

Cicero: Addressing Algorithmic and Architectural Bottlenecks in Neural Rendering by Radiance Warping and Memory Optimizations

Yu Feng*

Shanghai Jiao Tong University
University of Rochester
y-feng@sjtu.edu.cn

Zihan Liu

Shanghai Jiao Tong University
altair.liu@sjtu.edu.cn

Jingwen Leng

Shanghai Jiao Tong University
leng-jw@cs.sjtu.edu.cn

Minyi Guo

Shanghai Jiao Tong University
guo-my@cs.sjtu.edu.cn

Yuhao Zhu

University of Rochester
yzhu@rochester.edu

Abstract—Neural Radiance Field (NeRF) is widely seen as an alternative to traditional physically-based rendering. However, NeRF has not yet seen its adoption in resource-limited mobile systems such as Virtual and Augmented Reality (VR/AR), because it is simply extremely slow. On a mobile Volta GPU, even the state-of-the-art NeRF models generally execute only at 0.8 FPS. We show that the main performance bottlenecks are both algorithmic and architectural. We introduce, CICERO, to tame both forms of inefficiencies. We first introduce two algorithms, one fundamentally reduces the amount of work any NeRF model has to execute, and the other eliminates irregular DRAM accesses. We then describe an on-chip data layout strategy that eliminates SRAM bank conflicts. A pure software implementation of CICERO offers an $8.0\times$ speed-up and $7.9\times$ energy saving over a mobile Volta GPU. When compared to a baseline with a dedicated DNN accelerator, our speed-up and energy reduction increase to $28.2\times$ and $37.8\times$, respectively — all with minimal quality loss (less than 1.0 dB peak signal-to-noise ratio reduction).

Index Terms—Mobile Architecture; NeRF Acceleration; VR

I. INTRODUCTION

Neural Radiance Field (NeRF) [55] revives classic image-based rendering [76], [80] using modern deep learning techniques, and has become an attractive alternative to conventional photorealistic rendering methods such as path tracing [21], [66], [69]. However, NeRF is excessively slow [55]. Despite numerous efforts [15], [39], [53], [59], [64], [79], [87], NeRF rendering performance is still far from real-time on mobile devices. On a mobile Volta GPU on Nvidia’s Xavier SoC, common models like DirectVoxGO [79] merely achieve 0.8 Frame Per Second (FPS), while Instant-NGP [59] takes over 6 s to render a 800×800 frame.

Accelerating NeRF models is critical, but one must not over-specialize for a specific model because of the rapid evolution in algorithm design. Since its inception, NeRF models have gone through several major architectural changes, from the original grid/voxel-based design [79] to those using hierarchical data structures [59], [87] and, more recently, point-based rendering

through Gaussian splatting [16], [42], [84]. It is conceivable that NeRF models still have to go through a few more iterations to mature. Thus, any system and architecture support must address fundamental bottlenecks that are central to neural rendering rather than artifacts of specific models.

We identify inherent bottlenecks of NeRF models both in algorithm design and in the underlying hardware (Sec. II). Algorithmically, a NeRF model has to compute the radiance of millions of rays, each carrying hundreds of ray samples. Each ray sample executes a Multilayer Perceptron (MLP) inference, collectively incurring a high computational cost [55]. Architecturally, the actual computation of the MLPs introduces irregular memory accesses, stemming from the way that ray samples are grouped and accessed in NeRF. Consequently, NeRF rendering results in many irregular DRAM accesses and SRAM bank conflicts.

This paper proposes CICERO, an algorithm-architecture co-designed approach to tame both forms of inefficiency. Algorithm-wise, we introduce a plug-and-play extension to existing NeRF algorithms, fundamentally reducing the computational workload of a NeRF model (Sec. III). In particular, we exploit radiance proximity: the radiances of nearby rays emanating from the same physical point (radiant exitance) are approximately the same. We propose *sparse radiance warping* (SPARW), which avoids up to 88% of the radiance computation by reusing ray radiances computed in previous frames. Critically, SPARW is not a new NeRF model. Rather, it is an extension that can be easily integrated into virtually all existing NeRF methods, widening its applicability.

While SPARW avoids a large portion of radiance computation, SPARW does not completely eliminate it. Whenever radiance computation is needed, it is bottlenecked by irregular DRAM accesses and frequently on-chip SRAM bank conflicts. We further propose two optimization techniques that mitigate these memory inefficiencies in radiance computation.

To eliminate irregular DRAM accesses (Sec. IV-A), our idea is to convert NeRF inference from a *pixel-centric* order to

*Work done while at University of Rochester.

a *memory-centric* order. The pixel-centric rendering follows the order of rays (ultimately image pixels) and their samples, resulting in discontinuous memory accesses. The memory-centric order, in contrast, accesses voxels in the scene sequentially, inherently yielding full-streaming memory accesses.

To eliminate SRAM bank conflicts (Sec. IV-B), we explore a new data layout strategy in on-chip SRAMs. We show that the traditional feature-major data layout, which allocates all the channels of a feature vector in the same SRAM bank, necessarily introduces frequent SRAM bank conflicts. Instead, we propose a channel-major layout, where different channels of the same feature vector are spread across SRAM banks. With the co-designed hardware architecture, the new data layout can completely eliminate on-chip bank conflicts.

We integrate CICERO with three state-of-the-art NeRF models. A pure software implementation achieves an $8.0\times$ speed-up and $7.9\times$ energy saving over a mobile Volta GPU. Furthermore, compared to a baseline with a dedicated DNN accelerator, our speed-up and energy reduction increase to $28.2\times$ and $37.8\times$, respectively.

The contributions of this paper are as follows.

- We introduce SPARW, a novel algorithm that reduces up to 88% of the MLP computations in NeRF by exploiting radiance similarities between nearby rays.
- We propose a fully-streaming NeRF rendering algorithm that reduces the redundant DRAM access and ensures completely streaming DRAM accesses.
- We propose an on-chip data layout and the associated hardware support that eliminate SRAM bank conflicts.
- We demonstrate that CICERO achieves a $28.2\times$ speed-up and $37.8\times$ energy savings over a baseline that has a dedicated DNN accelerator while maintaining less than 1.0 dB degradation in Peak Signal-to-Noise Ratio (PSNR).

II. MOTIVATION

We begin by discussing NeRF vs. conventional rendering methods (Sec. II-A). We then overview the general pipeline of today’s NeRF rendering model (Sec. II-B). We then characterize NeRF algorithms to identify Feature Gathering as a performance bottleneck (Sec. II-C). Finally, we characterize the memory inefficiencies in Feature Gathering (Sec. II-D).

II-A. Why NeRF?

NeRF vs. traditional ray tracing (physically-based rendering) [69] is widely debated in graphics. Our work does not aim to settle that debate but, rather, to contribute to that comparison by allowing NeRF to be a more appealing option.

While NeRF is generally slower, it has two advantages compared to ray tracing. First, NeRF promises better rendering quality, because the complicated light-matter interactions are learned through data using deep learning methods rather than physically simulated. Second, ray tracing requires a more complicated setup, e.g., modeling the geometry of the scene and describing material properties. NeRF, in contrast, is a form of classic image-based rendering [76], [80], which uses a set of offline-captured images of the scene without any modeling.

From offline captured images of a scene, NeRF trains a differentiable model, which encodes the volume density and the light field in the scene (i.e., radiance of any ray) [30], [50]. At rendering time, given the camera pose where an image is to be rendered, the model is probed through the classic volume rendering [41], [49] to synthesize the image.

II-B. NeRF Rendering Pipeline

We focus on MLP-based models. More recent 3D Gaussian Splatting (3DGS) models [16], [42], [84] do away with MLPs; they are generally faster (still far from real-time on mobile devices) at the cost of much larger model sizes. How CICERO can be applied to 3DGS models will be discussed in Sec. VIII. State-of-the-art NeRF models have a general pipeline that consists of three stages: Indexing (\mathcal{I}), Feature Gathering (\mathcal{G}), and Feature Computation (\mathcal{F}), as illustrated in Fig. 1.

Indexing (\mathcal{I}). The entire 3D scene is initially partitioned into many adjacent voxels. Each voxel is a cube with eight vertices in the space. Each vertex carries a high-dimensional feature vector that is offline trained. During rendering, we first generate a ray for each pixel to be rendered. Each ray samples a set of points (e.g. S_1 , S_2 , and S_3 in Fig. 1) along its direction. Each ray sample, based on its sampled position in the space, calculates the ID of the voxel that contains the sample.

Feature Gathering (\mathcal{G}). Using the voxel ID, each ray sample finds the eight vertices of that voxel and gathers the features of the eight vertices. In the example in Fig. 1, S_1 would access the eight vertex features in V_3 . These features encode both the density and radiance field at the corresponding vertex locations. This ray sample then computes its feature by trilinearly interpolating the feature vectors of the eight vertices.

Feature Computation (\mathcal{F}). In Feature Computation, each ray sample passes its intermediate feature through a lightweight MLP model to obtain the actual density and radiance value of that sample. The final color of a ray (and thus the color of the pixel that is hit by the ray) would be computed by accumulating all ray samples along the ray direction.

II-C. Computation Characterizations

Performance and Model Size. Today’s NeRF models are not only slow, but they also have model sizes that do not fit in on-chip SRAMs. Fig. 2 shows the frame rate on a mobile Volta GPU [5] (y -axis) and model size (x -axis) of several state-of-the-art NeRF models [15], [19], [35], [39], [59], [79]. We also overlay the 60 FPS frame rate requirement. NeRF algorithms rarely achieve real-time rendering on today’s commodity mobile devices.

In addition, NeRF model sizes far exceed the on-chip SRAM sizes affordable on today’s mobile SoC, necessitating frequent DRAM accesses. A NeRF model’s size includes both the feature vectors of the voxels and the MLP model weights; the former dominates the total size. The feature vectors are usually at the order of 10 MB – 1,000 MB, whereas the MLP weights are generally small (10 KB – 100 KB).

