ARC: Warp-level Adaptive Atomic Reduction in GPUs to Accelerate Differentiable Rendering

Sankeerth Durvasula*
University of Toronto
Vector Institute
Toronto, Canada

Adrian Zhao*
University of Toronto
Vector Institute
Toronto, Canada

Fan Chen
University of Toronto
Toronto, Canada

Ruofan Liang
University of Toronto
Vector Institute
Toronto, Canada

Pawan Kumar Sanjaya
University of Toronto
Vector Institute
Toronto, Canada

Yushi Guan
University of Toronto
Vector Institute
Toronto, Canada

Christina Giannoula
University of Toronto
Vector Institute
Toronto, Canada

Nandita Vijaykumar
University of Toronto
Vector Institute
Toronto, Canada

Abstract

Differentiable rendering is widely used in emerging applications that represent any 3D scene as a model trained using gradient descent from 2D images. Recent works (e.g., 3D Gaussian Splatting) use rasterization to enable rendering photo-realistic imagery at high speeds from these learned 3D models. These rasterization-based differentiable rendering methods have been demonstrated to be very promising, providing state-of-art quality for various important tasks. However, training a model to represent a scene is still time-consuming even on powerful GPUs. In this work, we observe that the gradient computation step during model training is a significant bottleneck due to the large number of *atomic operations*. These atomics overwhelm the atomic units in the L2 cache of GPUs, causing long stalls.

To address this, we leverage the observations that during gradient computation: (1) for most warps, all threads atomically update the same memory locations; and (2) warps generate varying amount of atomic traffic. We propose ARC, a primitive that accelerates atomic operations based on two key ideas: First, we enable warp-level reduction at the GPU cores using registers to leverage the locality in intra-warp atomic updates. Second, we distribute atomic computation between the cores and the L2 atomic units to increase the throughput of atomic computation. We propose two implementations of ARC: ARC-HW, a hardware-based approach and ARC-SW, a software-only approach. We demonstrate significant speedups with ARC of 2.6× on average (up to 5.7×) for widely used differentiable rendering workloads.

CCS Concepts: • **Computer systems organization** → **Architectures**; • **Computing methodologies** → **Rendering**; *Machine learning*.

Keywords: Differentiable Rendering, Atomics, Gaussian Splatting, Machine Learning, Graphics Processing Unit

ACM Reference Format:
Sankeerth Durvasula, Adrian Zhao, Fan Chen, Ruofan Liang, Pawan Kumar Sanjaya, Yushi Guan, Christina Giannoula, and Nandita Vijaykumar. 2025. ARC: Warp-level Adaptive Atomic Reduction in GPUs to Accelerate Differentiable Rendering. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3669940.3707238

*Both authors contributed equally to this research.

ASPLOS '25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-0698-1/25/03
https://doi.org/10.1145/3669940.3707238

1 Introduction

Learned 3D scene representations have been a fast-advancing field in recent years. For example, Neural Radiance Fields (NeRFs) [22, 23, 28, 43, 50, 73, 76, 79, 93, 95, 110] and 3D Gaussian Splatting (3DGS) [26, 36, 46, 54–57, 71, 72, 74, 90, 96, 100, 101, 103, 105–107, 109, 114, 115] have demonstrated new capabilities with applications in many domains including entertainment, health care, robotics, and AR/VR. These 3D scene representations comprise *learned parameters* and provide state-of-art quality in scene reconstruction, among

many other advantages, compared to traditional explicit representations (e.g., meshes and point clouds). They have thus emerged as a transformative representation of 3D visual data, leading to high interest in the computer graphics and robotics communities.

Differentiable rendering is a key technique in 3D scene representations that leverages machine learning to solve fundamental tasks in computer graphics, such as scene reconstruction [55, 76] (deriving a 3D scene representation), and inverse rendering [80, 81] (estimating the shape, texture, lighting, and material of 3D objects) from a set of captured reference images. These tasks are fundamental to many important applications [97], such as photogrammetry, 3D modeling and scanning, 3D model creation tools, game engines, and AR/VR applications. With differentiable rendering, these tasks are formulated as a learning problem that can be solved using gradient descent-based optimization techniques.

NeRF [22, 23, 28, 43, 50, 73, 76, 79, 93, 95, 110] is a popular differentiable rendering technique where each scene is represented as a set of learned parameters/neural networks. These neural parameters are trained with gradient-based optimization to minimize the difference between ground truth images and rendered images from the NeRF model. The success of this approach has led to the development of several specialized frameworks/libraries for differentiable rendering [52, 63, 81, 83], native support for differentiable rendering in GPUs [21, 49], and accelerators for NeRF-based rendering and training [66, 67, 69, 77, 102]. NeRFs, however, use ray marching, which can be inefficient in rendering images.

More recent advancements in differentiable rendering over NeRFs are approaches such as 3DGS [26, 36, 46, 54–57, 71, 72, 74, 90, 96, 100, 101, 103, 105–107, 109, 114, 115] that leverage the high-speed *rasterization* pipeline [20] instead of ray marching. Rasterization requires the scene to be represented as a set of *geometric primitives* (i.e., meshes, triangles, points) in 3D space which can be rendered as 2D images at high speeds. Differentiable rendering with rasterization involves learning these primitives using similar gradient descent-based training. This approach [41, 55, 64, 80, 87] has demonstrated state-of-the-art capability in producing high-quality scene reconstructions at high speeds. 3DGS represents the scene with 3D Gaussian densities as its primitives and uses an efficient rasterizer [55] to render images.

The raster-based pipeline enables high-speed rendering (i.e. the forward pass). However, *training* raster-based models is still much slower compared to rendering. Training involves a forward pass (rendering an image), loss computation, gradient computation, and applying gradient updates to these learned parameters. A new model has to be trained for *each individual scene*. We perform a detailed performance analysis of training and find that the gradient computation step of the backward pass is a significant bottleneck. For example, in 3DGS workloads (see §6 for details), the gradient computation kernel takes on average 51.9% (up to 65.8%) of the overall training time on the NVIDIA RTX 4090 GPU (§3).

Our analysis shows that this bottleneck is primarily caused by a large number of atomic operations that accumulate gradients for the trained parameters. During gradient computation, each thread is associated with one pixel. The gradients for all primitives associated with the pixel are computed and accumulated. These gradient updates are performed using atomic operations, since multiple threads may update the same parameters. This leads to a massive number of atomic operations launched by each thread. These atomic operations cause significant contention at the atomic units in the L2 memory subpartitions (ROP units), causing long stalls at the streaming multiprocessors (SMs) (§3.2).

Our **goal** in this work is to accelerate raster-based differentiable rendering applications by accelerating atomic operations that constitute a significant bottleneck during gradient computation. From our analysis of atomic operations in gradient computation, we make two observations: **(1) Locality in intra-warp atomic updates:** Threads *within a warp* typically update the same parameters and thus the *same memory location*. We find that over 99% of warps have all their threads update the same memory location (§3.1). **(2) Only a subset of threads in a warp perform atomic updates:** There is significant variation in the number of threads within each warp that make gradient updates during gradient computation (§3.1) as some threads are made inactive due to failing condition checks in the code (i.e., control divergence). The number of threads executing atomics determines the atomic request traffic generated by the warp and varies across warps. Prior GPU approaches [19, 30, 32, 78] that aim to address bottlenecks due to atomic requests buffer and aggregate atomic updates in the L1 cache/scratchpad memory/local SRAM to reduce traffic in the interconnect and L2 atomic units (ROP units). While these approaches can partially alleviate atomic overheads, they do not fully leverage the intra-warp locality in atomic updates seen in differentiable rendering. Thus, the sheer number of atomic requests overwhelm the load-store units before the atomics can be aggregated at L1, making this approach less effective for differentiable rendering workloads (See §7.1 and §8).

In this work, we introduce Adaptive Atomic ReduCtion (ARC), a primitive that accelerates atomic updates in applications that (1) generate significantly large amounts of atomic requests and (2) typically have most threads within an warp performing atomic updates to the same memory locations. ARC is based on two key ideas: **(1)** We leverage intra-warp locality in atomic updates (Observation 1) to perform *warp-level reduction* at the core itself using registers. This significantly reduces the number of atomic operations that need to be sent to the L2 atomic units to update global memory. **(2)** We dynamically distribute the atomic computation between the cores and L2 atomic units to enable high throughput atomic updates by employing all atomic units.

Leveraging Observation 2, warps that only generate a few atomic updates are handled at the L2 atomic units. Warps, where most/all threads generate atomic updates, are first reduced at the SM using the proposed warp-level reduction. Implementing ARC requires addressing important design challenges (§4.1). We propose two ARC implementations: **ARC-HW**. We propose a hardware-software primitive that uses a low-overhead hardware module in the SMs to perform warp-level reduction using registers and leveraging the address coalescing unit. To distribute atomic computation between the SM and ROP units, we dynamically schedule the atomic computations based on their utilization. ARC-HW is exposed to the programmer using a new ISA instruction.

ARC-SW[†]. This software-only approach leverages existing warp-level primitives (such as `shfl`) to implement warp-level reduction at each SM sub-core. All atomic updates to any memory location involving more than a predefined number of threads in a warp are performed at the SM, and the others are performed at the ROP units. This predefined number is a tunable hyperparameter (the *balancing threshold*). ARC-SW can be directly used in many modern GPUs, but ARC-HW is more efficient as there are no overheads from additional instructions, redundant computation, or control flow; it requires less programmer intervention; and does not rely on instructions specific to some GPUs (§4.3).

We evaluate ARC across recent widely used differentiable rendering applications (3D Gaussian Splatting [55], NVDiffRec [80], Pulsar [64, 83]). With ARC-HW, we demonstrate an average speedup of 2.1× (up to 8.6×) for gradient computation using GPGPU-Sim GPU simulator [58]. With ARC-SW, we demonstrate a speedup of 1.8× on average (up to 7.2×) for gradient computation on the simulator. On a real NVIDIA RTX 4090 GPU, we demonstrate a speedup of 2.6× on average (up to 5.7×) on gradient computation, and an average end-to-end speedup of 1.4× (up to 2.4×). Our contributions are summarized as follows:

- This is the first work to perform a performance characterization of an important workload, rasterization-based differential rendering for 3D visual data including Gaussian Splatting, and identify atomic updates as a key bottleneck.
- We introduce ARC, a primitive to accelerate atomic computations in GPUs for raster-based differentiable rendering.
- We propose two implementations: ARC-HW, a low-overhead hardware atomic reduction framework, and ARC-SW, a SW-only implementation. We open source ARC-SW [12], which can be directly used to obtain significant speedups in raster-based differentiable rendering workloads.
- We evaluate ARC on both real hardware and using a simulator and demonstrate significant speedups for widely-used differentiable rendering applications.

[†]The open-source code repository for `ARC-SW` is publicly available at https://github.com/Accelsnow/gaussian-splatting-distwar, along with the extended original technical report [37].

2 Background

2.1 Atomic Processing in GPUs

Figure 1 shows a Streaming Multiprocessor (SM) of a GPU [3, 5, 6]. Each SM consists of multiple sub-cores ❶ with its own warp scheduler, register file, and execution units. Each sub-core sends local, global, and atomic memory requests to the MIO (Memory I/O Unit), which interfaces with the caches and memory subsystem through a queue [13] (sometimes called L1 instruction queue ❷). In this work, we refer to the unit that dispatches requests from the sub-cores to the caches and memory subsystem as the Load-Store Unit (LSU) (consistent with NVIDIA's NSIGHT terminology [13]). Atomic operations sent to the LSU are issued to the memory sub-partition ❸ via the interconnect. The memory subpartition contains compute units (known as ROP units) [5, 6, 17] that process the atomic requests in the L2 cache. The L2 cache is shared across all SMs [47, 89]. A large number of atomic requests may lead to contention at the ROP units.

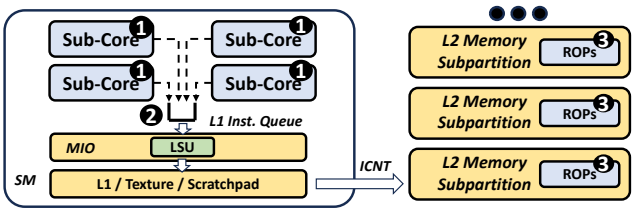


Figure 1. Atomic processing in a GPU.

