered complete, triggering early termination (T-exit). The currently implemented forward kernel performs complete sorting of all Gaussians before starting the rendering process. However, this approach potentially includes unnecessary computations, as Gaussians that do not contribute to the final pixel color are also sorted. This inefficiency is also highlighted in [2-gscore], which shows that the GPU rendering pipeline completes full sorting even in cases where early termination renders such computations unnecessary. Our analysis of the dataset revealed that, on average, approximately 20–30% of Gaussians (and up to 40% in some cases) did not need to be sorted, yet were unnecessarily processed. Since sorting incurs the second-highest latency after rendering, we optimized the current sorting method as described below to address these inefficiencies.

A. A Hybrid Approach: Combining Radix Sort and Bitonic Sort

To address the limitations of Radix Sort, we propose a hybrid approach.

Step 1: Perform Radix Sort on the upper n bits of the keys to divide the dataset into multiple buckets. This step ensures an efficient, high-level partitioning of data, enabling subsequent sorting to operate on smaller, more manageable subsets.

Step 2: Complete the sorting process within each bucket using Bitonic Sort. This allows for highly parallelized, in-place sorting with minimal memory overhead, which is particularly beneficial in constrained environments.

This hybrid method combines the strengths of both algorithms: the efficiency of Radix Sort for high-level categorization and the memory-efficient, parallel-friendly nature of Bitonic Sort for finer sorting.

In our target application for 3D Gaussian Splatting (3DGS), we apply Radix Sort to sort the tiles and the upper n bits of the keys, while using Bitonic Sort to sort the remaining bits in parallel within each tile. This approach optimizes performance for the specific characteristics of the dataset and the hardware constraints.

B. Optimized Bucket-Based Pre-processing

To efficiently handle large-scale datasets, we propose a bucket-based optimization technique using a hybrid sorting approach which is shown in Fig.6. This method combines Radix Sort for high-level partitioning and subsequent processing steps that enable early termination under certain conditions. The process is as follows:

A key, which has to be sorted in 3DGS, typically consists of a 12-bit Tile ID concatenated with a 32-bit Gaussian Depth. To ensure proper rendering, the keys need to be sorted in ascending order. For parallel processing, the Tile IDs must be sorted before rendering. Previously, Radix Sort was applied to all 44 bits of the keys. In contrast, our approach sorts only the upper 12 bits corresponding to the Tile ID along with an additional n bits.

This operation divides the data into 2^n distinct buckets based on the prefixes of the keys for each corresponding tile ID. For instance, when n=2, the dataset is divided into four buckets corresponding to prefixes 00,01,10, and 11. After sorting, the range of data for each bucket is recorded, including the pointers of start and end indices for each prefix. For example, the range of data with the prefix 00 is stored as $(start_{00}, end_{00})$. Similarly, ranges $(start_{01}, end_{01})$, $(start_{10}, end_{10})$, and $(start_{11}, end_{11})$ are stored.

C. Tile-Based Processing with Parallel Threads

Once the ranges are determined, threads are assigned to tiles corresponding to each bucket. All threads allocated to a specific tile operate within the range of data defined by the prefix of the key. For instance, in the first iteration, threads process the data within the 00-range. The data is further sorted based on depth, enabling efficient rendering of objects or scenes in the correct order. In subsequent iterations, the 01-range, 10-range, and 11-range are processed similarly.

The range-based structure provides opportunities for early termination. If an early exit signal is detected during processing (e.g., when specific conditions indicate that further computations or rendering are unnecessary), remaining ranges can be skipped entirely. This drastically reduces computational overhead and improves overall efficiency by avoiding unnecessary sorting and rendering operations as described in Fig.7. In addition to this, if the range of that particular bucket is zero, it skips the whole sorting and rendering process.