Performance Bottleneck. We characterize the performance bottlenecks across NeRF algorithms. Fig. 3 shows the execution breakdown of different stages across four popular NeRF

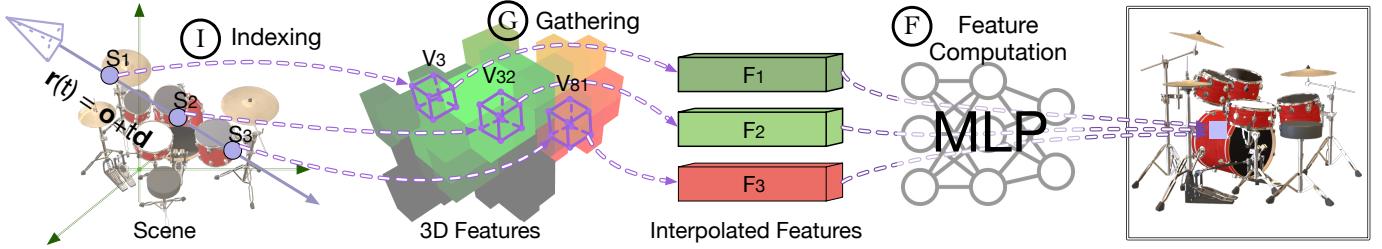


Fig. 1: The rendering pipeline of today’s NeRF algorithms. The computation flow is highlighted in purple. Each ray first samples points, S_1 , S_2 , and S_3 , along the ray direction. Each ray sample gathers and interpolates 3D features from eight vertices of the intersected voxel (V_3 , and V_{32} , and V_{81}). The interpolated features (F_1 , F_2 , and F_3) are then fed into the MLP to get the partial pixel values at the three ray samples. The final pixel value is accumulated from all partial pixel values [55].

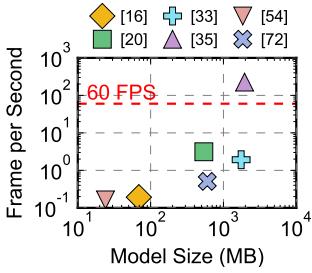


Fig. 2: Frame rate vs. model size on the Xavier SoC [5]. Models [15], [19], [35], [39], [59], [79] are named by reference numbers.

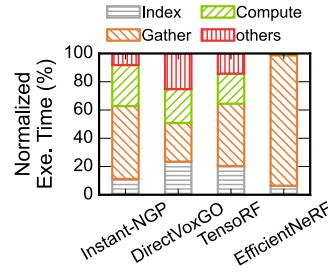


Fig. 3: Normalized execution breakdown across state-of-the-art NeRF algorithms [15], [39], [59], [79].

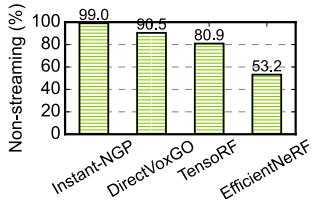


Fig. 4: Percentage of non-continuous DRAM accesses in feature gathering.

algorithms on a mobile Volta GPU [5]. All three stages take non-trivial execution time with Feature Gathering dominating the execution (>56% of execution time on average).

II-D. Memory Inefficiencies in NeRF

Given that Feature Gathering is memory-heavy, we further characterize its memory accesses. We find that, while Feature Gathering is computationally parallel (between rays and between samples on a ray), it is not memory-friendly, introducing both irregular DRAM accesses and frequent on-chip bank conflicts. The following paragraphs provide a quantitative analysis to illustrate these issues.

DRAM Access Inefficiency. The inefficiency in DRAM access can be attributed to two main factors: non-streaming DRAM access and redundant DRAM access, both stemming from the inherent *pixel-centric* rendering in NeRF models.

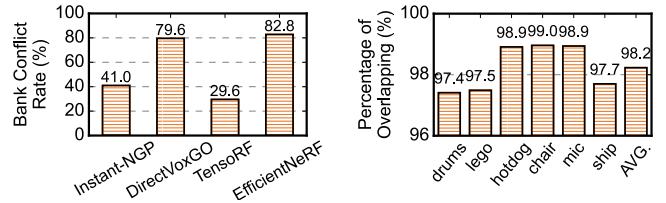


Fig. 5: Cache miss rate in feature gathering across common NeRF algorithms.

Fig. 6: SRAM bank conflict rate in feature gathering, assuming 16 banks and 16 concurrent ray queries.

Fig. 7: The overlapping percentage across six scenes in Synthetic-NeRF [55].

NeRF inference is parallelized across pixels, where adjacent pixels in the final image are computed simultaneously. This pixel-centric rendering introduces two levels of memory-access irregularities: inter-ray irregularity and intra-ray irregularity. Inter-ray irregularity means that rays of adjacent pixels might access non-continuous memory regions. This is because rays might diverge as they transport over space, even when their origins are spatially close.

Intra-ray irregularity is caused by sampling points along a single camera ray accessing discontinuous memory regions. Specifically, the feature vectors corresponding to the different ray samples can be stored at arbitrary memory locations. As shown in Fig. 1, ray samples, S_1 and S_2 , intersect voxels, V_3 and V_{32} , which are spatially distant and, thus, not stored continuously in DRAM. Fig. 4 shows the non-streaming DRAM access in four popular NeRF algorithms. On average, over 81% of DRAM access is non-streaming.

Irregular accesses mean that each voxel might be accessed multiple times during rendering, which leads to redundant DRAM accesses, given that the feature vectors cannot be stored completely on-chip (Fig. 2). Assuming a 2 MB on-chip buffer with oracle replacement [5], Fig. 5 shows the cache miss rates of different NeRF algorithms during feature gathering; the miss rate can be as high as 92% (average 38%). In reality, an even smaller buffer would be allocated to accommodate other operations and data structures.

SRAM Access Inefficiency. On-chip memory access in NeRF results in frequent bank conflicts. In conventional DNNs, the memory access patterns can be determined statically. Thus, bank conflicts can be eliminated through metic-

ulous data layout across SRAM banks [43], [90]. Conversely, the data access pattern of Feature Gathering in NeRF depends on the camera view and cannot be known offline.

Sec. IV-B will provide a detailed description of the causes of bank conflicts in Feature Gathering. Here, we simply show the bank conflict rate of Feature Gathering across NeRF algorithms (Fig. 6). Assuming a 2 MB buffer with 16 banks and 16 concurrent camera rays, the average bank conflict rate is 52%, with EfficientNeRF reaching as high as 83%. A larger number of concurrent rays would lead to a higher bank conflict rate. For instance, the bank conflict rate of Instant-NGP increases to 80% when the number of rays escalates to 64. Increasing the number of banks does reduce the bank conflict rate. However, heavily banked SRAM designs are highly undesirable due to costly crossbars [7], [32].

III. SPARSE RADIANCE WARPING

This section introduces, *sparse radiance warping* (SPARW), an algorithm that exploits the radiance similarity across rays from nearby camera views. We first provide an intuition of SPARW algorithm (Sec. III-A), followed by a description of the overall algorithm (Sec. III-B). Finally, we discuss two key aspects in the SPARW algorithm design that helps improve performance and rendering quality (Sec. III-C).

III-A. Intuition

The goal of SPARW is to reuse pixel values rendered in previous frames using a technique called image warping. Fig. 8 illustrates our idea, which starts from a previously rendered frame, called a reference frame, F_{ref} . For a given pixel in F_{ref} , say P_x , we can find the point in the scene P that is captured by that pixel. When rendering a new target frame F_{tgt} , the same point P is captured as a new pixel P_y in the target frame F_{tgt} . The assumption here is that if the camera poses of F_{tgt} and F_{ref} are sufficiently close, the radiance of both \overline{PP}_x ray and \overline{PP}_y ray are approximately similar. Thus, the pixel value P_x can be simply reused in P_y , avoiding rendering P_y through the compute-intensive NeRF model.

This warping idea avoids rendering pixels in F_{tgt} whose corresponding scene points are also captured by F_{ref} . The larger the overlap between F_{ref} and F_{tgt} is, the less NeRF computation is required. Fig. 7 characterizes the overlapping between two adjacent frames in the Synthetic-NeRF dataset [55]. More than 98% of pixels are overlapped (standard deviation: 1.7%), indicating that less than 2% of pixels require re-rendering. The same conclusion holds for real-world datasets: on the Unbounded-360 [10] and Tanks and Temples [44] datasets, only 4.3% and 4.9% pixels cannot be warped, respectively. The high overlap is not an artifact of a particular dataset but a fundamental attribute of real-time rendering, where consecutive frames are necessarily in close proximity because the observer/camera does not jump arbitrarily.

The non-overlapped pixels, called *disoccluded pixels*, arise when the previously occluded scene in F_{ref} becomes visible in F_{tgt} . Fig. 9 shows the effect of a naive warping. Without recalculating the disoccluded pixels, the rendered image F'_{tgt}

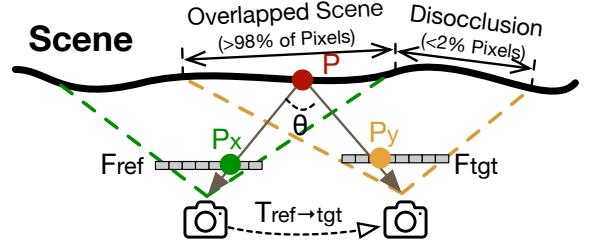


Fig. 8: Intuition of our SPARW algorithm. The radiance of the \overline{PP}_x ray can approximate the radiance of the \overline{PP}_y ray if the angle θ between these two rays is sufficiently close.