2.2 Differentiable Rendering for 3D Scene Reconstruction

Differentiable rendering is the backbone of important applications, including 3D modeling/reconstruction [48, 108], perception in autonomous driving [104, 112], SLAM [75, 103], AI-generated content [96], and medical imaging [70], and provides a powerful fundamental representation for 3D data. We describe differentiable rendering using a classic problem in graphics: 3D scene reconstruction, which involves creating a 3D scene representation from 2D images. 3D reconstruction has important applications in novel view synthesis, 3D scanning/modeling, and photogrammetry. With differentiable rendering, the scene is represented using a set of parameters (i.e., model) that are learned using gradient descent, similar to standard deep learning training. This process of training a model to represent a 3D scene is depicted in Figure 2.

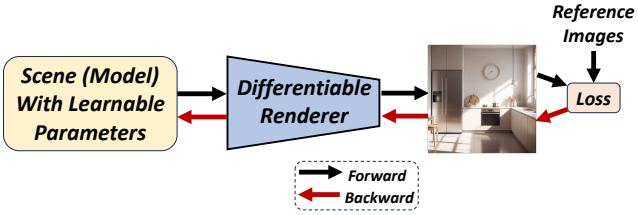


Figure 2. A generalized differentiable rendering training pipeline to train a *model* to learn a 3D scene.

The differentiable renderer produces 2D images of the represented 3D scene at different viewpoints during the forward pass (Figure 2). The difference between a rendered image and its corresponding reference/ground truth image (i.e., *loss*) is obtained by subtracting their RGB values. This loss is backpropagated to calculate gradients for all model parameters that minimize the loss using gradient descent-based optimization (the backward pass in Figure 2). This process is repeated for images from all viewpoints. Examples, also referred to as *implicit representations*, include neural radiance fields (NeRFs) [76] and 3D Gaussians [55]. These models represent scenes with differentiable, compact representations and achieve the state-of-art performance in novel-view synthesis. The high-quality model reconstruction and rendering, even from few input images, has spurred tremendous interest in vision and graphics communities and these learned models have emerged as a transformative representation for 3D scenes. These representations do have shortcomings compared to traditional representations (e.g., meshes, point clouds) as they are difficult to edit, can be computationally expensive, and learn surfaces inaccurately. Addressing these challenges is a very active area of ongoing research [23, 25, 39, 42, 65, 82, 100, 106].

2.3 Differentiable Rendering with Rasterization

NeRFs (neural radiance fields) are a differentiable rendering-based representation, but use *ray marching* to render images in the scene. More recent advances in differentiable rendering [55, 63, 64, 83, 87] propose to use *rasterization* instead of ray marching, enabling the rendering of images at much higher speeds. Rasterization requires the scene to be composed of several discrete 3D geometric elements, or *primitives* (e.g., triangles, points, ellipsoids). Each of these primitives is associated with shading information (e.g., color, opacity) and a position in space. Figure 3 depicts how these primitives ❶ (ellipsoids in this example) are rendered into 2D images ❷. Each pixel of the rendered image is thus influenced by a subset of the primitives in the scene. With differentiable rendering, all primitives are associated with a set of *learnable parameters* ❸ that are trained using gradient descent. For each training iteration (i.e., one image), the loss ❹ is backpropagated ❺ to compute the gradients for all the parameters associated with each primitive ❻ (only primitives that influence the current image). These parameters are updated with the computed gradients ❼, and the training iterations continue until convergence is achieved (i.e., the primitives are able to accurately represent the scene from all angles). 3D Gaussian Splatting [55], a state-of-art raster-based differentiable rendering method, models scenes with 3D Gaussians (seen as ellipsoids) as the geometric primitives.

3 Motivation

In this section, we profile important raster-based differentiable rendering workloads on the NVIDIA RTX 4090 GPU

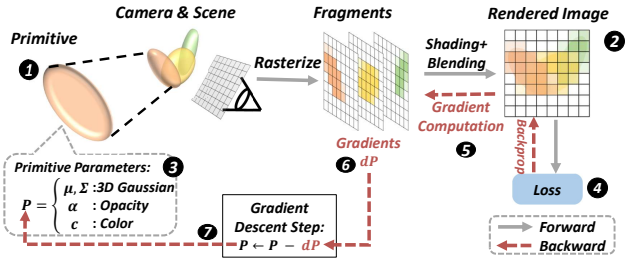


Figure 3. Differentiable rendering flow with rasterization.

(methodology is described in §6). Figure 4 depicts the breakdown of training time, including the forward pass (during which an image is rendered from the model), loss calculation (which involves computing the difference between ground truth and rendered image), and the gradient computation (which involves computing and updating the loss gradient with respect to model parameters). We make the following observations. First, we observe that on RTX 4090 on average 44% (up to 66%) of the total execution time is spent on the gradient computation step and is thus a significant bottleneck in most workloads. Second, this bottleneck is most pronounced for workloads such as 3D-DR and 3D-PL (see §6), taking up 65.8% and 62% of the overall runtime, respectively. This is because DR and PL are real-world scenes that require a large number of primitives (i.e., a large model) for accuracy. There is a larger increase in gradient computation time with scene size and complexity compared to the forward pass. This is because the forward pass has more parallelism that is extracted as the number of primitives increases, but gradient computation is limited by atomic operations, thus becoming a bigger bottleneck in more complex scenes.

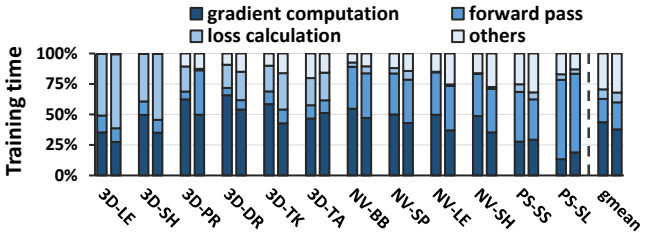


Figure 4. Training time breakdown on 4090 (left) and 3060 (right)

Figure 5 depicts the algorithm for the gradient computation step of differentiable rendering workloads. The input to the gradient computation kernel is a per-pixel list of primitives, where each list contains the IDs of primitives that influence the color of the corresponding pixel (discussed in §2.3). Each thread (one per pixel) iterates through a list of its associated primitives (line 2, 3). Several intermediate conditions (like $cond1, cond2$ in lines 5 and 9) determine if the current thread contributes to each primitive's gradients. Each thread then computes the gradient contribution of the primitive's parameters ($grad_t x1, grad_t x2, ...$). Finally, each

thread atomically adds its gradient contributions to the primitive's parameters (shown in lines 12-14).

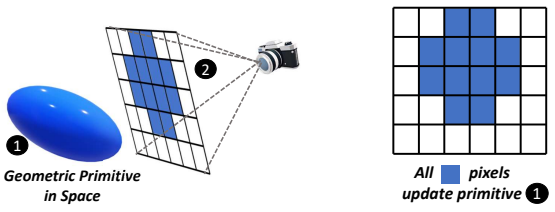
```
1:  function GradComputation(prims_per_thread)
2:      tid ← thread_idx                    ▷ Thread corr. to pixel
3:      for p : primitives[tid] do                    ▷ Iterate
4:          if cond1 then
5:              continue;          ▷ thread doesn't participate
6:          end if
7:          ...
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
```

Figure 5. Execution flow of the gradient computation step

3.1 Characteristics of Gradient Computation Step

We analyze the characteristics of the gradient computation step using real differentiable rendering workloads (See §6). We make two key observations from profiling the atomic operations in the gradient computation step:

Observation 1: Threads within a warp are likely to update the same parameters. Each primitive affects a region of pixels on the screen, called a "fragment" (§2.3). As a result, close-by pixels that belong to the same fragment update the same primitive. Figure 6 shows how adjacent/close-by pixels are part of the same fragment during rasterization. Figure 6a shows a primitive ❶ rasterized onto a screen ❷ as seen from the camera indicated by the blue pixels during rendering. In the gradient computation step, each of these blue pixels affected will update the primitive's gradient. Figure 6b shows a zoomed-in version of the captured image.

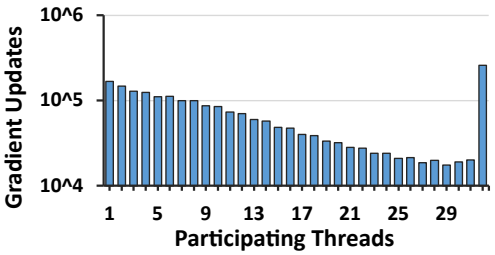


(a) Close-by pixels likely to be influenced by same primitive. **(b)** Gradients of affected pixels are atomically aggregated.

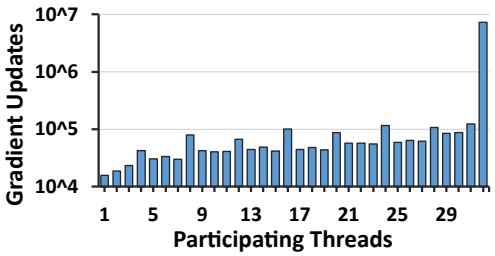
Figure 6. Close-by threads (corresponding to pixels) update the parameters of the same primitive.

Thus, threads within a warp (where each thread corresponds to one pixel and each warp corresponds to a local region of pixels) often compute the gradients for the parameters associated with the same primitive. This explains why these gradients must be *atomically* summed up across threads to update each parameter (Figure 5 line 12-14). We perform an experiment to determine the number of threads in each active warp that updates the same parameters and thus, the same memory locations. For 3D-PL (see §6 for workload details), we find that over 99% of warps have all their threads update the same memory location.

Observation 2: Only a fraction of threads within a warp perform atomic updates at any given time. From Figure 5, we see that the gradient computation step has certain dynamic conditions (*cond*1, *cond*2, ...) that cause some threads to skip the current iteration of gradient updates. Thus, only a fraction of all threads within a warp sends out atomic requests in one iteration. We measure the number of threads that typically participate in the atomic reduction in Figure 7 for two different workloads 3D-PR and NV-LG (see §6 for workload details). We observe that there is significant variation in the number of threads in a warp that participate in one reduction. Thus, each warp contributes a different amount of traffic to the LSU and the ROP units.



(a) 3D-PR workload



(b) NV-LG workload

Figure 7. Log-scale histograms of average number of active threads per warp that participate in gradient updates.

3.2 Atomic Bottleneck in the Gradient Computation

Given that each thread updates a number of primitives, each of which has many learned parameters, a massive number of atomic operations is generated. To evaluate the impact of this, we analyze the cycles during the gradient computation step when instructions are stalled from executing. Figure 8 depicts the breakdown of the number of cycles a warp is stalled per instruction using NVIDIA Nsight profiler [13] on two GPUs. We make two key observations: First, the LSU (load-store unit) stalls contribute over 60% of all stalls on average. The LSU stalls are caused due to the large number of memory requests (primarily atomic operations) to global

memory from each sub-core (§2.1). These workloads also have high cache hit rates (97% L2 hit rate on average on both RTX 4090 and RTX 3060 across all workloads), indicating that the memory stalls are not caused by cache misses, but atomic operations. Second, the RTX 4090 GPU has more stalls in issuing instructions to the LSU compared to the RTX 3060. This is because more recent GPUs have a higher SM to ROP unit ratio. In our experimental setup, the RTX 4090 has 4.57× more SMs than the RTX 3060 (128 SMs and 28 SMs, respectively). However, the RTX 4090 only has about 3.6× more ROP units (176 ROP units versus 48 ROP units).

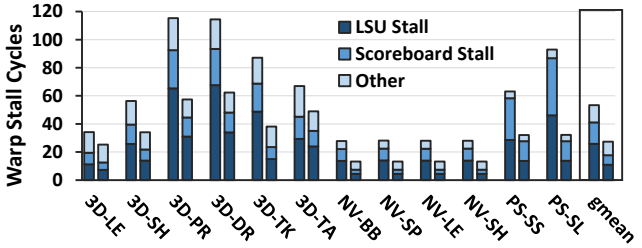


Figure 8. Breakdown of warp stalls on 4090 (left) and 3060 (right).

To this end, our **goal** is to enable faster raster-based differentiable rendering methods by accelerating atomic operations that constitute a significant bottleneck.

4 Approach

We introduce ARC, a primitive that enables fast atomic reduction in applications that **(1)** generate a large number of atomic requests, thus overwhelming the hardware queues and compute units that process atomics, and **(2)** typically have most threads within an warp performing atomic updates to the same memory locations.

The **key ideas** behind ARC is to (i) leverage the intra-warp locality in atomic updates to perform warp-level reduction in the SM itself using registers, and (ii) distribute atomic computation between the SM and L2 ROP units to enable high throughput atomic reduction. A high-level overview of ARC is shown in Figure 9. At each sub-core, ARC's scheduler ❶ determines whether each atomic update should be performed locally as warp-level reduction in the core (❸ and ❹) or using the atomic units at the L2 ROPs. ❷.