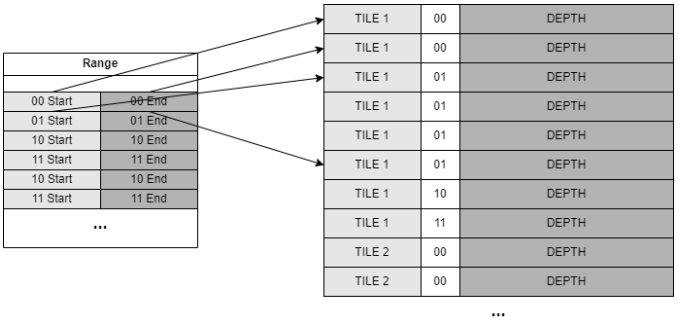

Fig. 6. Example of storing range of each tile's bucket

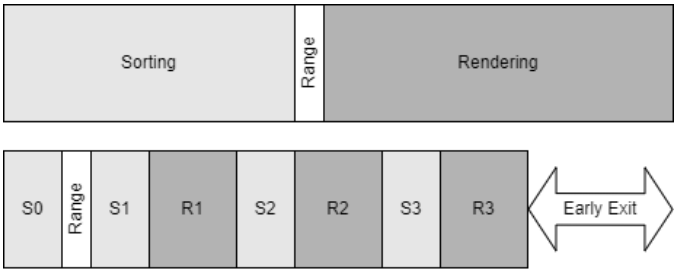

Fig. 7. Reduced latency with early exit

IV. Experiments

All experiments, including the baseline, were conducted on an NVIDIA RTX Titan (Turing microarchitecture). The datasets required up to 10GB of VRAM for rendering and a similar amount of host (CPU) memory. The environment utilized dual Intel Xeon Gold 6242 CPUs running on Ubuntu 18.04 LTS.

A. Performance Comparison of Gaussian Rendering with Varying Bucket Sizes

The table compares the changes in Frames Per Second (FPS) across various datasets during Gaussian rendering, based on the number of buckets and a baseline model. The baseline utilizes the vanilla implementation of Gaussian Splatting, while the proposed method employs Radix Sort to divide data into buckets and Bitonic Sort within each bucket to improve parallelism.

In this approach, the parameter n denotes the number of Most Significant Bits (MSB) used for partitioning the buckets. For example, when n=3, the data is divided into $2^3 = 8$ buckets. The best performance was observed with n=3 for datasets such as Hell Warrior, Trex, and Lego, where dividing

Dataset	Gaussian #	baseline	n=0	n=1	n=2	n=3	n=4
hell warrior	57882	298.1	299.8	302.2	292.7	**304.1**	277.6
jumping jacks	67200	218.9	**228.5**	225.9	226.0	226.2	226.1
bouncing balls	89023	79.8	78.2	80.1	80.7	80.0	**81.1**
trex	123593	160.9	158.6	156.2	160.2	**173.9**	150.5
lego	192771	171.2	155.9	170.6	173.5	**174.0**	170.2

Fig. 8. Average FPS with varying bucket size

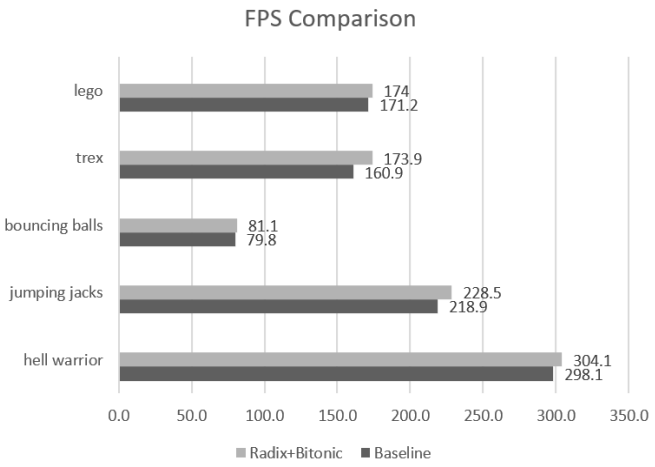

Fig. 9. Comparison of FPS between the baseline and the optimal hybrid method

the data into 8 buckets resulted in significant FPS improvements. Especially, proposed method achieved a remarkable 13 FPS increase over the baseline when n=3 for Trex dataset. The highest performance of 81.1 FPS was achieved with n=4 demonstrating that finer bucket divisions can be advantageous for Bouncing Balls dataset.

The reasons for the observed performance differences are as follows: Unlike the baseline, where all key values from all tiles are globally sorted in a single operation, our method assigns rendering threads to sort only the key values relevant to their respective tiles in a localized manner. This allows for more efficient sorting tailored to each tile.

Additionally, the Tile Exit mechanism helps avoid unnecessary sorting by terminating computations for tiles or buckets that do not significantly affect the rendering output. Furthermore, our implementation examines the range of the upper n-bits and skips sorting entirely for buckets where the difference between the range's start and end indices is zero. This approach effectively handles sparse bucket distributions and prevents redundant computations.

When n=0, bucket partitioning based on the upper bits of depth is not applied. However, each tile processes and sorts only the Gaussians relevant to that tile. In datasets such as Bouncing Balls, Trex, and Lego, performance decreases for n=0 due to the higher efficiency of Radix Sort implemented in cuBLAS compared to our Bitonic Sort for larger datasets. As n increases, the size of the data in each bucket decreases, reducing the difference in performance between Bitonic Sort and cuBLAS's Radix Sort. The tile exit and range skipping effect also becomes more pronounced, further improving performance. Conversely, in the Jumping Jack dataset, the best performance is observed at n=0. It seems that the data is evenly distributed across tiles, maximizing the benefits of tile-based Gaussian sorting than global sorting.