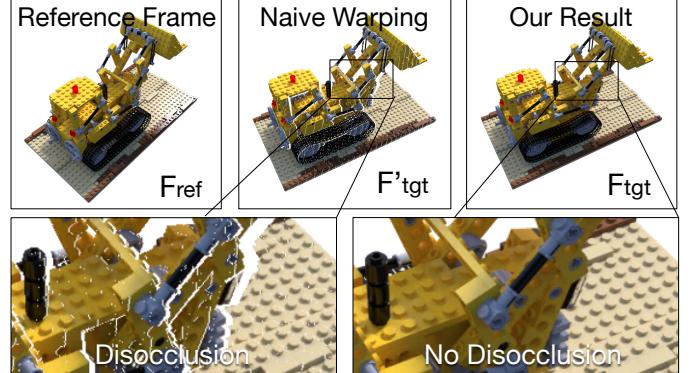


Fig. 9: Examples of a reference frame F_{ref} , a result of naive warping F'_{tgt} and our result F_{tgt} by SPARW. Note that disocclusions (missing pixels) are eliminated in our result.

has clear “holes”, because the disoccluded pixels cannot be warped from the reference frame. Our idea then is to calculate the disoccluded pixels using the original NeRF model, which now renders only a small amount of (e.g., 2%) sparsely disoccluded pixels in the target frame.

III-B. Basic Algorithm

In SPARW there are two rendering paths, which are illustrated in Fig. 10: a compute-intensive path (in green) to render reference frames (R_0 and R_1) using full-frame NeRF rendering, and a lightweight path (in orange) that uses the warping idea in Sec. III-A to render target frames ($T_0 - T_5$). We first describe how to warp from a reference frame to a target frame, then discuss the choice of reference frames.

Target Frame Rendering. There are four steps in rendering a target frame: ① point cloud conversion, ② transformation, ③ re-projection and ④ sparse NeRF rendering.

① Given a reference frame F_{ref} , we first convert F_{ref} into a point cloud P_{ref} , which represents the 3D scene in the reference camera coordinate system. The transformation uses scene depth and the camera’s intrinsic parameters. Mathematically, it can be expressed as follows:

$$P_{ref} = \begin{bmatrix} \frac{D_{ref}}{f} & 0 & -\frac{D_{ref}C_x}{f} \\ 0 & \frac{D_{ref}}{f} & -\frac{D_{ref}C_y}{f} \\ 0 & 0 & D_{ref} \end{bmatrix} \times F_{ref} \quad (1)$$

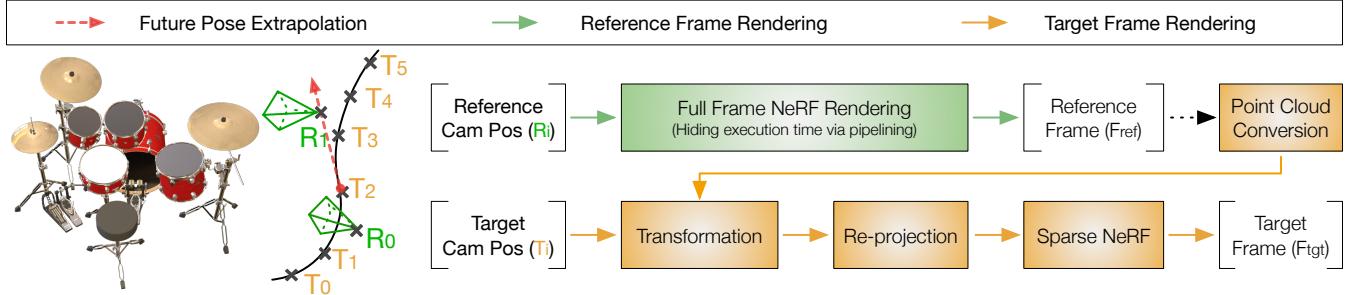


Fig. 10: An overview of the SPARW algorithm. Only reference frames (R_i) undergo full-frame NeRF inference as denoted by the green path. All target frames (T_i) are computed using the lightweight warping operations denoted by the orange path. The reference frames are not on the camera trajectory so reference frame rendering and target frame rendering can be overlapped (Fig. 11). Camera poses at reference frames are extrapolated using the poses of previous target frames.

where D_{ref} is the depths of points in P_{ref} corresponding to pixels in F_{ref} ; D_{ref} can be obtained through a standard rasterization pipeline (using depth buffer) [74]; f is the camera focal length; $[C_x, C_y]$ is the camera center. Both focal length and camera center are part of the camera’s intrinsic parameters [27].

② The P_{ref} calculated so far is expressed in the coordinate system of the reference frame. To render the target frame, we must transform the point cloud to the coordinate system of the target frame — using a simple linear transformation:

$$P_{tgt} = T_{ref \rightarrow tgt} \times P_{ref} \quad (2)$$

where $T_{ref \rightarrow tgt}$ is the transformation matrix between the reference camera pose R_i and the target camera pose T_i ; P_{tgt} denotes the point cloud in the target frame’s coordinate system.

③ Once we have P_{tgt} expressed at the current camera coordinate system, obtaining the frame at the current camera pose requires a standard perspective projection in the classic rasterization pipeline [74]:

$$F'_{tgt} = \begin{bmatrix} \frac{f}{D_{tgt}} & 0 & 0 & C_x \\ 0 & \frac{f}{D_{tgt}} & 0 & C_y \\ 0 & 0 & \frac{1}{D_{tgt}} & 0 \end{bmatrix} \times P_{tgt} \quad (3)$$

where D_{tgt} is the depth of all the points corresponding to the pixels in the target frame.

④ As shown in Fig. 9, naively warped frame F'_{tgt} contains disocclusion artifacts. To mitigate disocclusions, we simply run the original NeRF model for those disoccluded pixels,

$$F_{tgt} = F'_{tgt} \circledast \Gamma_{sp} \quad (4)$$

Γ_{sp} denotes sparse NeRF rendering of disoccluded pixels, and \circledast combines the warped pixels with the NeRF-rendered pixels.

Interestingly, “holes” in a target frame can be attributed to two factors: disocclusion and void (i.e., areas in the scene where there is nothing). To avoid unnecessary computation on the latter, we perform a simple depth test so that pixels whose depth is infinite are skipped in sparse NeRF rendering. The depth map of the current frame can be obtained through, again, the standard perspective projection. The overhead of

such a projection is minimal. In our measurement, the latency of processing, e.g., one million points, is less than one millisecond on a Nvidia Volta mobile GPU.

III-C. Key Design Decisions

Reference Frame Rendering. Target frame rendering relies on the existence of a reference frame, which in SPARW will be rendered via full-frame NeRF rendering. The key question is *how to choose the reference frames*. Technically any frame can be a reference frame, and the reference frames do not even have to be on the camera’s trajectory. In Fig. 10, the two reference frames R_0 and R_1 are indeed off the trajectory.

Rendering reference frames on the trajectory is a common strategy in previous work that uses temporal correlation [13], [26], [77], [92]. The advantage is that it is work-efficient: on-trajectory frames have to be rendered anyway. However, this approach also limits *when* a reference frame can be rendered: a reference frame can only be rendered when its actual camera pose is obtained. As a result, it necessarily serializes target frame rendering and reference frame rendering, as illustrated in Fig. 11a, because all frames (target or reference) must be rendered in the order in which the camera poses are obtained, which is necessarily sequential.

Instead, our observation is that the reference frames do not have to be actual frames that users see; they just provide information that can be reused in the target frames. As long as the reference frame is close to the camera trajectory, the radiance approximation still holds. This allows us to overlap reference frame rendering with target frame rendering.

This overlapping idea is illustrated in Fig. 10 with the corresponding timeline illustrated in Fig. 11b. The reference frame R_1 is off the trajectory. The camera position of R_1 is chosen so that it is close to the trajectory. In our implementation, we use the velocity at camera pose T_2 to extrapolate the position of R_1 [38]. This does not mean that rendering R_1 depends on the rendering result of T_2 ; rather, it depends only on the camera pose of T_2 , which is known before T_2 is rendered.

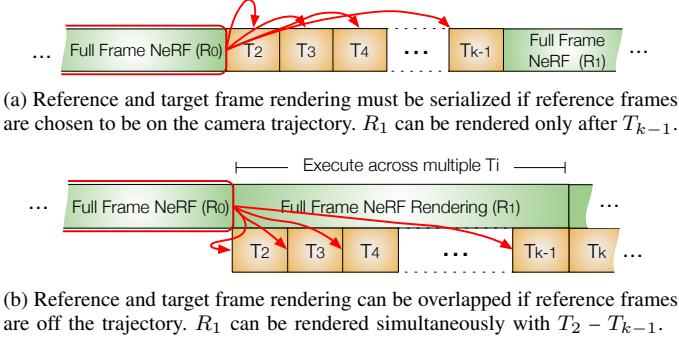


Fig. 11: Two choices of reference frames. Red arrows show how a reference frame is used to warp target frames, which is the same between the two methods. Our method (bottom) overlaps reference and target frame rendering.

Specifically, we use the position of the last two rendered frames, T_1 and T_2 , to calculate the velocity v at T_2 :

$$v = \frac{T_2 - T_1}{\Delta t}, \quad (5)$$

where Δt represents the interval between two consecutive frames. We then calculate the pose of the reference frame R_1 :

$$R_1 = T_2 + v \times t_r, \quad t_r = \frac{N}{2} \Delta t, \quad (6)$$

where N is the number of target frames that share the same reference frame (i.e., 4 in this example as $T_2 - T_5$ share R_1). Using $\frac{N}{2}$ allows the reference frame to be roughly at the center of, and thus increase the overlap with, its target frames.

Critically, while rendering R_1 using a compute-intensive full-frame NeRF model, we can *simultaneously* render the target frames $T_2 - T_5$ (using the lightweight warping computation) based on a previous reference frame R_0 , whose position itself is extrapolated from an earlier target frame such as T_0 . We show that one single reference can be used for up to 30 target frames with a minimal visual quality loss (Sec. VI-D). In this way, the long latency of rendering reference frames can be largely hidden behind rendering target frames.