We propose two implementations of ARC: **(i)** ARC-HW, a hardware-software cooperative approach that efficiently implements warp-level reduction at each sub-core and automatically distributes reduction computation between the core and L2 ROP units, and *(ii)* ARC-SW, a software-only approach that leverages existing warp-level primitives in some GPUs to implement the ideas in ARC. ARC-SW can be used directly in many existing GPUs using our open-source implementation. However, ARC-HW is more efficient because it eliminates additional instruction overheads from handling divergence and control flow in software, can be implemented

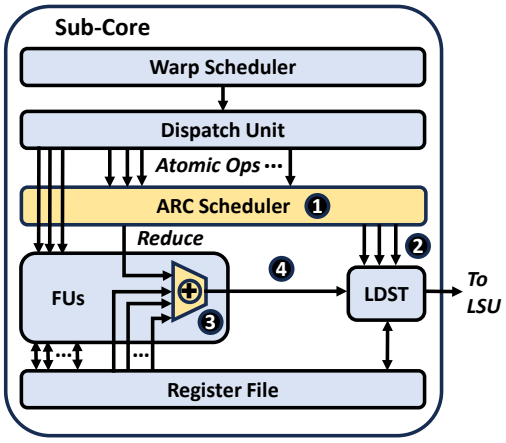


Figure 9. High-level overview of ARC. FUs refers to functional units (e.g., IADD, FFMA), and LDST refers to memory units within each sub-core that handles and forwards memory requests to the LSU.

flexibly in GPUs that do not support these primitives (e.g., Intel GPUs), and requires less programmer effort (See §4.5).

4.1 Design Challenges of ARC

ARC addresses two design challenges:

Challenge ❶: All threads in warp may not generate atomic updates. Only a subset of threads in a warp typically generate atomic updates at any given time (as discussed in §3.1). Existing warp-level instructions thus cannot be directly used to perform warp-level reduction for differentiable rendering workloads. This irregularity poses challenges in developing an efficient implementation of warp-level reduction at the core level.

Challenge ❷: Dynamic scheduling of atomic computation between the core and L2. To meet the high throughput requirements for atomic computations in differentiable rendering, it is critical to effectively use both existing ROP units at the L2 as well as the proposed warp-level reduction at the sub-cores. Thus, ARC must schedule the atomic operations efficiently at runtime based on the utilization of the atomic units at the sub-cores and the L2.

4.2 Warp-Level Reduction with Primitives and Software Libraries

Existing shuffle primitives do not take predicate registers and do not have logic masks for filtering active threads. The mask argument in `shfl_sync` primitive [11] is a synchronization mask; it does not prevent fetching from or saving to inactive threads, thus causing undefined behavior. An "if" condition does not solve the problem, as the inactive threads have already diverged due to earlier condition checks. CUDA software libraries such as CCCL [14] and CUB [15] provide highly optimized implementations for warp-level reduction using existing instructions, but these libraries assume that *all*

threads within a warp are active. For differentiable rendering workloads, all threads often do not participate in reduction. These approaches do not address either Challenge ❶ or ❷. We quantitatively compare ARC with warp-level reduction of the CCCL library in §7.2.

4.3 Key Components of ARC-HW

ARC-HW uses hardware support to perform warp-level reduction and automatically distributes atomic computation between the SMs and ROP units. ARC-HW is exposed to the programmer with a new instruction, named `atomred`. We next describe the workflow of ARC-HW:

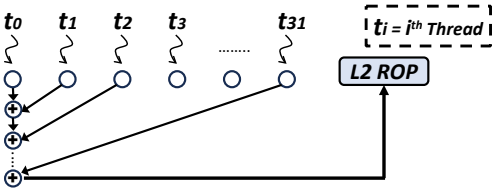
Identifying Active Threads. To address Challenge ❶, ARC-HW identifies sets of threads within a warp that update the same parameter (and thus memory location). This is done by leveraging the address coalescing module that is used to coalesce common memory addresses of threads performing a load operation. In this step, for each memory location being updated atomically in the warp, the corresponding active threads are identified (generating an *atomic transaction*). Only these threads participate in the subsequent reduction steps since other threads are not eligible for a warp-level reduction (because they are either inactive or do not update the same memory location).

Warp-level Reduction. In order to perform the warp-level reduction at the sub-core, ARC-HW introduces additional logic per sub-core, which is referred to as the *reduction unit*. The reduction unit contains a single floating point unit which serially sums the gradients. The result of the reduction is then sent to the LSU to atomically update the parameter in global memory with the reduced gradients.

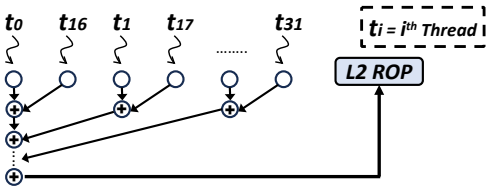
Scheduling Atomic Updates Between Core and L2 ROP. To address Challenge ❷, ARC employs a greedy scheduling algorithm and schedules the atomic transaction either at the SM reduction unit or the LSU (for computation at the ROPs) depending on which queue is free. When the rate of atomic requests overwhelms the available atomic throughput of the ROP units, the requests get stalled at the LSU queue. Such atomic stalls propagate the memory pipeline and eventually lead to stalls within the sub-core at the LDST units (Figure fig. 9). ARC scheduler leverages this behavior and observes the atomic stalls at the LDST units to determine the availability of the ROP units. When an atomic memory transaction is generated, if the ROP units are not stalled, the ARC scheduler schedules the atomic update instructions directly to the ROP units. Otherwise, the atomic updates are reduced using ARC-HW's reduction unit in the sub-core. The scheduling algorithm may benefit from additional information such as the number of concurrent requests and future request predictions; however, acquiring such information requires additional logic and hardware overhead, which adds to the overall design complexity. Thus, we choose the greedy algorithm that achieves high performance with negligible hardware modifications.

4.4 Key Components of ARC-SW

ARC-SW is implemented and exposed to the programmer as a function call that can be inserted into GPU code. We describe how we implement ARC-SW using existing instructions and warp-level primitives.



(a) Serialized reduction.



(b) Butterfly reduction.

Figure 10. Overview of reduction algorithms of ARC.

Warp-level Reduction (Challenge ❶). We propose two approaches to perform warp-level reduction that addresses Challenge ❶, each of which has different tradeoffs. These approaches are outlined below:

(1) Serialized Reduction: Within each warp, we first determine a set of threads that atomically update the same parameter (and thus memory location). One thread out of this group then iterates through all the gradients (one from each thread), as shown in Figure 10a. The accumulated result is then added to the parameter using a regular atomic add operation. The serial nature of this scheme is inefficient. However, when the warp has threads updating multiple parameters, the reductions can be parallelized.

(2) Butterfly Reduction: Figure 10b shows how butterfly reduction is performed for threads in a warp. We first check whether all the threads in a warp update the same primitive. If so, we use a reduction tree to sum the gradients. For this implementation to work, it requires all threads to be active, or for threads that are inactive, we must add a zero value. This introduces some redundant computation. Thus, the programmer has to ensure there is no control flow divergence and all threads are active and assign zero-value updates to threads that were originally inactive. Butterfly reduction is the most efficient, when only one parameter is being updated by the warp and most threads are active (less redundant updates).

Scheduling Atomic Updates Between Core and L2 ROP (Challenge ❷). As discussed in §3.1, the amount of contention at the LSU that is contributed to by each warp depends on the number of active threads producing atomic requests. Additionally, the active thread count also reflects

the amount of reduction "work" to be done in the SM (if the atomic update is scheduled for warp-level reduction). To address Challenge ❷, we determine whether the atomic updates should be performed using a warp-level reduction at the core or at the L2 ROPs, by comparing the number of threads in the warp that actively update one parameter over a *predefined threshold*. We call this the balancing threshold, as it balances the atomic computation between ROP units and the SMs. This scheduling is performed for each set of threads in a warp that updates one parameter. The optimal *balancing threshold* depends on the amount of contention in atomic units, which depends on the following:

- **Dataset and workload:** The number of atomic updates depends on factors such as the camera resolution, model architecture, and the complexity of the scene being learned.
- **GPU architecture:** The ratio of SMs to ROP units impacts the contention at cores and ROP units.
- **Reduction method:** The choice of using the butterfly or serial reduction method also affects the contention at the atomic units.

Due to the complexity in determining the threshold analytically, we treat the balancing threshold as a hyperparameter that needs to be tuned for each workload. We discuss in detail how we use the balancing threshold in §5 and evaluate the impact of this hyperparameter in §7.2.

4.5 Advantages of ARC-HW over ARC-SW

ARC-HW addresses five limitations of ARC-SW. (i) ARC-SW incurs additional overhead in performing redundant computations during warp-level reduction, because not all threads produce gradient updates in every iteration (when doing butterfly reduction). (ii) ARC-SW introduces overhead with control flow instructions when performing warp-level reduction. (iii) Programmer intervention is required to integrate ARC-SW (code must be converted as in §5.5). (iv) ARC-SW requires tuning the balancing threshold hyperparameter to get the best possible speedup. (v) Not all GPUs currently support the instructions used in ARC-SW (e.g., Intel GPUs).

5 Detailed Design

5.1 Design of ARC-HW

Programmer Interface. ARC is exposed to the programmer as a new instruction `atomred`:

```
atomred [%addr], %f
```

`%addr` is the memory location of the parameter that needs to be atomically updated and `%f` is the floating point value to be added. This instruction is similar to the baseline atomic instruction, except that there is no return value.

ARC comprises two key components:

ARC Scheduler. `atomred` instructions are first dispatched to this unit (shown in Figure 11), which decides whether a warp-level reduction should be performed for the current `atomred` instruction ❶. The scheduler leverages the execution status of previous atomic instructions to determine the

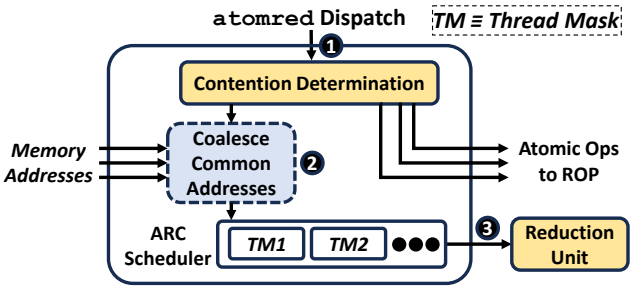


Figure 11. ARC scheduler module.

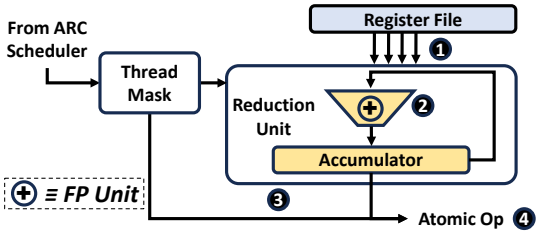


Figure 12. ARC's reduction computation unit

ROP utilization. If the ROP units are free, the scheduler sends the `atomred` instruction as a normal atomic instruction to the address coalescing unit to generate regular atomic requests that are sent to the ROP units. If the ROP units are busy, the `atomred` instruction is scheduled for warp-level reduction. To perform warp-level reduction, we must first identify which threads update the same parameter (i.e., memory location). To do this, we leverage the same address coalescing unit for generating coalesced addresses for memory transactions to generate a thread mask. Threads marked in the mask are threads that update the same parameter. This thread mask and memory addresses generated by the coalescing unit are sent to the *reduction unit* to perform warp-level reduction.

ARC Reduction Unit. The reduction unit performs warp-level reductions by serially aggregating atomic add instructions from each transaction and then issuing a single atomic add request to the memory interface (shown in Figure 12). The thread masks and memory addresses sent by the ARC scheduler are used to serially fetch values ❶ to be reduced from the register file and accumulate the result ❷❸. Finally, the sum and the thread mask are used to generate a new atomic transaction, which is issued as a normal atomic update ❹ to be processed by the L2 ROP units. We add a dedicated FPU per sub-core in the reduction unit to perform the warp-level reduction. Existing FPUs could be used; however, it might be challenging to re-design the 32-lane-wide pipeline to perform serial reduction. Instead, dedicated per-sub-core FPUs have low area overheads (§5.4).

