Atomic Aggregation on 3D Gaussian Splatting

Fan Chen, Keyi Zhang, Xin Peng

Abstract—Differential rendering has surfaced as a promising technique within the domain of visual computing applications, involving the training of a model through gradient descent from 2D images to represent a 3D scene. Recent research, exemplified by 3D Gaussian Splatting [1], integrates differential rasterization employing Gaussian primitives. This approach demonstrates superior rendering quality while concurrently upholding a high rendering speed. Despite these advancements, it is noteworthy that training such a model remains a time-consuming task, even with the utilization of robust GPUs. In the context of this study, we have identified a notable bottleneck associated with the substantial volume of atomic operations in gradient computation. These atomic operations overwhelm the atomic units within the L2 subpartitions, leading to long stalls. To ameliorate this issue, our approach leverages two key observations: (1) all active threads within the same warp collectively update the same memory location, and (2) only select threads in a warp initiate atomic updates. Our proposed methodology employs a warp-level primitive with low overhead to facilitate warp-level reduction, harnessing the inherent locality in intra-warp atomic updates. The experimental showcases substantial speed enhancements with an average improvement of 3.04× (and a peak improvement of 5.2×) for gradient computation. Furthermore, an average speedup of 1.58× (and a maximum speedup of 2.05×) is observed for the overall application.

```
1  #define FULL_MASK 0xffffffff
2  for (int offset = 16; offset > 0; offset /= 2)
3      val += __shfl_down_sync(FULL_MASK, val, offset);
```

Fig. 1: Typical implementation of reduction. [2]

1 Introduction

Differential rendering has emerged as a pivotal paradigm within the realm of computer graphics and visual computing. This innovative technique involves the computation of gradients with respect to scene parameters, enabling the generation of realistic images by efficiently capturing variations in lighting, geometry, and material properties. By differentiating through the entire rendering process, from scene representation to pixel colors, differential rendering offers a powerful framework for tasks such as image synthesis, novel view synthesis, and 3D scene reconstruction.

For example, recent advancements in Neural Radiance Field (NeRF) [3] technology have demonstrated significant promise in novel view synthesis. NeRF achieves this by optimizing deep fully-connected neural networks that take 5D coordinates as input, subsequently producing volume density and view-dependent RGB color. Despite the commendable rendering quality achieved by NeRF, its drawback lies in prolonged training and inference times, rendering is impractical for real-time applications. State-of-the-art techniques such as instant-ngp [4] have addressed this concern by leveraging a hash-grid data structure, resulting in remarkably accelerated convergence times. Another notable approach, termed 3D Gaussian Splatting [1], employs differential rasterization utilizing Gaussian primitives, demonstrating superior rendering quality while maintaining comparable convergence speeds to instant-ngp.

Rendering scenes with 3D Gaussian Splatting (3DGS) [1], can achieve high-speed performance through a raster-based rendering pipeline. However, it is noteworthy that the training of these models remains a computationally demanding process, particularly when leveraging powerful GPU architectures. In this work, we conduct an detailed performance analysis of the 3DGS application, revealing that a substantial bottleneck arises from the considerable volume of atomic updates during the backward pass. Specifically, our investigation discloses that the backward pass constitutes an average of 46.95% (with peaks reaching up to 59.20%) of the overall training time on a RTX 4090 GPU.

Within the gradient computation phase of 3DGS, individual threads are linked to specific pixels. Maintaining correctness requires these updates to occur atomically, given the potential for multiple threads to simultaneously modify the same set of parameters. However, as each thread is responsible for updating multiple parameters, this approach results in a significant proliferation of atomic updates. Consequently, this surge in atomic updates precipitates intense contention at the atomic units within the L2 memory subpartition, leading to extended stalls in the GPU streaming multiprocessors (SM).