Deciding When to Warp. A potential limitation of our SPARW algorithm is that the radiance approximation does not hold well when 1) consecutive frames have large differences in camera poses (i.e., θ in Fig. 8 is too large) and 2) the surface material is non-diffuse. As a simple heuristic to mitigate such issues, we allow warping only when the angle subtended by a ray in the reference frame and the corresponding ray in the target frame (i.e., θ in Fig. 8) is smaller than a threshold ϕ . We provide a more comprehensive discussion of this warping heuristics in Sec. VIII.

IV. MEMORY OPTIMIZATIONS

While SPARW reduces NeRF computation of target frames, reference frames still execute full-frame NeRF rendering, which is bottlenecked by redundant and irregular DRAM accesses of feature vectors and the frequent on-chip bank conflicts as shown in Sec. II-D. This section first describes an algorithmic optimization that eliminates redundant DRAM

accesses and guarantees fully-streaming DRAM accesses (Sec. IV-A). We then discuss a new on-chip data layout that eliminates SRAM bank conflicts (Sec. IV-B). Finally, we describe our co-designed hardware architecture that unleashes the two memory optimizations (Sec. IV-C).

IV-A. Fully-Streaming NeRF Rendering

Architectural Assumptions. We assume a DNN accelerator for MLP operations. The accelerator has an on-chip buffer (scratchpad) to store 3D voxel encoding (feature vectors) in the NeRF model. However, this buffer is generally too small (1 MB – 3 MB) to hold the entire 3D voxel encoding (10 MB – 1000 MB). Additionally, there is a dedicated on-chip buffer to store MLP weights of NeRF models; these weights are generally small (10 KB – 100 KB).

Memory-Centric Rendering. Recall from Sec. II-D that redundant and irregular DRAM accesses are caused by the *pixel-centric* rendering, where we parallelize the computation across all the pixels in the order that they appear in the final image. As a result, the ray samples during Feature Gathering (\mathcal{G}) access discontiguous memory regions.

Instead, we propose a *memory-centric* rendering, where the order of computation is based on where the ray samples reside in the DRAM. At a high level, we sequentially read, in chunks, voxel features that are contiguously laid in memory. As each chunk is loaded to the on-chip SRAM, we render the ray samples whose feature vectors happen to reside in the chunk. We throw away the chunk only after all the associated ray samples have been computed. In this way, we guarantee that each voxel feature is read only once and the DRAM accesses to the voxel features are fully streaming.

Memory-centric rendering incurs no storage overhead: the vertex features are stored in memory *as is* without duplication. What is being reordered is the order we *access* the features. We use Fig. 12 to illustrate the idea and how the three stages in NeRF rendering are changed accordingly.

Indexing (\mathcal{I}). We first group all the voxel features into macro voxels (MVoxels). All the data in a MVoxel is loaded to the SRAM together when the MVoxel is loaded. In the example of Fig. 12, we combine every 2×2 voxels into one MVoxel. In reality, we guarantee that the data size of one MVoxel is smaller than the on-chip buffer size. We store vertex features within one MVoxel continuously in the DRAM, and store MVoxels continuously in the DRAM.

We then compute a Ray Index Table (RIT), where each MVoxel has an entry. Each entry records the IDs of all the ray samples whose features reside in that particular MVoxel. Note that the ray sample-to-voxel mapping has to be calculated in the original NeRF models too; we simply group all such calculations and store the results in a table.

Feature Gathering (\mathcal{G}) and Feature Computation (\mathcal{F}). During gathering, we load the MVoxels from the DRAM to the SRAM sequentially. When an MVoxel is loaded, we look up the RIT to find all the ray samples that can be computed. A standard double buffer can be used to overlap MVoxel loading

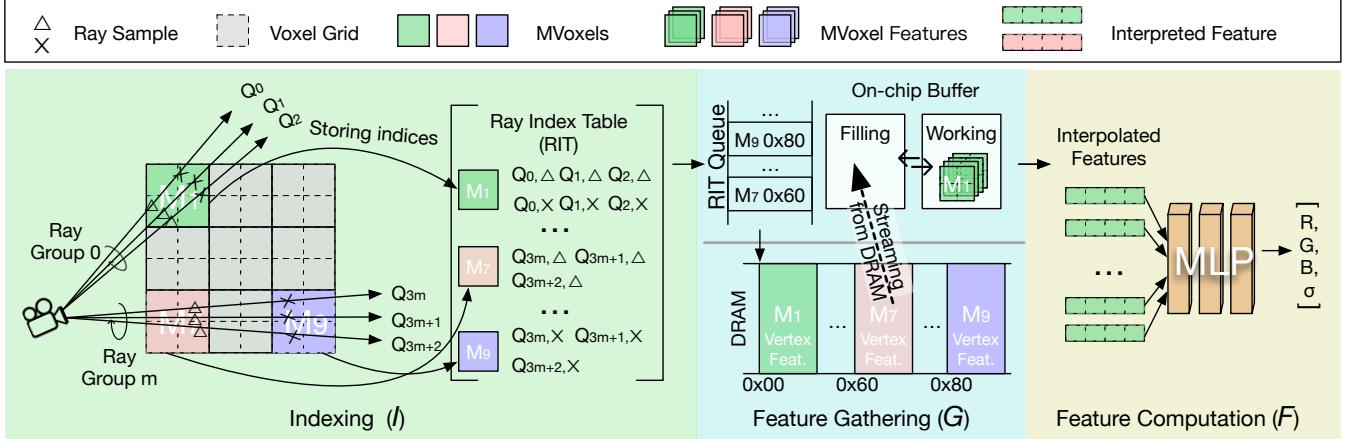


Fig. 12: Fully-streaming NeRF rendering algorithm. We first group all the voxels into MVoxels, which are continuously stored in the DRAM. The Ray Index Table (RIT) records, for each MVoxel, the IDs of ray samples whose features reside in the MVoxel. During feature gathering (\mathcal{G}), the entries in the RIT are sequentially accessed, essentially streaming the MVoxels from the DRAM. Each time an MVoxel is loaded on-chip, we process all the ray samples whose feature vectors are in that MVoxel. The Feature Computation stage is unchanged.

and the on-chip computation. Feature Computation remains unchanged compared to the baseline NeRF rendering.

Accommodating Hierarchical Data Encodings. Some NeRF algorithms, instead of storing voxel features directly, use hierarchical data structures such as hierarchical voxel grid [19], [59], [79], hashing [59], [64], and factorized tensor [15] to index the features.

To accommodate our fully-streaming data flow with these hierarchical data structures, we first partition 3D voxels at each level into MVoxel grids. During Feature Gathering, we group all rays into small ray groups (Fig. 12) and collect features level-by-level for a given ray group. Once we have traversed all levels, we can then compile all the vertex features necessary for this ray group. When the 3D voxel dimensions in the last several levels are too large, loading each MVoxel entirely would lead to low utilization of voxels, wasting DRAM bandwidth. In that case, we revert back to the original (non-streaming) data flow. This reversion happens in, for instance, Instant-NGP [59] from level 5 (out of 8 levels) onwards. As a result, about half of the DRAM traffics on Instant-NGP are non-streaming (which is faithfully captured in evaluation).

IV-B. Bank Conflict-Free Interleaving

With fully-streaming DRAM accesses, the inefficiency shifts to the on-chip SRAM, which experiences frequent bank conflicts (Fig. 6), which arise when different rays access the vertex features located at the same bank, leading to stalls in Feature Gathering (\mathcal{G}). Critically, unlike conventional DNNs where one can orchestrate data layout offline to avoid bank conflicts [43], [90], the SRAM access pattern of ray samples is known only at the run time, because the exact ray samples depend on the run-time camera pose information.

The reason behind bank conflicts in Feature Gathering has to do with the way feature vectors are laid out in SRAM; Fig. 13a illustrates this point. State-of-the-art NeRF accelerators [47],

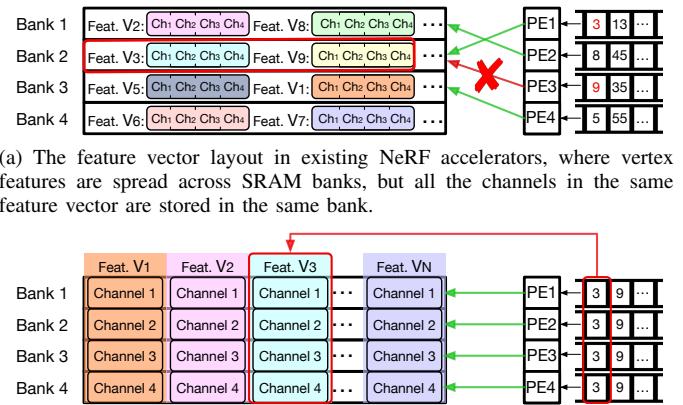


Fig. 13: A comparison between the original feature vector layout (Fig. 13a) and our data layout (Fig. 13b). The example has four banks and four concurrent PEs. In Fig. 13a, a bank conflict occurs when PE_1 and PE_3 (each collecting features for a different ray sample) access two different features from bank 2. Our data layout (Fig. 13b) eliminates bank conflicts by 1) spreading channels across banks and 2) having each PE collect a particular channel across different ray samples.

[52] store feature vectors in the SRAM using a *feature-major* order, where all the channels of a feature vector are stored in the same SRAM bank. Assume in this example we have four PEs, each responsible for collecting the feature vector for a particular ray sample. PE_1 and PE_3 are responsible for two ray samples, which require feature vectors 3 and 9, respectively. However, the two feature vectors happen to reside in the same bank, causing a bank conflict.

To address this issue, we propose a *channel-major* layout, as illustrated in Fig. 13b, where different channels of the same feature vector are spread across banks. For instance, bank

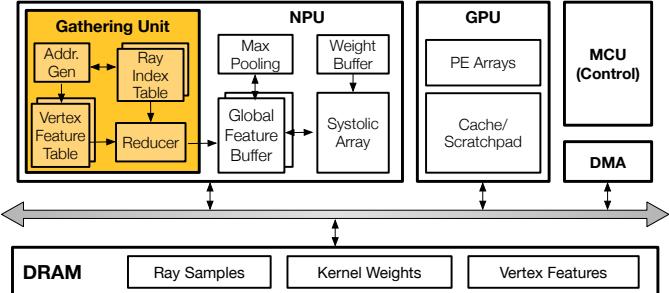


Fig. 14: The SoC architecture; the uncolored is the baseline architecture and we augment a standard systolic array-based NPU with a Gathering Unit (GU); colored. The GU executes Feather Gathering (\mathcal{G}) and the MAC array executes Feature Computation (\mathcal{F}). GPU executes the rest, e.g., Ray Indexing (\mathcal{I}) and the first three steps in SPARW for the target frames.