5.2 Atomic Additions and Commutativity

ARC-HW assumes that the atomic add operations are commutative, i.e., the order with which the addition operations to the same memory location are performed does not affect

the result. Thus, the proposed `atomred` instruction does not provide any ordering guarantees and should only be used for commutative atomic additions. For the differentiable rendering workloads, we find that all atomic operations are commutative. We note that floating point additions may produce slightly different results based on the order of operations, but differential rendering workloads, similar to machine learning workloads [84], are resilient to these small errors. Thus, the baseline differentiable rendering applications are also written without imposing any order in the atomic additions.

5.3 Coherence and Consistency Considerations

ARC does not locally buffer any memory locations, but only partially aggregates atomic updates at the SM registers. These updates are not buffered —once the additions across the threads in a warp are reduced, the aggregated atomic update is sent to the L2 ROP units to update the memory location. For the locally aggregated values in the ARC reduction units to be visible to the rest of the GPU, GPU coherence protocols (e.g., [62, 86, 91]) must be updated to include ARC reduction units. In differentiable rendering workloads, the same data is never read and updated across multiple SMs concurrently. Thus, it will not add significant overhead. However, for other GPU workloads with different access patterns, the coherence protocols may incur additional overheads.

We assume that all code is data race-free with the SC-for-HRF [51] consistency model (similar to prior works [30, 32]). We assume all uses of the proposed `atomred` instruction are commutative and relaxed, and the output is not being used to influence the control flow of the program. Additionally, we assume that if any kernel leverages ARC, the memory addresses accessed atomically are always accessed atomically within the same kernel (i.e. respects strong atomicity [24, 30] within a kernel). Differentiable rendering workloads respect strong atomicity in the gradient computation kernel, as all gradient updates are atomic and there are no accesses to the gradients that are non-atomic within the gradient computation kernel. If a kernel that leverages ARC violates strong atomicity, no guarantee is provided by ARC regarding the ordering between `atomred` and non-atomic requests.

5.4 Area Overhead

ARC introduces control logic in the frontend to process the `atomred` instruction, and control logic in the scheduler to determine the stall status of the LSU. In the reduction unit, ARC only introduces a floating-point unit (FPU) per sub-core for the reduction computation, and a few registers to store inputs and accumulate results. We use Yosys [16] to evaluate this area overhead, which is estimated under $70K$ transistors per FPU. In a GPU with 128 SMs and 4 sub-cores per SM (as in NVIDIA RTX 4090), ARC introduces $128 \times 4 \times 70K = 35840K$ additional transistors. Since ARC does not require any additional SRAM buffer, the area overhead is small compared to the total transistor count of modern GPUs: RTX 4090 includes 76 billion transistors, thus ARC would incur a very modest area overhead of $\sim 0.047\%$.

5.5 Design of ARC-SW

ARC-SW is exposed to the programmer as a function call that is invoked during gradient computation, directly replacing the atomic instructions in Figure 5 (lines 12-14) and is called by all threads. The function's prototype is provided in Figure 13. It takes as input the primitive to be updated by the thread, the primitive's parameters, and the gradients generated by the calling thread for all the primitive's parameters.

```
1  // Input: primitive index, pointers to parameter
2  // gradients, values to be accumulated,
3  // balancing threshold
4  template<typename ATOM_T>
5  void reduce_arc(int idx, ATOM_T** ptr,
6   ATOM_T *val, int num_params,int balance_thr){
7      ... <balanced reduction implementation>
8  }
```

Figure 13. Function prototype of ARC-SW.

```
1  /* balanced reduction implementation */
2  // when num. of active threads exceeds threshold,
3  // do butterfly or serialized reduction
4  int num_active_threads = __popc(__match(idx));
5  if (num_active_threads >= balance_thr){
6      ... <serialized OR butterfly implementation>
7  } else {
8      ... <perform baseline atomic add>
9  }
```

Figure 14. Dynamically determining whether to send atomic requests to ROP units or perform warp-level reduction (§4.4).

Figure 14 shows the implementation of the balanced reduction. Each thread determines how many other threads in the warp are updating the same primitive (done using `__match` instruction [10]). If this thread count is less than the balancing threshold, the function simply sends the baseline CUDA `atomicAdd` [9] (line 8), which leverages the ROP units and does not perform warp-level reduction. Otherwise, it performs warp reduction using either serialized reduction (§5.5.1) or butterfly reduction (§5.5.2).

5.5.1 ARC-SW with Serialized Reduction (SW-S) If execution enters the warp reduction branch (line 6 Figure 14), as discussed in §4.4, the warp-level reduction is performed serially (Figure 15). For each primitive, a leader thread is identified (active thread in the warp with the lowest lane ID, that updates the same primitive, line 6). This thread serially accumulates gradients across all active threads in a warp for all the parameters (lines 10-15).

Limitations: The primary limitation is the inefficient serial reduction with execution time proportional to the number of active threads per primitive. This also involves additional control flow overheads (lines 10, 12, 13, 17, 18).

```
1  /* serialized reduction implementation */
2  // a mask of warp threads updating the same primitive.
3  int prim_mask = __match(idx);
4  // thread within a warp with lowest lane id
5  // with the same primitive is the leader
6  int leader = <lowestLaneIdUpdatingSamePrimitive>
7
8  /* leader accumulates parameters from threads updating
9  the same primitive (threads with same prim_mask) */
10 for src_lane <- next_lane_id(prim_mask){
11   // get data from next active lane
12   if (laneId==leader || laneId==src_lane)
13     for (int i = 0; i < len; ++i)
14       val[i] += __shfl(val[i], src_lane);
15 }
16 // leader sends an atomicAdd per parameter
17 if (laneId == leader)
18   for (int i = 0; i < num_params; ++i)
19     atomicAdd(ptr[i], val[i]);
```

Figure 15. Serialized reduction in SMs (SW-S).

```
1  /* butterfly reduction implementation */
2  // reduction only performed when all threads
3  // are updating the same primitive
4  bool all_same = __match(idx) == 0xffffffff;
5  if (all_same) {
6    // parallel butterfly reduction tree
7    for(int offs = 16; offs >= 1; offs /= 2)
8      val[i] += __shfl(val[i], offset);
9    // thread 0 sends an atomicAdd per reduced gradient
10   if (laneId == 0)
11     atomicAdd(ptr[i], val[i]);
12 } else if (was_active) {
13 /* if balance threshold not met or bfly reduction
14 ineligible, do baseline atomicadd */
15   for (int i = 0; i < num_params; ++i)
16     atomicAdd(ptr[i], val[i]);
17 }
```

Figure 16. Butterfly reduction in SMs (SW-B).

5.5.2 ARC-SW with Butterfly Reduction (SW-B)

As discussed in §3.1, over 99% of warps in many workloads have all active threads update the same primitive's parameters. In these cases, a parallelized reduction tree can be used for fast warp-level reduction. Figure 16 presents our implementation. We propose an efficient implementation that requires (1) all threads in a warp update the same primitive (line 4), and (2) all threads actively participate in the reduction. The programmer can use SW-B only if the first condition is met. To ensure all threads participate in the reduction, the previously inactive threads must now be made to generate zero value gradient updates. A `was_active` flag is set to false for these updates. If condition (1) is not met, atomic add operations are used for reduction (line 16-17). Otherwise, a butterfly reduction is performed using `shfl` instructions (line 8-9).

Limitations: SW-B adds redundant computation by making inactive threads perform zero value gradient updates, making reduction for warps with many inactive threads inefficient. Using SW-B also requires changes to the kernel code shown with an example in Figure 17, where the code is transformed to ensure all threads participate in the reduction. This transformation can be non-trivial in some applications.

```
1:  function GradComputeBFLY(prims_per_thread)
2:      tid = thread_idx
3:      prims_per_thread = primitives[tid]
4:      for p in prims_per_thread do
5:          was_active = true;                    ▷ active by default
6:          if cond1 then
7:              was_active = false;               ▷ Don't skip, mark inactive
8:          end if
9:          ...
10:         if cond2 then
11:             was_active = false;               ▷ Don't skip, mark inactive
12:         end if
13:         ...
14:         if not was_active then
15:             grad_{x1,...xN} = 0               ▷ Inactive, assign zero gradients
16:         end if
17:         grad_ptrs = array[p.grad_{x1,...xN}]
18:         grad_vals = array[grad_{x1,...xN}]
19:         REDUCE_ARC _BFLY(p, g_ptrs, g_vals, N, was_active)
20:     end for
21: end function
```

Figure 17. Outline of a modified gradient computation kernel (Figure 5) that integrates the SW-B primitive.

5.5.3 Determining Balancing Threshold

The balancing threshold significantly impacts speedups (evaluated in §7.2) and needs to be tuned for best results. The balancing threshold has only 32 possible values ($0-31$), and the gradient compute kernel is called 100000s of times during training. Thus, we present a simple method to tune the threshold automatically: We execute one iteration of the gradient computation kernel using all 32 values of the threshold and select the value that provides the largest speedup. We repeat this profiling every N iteration (2000 in our evaluation). This profiling step adds a negligible amount of overhead, as the number of profiling iterations is significantly fewer than the number of training iterations.

5.6 Applicability to Other Workloads

Important deep learning and graph analytics applications are also bottlenecked by a large number of atomic requests [32]. Unlike differentiable rendering workloads, however, these workloads have more irregular memory accesses, and thus exhibit a low degree of intra-warp locality between the addresses updated atomically by threads of a warp. For example, we profile the pagerank workload from the Pannotia GPGPU benchmark [27], implemented as a strongly atomic kernel. We find that although 89.2% of global memory accesses that reach the L2 are global atomics, fewer than 0.1% of warps have all active threads atomically updating the same address (i.e., low intra-warp locality). Compared to this, differentiable rendering workloads have a very high intra-warp locality that we primarily leverage with ARC (§3.1 Observation 1), where on average 99% of the warps have all active threads atomically updating the same memory location. Thus, ARC

cannot provide performance benefits to these workloads. When ARC is not used, there are no performance overheads for those workloads as the ARC reduction unit is bypassed.

6 Methodology

Evaluation Platform. We implement and evaluate ARC-SW on a x86_64 Linux system with an Intel Core i9-13900KF CPU and the NVIDIA RTX4090 and RTX3060 GPUs. We model and evaluate ARC-HW using the GPGPU-Sim simulator [58] with the RTX4090 and RTX3060 GPU configurations (shown in Table 1). We measure the energy consumption using pyNVML [1, 4] and pyRAPL [8].

Table 1. Simulated GPU configurations

4090-Sim	**SM Resources**	128 SMs, 32768 Registers, 176 ROPs
	Shader Core	2.24GHz; 4 sub-cores per SM
	Cache Model	128KB L1 per SM, 72MB L2
	DRAM	12-channel; 16-bank; 24GB GDDR5
3060-Sim	**SM Resources**	28 SMs, 32768 Registers, 48 ROPs
	Shader Core	1.32GHz; 4 sub-cores per SM
	Cache Model	128KB L1 per SM, 3MB L2
	DRAM	12-channel; 16-bank; 12GB GDDR5

Table 2. Workloads and datasets.

Workloads	Dataset Identifier (Dataset Name)
3DGS (3D)	LE, SH (NerfSynthetic - Lego, Ship [76]), PR, DR (DB COLMAP - Playroom, DrJohnson [18]), TK, TA (Tanks&Temples - Truck, Train [61]),
NvDiff (NV)	BB, SP (Keenan Crane - Bob, Spot [31]), LE, SH (NerfSynthetic - Lego, Ship [76])
pulsar (PS)	SS, SL (Synthetic Spheres - Small, Large)

Workloads. We evaluate ARC using 3 widely used raster-based differentiable rendering applications: (i) **3DGS** [55] represents scenes with a set of 3D Gaussians; (ii) **NvDiffRec** [80] includes various differentiable rendering tasks, and we use differentiable rendering to learn the parameters of specular cubemap texture from a set of mesh images, and (iii) **Pulsar** [64] represents the scene with a set of spheres and an efficient sphere rasterizer with a widely-used implementation in Pytorch3D [83]. We use the datasets listed in Table 2. For pulsar, we use two synthesized datasets comprising 3D spheres (PS-SS and PS-SL).