The primary objective of our work is to enhance the efficiency of the training pipeline for 3DGS by consolidating atomic instructions within the backward pass. Two key observations guide our pursuit of this objective: (1) Threads within the same warp exclusively update identical memory locations. (2) Atomic updates are solely performed by a subset of threads within a warp: The number of threads engaging in gradient updates at any given time varies due to certain threads becoming inactive as a consequence of failed condition checks in the code.

In this work, we present a software-based strategy aimed at speeding up the training pipeline within 3DGS application. Our approach hinges on two principal concepts: (1) Exploiting intra-warp locality in atomic updates (Observation 1), we conduct warp-level reduction directly at the core by utilizing registers. (2) Aligned with Observation 2, we exclude warps that do not initiate any atomic updates. The software approach we propose leverages established warp-level primitives, such as `__shfl_down_sync`, to effectuate

warp-level reduction at each SM sub-core.

We evaluate our software-based approach on six different scenes. We demonstrate a speed up of 3.04x on average (and a peak improvement of 5.2×) for gradient computation and an average speed up of 1.58x (and a maximum speedup of 2.05×) on the overall 3DGS application on a real NVIDIA RTX 4090 GPU. Our contributions are summarized as follow:

- We conduct an exhaustive performance analysis on the training pipeline of 3DGS and discern atomic updates as a pivotal bottleneck.
- We introduce a software approach that uses warp-level reduction to reduce the number of atomic updates.
- We evaluate our approach on 3DGS application and demonstrate significant speed up.

2 Related Work

2.1 Atomic Processing in GPU

In GPU processing, atomic updates are typically associated with individual threads, where each thread is responsible for updating specific data or memory locations. To ensure correctness in a parallel computing environment, atomic operations are employed when multiple threads might attempt to modify the same memory location simultaneously. In the GPU architecture, every Streaming Multiprocessor (SM) comprises multiple sub-cores, each sends the dispatch of local and global atomic memory requests to the Memory I/O Units (MIO). These requests are subsequently routed through the interconnect to the memory subpartition and processed near L2 cache.

2.2 Warp Reduction

Warp reduction is a parallel computing technique commonly used in GPU programming to aggregate and combine data across threads within a warp. In the context of differentiable rendering or similar applications, it is often employed to reduce the number of atomic updates and enhance computational efficiency. Fig.1 shows how warp reduction is implemented using primitive `__shfl_down_sync`. A warp comprises 32 lanes, with each thread occupying one lane. For a thread at lane `X` in the warp, `__shfl_down_sync(MASK, val, offset)` gets the value of the `val` variable from the thread at lane `X+offset` of the same warp. The data exchange is performed between registers, and more efficient than going through shared memory, which requires a load, a store and an extra register to hold the address [2]. The thread at lane 0 will possess a `val` value equal to the sum of the `val` values across all 32 threads within the warp. It is important to note that for this process to execute accurately, both threads at lane X and at lane `X+offset` must be marked as active in the `MASK`. Traditional warp reduction employs a full mask, requiring the participation of all threads within the warp.

3 Motivation

3.1 Atomic Reduction Bottleneck

In this section, an in-depth performance analysis of 3DGS application is conducted. The profiling of the training pipeline is facilitated using Nsight System [5]. Our scrutiny reveals that the backward pass, specifically in the gradient computation step, constitutes the most time-consuming aspect, as illustrated in Fig. 2. Several key observations emerge from our analysis. Firstly, on average 47.0% (with a maximum of 59.2%) of the total execution time is allocated to gradient computation, establishing it as a significant bottleneck within the 3DGS application. Secondly, we discern that the time required for gradient computation escalates with scene size and complexity. In contrast, the forward pass and loss computation exhibit independence from scene complexity. Consequently, gradient computation emerges as a more pronounced bottleneck in scenarios characterized by increased scene intricacy.