1 stores the first channel of all feature vectors within one MVoxel. In cases where the feature channel size exceeds the bank size, the storing sequence restarts from bank 1.

During feature gathering, instead of parallelizing ray samples across PEs, we parallelize channels across PEs. Each PE is responsible for gathering one channel of all the ray samples rather than gathering all channels of one individual ray sample. That is, each PE is dedicated to a specific bank. For instance, in Fig. 11b, the four PEs are collecting the four channels of the same feature vector 3 (required by one ray sample).

IV-C. Hardware Support

We extend a standard NPU architecture to support the two memory optimizations. Fig. 14 shows the SoC architecture, in which we augment the NPU with the new Gathering Unit (GU). The baseline SoC consists of mainly a GPU and an NPU. The GU in the NPU executes Feather Gathering (\mathcal{G}) and the MAC array in the NPU executes Feature Computation (\mathcal{F}). GPU executes the rest of the computations, e.g., Ray Indexing (\mathcal{I}) and the first three steps in SPARW for the target frames.

We describe the detailed GU architecture in Fig. 15. The GU has a dedicated buffer to store the RIT, and a dedicated Vertex Feature Table (VFT) to store MVoxels streamed from the DRAM as described in Sec. IV-A. The MVoxels are stored in the VFT using the data layout described in Sec. IV-B. Once RIT entries are loaded to RIT, the Address Generation logic takes one RIT entry and computes the addresses of the vertex features corresponding to the ray sample in that RIT entry.

Since VFT is free from bank conflicts, the VFT is designed as B individual SRAM arrays *without* the area-heavy crossbar. All the channels of the same feature vector are accessed simultaneously; thus, it takes one cycle to read one vertex feature. Since there are eight vertices of a voxel, it takes 8 cycles to read all the feature vectors of a given ray sample. Each bank is equipped with M ports to allow M feature vectors to be read simultaneously, which in turn allows M ray samples to be processed in parallel.

To calculate the feature vectors for feature computation, the GU uses $B \times M$ reducers, each of which performs the trilinear

interpolation operation required by NeRF models to calculate the final feature vector of a ray sample. The interpolated feature vectors are stored in the Feature FIFO before being written to the global feature buffer in the NPU.

SoC Integration. Our hardware extensions are limited to the NPU without changing the GPU hardware. Our design is agnostic to and, thus, integrates well with different GPU architectures — for two reasons. First, our hardware augmentation, i.e., the Gathering Unit, is limited to the NPU, whose communication with the GPU is dealt with by standard SoC-level interconnect (e.g., AXI) and thus accommodates different GPUs. Second, the interaction between the NPU and the GPU is minimal in our design: the GPU simply sends the Ray Index Table through the DMA to the NPU.

Broad Applicability. There is a long history of co-opting graphics hardware for non-graphics workloads, starting from using shader cores for general-purpose parallel computing [65] to using recent ray tracing hardware for neighbor search [62], [91] and database operations [61]. Our GU has the potential to be used beyond NeRF. In essence, the GU performs a parallel gather operation followed by a parallel reduction on the gathered data. Thus, the GU can find uses in domains such as neighbor search [58], [68] and sparse linear algebra [20], [22]. To accommodate different gather-reduction operations in other domains, the GU could be made programmable to allow 1) the RIT to store indices for to-be-gathered elements, 2) the address generation unit to map an element index to the element’s address, 3) the VFT to store the gathered data, and 4) the Reducer to implement different reduction operations required in different applications.

V. EXPERIMENTAL SETUP

Hardware Details. The NPU is a systolic-array-based DNN accelerator, which has a 24×24 MAC array, where each MAC unit mimics the design of that in the TPU [40]. The NPU also consists of a scalar unit, which can parallelize element-wise updates such as ReLU and Max-Pooling operation. The Global Feature Buffer is configured to be double-buffered with a size of 1.5 MB at a granularity of 32 KB. We reserve a dedicated 96 KB weight buffer to store MLP weights.

In our GU design, the RIT is double-buffered, each sized at 6 KB to store 128 entries. Each entry is 48 Bytes to accommodate eight vertex indices and their associated weights for linear interpolation (4-byte for one vertex index and 2-byte for one weight value). The Vertex Feature Buffer is 32 KB (organized as $B = 32$ banks each with $M = 2$ ports), which can store a MVoxel ($8 \times 8 \times 8$ points) with 32 channels. When the channel size of a vertex feature is greater than 32, we partition the vertex features into segments along the channel direction and load each segment sequentially.

Experimental Methodology. We directly time the GPU execution as well as the kernel launch on the mobile Volta GPU on Nvidia’s Xavier SoC [5]. The GPU power is directly measured using the built-in power sensing circuitry. We synthesize, place, and route the datapath of the systolic array and the gathering unit using an EDA flow consisting of Synopsys

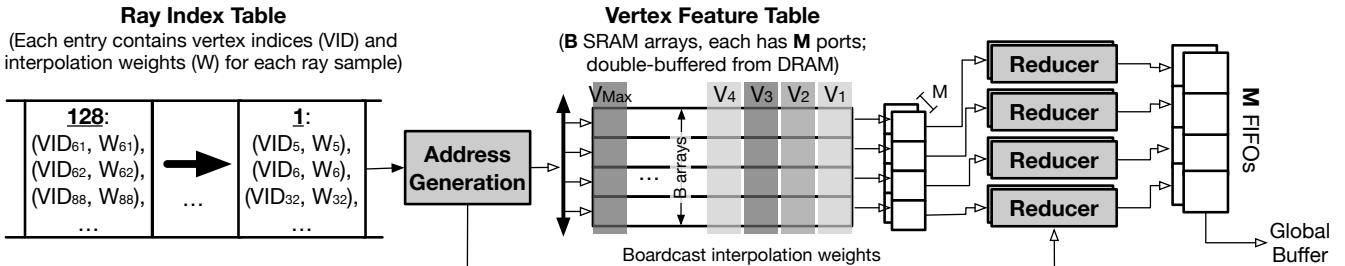


Fig. 15: Gathering unit. Each RIT entry stores the information of a ray sample. In particular, each entry has the eight vertex IDs (VIDs) of a ray sample’s voxel as well as the corresponding weights (W) for trilinear interpolation. The Address Generation logic uses the VIDs to access the corresponding vertex features from the VFT, which stores the features of a M-Voxel using the channel-major layout. The VFT has B individual SRAM arrays, each has M ports, supporting retrieving features for M ray samples simultaneously. $B \times M$ number of reducers perform trilinear interpolation required by the Feather Gathering (\mathcal{G}) stage to calculate the final feature vector for M ray samples. The weights are broadcast to the reducers. A FIFO holds the interpolated results before writing to the Global Feature Buffer for subsequent MLP computation.

and Cadence tools with the TSMC 16 nm FinFET technology and scale the results to 12 nm using the DeepScaleTool [72], [78] so that the results can be comparable with the mobile Volta GPU on Nvidia’s Xavier SoC in 12 nm node [5].

The SRAMs are generated using the Arm Artisan memory compiler. Power is estimated using Synopsys PrimeTimePX with annotated switching activities. The DRAM is modeled after Micron 16 Gb LPDDR3-1600 (4 channels) according to its datasheet [3]. The DRAM energy is obtained using Micron System Power Calculators [4]. On average, the energy ratio between a random DRAM access and a streaming DRAM access is about 3:1, and the energy ratio between a random DRAM access and an SRAM access is about 25:1.

We build a cycle-level simulator of the architecture with the latency of each component parameterized from measurements (for GPU) and post-synthesis results of the NPU design.

Area Overhead. CICERO introduces minimal area overhead with GU augmentation. The major overhead is from 44 K SRAM introduced from RIT buffer and VFT buffer. The additional area overhead (0.048 mm^2) compared to baseline NPU is less than 2.5%, in which the CICERO-specific portion is almost negligible compared to the entire SoC area, such as 350 mm^2 for Nvidia Xavier [6] and 108 mm^2 for Apple A15 [2]. We also removed the crossbar connections in VFT buffer due to our interleaving access pattern in feature gathering. In comparison, a heavily banked SRAM with a crossbar would introduce an additional 0.036 mm^2 of area overhead.

NeRF Algorithms. CICERO can accommodate arbitrary NeRF algorithms. To demonstrate the flexibility of CICERO, we evaluate three different NeRF algorithms: INSTANT-NGP [59], DIRECTVoxGO [79] and TENSORF [15], with varying model size-computation trade-offs (Fig. 2). The three networks also cover different feature representations: voxel grid for DirectVoxGO, hierarchical hashmaps for Instance-NGP, and factorized tensor for TensorRF. We evaluate algorithm quality on a standard metric, PSNR.

Datasets. We use Synthetic-NeRF [55], a synthetic dataset with eight different scenes. We also use two real-world

datasets, Unbounded-360 [10] (Bonsai trace) and Tanks and Temples [44] (Ignatius trace) to evaluate SPARW in real-world scenarios. Generating a mesh from images in the wild is a mature field (photogrammetry). We use Agisoft Metashape [1], a well-known photogrammetry tool, to generate meshes for the two real-world datasets.

Baseline. Our baseline is the SoC in Fig. 14 without the GU. It executes Ray Indexing (\mathcal{I}) and Feature Gathering (\mathcal{G}) on GPU and Feature Computation (\mathcal{F}) on NPU for all frames.