7 Evaluation

We evaluate both the software implementation (ARC-SW) and the hardware implementation (ARC-HW). The baseline configuration sends all atomic requests to the ROP units using the atomicAdd primitive [9]. For ARC-SW, we evaluate two different configurations: (i) SW-B-X: an implementation of ARC-SW using butterfly reduction, with balancing threshold X. (ii) SW-S-X: an implementation of ARC-SW using serialized reduction, with balancing threshold X. We refer to the configurations of SW-B-X and SW-S-X with the best-performing balancing threshold as SW-B and SW-S, respectively. We compare both configurations against CCCL, which uses the NVIDIA CCCL library [14, 15] to perform warp-level reduction. We test software approaches on real hardware: (i) 4090: NVIDIA RTX 4090 GPU and (ii) 3060: NVIDIA RTX 3060 GPU. Each result is the arithmetic mean of 10 runs of the same configuration to ensure speedups are statistically significant. For ARC-HW, we compare ARC-HW against LAB [32], the closest prior work that dynamically partitions the local L1/shared memory SRAM to buffer and aggregate atomic requests. We use two LAB configurations: (i) LAB-ideal: an idealized implementation of LAB. We dedicate an additional and separate SRAM buffer with the same capacity as the L1/shared memory SRAM that does not incur contention in the LSU, and we assume no tag lookup overheads, MSHR queuing delays, etc. Commutative atomic requests are sent to LAB-ideal's dedicated SRAM buffer, and other requests are sent to L1/shared memory. (ii) LAB: an implementation of LAB where the L1/shared memory SRAM is partitioned between atomic and non-atomic requests (no additional SRAM). Differentiable rendering workloads use some shared memory, thus there is less available L1/shared memory SRAM that can be allocated to LAB. The partition size for LAB was chosen to be the empirically best performing for each workload. Additionally, we compare against PHI [78], which buffers and aggregates atomic updates in the L1 caches. We evaluate ARC-HW, LAB-ideal, LAB and PHI on both simulator configurations: 4090-Sim and 3060-Sim.

7.1 Performance analysis of ARC-HW.

Figures 18 and 19 depict the speedup achieved by ARC-HW, LAB, LAB-ideal and PHI on 4090-Sim and 3060-Sim, normalized to the baseline. Figures 20 and 21 depict the reduction in shader stalls due to atomic operations over ARC-HW, LAB and LAB-ideal on 4090-Sim and 3060-Sim, normalized to the baseline. We make four observations:

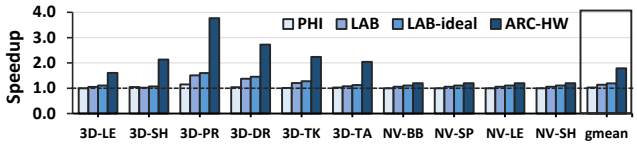


Figure 18. Gradient computation speedup of ARC-HW, PHI, LAB and LAB-ideal on 3060-Sim, normalized to baseline.

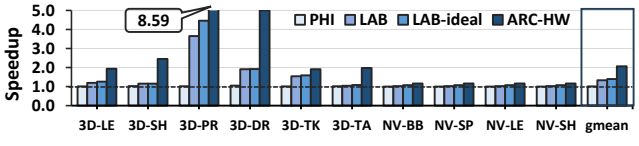


Figure 19. Gradient computation speedup of ARC-HW, PHI, LAB and LAB-ideal on 4090-Sim, normalized to baseline.

First, ARC-HW demonstrates significant performance improvements, achieving 2.06× speedup on average (up to 8.59×) on 4090-Sim and 1.73× speedup on average (up to 3.77×) on 3060-Sim. Figures 20 and 21 demonstrate 2.43× and 2.28× average reduction in shader atomic stalls on 4090-Sim

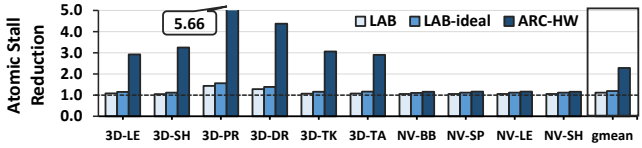


Figure 20. Reduction in shader atomic stalls on 3060-Sim, normalized to baseline.

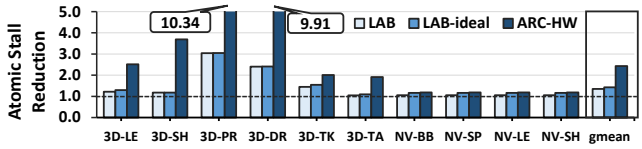


Figure 21. Reduction in shader atomic stalls on 4090-Sim, normalized to baseline.

and 3060-Sim, respectively, as a result of ARC-HW significantly reducing the atomic memory transactions. Second, LAB-ideal achieves an average speedup of 1.40× and 1.20× on average (up to 3.05× and 1.56×) on 4090-Sim and 3060-Sim, respectively, which are lower compared to the speedups achieved by ARC-HW. ARC-HW's performance benefits over LAB come from (a) higher atomic processing throughput as ARC performs reduction at each of the 4 sub-cores, rather than just the overall core, (b) there is no eviction/capacity issue in buffers since the reduction is only intra-warp and happens directly in the registers, and (c) further improved throughput by distributing atomic computation between the ROP units and the sub-cores. Thus, ARC-HW is more effective in reducing the atomic stalls as demonstrated in Figures 20 and 21, where LAB-ideal achieves 1.43× and 1.19× average reduction in shader atomic stalls on 4090-Sim and 3060-Sim, respectively. Third, LAB-ideal marginally outperforms LAB by 1.05× on average on both 4090-Sim and 3060-Sim. This is because of the smaller space available for atomic buffering and the slightly higher L1 miss rate from the reduced capacity for the non-atomic requests. Fourth, PHI provides only small performance improvements (1.01× and 1.03× on average on 4090-Sim and 3060-Sim) over the baseline. This approach buffers atomic updates at L1 caches and does not perform warp-level reduction at the sub-cores. Thus, a large number of atomic updates overwhelms the LSU when buffering updates, and does not alleviate the atomic stalls in these workloads. Moreover, PHI performs L1 address lookups for each atomic operation, which incurs additional overheads and also impacts the performance of non-atomic operations.

We conclude that ARC-HW is an efficient approach to accelerate differentiable rendering applications by alleviating the overheads of atomic operations in gradient computations.

7.2 Software-only Approaches

Performance Analysis. Figure 22 shows the speedup of ARC-SW for end-to-end runtime (including the forward pass) and for gradient computation alone on real hardware, normalized to the baseline. We make two observations.

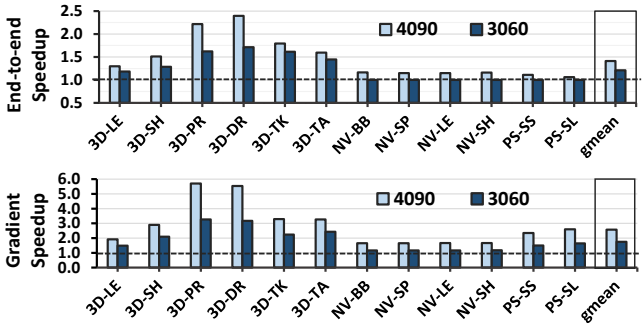


Figure 22. End-to-end and gradient computation speedup on 4090 and 3060 with ARC-SW, normalized to baseline.

First, both SW techniques, SW-B and SW-S, significantly outperform the baseline on average on both GPUs. For the gradient computation, ARC-SW achieves an average speedup of 2.44× (up to 5.7×) on 4090, and 1.74× (up to 3.27×) on 3060. For the entire differentiable rendering pipeline, ARC-SW achieves an average speedup of 1.41× on 4090 (up to 2.4×), and 1.21× (up to 1.71×) on 3060.

Second, we observe higher speedups on 4090, compared to that of 3060, because the atomic processing bottleneck is more pronounced in 4090 which has a lower ROP to SM ratio (128 SMs and 176 ROP units in 4090 versus 28 SMs and 48 ROPs in 3060). Third, SW-B performs as well as or much better than SW-S, which performs the reduction serially. However, there are some workloads (PS-SS and PS-SL) that cannot use SW-B, because it was difficult to eliminate thread divergence, the key requirement for butterfly reduction (§5.5.2). Fourth, we observe significant performance speedups on 3D-PR and 3D-DR, because the datasets PR, DR are large-scale, photorealistic scenes that require many more geometric primitives (Gaussians for 3DGS) for accurate scene representation compared to smaller scenes. Therefore, a larger number of parameters needs to be atomically updated during gradient computation, making the atomic bottleneck more pronounced. Finally, we observe smaller end-to-end speedups in NV and PS. NV has much fewer warp stalls compared to 3D in the baseline application (Figure 8). This leads to a less contended LSU, which diminishes the speedups achieved by ARC-SW. In PS, although the LSU is heavily contended in gradient computation (Figure 8), the gradient computation is not the main bottleneck (Figure 4).

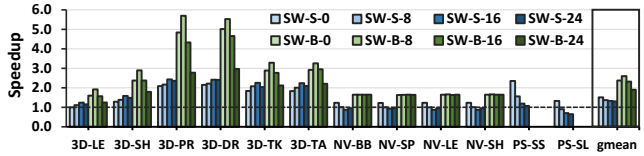


Figure 23. Sensitivity of SW-S and SW-B to balancing threshold X. SW-B cannot be used for PS-SS and PS-SL.

Impact of the Balancing Threshold. In Figure 23, we depict the sensitivity of SW-S and SW-B to the balancing

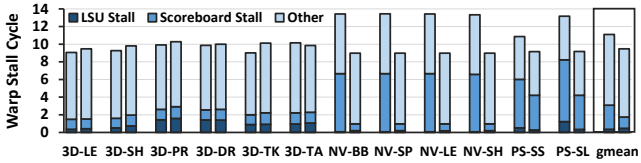


Figure 24. Breakdown of warp stall cycles during gradient computation using ARC-SW on 4090 (left) and 3060 (right).

threshold X for gradient computation on 4090. We make two observations. First, the best-performing balancing threshold varies across workloads/datasets. For most workloads, we achieve the highest speedup when X is set to a value that ensures that the atomic updates are distributed between the ROP units and the SMs for both SW-S and SW-B. Setting X to be 0 or 24 leads to contention in either the sub-core reduction unit or the ROP units, respectively. Second, in NV and PS workloads, choosing sub-optimal balancing thresholds can result in slowdowns: in some compute-bound workloads, the additional instructions required can incur significant overheads. In these cases, balancing thresholds that favor the ROP unit should be chosen.

Elimination of LSU Stalls. To better understand the observed performance benefits, we measure the number of stall cycles per instruction in Figure 24 using the NVIDIA NSIGHT Compute [7] profiler. We find significantly fewer mean stalls per instruction across all workloads compared to baseline (Figure 8): 10.3 cycles versus 38.3 cycles on average thanks to significantly fewer stalls in atomics (LSU stalls).

Comparing ARC-SW to ARC-HW. We evaluate ARC-SW in the simulator to compare it against ARC-HW. Figure 25 depicts the speedup of ARC-HW normalized to ARC-SW. We find that ARC-HW consistently outperforms ARC-SW by 1.13× on average (up to 1.29×) on 4090-Sim, and 1.14× on average (up to 1.26×) on 3060-Sim. ARC-HW is more efficient since it does not incur the overhead of many additional instructions, control flow, and redundant computation.

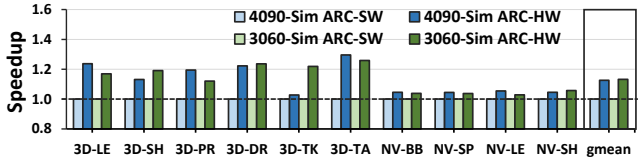


Figure 25. Gradient computation speedup of ARC-HW and ARC-SW in the simulator, normalized to ARC-SW.

Comparing Against Software Warp-level Reduction. Figure 26 compares ARC-SW over NVIDIA CCCL library [14], the state-of-art library for software warp-level reduction. We note that significant engineering efforts were needed to make CCCL work correctly for these workloads. We present the gradient computation speedup of ARC-SW and CCCL normalized to the atomicAdd baseline on 4090.

We make two observations: **(1)** ARC-SW outperforms CCCL significantly, achieving 1.58× higher speedup on average

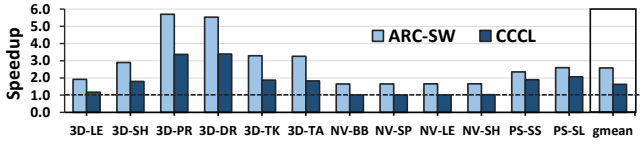


Figure 26. Gradient computation speedup of ARC-SW over CCCL on 4090, normalized to the baseline.

(up to 1.78×) across all workloads. CCCL requires all threads in the same warp to be active (§4.2), but the gradient computation steps of raster-based differentiable methods have dynamic conditions that often do not satisfy this requirement (§3.1). Moreover, ARC-SW effectively distributes reduction between SMs and ROP units (§4.4). **(2)** CCCL yields marginal performance improvements on NvDiff workloads because NvDiff has many inactive threads in the most warps. Thus, CCCL fails to capture most reduction opportunities.