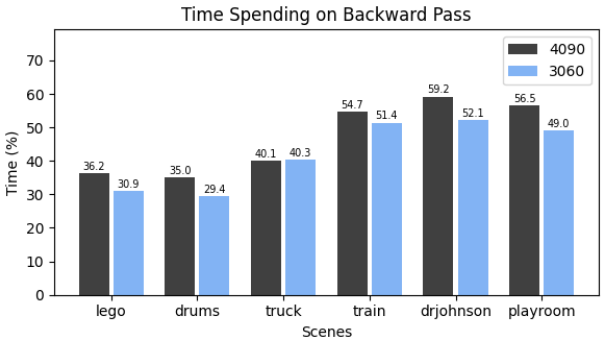

Fig. 2: The amount of time spending on backward pass using 4090 (left) and 3060 (right)

```
1:  function GradComputation(prims_per_thread)
2:      tid ← thread_idx                          ▷ Thread corr. to pixel
3:      for p : primitives[tid] do                                ▷ Iterate
4:          if cond1 then
5:              continue;          ▷ thread doesn't participate
6:          end if
7:          ...
8:          if cond2 then
9:              continue;          ▷ thread doesn't participate
10:         end if
11:         ...            ▷ Gradient computation is done here
12:         AtomicAdd(p.grad_x1, grad_t x1)
13:         AtomicAdd(p.grad_x2, grad_t x2)
14:         AtomicAdd(p.grad_x3, grad_t x3)
15:     end for
16: end function
```

Fig. 3: Outline of the gradient computation step

The input to the gradient computation kernel consists of a per-pixel list of primitives, where each list enumerates the IDs of primitives influencing the color of the corresponding pixel. The gradient computation within the backward step of 3DGS is illustrated in Fig.3. Each thread, representing one per pixel, iterates through its associated list of primitives (line 2, 3). Various intermediate conditions, denoted as $cond1, cond2$ in lines 5 and 9, determine the thread's contribution to the gradients of each primitive. Subsequently, the thread computes the gradient contribution of the primitive's parameters ($grad_t x1, grad_t x2, ...$). Ultimately, each thread executes an atomic add operation (depicted in lines 12-14) to atomically accumulate its gradient contributions to the parameters of the primitive. This atomic operation is

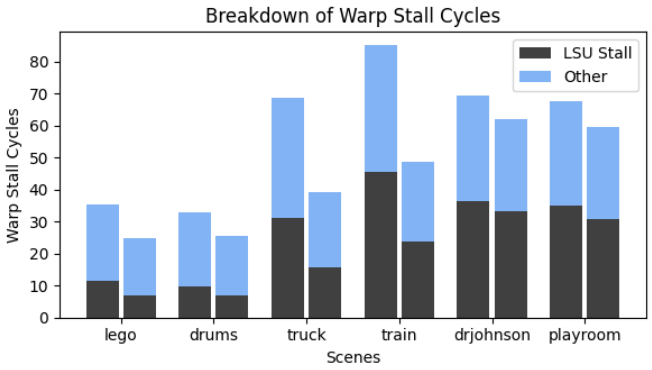

Fig. 4: Breakdown of warp stalls on `4090`(left), `3060`(right).

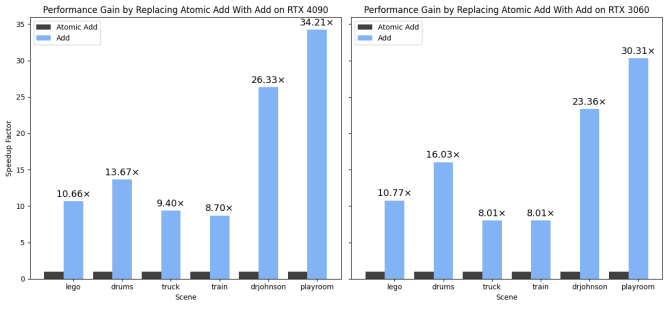

Fig. 5: The performance gain by replacing atomic_add with add 4090(left), 3060(right).

imperative due to the potential scenario where multiple threads may concurrently update the parameters of the same primitive.