Variants. We evaluate three variants of CICERO to decouple the contribution proposed in our paper:

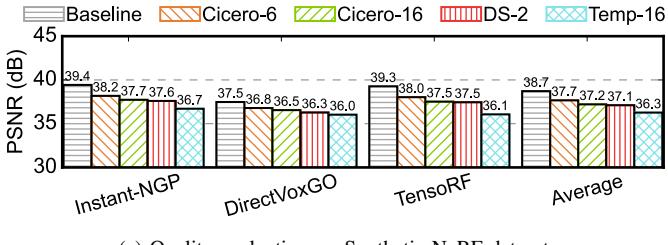
- SPARW: only performs sparse radiance warping with the same hardware configuration as the baseline.
- SPARW +FS: same as SPARW except it includes the fully-streaming NeRF rendering.
- CICERO: the full version of CICERO, which includes sparse radiance warping, fully-streaming NeRF rendering, and bank conflict-free interleaving (with GU support).

Application Scenarios. We evaluate two application scenarios that commonly exist in AR/VR applications:

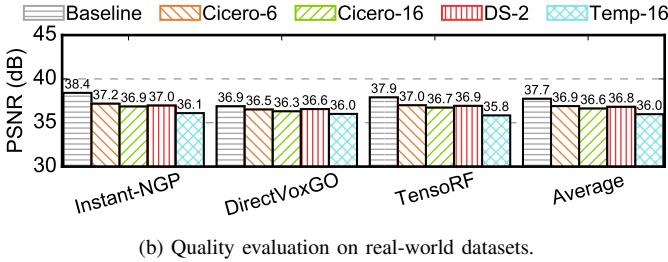
- **Local Rendering**: All the computations are executed on the standalone device with the hardware described above.
- **Remote Rendering**: Many VR devices, such as the Oculus Quest series, can be tethered wirelessly to a remote machine (e.g., a nearby workstation or even the cloud) to accelerate rendering, whereas the local device is used for display and lightweight processing. How to effectively leverage the remote rendering paradigm is an active area of research, and our evaluation aims to demonstrate a particular use of remote rendering by offloading the reference frame rendering in our SPARW algorithm to a remote 2080Ti GPU via a wireless connection. The wireless communication energy is modeled as 100 nJ/B with a speed of 10 MB/s [54].

VI. EVALUATION

We first demonstrate that CICERO achieves quality levels comparable to the baseline (Sec. VI-A). Meanwhile, we show



(a) Quality evaluation on Synthetic-NeRF dataset.



(b) Quality evaluation on real-world datasets.

Fig. 16: Image quality comparison. CICERO-6 and CICERO-16 use a warping window size (i.e., the number of target frames a reference frame is used for) of 6 and 16, respectively.

that even a pure software implementation of CICERO delivers significant speedups and energy reductions on a mobile Volta GPU (Sec. VI-B). The speedup and energy reduction increase when the baseline hardware incorporates a dedicated DNN accelerator (Sec. VI-C). We next perform a sensitivity study to understand CICERO’s performance and energy savings under different settings (Sec. VI-D). We show that CICERO achieves better speedups compared to prior NeRF accelerators (Sec. VI-E). Finally, we discuss the effectiveness of CICERO on real-world datasets (Sec. VI-F.) Only results in Sec. VI-F make use of the warping heuristics discussed in Sec. III-C.

VI-A. Rendering Quality

Fig. 16 shows the rendering quality of applying our SPARW algorithm to NeRF algorithms on both Synthetic-NeRF dataset (Fig. 16a) and real-world scenes (Fig. 16b). We use *warping window* to denote the number of target frames that reuse a single reference frame. We consider two warping window sizes, 6 and 16. In addition to the baseline algorithms, we also compare against two variants, DS-2 and TEMP-16. DS-2 first downsamples the frame by 2 for NeRF rendering and then up-samples it to the original resolution via bilinear interpolation. TEMP-16 is a method that uses a previously rendered frame as a reference frame with a warping window of 16 frames.

On both datasets, CICERO-6 retains an average PSNR drop within 1.0 dB compared to the original algorithms. Despite CICERO-16 dropping the average quality by 1.3 dB, it still has better quality compared against DS-2 and TEMP-16 on Synthetic-NeRF dataset. TEMP-16 is the worst because it warps from previous frames and accumulates errors. The quality of CICERO-6 is only slightly better than DS-2 on the real-world datasets, which use a low temporal resolution (1 FPS), for which the radiance approximation does not hold well. We will further discuss this in Sec. VI-F.

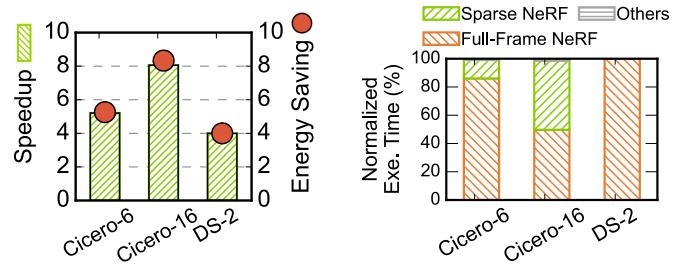


Fig. 17: Speedup and energy savings of CICERO against DS-2. Numbers are normalized by GPU baselines.

Fig. 18: GPU execution distribution of CICERO and DS-2. Full-Frame NeRF in CICERO is amortized across frames.

VI-B. Results on GPU

This section evaluates the performance of a pure software implementation of CICERO. Fig. 17 compares the FPS of CICERO-6 and CICERO-16 with DS-2 and the results are normalized to the FPS of a mobile Volta GPU. On average, CICERO-16 achieves 8.0× speedup and 7.9× energy saving compared to the original algorithms. In comparison, the DS-2 achieves only 4.0× speedup and 4.0× energy reduction. Even with a warping window of 6, CICERO-6 is faster than DS-2.

Fig. 18 shows the execution time distribution of both CICERO and DS-2. 86.1% of execution time in CICERO-6 is due to reference frame rendering. As the warping window size increases to 16, the percentage of Full-Frame NeRF decreases to 49.7%, while the execution time of sparse NeRF increases to 48.9%. Overall, the major latency bottleneck even with SPARW is still NeRF rendering, not the warping operations in SPARW. The “Others” category includes all the non-NeRF operations in SPARW, which is negligible.

VI-C. Performance and Energy

The speedup and energy reduction are even higher when the baseline SoC uses a dedicated NPU to execute the MLPs. To demonstrate that, we evaluate two different application scenarios: local rendering vs. remote rendering, both are common in VR. Unless stated otherwise, we use a warping window of 16 for our evaluation.

Local Rendering. Fig. 19a shows the speedup and normalized energy comparison in a local rendering scenario. All results are normalized with the baseline. On average, SPARW achieves 8.1× speedup and 8.1× energy saving on the same hardware configuration as the baseline. With the additional assistance from fully-streaming NeRF rendering, SPARW +FS achieves an additional 1.2× speedup and 1.6× energy saving under the same hardware configuration.

Two factors help SPARW +FS improve upon the baseline. First, SPARW algorithm reduces the amount of full-frame NeRF computation. Second, fully-streaming NeRF rendering reduces redundant DRAM accesses. With GU hardware support, CICERO further boosts the speedup and energy saving to 28.2× and 37.8×, respectively.

Remote Rendering. One factor that prevents CICERO from achieving higher speedup is resource contention: even though

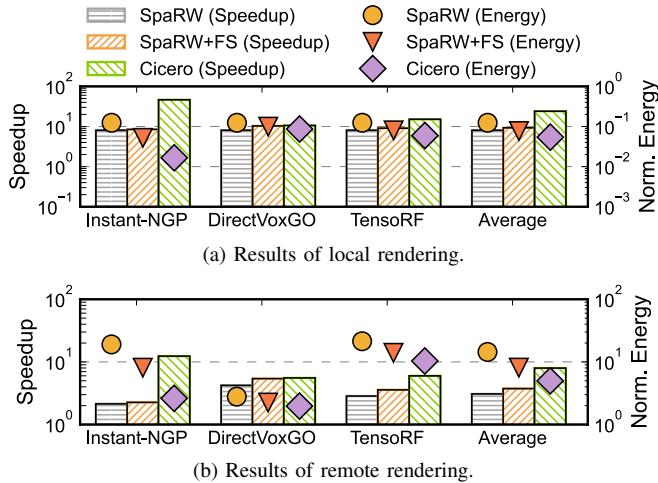


Fig. 19: End-to-end speedup and normalized energy of our variants over the baseline with a GPU and an NPU. We evaluate two application scenarios: local rendering and remote rendering. All values are normalized to the baseline.

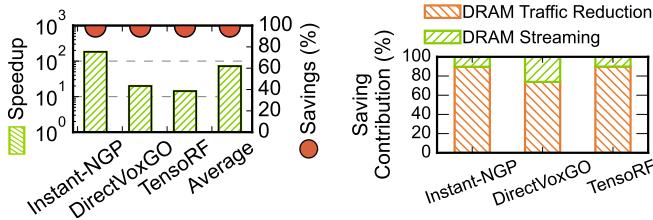


Fig. 20: Speedup and energy savings of feature gathering.

Fig. 21: Memory energy saving contribution.

algorithmically reference frame and target frame rendering can be overlapped, as described in Fig. 11, they compete for the same NPU and GPU resources. With additional resources on a remote machine, CICERO further boosts the performance.

Fig. 19b shows the speedup and energy comparison. In the baseline, the entire NeRF rendering executes on the remote GPU. In our system, we map the reference frame NeRF rendering to the remote GPU and render the target frames locally. In both cases, the remote GPU and the local device communicate the pixel data of the rendered frames.

SPARW achieves a $3.1\times$ speedup against the baseline, while SPARW +FS achieves a $3.8\times$ speedup by applying fully-streaming NeRF rendering. With GU hardware support, CICERO further improves the speedup to $8.0\times$. In all cases, data communication between the remote GPU and the local device is not a bottleneck: the communication latency is 0.02% of the average frame latency in CICERO.