7.3 Energy analysis

We evaluate energy efficiency of ARC-SW and ARC-HW. For ARC-SW, we measure energy consumption on real GPUs, depicted in Figure 27 and normalized to baseline. We observe that ARC-SW reduces energy consumption by 2.8× on average on 4090 and by 1.7× on average on 3060. This is due to the significantly faster execution as well as the fewer memory requests (atomic requests) from the SMs to the ROP units which leads to less energy consumption at the interconnect. For ARC-HW, we measure its energy reduction in the simulator with different configurations, depicted in Figure 28 and normalized to the simulator baseline. We observe that ARC-HW significantly reduces energy consumption by 3.9× on average on 4090-Sim and 2.55× on average on 3060-Sim.

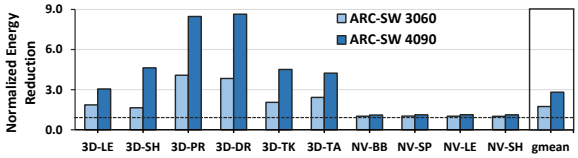


Figure 27. Normalized energy reduction in gradient computation with ARC-SW on 4090 and 3060.

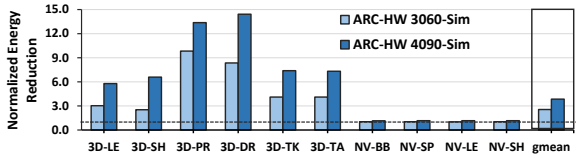


Figure 28. Normalized energy reduction in gradient computation with ARC-HW on 4090-Sim and 3060-Sim.

8 Related Work

To our knowledge, this is the first work to *(i)* characterize emerging differentiable rendering workloads and identify atomic operations to be a key bottleneck; and *(ii)* propose an efficient method to leverage warp-level reduction and

existing atomic units to accelerate atomic processing in GPUs for raster-based differentiable rendering.

Differentiable Rendering Techniques. Prior works propose software techniques [29, 43, 68, 79, 93] or hardware accelerators [44, 66, 69, 102] to accelerate training and/or rendering for NeRFs [76, 79]. These works target one class of differentiable rendering applications in which the primary bottleneck is the large number of computations and memory accesses performed. Instead, our work targets raster-based differentiable rendering methods that significantly reduce the number of computations needed, thus being a more efficient approach than NeRFs. These methods are highly bottlenecked by atomic operations during training, a problem that is not addressed by prior work. Applying ARC in NeRFs would not provide significant speedups, because (i) for NeRFs, atomics operations only constitute a secondary bottleneck, and (ii) NeRFs have low intra-warp locality in atomic updates (the key insight that ARC leverages) as they typically use hash tables/octtrees/etc. to store learned parameters that lead to more irregular access patterns.

Efficient Atomic Operations in GPUs. Prior works [19, 40, 85, 91, 92, 94] propose cache coherence protocols that can accelerate atomic requests in GPUs. However, they require non-trivial changes to GPU cache coherence protocols at the L1. Atomic operations can instead be implemented with Remote Memory Operations (RMOs) [45, 47, 60, 88, 89, 98], which is hardware support for atomic operations: this involves using arithmetic logic near shared caches/main memory to perform efficient atomic operations without complex changes to cache coherence protocols and is used in modern GPUs [99]. However, as demonstrated in this work (§3.2), this can lead to bottlenecks at the LSU and memory subsystem due to high atomic traffic in workloads where a large number of threads perform atomic operations. To address this bottleneck, prior works propose to buffer atomic operations in local SRAM in each SM and thus reduce the contention near the L2 atomic units: LAB [32] dynamically reserves a partition of the L1/shared memory SRAM in each SM to aggregate the atomic requests. PHI [78] aggregates commutative atomic requests at the L1 cache to reduce memory traffic. These works demonstrate significant speedups in applications with a large amount of atomic traffic, such as graph applications, histograms, and ML training. These works, however, do not fully leverage the high intra-warp atomic locality seen in differentiable rendering workloads. We quantitatively compare against LAB [32] and PHI [78] in §7.1 and demonstrate that ARC performs better by leveraging intra-warp locality in atomic updates to enable warp-level reduction using registers in the GPU sub-cores. ARC significantly reduces the number of atomic updates sent to the LSU and memory hierarchy, and our dynamic scheduling scheme (§4.3) leverages both the ROP units and the SMs to enable high atomic processing throughput. DAB [30] aims to achieve deterministic atomic execution by buffering and fusing atomic requests in dedicated atomic buffers at each SM. However, scheduling atomic operations with deterministic orderings (determinism-aware schedulers) introduces additional overheads that can lead to > 20% slowdowns over non-deterministic application baselines, as reported by DAB.

Software-based Warp-level Reduction. Software frameworks and libraries [2, 14, 15, 34, 38] provide functions to perform warp-level and block-level reduction. These libraries require all threads of the warp to be active for reduction computation. ARC-SW efficiently performs updates to all parameters associated with a primitive, even when *a subset* of warp threads is active. These libraries also do not use both the SM and L2 atomic units to increase atomic processing throughput. We compare ARC with CCCL library in §7.2.

In-Register Parameter Caching. Prior works leverage GPU register files as additional on-chip memory to cache partial updates of model parameters and exploit data locality [33, 35, 53, 59, 111, 113]. These works demonstrate speedups from reducing off-chip memory accesses. ARC also uses register files to temporarily store parameter updates. However, unlike these works that primarily optimize for memory loads, ARC is an atomic reduction technique that aims at alleviating the significant memory traffic from *atomic operations* which can constitute a significant bottleneck in differentiable rendering workloads. ARC is designed to leverage any intra-warp locality in atomic updates by using registers to perform atomic reductions. Thus, instead of buffering for reuse, ARC requires the addition of an FPU to perform the sub-core reduction of atomic updates. This reduces the amount of atomic contention at the L2 ROP units. The ARC scheduler also dynamically detects threads eligible for reduction and schedules atomic operations between the ARC reduction unit and the ROP units to achieve optimal atomic throughput.

9 Conclusion

We introduce ARC, a novel primitive that enables fast processing of atomic reduction operations in applications that **(1)** generate a massive number of atomic requests and **(2)** have many threads within each warp atomically updating a common parameter. We demonstrate that both the hardware-software primitive ARC-HW and the software-only primitive ARC-SW can effectively alleviate the atomic processing bottleneck and accelerate raster-based differentiable rendering workloads, which is an important emerging class of applications in visual computing. ARC is a general atomic primitive that can also be used to accelerate other workloads with similar atomic characteristics.

Acknowledgments

We thank the reviewers for their valuable feedback and the members of the embARC research group for all their help and the stimulating research environment they provide. This research was supported by the Sony Research Award program and NSERC Alliance.

A Artifact Appendix

A.1 Abstract

This Artifact Appendix describes how to reproduce the ARC-SW end-to-end speedup results in §7.2 (Fig. 22 and Fig. 26). Since ARC-SW is a modular extension to the gradient computation kernel, it is integrated into the existing code bases of the threedifferentiable rendering applications we use in the paper (`3DGS`, `NvDiff`, `pulsar`). The 3 different applications have different software dependencies, setup workflows and execution environments and could not be condensed into one set of instructions and submission, thus we present the workflow and setup to reproduce the results for the `3DGS` workloads which includes the majority of the workloads in our evaluation and the workloads which achieve the highest speedups with our approach. `3DGS` is one of the state-of-the-art and widely-used raster-based differentiable rendering methods (described in §2.3).

The code provided in the artifact is developed based on the original `3DGS`'s open-source implementation. The training process contains model evaluation checkpoints for correctness checks and end-to-end training time at the end as performance measurement. A successful training process with either ARC-SW variant (ARC-SW-B or ARC-SW-S) should report similar Peak-Signal-to-Noise-Ratio (PSNR↑) and L1 loss (L1↓) values, while achieving similar speedups. Note that since the training process is stochastic, both PSNR and L1 values will have minor differences across runs even with the same configuration.

To facilitate the AE process, we provide a tarball file, which contains the ARC-SW source code, experiment datasets, and the automated workflow scripts to execute the `3DGS` training.

A.2 Artifact check-list (meta-information)

- **Compilation:** GCC 12.1.0, public, not included; NVIDIA CUDA Compiler (NVCC) from CUDA Toolkit 12.2, public, not included.
- **Data set:** DB COLMAP, Tanks and Temples, NeRF Synthetic. All are publicly available and included.
- **Run-time environment:** Linux x86_64. Requires CUDA 12.2, Python 3.11 and Conda. Root access only required for CUDA installation.
- **Hardware:** NVIDIA RTX 4090 and RTX 3060 GPUs to accurately reproduce all results. NVIDIA GPUs with a minimum compute capability of 8.6 and at least 8GB GPU memory may also be used to validate the results.
- **Run-time state:** No other GPU-intensive task should be running during the AE process.
- **Metrics:** Correctness metrics (model quality): Peak-Signal-to-Noise-Ratio (PSNR↑), L1 loss (L1↓). Performance metric: end-to-end runtime.
- **Output:** A CSV file containing all configurations and raw metrics, and a PNG figure that reconstructs Fig. 22 and Fig. 26 in the paper.
- **Experiments:** Install required run-time environment, fetch and extract the provided tarball file
- **How much disk space required (approximately)?:** 10GB disk space required.
- **How much time is needed to prepare workflow (approximately)?:** About 10 minutes (download time excluded).
- **How much time is needed to complete experiments (approximately)?:** About 2 hours on a system with RTX 4090.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Gaussian-Splatting License (the same license as the original `3DGS`)
- **Archived (provide DOI)?:** 10.5281/zenodo.14053431

A.3 Description

A.3.1 How to access Download the tarball file *scar_ae.tar.gz* from Zenodo: 10.5281/zenodo.14053431

A.3.2 Hardware dependencies AE should be run on a host machine with NVIDIA RTX 4090 GPU and/or NVIDIA RTX 3060 GPU. To reproduce all results, it is required to run AE once on each GPU, as Fig. 22 contains results collected from both RTX 4090 and RTX 3060. On a platform with multiple GPUs installed, our script allows for the selection of a specific CUDA device to run the AE on.

If RTX 4090 and RTX 3060 are unavailable on any AE platform, ARC-SW may also be evaluated on a different NVIDIA GPU with a minimum compute capability of 8.6 and at least 8GB GPU memory. In this case, the observed speedups may show different trends compared to Fig. 22 and Fig. 26.

A.3.3 Software dependencies The verified operating systems and dependencies are listed below:

- Ubuntu 22.04.3 LTS or Manjaro 24.1
- Python 3.11
- CUDA Toolkit 12.2
- GCC 12.1.0 or GCC 12.3.0
- Conda installation

A.3.4 Data sets We evaluate across three publicly available datasets, listed below:

- DB COLMAP - Playroom, DrJohnson
- Tanks and Temples - Truck, Train
- Nerf Synthetic - Lego, Ship

A.4 Installation

Assuming the provided tarball file *scar_ae.tar.gz* has been downloaded and all software dependencies have been installed, please execute the following commands in sequence to install the environment:

```
$ tar -zxvf scar_ae.tar.gz
$ cd scar_ae
$ conda env create --file environment.yml
$ conda activate scar_ae
```

A.5 Experiment workflow

We provide a single Python script that runs all AE experiments and produces a single CSV file that contains the raw data required to reproduce Fig. 22 and Fig. 26. To run this

script, execute the following command (continuing from installation):

```
$ python run_ae.py 0
```

This script uses CUDA device 0 by default. If the AE platform has more than one NVIDIA GPU installed, the argument 0 may be changed to the target CUDA device ID to run the AE on.

A.6 Evaluation and expected results

When `run_ae.py` finishes successfully, the output CSV file `ae_result.csv` will contain raw experiment results from all combinations of the three parameters:

4 backward (gradient computation) kernel implementations:

- The original 3DGS implementation (baseline)
- ARC-SW-S (§5.5.1)
- ARC-SW-B (§5.5.2)
- CCCL library implementation (§7.2)

6 workloads from 3 datasets:

- NeRF Synthetic Ship
- NeRF Synthetic Lego
- DB COLMAP Playroom
- DB COLMAP DrJohnson
- Tanks and Temples Truck
- Tanks and Temples Train

4 `balancing_thresholds` (§4.4): an integer selected among 0, 8, 16, 24, 32. ARC-SW warp reduction is performed if and only if the number of active threads is greater than or equal to `balancing_threshold`. `balancing_threshold` has no effect when `bw_implementation` is set to `org` or `CCCL`.