Given that each thread is responsible for updating a multitude of primitives, each of which encompasses numerous learned parameters, a substantial volume of atomic operations is generated. To assess the repercussions of this, we analyze the cycles during the gradient computation step when instructions experience stalls on two GPUs. The breakdown of the number of cycles a warp is stalled per instruction on the NVIDIA RTX 4090 and RTX 3060 GPUs is illustrated in Fig.4, utilizing NVIDIA NSIGHT Compute [6]. Our analysis yields the following observation: LSU (Load-Store Unit) stalls contribute to over 60% of all stalls on average. These stalls within the LSU are attributed to the considerable number of memory requests, primarily in the form of atomic operations, directed to global memory from each sub-core.

We conduct an empirical investigation to quantify the impact of atomic updates on the overall performance of the training pipeline. Specifically, we substitute all instances of `atomic_add` in gradient computation with `add`, aiming to discern the influence of these atomic updates on performance dynamics. The outcomes of this experiment are presented in Fig. 5. The elimination of `atomic_add` yields a notable speedup of 17.16× on average (with a maximum of 34.21×). This compelling result underscores the substantial role played by atomic updates as a significant bottleneck in the context of gradient computation.

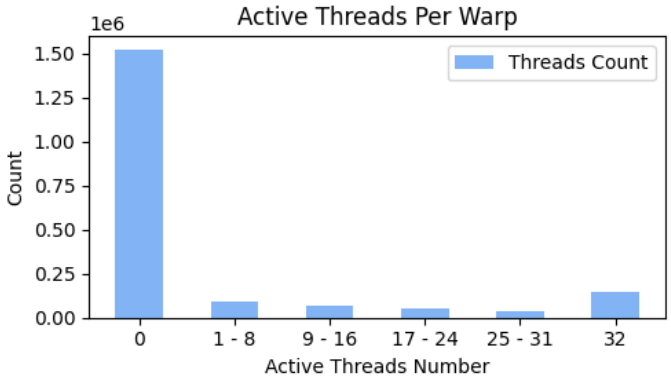

Fig. 6: Average number of active threads per warp participating in atomic updates.

3.2 Key Observation

We make the following observations from profiling atomic operations in the gradient computation step.

3.2.1 *Observation 1*

Threads within a common warp concurrently update a shared memory location. In the implementation of the gradient computation kernel, each engaged thread (as depicted in Fig. 3) is endowed with a global identifier. Importantly, this global identifier remains consistent across all threads within a given warp. Leveraging this shared global identifier, each thread retrieves the primitive from a shared array and subsequently performs atomic updates to modify the parameters associated with that primitive.

3.2.2 *Observation 2*

A subset of threads within a warp exclusively engages in atomic updates during any given instance. As evident from Fig. 3, the gradient computation step incorporates dynamic conditions $(cond1, cond2, ...)$ that lead certain threads to bypass the ongoing iteration of gradient updates. Consequently, only a fraction of the entire warp's threads initiate atomic requests during a singular iteration.

To quantify the extent of thread participation in the atomic reduction process for 3DGS, we examine the number of threads typically involved in atomic reduction, as illustrated in Fig. 6. Our observations reveal notable variability in the count of participating threads within a warp during a single reduction cycle. Consequently, each warp contributes a distinct level of traffic to the Load Store Unit, accentuating the non-uniformity in warp-level engagement in the atomic reduction operation.

In this work, our objective is to enhance the efficiency of the training pipeline in the context of the 3D Gaussian Splatting application. This enhancement specifically targets the acceleration of atomic operations, which emerge as a prominent bottleneck within the gradient computation step. In the subsequent section, we expound upon the utilization of these insights to formulate a software-based approach aimed at mitigating the identified bottleneck.