Notably, the baseline in this scenario consumes lower energy than all three variants of CICERO. This is because when all the computations are offloaded to the remote GPU in the baseline, the main energy consumption of the local device is simply wireless communication. Transferring one frame consumes almost $5\times$ lower energy than rendering a frame in CICERO.

Feature Gathering (\mathcal{G}). Fig. 20 demonstrates the speedup and energy reduction brought by GU compared to the GPU

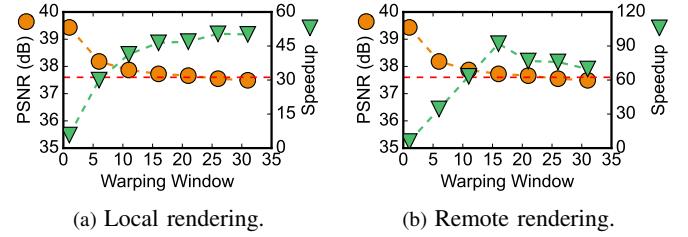


Fig. 22: Sensitivity of speedup and quality on Instant-NGP to warping window size under the two scenarios. The speedup is normalized to the baseline under the local rendering scenario. The red dash line shows the quality of DS-2.

execution. Overall, our GU achieves $72.2\times$ speedup while contributing to 99.9% of the energy reduction. This is attributed not only to our hardware acceleration of the Gather stage, but also to our data placement strategy that eliminates bank conflicts. For instance, Instant-NGP uses hash tables which causes severe SRAM bank conflicts. CICERO eliminates the irregular accesses entirely. Coupled with GU, we achieve a $182.4\times$ speedup on Instant-NGP.

Memory Saving Contribution. Not only does CICERO eliminate non-streaming DRAM access, it also reduces the overall DRAM traffic. Fig. 21 plots the percentage of DRAM energy reduction attributed to DRAM traffic reduction and converting random DRAM accesses to streaming accesses. On average, 84.5% of energy reduction is from DRAM traffic reduction. This shows that by grouping/loading a cluster of voxels together, CICERO effectively improves the reuse of each voxel, thus reducing the overall DRAM access. The rest of the energy reduction (15.5%) is from converting non-streaming DRAM access to streaming DRAM access. Although reducing bank conflicts does not reduce overall energy consumption, it does improve the performance of feature gathering (Fig. 20).

VI-D. Sensitivity Study

Warping Window. Fig. 22 illustrates how the overall speedup and quality vary under different warping window lengths (energy follows a similar trend as speed-up). For simplicity, we focus on Instant-NGP.

Fig. 22a shows the sensitivity to the warping window length in the local rendering scenario. Unsurprisingly, the quality of CICERO gradually decreases as the warping window increases. Nevertheless, CICERO still retains higher quality compared to DS-2 in Fig. 16, even at a warping window of 21. Similarly, the speedup of CICERO gradually plateaus and starts to decline as the warping window reaches 26. The decline is due to the growing number of empty pixels caused by disocclusions (Fig. 8). Consequently, the workload of sparse NeRF rendering, which aims to fill the disocclusions, gradually increases and eventually becomes dominant.

In comparison, Fig. 22b shows the sensitivity in the remote rendering scenario. One interesting observation is that the speedup of CICERO first increases linearly as the warping window increases from 1 to 16. This is because the on-device

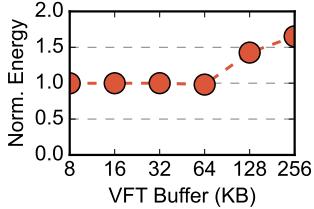


Fig. 23: Sensitivity of GU energy consumption to different VFT buffer sizes.

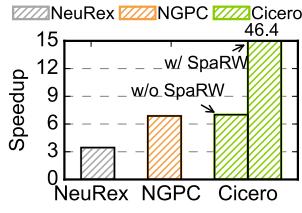


Fig. 24: Comparison of speedup between CICERO and prior NeRF accelerators.

execution can be hidden behind the execution time of full-time NeRF rendering offloaded to the workstation. However, when the warping window reaches 16, the on-device execution time can no longer be hidden and becomes the dominant factor.

GU. Fig. 23 shows the sensitivity of GU energy consumption to various VFT buffer sizes. To ensure each bank can hold one channel value of a MVoxel, we maintain the bank size at 1KB while sweeping the VFT buffer sizes (effectively bank sizes). To accommodate the same number of reducers in GU, we increase the number of ports per bank. As shown in Fig. 23, the overall GU energy consumption remains relatively constant as VFT buffer size increases from 8 KB to 64 KB. However, beyond 64 KB, the overall GU energy starts to rise.

VI-E. NGPC and NeuRex Comparison

We also compare against two prior NeRF accelerators: NEUREX [47] and NGPC [56]. Notably, both two accelerators are tailored to one particular NeRF algorithm, Instant-NGP, whereas CICERO generally applies to any NeRF algorithms.

Fig. 24 compares CICERO with the two accelerators on Instant-NGP. All values are normalized to the GPU baseline. We use the data reported in the NEUREX paper¹ and implement the NGPC architecture based on the paper’s description. For a fair comparison, we configure CICERO to use the same amount of PEs (24×24) as NGPC; NeuRex has a higher PE count (32×32) as reported in the paper. Our accelerator uses a 32 KB feature buffer, and the two baseline accelerators have larger on-chip feature buffers as described in their respective papers (i.e., 16 MB for NGPC and 64 KB for NeuRex).

Overall, CICERO without SPARW algorithm demonstrates a $2.0\times$ speedup over NEUREX. The speedup against NEUREX is attributed to hardware augmentation of GU in CICERO, which eliminates the SRAM bank conflicts in feature gathering. By contrast, NGPC design inherently avoids SRAM bank conflicts (because they use one bank for all the feature vectors in one Instant-NGP level). CICERO without SPARW achieves a similar speed. However, NGPC requires a 16 MB on-chip buffer dedicated to storing feature encodings, which is unrealistic for a mobile SoC. In contrast, with fully-streaming

¹The original NeuRex paper compares against Xavier NX (21 TOPS, 384 core) and our GPU baseline is Xavier (32 TOPS, 512 core). To convert the result to the Xavier baseline, we use the actual execution time of Instant-NGP on Xavier NX and NeuRex’s speedup numbers reported in the original paper to calculate the absolute execution time of NeuRex. Based on that, we calculate the speed-up of NeuRex over Xavier used in Fig. 24.

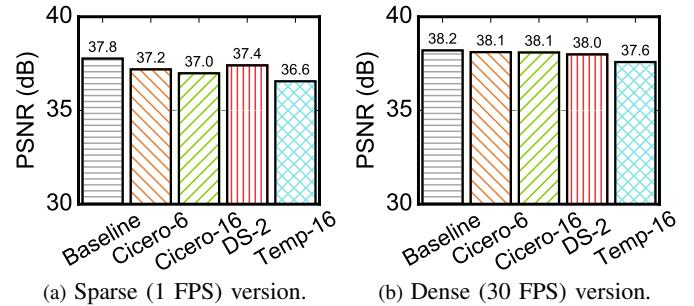


Fig. 25: PSNR comparison on the *Ignatius* scene in the Tanks and Temples dataset. The temporally dense sequence is more representative of real-time VR rendering, which we target.

rendering algorithm, our on-chip SRAM size is only 32 KB. With our SPARW algorithm, CICERO boosts the speedup to $16.4\times$ and $8.2\times$ against NEUREX and NGPC, respectively.

VI-F. Discussion on Real-World Scenes

We use the *Ignatius* scene in the Tanks and Temples dataset to discuss the effectiveness and limitations of SPARW on real-world scenes. Fig. 25a shows the results. Both CICERO-6 and CICERO-16 have lower quality compared to DS-2. This is because the temporal resolution of the scene is extremely low (1 FPS). Thus, consecutive frames have large differences in camera poses (i.e., θ in Fig. 8 is too large), so the radiance approximation does not hold well for non-diffuse surface.

We hasten to stress that the lower quality of SPARW here is *not* fundamental to the algorithm but an artifact of the low-FPS dataset. To evaluate CICERO in scenarios more representative of real-time VR rendering, we use the raw video sequence from the dataset captured at 30 FPS. The results are shown in Fig. 25b. In this more realistic scenario, CICERO-16 has little quality loss over the baseline and has a similar quality compared to DS-2 but is about $4\times$ faster.

While real-time VR rendering, which we target, usually has a high temporal resolution (> 30 FPS), in scenarios where a dataset has low temporal resolution, our warping heuristics in Sec. III-C can be used to mitigate the rendering quality loss. Fig. 26 shows the speed-up and PSNR of CICERO-16 across different warping thresholds on the challenging 1-FPS sequence of *Ignatius*. As ϕ reduces toward the left, the quality increases, since fewer pixels are warped and more pixels are NeRF-rendered, which also means the performance reduces. At a threshold of 4° , SPARW has a quality drop within 0.1 dB and a speed-up of $4.3\times$.

VII. RELATED WORK

NeRF Acceleration. NeRF rendering has drawn considerable attention in the last two years. Recent works have proposed several accelerators for NeRF algorithms [28], [47], [51], [52], [56], [71]. However, prior designs are tailored to individual NeRF algorithms—one accelerator for one algorithm. For example, Instant-3D [52] and NEUREX [47] accelerate the training and inference of Instant-NGP [59], respectively.

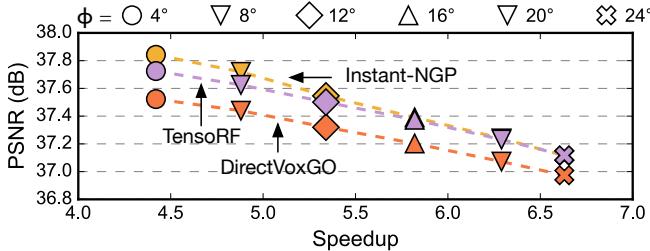


Fig. 26: Speed-up and PSNR of CICERO-16 under different warping thresholds ϕ on the 1-FPS sequence. We forgo warping if the angle subtended by a ray in the reference frame and the corresponding ray in the target frame is greater than ϕ .