At n=4, where the dataset is divided into 16 buckets, performance degrades due to underutilization, as the data size

	Total Gaussians	Gaussians per tile	T-Exited tiles (%)
Lego	192771	864	7%
T-rex	123593	1037	27%
BouncingBalls	89023	1845	17%
JumpingJacks	67200	567	22%

Fig. 10. Ratio of T-Exit Tiles to Total Tiles

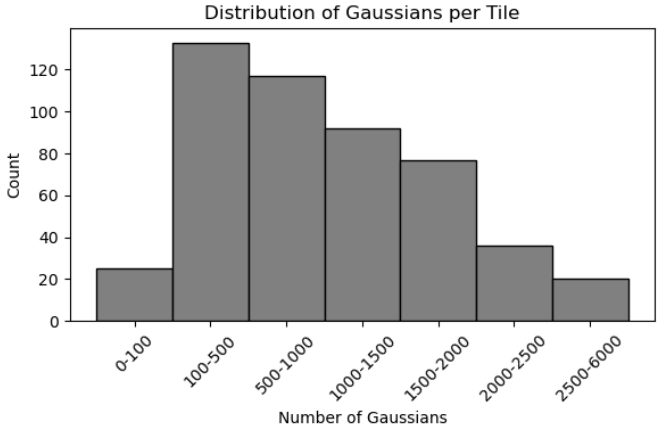

Fig. 11. Distribution of the Number of Gaussians Assigned to Tiles

per bucket becomes too small to effectively utilize GPU resources. These results highlight the importance of tailoring the partitioning strategy to the dataset characteristics, as different datasets respond differently to changes in bucket size.

V. Discussion

A. Low early termination ratio

The performance improvements in our implementation are noticeable when T-exit occurs frequently. However, contrary to our expectation that many tiles would contribute minimally to the final pixel values, the results often showed otherwise. The Fig. 10 illustrates the proportion of gaussians in randomly selected tiles where T-exit occurred. While our analysis confirms that T-exit can reduce the overhead of sorting, the limited proportion of T-exit events restricted the potential performance gains. This suggests that the intersection test in the preprocessing stage led to a significant number of false-allocated tiles. Therefore, improving the simplified intersection test to reduce such false allocations is a necessary area for future research.

B. Varying ranges of gaussians

The number of gaussians assigned to each tile varies significantly, which is attributed to the characteristics of the trained model. Particularly in lightweight models, there were many cases where specific regions contained very few Gaussians, as shown in the left-skewed distribution in the Fig. 11. In such cases, dividing and sorting Gaussians into buckets for rendering could worsen resource utilization efficiency. Fortunately, after performing Radix Sort, the number of Gaussians in each bucket can be determined in advance. When the total number of gaussians assigned to all buckets in a tile falls below a certain threshold, we configured Bitonic Sort to sort all tiles in a single operation, thereby mitigating performance degradation. However, this issue still has room for further optimization.

VI. Conclusion

Our works reveal that the proposed method consistently outperforms the baseline implementation of 3D Gaussian Splatting, achieving FPS improvements ranging from 1.59% to 8.09%. The hybrid sorting strategy showed adaptability to dataset-specific characteristics, with configurations optimized for varying Gaussian distributions and tile compositions. Our hybrid approach not only enhanced performance but also reduced memory usage compared to a Radix Sort solution, making it more suitable for memory-constrained environments like mobile devices and virtual reality systems. Furthermore, the bucket-based optimization introduced early termination, minimizing redundant computations and improving GPU resource utilization. The results demonstrate that carefully designed hybrid sorting mechanisms can mitigate the inherent inefficiencies of existing algorithms, providing scalable and efficient solutions for real-time rendering applications.

References

[1] Junseo Lee, Seokwon Lee, Jungi Lee, Junyong Park, and Jaewoong Sim. 2024. 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 (ASPLOS '24), Vol. 3. Association for Computing Machinery, New York, NY, USA, 497–511. https://doi.org/10.1145/3620666.3651385

[2] Kerbl, B., Kopanas, G., Leimkühler, T., & Drettakis, G. (2023). 3D Gaussian splatting for real-time radiance field rendering. ACM Transactions on Graphics, 42(4), Article 1. https://doi.org/10.1145/3592433