4 Proposed Method

We present a software-based methodology designed to facilitate rapid atomic reduction in the context of 3D Gaussian

```
1: function GRADCOMPUTATION(prims_per_thread)
2:     tid ← thread_idx                          ▷ Thread corr. to pixel
3:     for p : primitives[tid] do                                ▷ Iterate
4:         skip ← false
5:         if COND1 then
6:             skip ← true                       ▷ Mark inactive status
7:         end if
8:         ...
9:         if COND2 then
10:            skip ← true                       ▷ Mark inactive status
11:        end if
12:        ...
13:        if SKIP then
14:            grad_x1 ← 0
15:            grad_x2 ← 0
16:            grad_x3 ← 0
17:        end if
18:        REDUCTION(grad_t x1)
19:        REDUCTION(grad_t x2)
20:        REDUCTION(grad_t x3)
21:        if LANE_ID == 0 then
22:            ATOMICADD(p.grad_x1, grad_t x1)
23:            ATOMICADD(p.grad_x2, grad_t x2)
24:            ATOMICADD(p.grad_x3, grad_t x3)
25:        end if
26:    end for
27: end function
```

Fig. 7: Outline of the gradient computation step using reduction

```
1: function GRADCOMPUTATION(prims_per_thread)
2:     tid ← thread_idx                          ▷ Thread corr. to pixel
3:     for p : primitives[tid] do                                ▷ Iterate
4:         skip ← false
5:         if COND1 then
6:             skip ← true                       ▷ Mark inactive status
7:         end if
8:         ...
9:         if COND2 then
10:            skip ← true                       ▷ Mark inactive status
11:        end if
12:        ...
13:        if SKIP then
14:            grad_x1 ← 0
15:            grad_x2 ← 0
16:            grad_x3 ← 0
17:        end if
18:        active_count ←
   __popc(__ballot_sync(__activemask(), !skip))
19:        if !ACTIVE_COUNT then
20:            continue;                         ▷ warp doesn't participate
21:        end if
22:        REDUCTION(grad_t x1)
23:        REDUCTION(grad_t x2)
24:        REDUCTION(grad_t x3)
25:        if LANE_ID == 0 then
26:            ATOMICADD(p.grad_x1, grad_t x1)
27:            ATOMICADD(p.grad_x2, grad_t x2)
28:            ATOMICADD(p.grad_x3, grad_t x3)
29:        end if
30:    end for
31: end function
```

Fig. 8: Outline of the gradient computation step using reduction and skip non-participating warps

Splatting (3DGS). This approach is intricately tailored to address scenarios where the application generates substantial atomic requests, particularly emphasizing the intricacy arising from concurrent updates within the same warp targeting a shared memory location.

The central concept of this methodology is to exploit intra-warp locality to execute warp-level reduction prior to initiating any atomic updates. Rather than dispatching atomic updates individually, the proposed approach consolidates all updates and transmits a singular atomic update at the conclusion, a strategic maneuver facilitated by Observation 1.

4.1 Design Challenges

The occurrence of atomic updates across all threads within a warp is not guaranteed. As depicted in Fig. 3, certain threads may abstain from participating in the parameter update process due to unsuccessful conditions. Conventional warp-level primitives, such as `__shfl_down_sync()`, necessitate a mask input to specify participating threads, and it mandates the simultaneous activity of both thread i and thread (i + offset). Given the dynamic variation in the number of active threads within a warp, implementing an efficient warp-level reduction at the core level poses a substantial challenge.

4.2 Detailed Design

The conventional reduction technique necessitates thread convergence when `__shfl_down_sync()` is invoked. In order to realize this synchronization, we introduce a variable named `skip`. When a thread refrains from initiating an atomic update—rather than proceeding with divergence—we set `skip` to true. Conversely, when a thread initiates an atomic update, `skip` is assigned the value false. Threads with `skip` set to true are then assigned a value of 0, compelling their participation in the reduction process. Although this addition of 0 does not alter the final result, it does impact performance by introducing superfluous computation. Therefore, this approach attains optimal efficiency when the majority of threads are active, minimizing the introduction of redundant computations. The implementation details are illustrated in Fig. 7.