There are 8 columns in the CSV file, listed below:

- BW Implementation
- Balance Threshold
- Dataset
- Train PSNR↑
- Train L1↓
- Test PSNR↑
- Test L1↓
- End-to-end Training Time

Each row in the CSV file corresponds to the results of one specific experiment configuration. We expect the PSNR and L1 values to be similar across all experiments on the same dataset. We additionally provide a Python script to automatically generate a figure that reconstructs both Fig. 22 and Fig. 26 from the CSV file, which can be run with the following command after the CSV file has been successfully generated:

```
$ python gen_ae_figs.py
```

We expect that the generated figure `scar_ae_fig.png` should depict results similar to Fig. 22 and Fig. 26 with minor differences, primarily due to the stochastic nature of the application, system noises and different testing platforms.

References

[1] 2012. PyNVML. https://pythonhosted.org/nvidia-ml-py/.

[2] 2014. Faster Parallel Reductions on Kepler. https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/.

[3] 2017. NVIDIA TESLA V100 GPU ARCHITECTURE. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[4] 2017. NVML GPU Power Measurement. https://github.com/kajalv/nvml-power.

[5] 2021. NVIDIA AMPERE GA102 GPU ARCHITECTURE WHITEPAPER. https://images.nvidia.com/aem-dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf.

[6] 2023. NVIDIA ADA GPU ARCHITECTURE. https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf.

[7] 2023. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute.

[8] 2023. PyRAPL. https://github.com/powerapi-ng/pyRAPL.

[9] 2024. CUDA C++ Programming Guide - 7.14. Atomic Functions. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions.

[10] 2024. CUDA C++ Programming Guide - 7.20. Warp Match Functions. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#warp-match-functions.

[11] 2024. CUDA C++ Programming Guide - 7.22 Warp Shuffle Functions. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#warp-shuffle-functions.

[12] 2024. DISTWAR repository. https://github.com/Accelsnow/gaussian-splatting-distwar.

[13] 2024. Nsight Compute Documentation. https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#metrics-guide.

[14] 2024. NVIDIA cccl library. https://github.com/NVIDIA/nccl.

[15] 2024. NVIDIA cub library. https://nvlabs.github.io/cub/.

[16] 2024. Yosys Open SYnthesis Suite :: About. https://yosyshq.net/yosys/

[17] Tor M Aamodt, Wilson Wai Lun Fung, Timothy G Rogers, and Margaret Martonosi. 2018. *General-purpose graphics processor architectures*. Springer.

[18] Josh Abramson, Arun Ahuja, Iain Barr, Arthur Brussee, Federico Carnevale, Mary Cassin, Rachita Chhaparia, Stephen Clark, Bogdan Damoc, Andrew Dudzik, et al. 2020. Imitating interactive intelligence. *arXiv preprint arXiv:2012.05672* (2020).

[19] Johnathan Alsop, Marc S Orr, Bradford M Beckmann, and David A Wood. 2016. Lazy release consistency for GPUs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–14.

[20] Edward Angel. 1996. *Interactive Computer Graphics: A top-down approach with OpenGL*. Addison-Wesley Longman Publishing Co., Inc.

[21] Sai Praveen Bangaru, Lifan Wu, Tzu-Mao Li, Jacob Munkberg, Gilbert Bernstein, Jonathan Ragan-Kelley, Frédo Durand, Aaron Lefohn, and Yong He. 2023. Slang. d: Fast, modular and differentiable shader programming. *ACM Transactions on Graphics (TOG)* 42, 6 (2023), 1–28.

[22] Jonathan T Barron, Ben Mildenhall, Dor Verbin, Pratul P Srinivasan, and Peter Hedman. 2022. Mip-nerf 360: Unbounded anti-aliased neural radiance fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 5470–5479.

[23] Jonathan T Barron, Ben Mildenhall, Dor Verbin, Pratul P Srinivasan, and Peter Hedman. 2023. Zip-nerf: Anti-aliased grid-based neural radiance fields. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 19697–19705.

[24] Colin Blundell, E Christopher Lewis, and Milo MK Martin. 2006. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters* 5, 2 (2006), 17–17.

[25] Ang Cao and Justin Johnson. 2023. Hexplane: A fast representation for dynamic scenes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 130–141.

[26] David Charatan, Sizhe Lester Li, Andrea Tagliasacchi, and Vincent Sitzmann. 2024. pixelsplat: 3d gaussian splats from image pairs for scalable generalizable 3d reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 19457–19467.

[27] Shuai Che, Bradford M Beckmann, Steven K Reinhardt, and Kevin Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 185–195.

[28] Anpei Chen, Zexiang Xu, Andreas Geiger, Jingyi Yu, and Hao Su. 2022. Tensorf: Tensorial radiance fields. In *European Conference on Computer Vision*. Springer, 333–350.

[29] Zhiqin Chen, Thomas Funkhouser, Peter Hedman, and Andrea Tagliasacchi. 2023. Mobilenerf: Exploiting the polygon rasterization pipeline for efficient neural field rendering on mobile architectures. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 16569–16578.

[30] Yuan Hsi Chou, Christopher Ng, Shaylin Cattell, Jeremy Intan, Matthew D Sinclair, Joseph Devietti, Timothy G Rogers, and Tor M Aamodt. 2020. Deterministic atomic buffering. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 981–995.

[31] Keenan Crane, Ulrich Pinkall, and Peter Schröder. 2013. Robust fairing via conformal curvature flow. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 1–10.

[32] Preyesh Dalmia, Rohan Mahapatra, and Matthew D Sinclair. 2022. Only buffer when you need to: Reducing on-chip gpu traffic with reconfigurable local atomic buffers. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 676–691.

[33] Sina Darabi, Mohammad Sadrosadati, Negar Akbarzadeh, Joël Lindegger, Mohammad Hosseini, Jisung Park, Juan Gómez-Luna, Onur Mutlu, and Hamid Sarbazi-Azad. 2022. Morpheus: Extending the last level cache capacity in GPU systems using idle GPU core resources. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 228–244.

[34] Simon Garcia De Gonzalo, Sitao Huang, Juan Gómez-Luna, Simon Hammond, Onur Mutlu, and Wen-mei Hwu. 2019. Automatic generation of warp-level primitives and atomic instructions for fast and portable parallel reduction on GPUs. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 73–84.

[35] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. 2016. Persistent rnns: Stashing recurrent weights on-chip. In *International Conference on Machine Learning*. PMLR, 2024–2033.

[36] Bardienus P Duisterhof, Zhao Mandi, Yunchao Yao, Jia-Wei Liu, Mike Zheng Shou, Shuran Song, and Jeffrey Ichnowski. 2023. Md-splatting: Learning metric deformation from 4d gaussians in highly deformable scenes. *arXiv preprint arXiv:2312.00583* (2023).

[37] Sankeerth Durvasula, Adrian Zhao, Fan Chen, Ruofan Liang, Pawan Kumar Sanjaya, and Nandita Vijaykumar. 2023. DISTWAR: Fast Differentiable Rendering on Raster-based Rendering Pipelines. *arXiv preprint arXiv:2401.05345* (2023).

[38] Ian J Egielski, Jesse Huang, and Eddy Z Zhang. 2014. Massive atomics for massive parallelism on GPUs. *ACM SIGPLAN Notices* 49, 11 (2014), 93–103.

[39] Jiemin Fang, Taoran Yi, Xinggang Wang, Lingxi Xie, Xiaopeng Zhang, Wenyu Liu, Matthias Nießner, and Qi Tian. 2022. Fast dynamic radiance fields with time-aware neural voxels. In *SIGGRAPH Asia 2022 Conference Papers*. 1–9.

[40] Sean Franey and Mikko Lipasti. 2013. Accelerating atomic operations on GPGPUs. In *2013 Seventh IEEE/ACM International Symposium on Networks-on-Chip (NoCS)*. IEEE, 1–8.

[41] Linus Franke, Darius Rückert, Laura Fink, and Marc Stamminger. 2024. TRIPS: Trilinear Point Splatting for Real-Time Radiance Field Rendering. In *Computer Graphics Forum*. Wiley Online Library, e15012.

[42] Sara Fridovich-Keil, Giacomo Meanti, Frederik Rahbæk Warburg, Benjamin Recht, and Angjoo Kanazawa. 2023. K-planes: Explicit radiance fields in space, time, and appearance. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 12479–12488.

[43] Sara Fridovich-Keil, Alex Yu, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. 2022. Plenoxels: Radiance fields without neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 5501–5510.

[44] Yonggan Fu, Zhifan Ye, Jiayi Yuan, Shunyao Zhang, Sixu Li, Haoran You, and Yingyan Lin. 2023. Gen-NeRF: Efficient and Generalizable Neural Radiance Fields via Algorithm-Hardware Co-Design. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–12.

[45] Masaaki Fushimi, Takahiro Kawashima, Takafumi Nose, Nobutaka Ihara, Shinji Sumimoto, and Naoyuki Shida. 2019. A Memory Saving Communication Method Using Remote Atomic Operations. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. 36–42.

[46] Sharath Girish, Kamal Gupta, and Abhinav Shrivastava. 2023. Eagles: Efficient accelerated 3d gaussians with lightweight encodings. *arXiv preprint arXiv:2312.04564* (2023).

[47] Gottlieb, Grishman, Kruskal, McAuliffe, Rudolph, and Snir. 1983. The NYU ultracomputer—Designing an MIMD shared memory parallel computer. *IEEE Transactions on computers* 100, 2 (1983), 175–189.

[48] Antoine Guédon and Vincent Lepetit. 2024. Sugar: Surface-aligned gaussian splatting for efficient 3d mesh reconstruction and high-quality mesh rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 5354–5363.

[49] Yong He, Kayvon Fatahalian, and Tim Foley. 2018. Slang: language mechanisms for extensible real-time shading systems. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 1–13.

[50] Peter Hedman, Pratul P Srinivasan, Ben Mildenhall, Jonathan T Barron, and Paul Debevec. 2021. Baking neural radiance fields for real-time view synthesis. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 5875–5884.

[51] Lee Howes and Aaftab Munshi. 2015. The OpenCL specification, version 2.0. *Khronos Group* (2015).

[52] Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, and Delio Vicini. 2022. DR. JIT: a just-in-time compiler for differentiable rendering. *ACM Transactions on Graphics (TOG)* 41, 4 (2022), 1–19.

[53] Hyeran Jeon, Hodjat Asghari Esfeden, Nael B Abu-Ghazaleh, Daniel Wong, and Sindhuja Elango. 2019. Locality-aware gpu register file. *IEEE Computer Architecture Letters* 18, 2 (2019), 153–156.

[54] Nikhil Keetha, Jay Karhade, Krishna Murthy Jatavallabhula, Gengshan Yang, Sebastian Scherer, Deva Ramanan, and Jonathon Luiten. 2024. SplaTAM: Splat Track & Map 3D Gaussians for Dense RGB-D SLAM. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 21357–21366.

[55] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 2023. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics (ToG)* 42, 4 (2023), 1–14.

[56] Leonid Keselman. 2023. *Gaussian Representations for Differentiable Rendering and Optimization*. Ph. D. Dissertation. Carnegie Mellon University.

[57] Leonid Keselman and Martial Hebert. 2023. Flexible Techniques for Differentiable Rendering with 3D Gaussians. *arXiv preprint arXiv:2308.14737* (2023).

[58] Mahmoud Khairy, Zhesheng Shen, Tor M Aamodt, and Timothy G Rogers. 2020. Accel-Sim: An extensible simulation framework for validated GPU modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 473–486.

[59] Farzad Khorasani, Hodjat Asghari Esfeden, Nael Abu-Ghazaleh, and Vivek Sarkar. 2018. In-register parameter caching for dynamic neural nets with virtual persistent processor specialization. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 377–389.

[60] Benjamin Klenk, Nan Jiang, Greg Thorson, and Larry Dennison. 2020. An in-network architecture for accelerating shared-memory multiprocessor collectives. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 996–1009.

[61] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. 2017. Tanks and temples: Benchmarking large-scale scene reconstruction. *ACM Transactions on Graphics (ToG)* 36, 4 (2017), 1–13.

[62] Konstantinos Koukos, Alberto Ros, Erik Hagersten, and Stefanos Kaxiras. 2016. Building heterogeneous unified virtual memories (uvms) without the overhead. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 1 (2016), 1–22.

[63] Samuli Laine, Janne Hellsten, Tero Karras, Yeongho Seol, Jaakko Lehtinen, and Timo Aila. 2020. Modular Primitives for High-Performance Differentiable Rendering. *ACM Transactions on Graphics* 39, 6 (2020).