In addition to employing reduction to minimize the frequency of atomic requests, we utilize Observation 2 to further enhance the efficiency of the gradient computation step. Observation 2 elucidates that only a subset of threads within a warp actively engages in atomic updates. Our findings indicate the existence of certain warps that entirely abstain from sending any atomic updates. In such cases, we enforce thread convergence and execute reduction on 0, followed by an unnecessary atomic addition on 0. To address this inefficiency, prior to commencing the reduction process, we leverage warp-level primitives such as `__ballot_sync()` and `__popc()` to ascertain the number of threads within the warp that will participate in updating the primitive. If this count is determined to be 0, we expedite the process by advancing to the next iteration of the loop. The implementation details are illustrated in Fig. 8.

TABLE 1: CPU system configuration

CPU 3.6GHz Rocket-lake-like, OOO 4-wide dispatch window, 128-entry ROB; 32 entry LSQ
L1D + L1I Cache 2.3MB, 4 way LRU, 1 cycle; 64 Byte line; MSHR size: 10; stride prefetcher
L2 Cache 32MB, 8 way LRU, 4 cycle; 64 Byte line; MSHR size: 10; stride prefetcher
L3 Cache 36MB, 16 way LRU, 20 cycle; 64 Byte line; MSHR size: 64; stride prefetcher
DRAM 2-channel; 16-bank; open-row policy, 4GB DDR4

TABLE 2: Workloads and datasets

Workloads	Dataset identifier	Dataset name
3DGS [1]	lego	NerfSynthetic-Lego [3]
	drums	NerfSynthetic-Drums [3]
	playroom	DB COLMAP Playroom [7]
	drjohnson	DB COLMAP DR. Johnson [8]
	truck	Tanks and Temples-Truck [9]
	train	Tanks and Temples-Train [9]

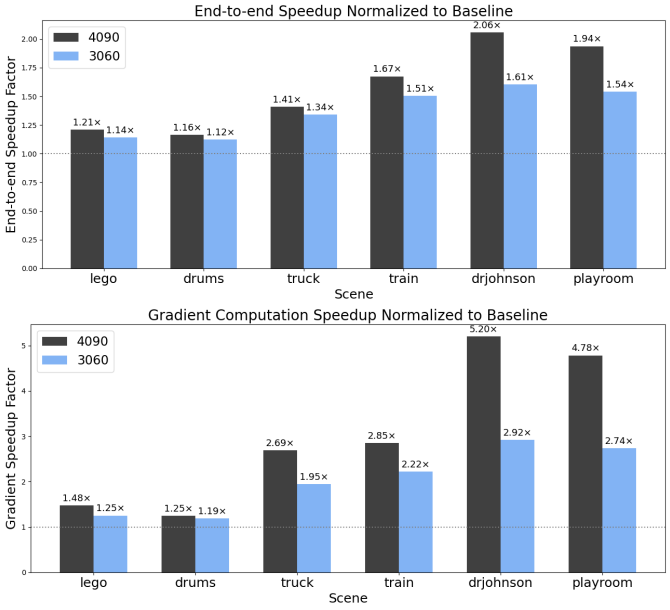

Fig. 9: End-to-end and gradient computation speedup normalized to baseline on 4090 and 3060.

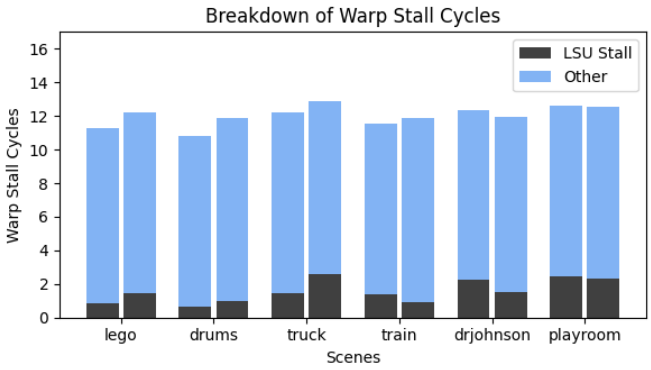

Fig. 10: Breakdown of warp stalls during gradient computation on 4090 (left) and 3060 (right).

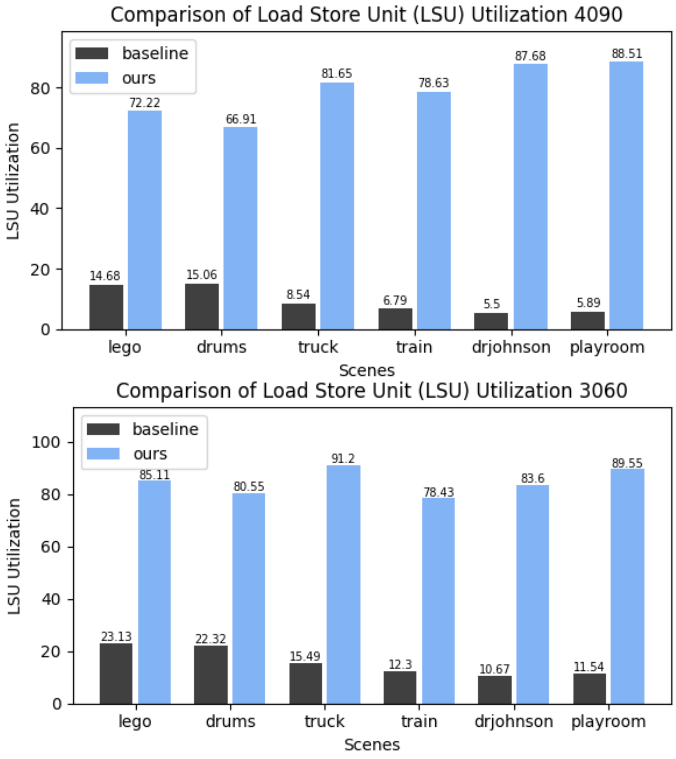

Fig. 11: Comparison of Load Store Unit (LSU) utilization between the baseline and our proposed approach.

5 Experimental Results

We implement and assess our software-based approach on real hardware setups with an Intel Core i9 13900KF CPU (specifications shown in Table 1) and the NVIDIA RTX 4090 and RTX 3060 GPUs.

We assess the efficacy of our software approach utilizing the datasets enumerated in Table 2.

Fig.9 illustrates the normalized speedup for the end-to-end runtime, encompassing the forward pass, as well as the normalized speedup specifically for the gradient computation. The presented speedups in both graphs are realized on real hardware and normalized in relation to the baseline.

Fig.10 illustrates the average number of warp stalls per instruction, along with its breakdown on both the RTX 4090 and RTX 3060. Additionally, Fig.11 presents the Load Store Unit utilization for both the baseline and our proposed approach on the RTX 4090 and RTX 3060. Furthermore, Table 3 shows the Peak Signal-to-Noise Ratio (PSNR) results for six datasets, acquired during a 5-minute training period on both the RTX 4090 and RTX 3060, employing both the baseline and our approach.

Firstly, our software approach consistently surpasses the baseline performance on both GPUs. Specifically, for the gradient computation, we achieve an average speedup of 3.04× (with a maximum of 5.2×) on the RTX 4090 and 2.04× (with a maximum of 2.92×) on the RTX 3060. In the broader context of the entire training pipeline, our approach secures an average speedup of 1.58× on the RTX 4090 (with a maximum of 2.05×) and 1.37× (with a maximum of 1.61×) on the RTX 3060.

Secondly, notable increases in speedup are observed, particularly in scenes labeled playroom and drjohnson. The datasets corresponding to these scenes are characterized

TABLE 3: PSNR on different approaches.