[64] Christoph Lassner and Michael Zollhofer. 2021. Pulsar: Efficient sphere-based neural rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 1440–1449.

[65] Verica Lazova, Vladimir Guzov, Kyle Olszewski, Sergey Tulyakov, and Gerard Pons-Moll. 2023. Control-nerf: Editable feature volumes for scene rendering and manipulation. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 4340–4350.

[66] Junseo Lee, Kwanseok Choi, Jungi Lee, Seokwon Lee, Joonho Whangbo, and Jaewoong Sim. 2023. NeuRex: A Case for Neural Rendering Acceleration. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–13.

[67] Chaojian Li, Sixu Li, Yang Zhao, Wenbo Zhu, and Yingyan Lin. 2022. RT-NeRF: Real-Time On-Device Neural Radiance Fields Towards Immersive AR/VR Rendering. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 1–9.

[68] Ruilong Li, Hang Gao, Matthew Tancik, and Angjoo Kanazawa. 2023. Nerfacc: Efficient sampling accelerates nerfs. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 18537–18546.

[69] Sixu Li, Chaojian Li, Wenbo Zhu, Boyang Yu, Yang Zhao, Cheng Wan, Haoran You, Huihong Shi, and Yingyan Lin. 2023. Instant-3D: Instant Neural Radiance Field Training Towards On-Device AR/VR 3D Reconstruction. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–13.

[70] Yingtai Li, Xueming Fu, Shang Zhao, Ruiyang Jin, and S Kevin Zhou. 2023. Sparse-view ct reconstruction with 3d gaussian volumetric representation. *arXiv preprint arXiv:2312.15676* (2023).

[71] Yiqing Liang, Numair Khan, Zhengqin Li, Thu Nguyen-Phuoc, Douglas Lanman, James Tompkin, and Lei Xiao. 2023. GauFRe: Gaussian Deformation Fields for Real-time Dynamic Novel View Synthesis. *arXiv preprint arXiv:2312.11458* (2023).

[72] Jiaqi Lin, Zhihao Li, Xiao Tang, Jianzhuang Liu, Shiyong Liu, Jiayue Liu, Yangdi Lu, Xiaofei Wu, Songcen Xu, Youliang Yan, et al. 2024. Vastgaussian: Vast 3d gaussians for large scene reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 5166–5175.

[73] Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. 2020. Neural sparse voxel fields. *Advances in Neural Information Processing Systems* 33 (2020), 15651–15663.

[74] Jonathon Luiten, Georgios Kopanas, Bastian Leibe, and Deva Ramanan. 2024. Dynamic 3d gaussians: Tracking by persistent dynamic view synthesis. In *2024 International Conference on 3D Vision (3DV)*. IEEE, 800–809.

[75] Hidenobu Matsuki, Riku Murai, Paul HJ Kelly, and Andrew J Davison. 2024. Gaussian splatting slam. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 18039–18048.

[76] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. 2021. Nerf: Representing scenes as neural radiance fields for view synthesis. *Commun. ACM* 65, 1 (2021), 99–106.

[77] Muhammad Husnain Mubarik, Ramakrishna Kanungo, Tobias Zirr, and Rakesh Kumar. 2023. Hardware Acceleration of Neural Graphics. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–12.

[78] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2019. PHI: Architectural support for synchronization-and bandwidth-efficient commutative scatter updates. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 1009–1022.

[79] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. 2022. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Transactions on Graphics (ToG)* 41, 4 (2022), 1–15.

[80] Jacob Munkberg, Jon Hasselgren, Tianchang Shen, Jun Gao, Wenzheng Chen, Alex Evans, Thomas Müller, and Sanja Fidler. 2022. Extracting Triangular 3D Models, Materials, and Lighting From Images. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 8280–8290.

[81] Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. 2019. Mitsuba 2: A retargetable forward and inverse renderer. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–17.

[82] Albert Pumarola, Enric Corona, Gerard Pons-Moll, and Francesc Moreno-Noguer. 2021. D-nerf: Neural radiance fields for dynamic scenes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 10318–10327.

[83] Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari. 2020. Accelerating 3d deep learning with pytorch3d. *arXiv preprint arXiv:2007.08501* (2020).

[84] Brandon Reagen, Udit Gupta, Lillian Pentecost, Paul Whatmough, Sae Kyu Lee, Niamh Mulholland, David Brooks, and Gu-Yeon Wei. 2018. Ares: A framework for quantifying the resilience of deep neural networks. In *Proceedings of the 55th Annual Design Automation Conference*. 1–6.

[85] Xiaowei Ren and Mieszko Lis. 2017. Efficient sequential consistency in gpus via relativistic cache coherence. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 625–636.

[86] Xiaowei Ren, Daniel Lustig, Evgeny Bolotin, Aamer Jaleel, Oreste Villa, and David Nellans. 2020. Hmg: Extending cache coherence protocols across modern hierarchical multi-gpu systems. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 582–595.

[87] Darius Rückert, Linus Franke, and Marc Stamminger. 2022. Adop: Approximate differentiable one-pixel point rendering. *ACM Transactions on Graphics (ToG)* 41, 4 (2022), 1–14.

[88] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. 2015. Evaluating the cost of atomic operations on modern architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 445–456.

[89] Steven L Scott. 1996. Synchronization and communication in the T3E multiprocessor. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*. 26–36.

[90] Yahao Shi, Yanmin Wu, Chenming Wu, Xing Liu, Chen Zhao, Haocheng Feng, Jingtuo Liu, Liangjun Zhang, Jian Zhang, Bin Zhou, et al. 2023. Gir: 3d gaussian inverse rendering for relightable scene factorization. *arXiv preprint arXiv:2312.05133* (2023).

[91] Matthew D Sinclair, Johnathan Alsop, and Sarita V Adve. 2015. Efficient GPU synchronization without scopes: Saying no to complex consistency models. In *Proceedings of the 48th International Symposium on Microarchitecture*. 647–659.

[92] Inderpreet Singh, Arrvindh Shriraman, Wilson WL Fung, Mike O'Connor, and Tor M Aamodt. 2013. Cache coherence for GPU architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 578–590.

[93] Cheng Sun, Min Sun, and Hwann-Tzong Chen. 2022. Improved direct voxel grid optimization for radiance fields reconstruction. *arXiv preprint arXiv:2206.05085* (2022).

[94] Abdulaziz Tabbakh, Xuehai Qian, and Murali Annavaram. 2018. G-tsc: Timestamp based coherence for gpus. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 403–415.

[95] Matthew Tancik, Ethan Weber, Evonne Ng, Ruilong Li, Brent Yi, Terrance Wang, Alexander Kristoffersen, Jake Austin, Kamyar Salahi, Abhik Ahuja, et al. 2023. Nerfstudio: A modular framework for neural radiance field development. In *ACM SIGGRAPH 2023 Conference Proceedings*. 1–12.

[96] Jiaxiang Tang, Jiawei Ren, Hang Zhou, Ziwei Liu, and Gang Zeng. 2023. Dreamgaussian: Generative gaussian splatting for efficient 3d content creation. *arXiv preprint arXiv:2309.16653* (2023).

[97] Ayush Tewari, Ohad Fried, Justus Thies, Vincent Sitzmann, Stephen Lombardi, Kalyan Sunkavalli, Ricardo Martin-Brualla, Tomas Simon, Jason Saragih, Matthias Nießner, et al. 2020. State of the art on neural rendering. In *Computer Graphics Forum*, Vol. 39. Wiley Online Library, 701–727.

[98] Xi Wang, Brody Williams, John D Leidel, Alan Ehret, Michel Kinsy, and Yong Chen. 2020. Remote atomic extension (rae) for scalable high performance computing. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[99] Craig M Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. 2011. Fermi GF100 GPU architecture. *IEEE Micro* 31, 2 (2011), 50–59.

[100] Guanjun Wu, Taoran Yi, Jiemin Fang, Lingxi Xie, Xiaopeng Zhang, Wei Wei, Wenyu Liu, Qi Tian, and Xinggang Wang. 2024. 4d gaussian splatting for real-time dynamic scene rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 20310–20320.

[101] Rundi Wu, Ben Mildenhall, Philipp Henzler, Keunhong Park, Ruiqi Gao, Daniel Watson, Pratul P Srinivasan, Dor Verbin, Jonathan T Barron, Ben Poole, et al. 2024. Reconfusion: 3d reconstruction with diffusion priors. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 21551–21561.

[102] Songm Xinkai, Yuanbo Wen, Xing Hu, Tianbo Liu, Haoxuan Zhou, Husheng Han, Tian Zhi, Zidong Du, Lim Wei, Rui Zhang, Chen Zhang, Lin Gao, Qi Guo, and Tianshi Chen. 2023. ARTist: A Fully Fused Accelerator for Real-Time Learning of Neural Scene Representation. In *Proceedings of the 56th International Symposium on Microarchitecture*. 1–13.

[103] Chi Yan, Delin Qu, Dan Xu, Bin Zhao, Zhigang Wang, Dong Wang, and Xuelong Li. 2024. Gs-slam: Dense visual slam with 3d gaussian splatting. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 19595–19604.

[104] Yunzhi Yan, Haotong Lin, Chenxu Zhou, Weijie Wang, Haiyang Sun, Kun Zhan, Xianpeng Lang, Xiaowei Zhou, and Sida Peng. 2024. Street gaussians for modeling dynamic urban scenes. *arXiv preprint arXiv:2401.01339* (2024).

[105] Chen Yang, Sikuang Li, Jiemin Fang, Ruofan Liang, Lingxi Xie, Xiaopeng Zhang, Wei Shen, and Qi Tian. 2024. GaussianObject: Just Taking Four Images to Get A High-Quality 3D Object with Gaussian Splatting. *arXiv preprint arXiv:2402.10259* (2024).

[106] Ziyi Yang, Xinyu Gao, Wen Zhou, Shaohui Jiao, Yuqing Zhang, and Xiaogang Jin. 2024. Deformable 3d gaussians for high-fidelity monocular dynamic scene reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 20331–20341.

[107] Zeyu Yang, Hongye Yang, Zijie Pan, and Li Zhang. 2024. Real-time Photorealistic Dynamic Scene Representation and Rendering with 4D Gaussian Splatting. *International Conference on Learning Representations (ICLR)*.

[108] Zeyu Yang, Hongye Yang, Zijie Pan, Xiatian Zhu, and Li Zhang. 2023. Real-time photorealistic dynamic scene representation and rendering with 4d gaussian splatting. *arXiv preprint arXiv:2310.10642* (2023).

[109] Taoran Yi, Jiemin Fang, Guanjun Wu, Lingxi Xie, Xiaopeng Zhang, Wenyu Liu, Qi Tian, and Xinggang Wang. 2023. GaussianDreamer: Fast Generation from Text to 3D Gaussian Splatting with Point Cloud Priors. *arXiv preprint arXiv:2310.08529* (2023).

[110] Alex Yu, Ruilong Li, Matthew Tancik, Hao Li, Ren Ng, and Angjoo Kanazawa. 2021. Plenoctrees for real-time rendering of neural radiance fields. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 5752–5761.

[111] Lingqi Zhang, Mohamed Wahib, Peng Chen, Jintao Meng, Xiao Wang, Toshio Endo, and Satoshi Matsuoka. 2023. PERKS: a Locality-Optimized Execution Model for Iterative Memory-bound GPU Applications. In *Proceedings of the 37th International Conference on Supercomputing*. 167–179.

[112] Xiaoyu Zhou, Zhiwei Lin, Xiaojun Shan, Yongtao Wang, Deqing Sun, and Ming-Hsuan Yang. 2024. Drivinggaussian: Composite gaussian splatting for surrounding dynamic autonomous driving scenes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 21634–21643.

[113] Feiwen Zhu, Jeff Pool, Michael Andersch, Jeremy Appleyard, and Fung Xie. 2018. Sparse persistent RNNs: Squeezing large recurrent networks on-chip. *arXiv preprint arXiv:1804.10223* (2018).

[114] Wojciech Zielonka, Timur Bagautdinov, Shunsuke Saito, Michael Zollhöfer, Justus Thies, and Javier Romero. 2023. Drivable 3D Gaussian Avatars. *arXiv preprint arXiv:2311.08581* (2023).

[115] Zi-Xin Zou, Zhipeng Yu, Yuan-Chen Guo, Yangguang Li, Ding Liang, Yan-Pei Cao, and Song-Hai Zhang. 2024. Triplane meets gaussian splatting: Fast and generalizable single-view 3d reconstruction with transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 10324–10335.