	4090		3060	
	Baseline	**Ours**	**Baseline**	**Ours**
lego	35.793	35.682	33.708	34.264
drums	29.990	30.124	28.856	29.034
playroom	32.843	34.975	29.988	30.195
drjohnson	29.549	32.160	28.025	29.076
truck	24.779	25.851	23.723	24.170
train	23.591	23.638	19.393	21.371

by their largescale and photorealistic nature, necessitating a greater number of geometric primitives (specifically, Gaussians for 3D Gaussian Splatting) for accurate scene representation in comparison to smaller scenes. Consequently, this results in a higher count of parameters requiring atomic updates during gradient computation, thereby accentuating the impact of the atomic bottleneck.

Thirdly, a notable reduction in warp stall cycles is evident, as depicted in Figure 10. On average, the warp stall cycles decrease by a factor of $4.28\times$ (with a maximum reduction of $7.38\times$). The proportion of warp stalls attributed to LSU stalls exhibits an average decrease of $3.6\times$ (with a maximum of $6.41\times$). In alignment with this observation, the utilization of the Load Store Unit (LSU) is markedly higher in our approach compared to the baseline. On average, the LSU utilization using our approach is $8.05\times$ higher (with a maximum of $15.94\times$). This increase in utilization is attributed to the reduction in the number of atomic updates being sent to the LSU.

Fourthly, to underscore the impact of our approach on rendering results, we conduct model training on datasets for a duration of 300 seconds, employing both our approach and the baseline. The outcomes are elucidated in Table 3. We discern an average increase of 0.858 dB in Peak Signal-to-Noise Ratio (PSNR), with a maximum improvement of 2.611 dB. Notably, more complex scenes exhibit a more pronounced increase in rendering results, as they derive greater benefits from our approach.

6 Conclusion

In this work, we present a software-based methodology designed to expedite the execution of atomic reduction operations in the context of 3D Gaussian Splatting. The fundamental concept underpinning our approach involves harnessing established primitives, such as `__shfl_down_sync()`, to conduct warp-level reduction prior to initiating atomic updates. Additionally, we strategically omit the involvement of warps that refrain from participating in parameter updates, guided by the observation that a considerable proportion of warps abstains from transmitting any atomic updates. Our findings substantiate that our proposed approach effectively mitigates the bottleneck associated with atomic processing in 3D Gaussian Splatting applications.

References

[1] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis, "3d gaussian splatting for real-time radiance field rendering," *ACM Transactions on Graphics*, vol. 42, no. 4, July 2023. [Online]. Available: https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/

[2] "Using cuda warp-level primitives," https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/, accessed: 2023-11-20.

[3] 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.

[4] T. Müller, A. Evans, C. Schied, and A. Keller, "Instant neural graphics primitives with a multiresolution hash encoding," *ACM Transactions on Graphics (ToG)*, vol. 41, no. 4, pp. 1–15, 2022.

[5] "Nvidia nsight systems," https://developer.nvidia.com/nsight-systems.

[6] "Nvidia nsight compute," https://developer.nvidia.com/nsight-compute, accessed: 2023-11-20.

[7] J. Abramson, A. Ahuja, I. Barr, A. Brussee, F. Carnevale, M. Cassin, R. Chhaparia, S. Clark, B. Damoc, A. Dudzik *et al.*, "Imitating interactive intelligence," *arXiv preprint arXiv:2012.05672*, 2020.

[8] S. Prakash, T. Leimkühler, S. Rodriguez, and G. Drettakis, "Hybrid image-based rendering for free-view synthesis," *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 4, no. 1, pp. 1–20, 2021.

[9] A. Knapitsch, J. Park, Q.-Y. Zhou, and V. Koltun, "Tanks and temples: Benchmarking large-scale scene reconstruction," *ACM Transactions on Graphics (ToG)*, vol. 36, no. 4, pp. 1–13, 2017.