NGPC [56] accelerates a range of neural graphic algorithms with similar hierarchical feature encodings. ICARUS [71] and RT-NeRF [51], on the other hand, design specialized architecture to speed up NeRF [55] and TensorRF [15], respectively. Meanwhile, Gen-NeRF [28] accelerates IBRNet [82] for novel view synthesis. In contrast, CICERO proposes solutions that are generally applicable to a range of existing NeRF algorithms.

Memory Optimizations in Rendering. Our idea of full-streaming DRAM accesses is inspired by ray reordering techniques in conventional ray tracing that increase memory access locality by grouping nearby rays [8], [11], [31], [70], [75].

Our approach has three main differences. First, we change the basic unit of reordering from rays to ray samples to accommodate the nature of NeRF. Second, existing techniques usually manipulate rays *dynamically*, since (secondary) rays are spawned at run time, which complicates the hardware design (e.g., dynamic identification of nearby rays, dynamic buffer management). In contrast, we exploit the nature of NeRF where all ray samples are known statically and reorder ray samples only once at the beginning. Third, ray reordering in ray tracing usually can only afford local reordering because rays are dynamically spawned, whereas we reorder ray samples *globally* to guarantee fully-streaming DRAM accesses.

Real-Time VR Rendering. To achieve real-time VR rendering, prior works have leaned on image-based rendering or remote rendering [12], [34], [36], [48], [57], [81]. Some systems directly stream rendered videos to clients, but they often suffer from bandwidth limits and require efficient video encoding [63]. Other works transmit one frame that can be reused to render multiple viewpoints, but often require complex encoding methods to store texture and geometric information to address disocclusions [36], [57]. By leveraging the computation characteristics of NeRF, CICERO proposes a straightforward yet effective approach to resolve disocclusions, achieving photo-realistic visual quality.

Warping in Image-based Rendering. SPARW belongs to a class of imaging warping techniques initially proposed for image-based rendering [17], [18], of which NeRF is a recent development. We refer interested readers to Chaurasia et al. [14] and Szeliski [80] for brief surveys. Recent studies generalize warping to use temporal correlations across frames for reducing computation in real-time vision [13], [26], [37],

[77], [86], [92], increase super-resolution [85] and depth estimation quality [24], and accelerate rendering [88], [89].

We are the first to apply temporal warping in NeRF (as a way of view synthesis). Prior temporal warping techniques serialize the processing of the reference frame and the target frame, as the latter depends on the former. SPARW’s main novelty is the observation that the reference frames merely provide useful information for target frames later. We break the dependencies between reference and target frames, which allows us to overlap reference and target frame rendering.

VIII. LIMITATIONS AND FUTURE WORK

Warping. The warping heuristics discussed in Sec. III-C is nothing more than an engineering hack to accommodate non-diffuse surfaces. Ideally, we want a “transfer function” that transforms the radiance of a ray to the radiance of another ray. The SPARW essentially uses an identity function (conditioned upon the warping threshold) as a special-case transfer function.

The exact transfer function depends on the material property of the surface. An interesting future direction is to learn the material properties [9], [29], e.g., Bidirectional Reflectance Distribution Function (BRDF) and Bidirectional Subsurface Scattering Function Function (BSSDF) [69], potentially jointly with the NeRF model. The material property could then be used to better transfer pixels in the reference frame, similar to classic irradiance caching [45], [83] and radiance caching [46], [60], [73], instead of simply reusing the pixel values.

SPARW also exposes potential aliasing issues across the boundary between warped pixels and NeRF-rendered pixels. One simple solution would be blended across the regions using techniques from classic foveated rendering [33], [67].

3DGS. Similar to MLP-based models, 3DGS follows the pixel-centric approach during image rendering. Thus, our ideas in principle can also be applied to 3DGS. Furthermore, 3DGS avoids MLP computations and uses point cloud-based representations, which might lead to higher memory consumption [25] and irregular memory access [23] during Feature Gathering, warranting future investigations.

IX. CONCLUSION

We reduce over 95% of the MLP computation in NeRF by warping radiances computed in previous frames with less than 1 dB PSNR loss. We also show that transforming NeRF inference from a ray-centric order to a scene-centric order leads to a completely sequential DRAM access. Finally, we show that laying feature vectors in a channel-major, rather than a feature-major, order eliminates on-chip SRAM bank conflicts. Collectively, we demonstrate over an order of magnitude speed-up and energy saving over a mobile Volta GPU.

X. ACKNOWLEDGEMENTS

We thank anonymous reviewers from ISCA for their comments and Weikai Lin from University of Rochester for invaluable discussion and technical support. Jingwen Leng is the corresponding author. The work is partially supported by the National Key R&D Program of China under Grant 2022YFB4501400, NSFC grant (62072297 and 62222210).

REFERENCES

- [1] “Agisoft metashape,” <https://www.agisoft.com/>.
- [2] “Apple A15 Die Shot and Annotation - IP Block Area Analysis.” [Online]. Available: <https://www.semianalysis.com/p/apple-a15-die-shot-and-annotation>
- [3] “Micron 178-Ball, Single-Channel Mobile LPDDR3 SDRAM Features.” [Online]. Available: https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/mobile-dram/low-power-dram/lpddr3/178b_8-16gb_2c0f_mobile_lpddr3.pdf
- [4] “Micron System Power Calculators.” [Online]. Available: <https://www.micron.com/support/tools-and-utilities/power-calc>
- [5] “Nvidia reveals xavier soc details.” [Online]. Available: <https://www.forbes.com/sites/moorinsights/2018/08/24/nvidia-reveals-xavier-soc-details/amp/>
- [6] “NVIDIA’s Xavier System-on-Chip, HotChips 30.” [Online]. Available: <https://fuse.wikichip.org/news/1618/hot-chips-30-nvidia-xavier-soc/>
- [7] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, “Garnet: A detailed on-chip network model inside a full-system simulator,” in *2009 IEEE international symposium on performance analysis of systems and software*. IEEE, 2009, pp. 33–42.
- [8] T. Aila and T. Karras, “Architecture considerations for tracing incoherent rays,” in *Proceedings of the Conference on High Performance Graphics*, 2010, pp. 113–122.
- [9] D. Azinovic, T.-M. Li, A. Kaplanyan, and M. Nießner, “Inverse path tracing for joint material and lighting estimation,” in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 2447–2456.
- [10] J. T. Barron, B. Mildenhall, D. Verbin, P. P. Srinivasan, and P. Hedman, “Mip-nerf 360: Unbounded anti-aliased neural radiance fields,” *CVPR*, 2022.
- [11] J. Bikker, “Improving data locality for efficient in-core path tracing,” in *Computer Graphics Forum*, vol. 31, no. 6. Wiley Online Library, 2012, pp. 1936–1947.
- [12] K. Boos, D. Chu, and E. Cuervo, “Flashback: Immersive virtual reality on mobile devices via rendering memoization,” in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, 2016, pp. 291–304.
- [13] M. Buckler, P. Bedoukian, S. Jayasuriya, and A. Sampson, “Eva²: Exploiting temporal redundancy in live computer vision,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 533–546.
- [14] G. Chaurasia, A. Nieuwoudt, A.-E. Ichim, R. Szeliski, and A. Sorkine-Hornung, “Passthrough+ real-time stereoscopic view synthesis for mobile mixed reality,” *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 3, no. 1, pp. 1–17, 2020.
- [15] A. Chen, Z. Xu, A. Geiger, J. Yu, and H. Su, “Tensorf: Tensorial radiance fields,” in *European Conference on Computer Vision*. Springer, 2022, pp. 333–350.
- [16] G. Chen and W. Wang, “A survey on 3d gaussian splatting,” *arXiv preprint arXiv:2401.03890*, 2024.
- [17] S. E. Chen, “Quicktime vr: An image-based approach to virtual environment navigation,” in *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, 1995, pp. 29–38.
- [18] S. E. Chen and L. Williams, “View interpolation for image synthesis,” in *Seminal Graphics Papers: Pushing the Boundaries, Volume 2*, 2023, pp. 423–432.
- [19] Z. Chen, T. Funkhouser, P. Hedman, and A. Tagliasacchi, “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*, 2023, pp. 16569–16578.
- [20] T. A. Davis, “Algorithm 1000: Suitesparse: Graphblas: Graph algorithms in the language of sparse linear algebra,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 4, pp. 1–25, 2019.
- [21] Y. Deng, Y. Ni, Z. Li, S. Mu, and W. Zhang, “Toward real-time ray tracing: A survey on hardware acceleration and microarchitecture techniques,” *ACM Computing Surveys (CSUR)*, vol. 50, no. 4, pp. 1–41, 2017.
- [22] I. S. Duff, M. A. Heroux, and R. Pozo, “An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 28, no. 2, pp. 239–267, 2002.
- [23] Y. Feng, G. Hammonds, Y. Gan, and Y. Zhu, “Crescent: taming memory irregularities for accelerating deep point cloud analytics,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 962–977.
- [24] Y. Feng, P. Hansen, P. N. Whatmough, G. Lu, and Y. Zhu, “Fast and accurate: Video enhancement using sparse depth,” in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, 2023, pp. 4492–4500.
- [25] Y. Feng, B. Tian, T. Xu, P. Whatmough, and Y. Zhu, “Mesorasi: Architecture support for point cloud analytics via delayed-aggregation,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1037–1050.
- [26] Y. Feng, P. Whatmough, and Y. Zhu, “Asv: Accelerated stereo vision system,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 643–656.
- [27] D. A. Forsyth and J. Ponce, *Computer vision: a modern approach*. prentice hall professional technical reference, 2002.
- [28] Y. Fu, Z. Ye, J. Yuan, S. Zhang, S. Li, H. You, and Y. Lin, “Gen-nerf: Efficient and generalizable neural radiance fields via algorithm-hardware co-design,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–12.
- [29] I. Gkioulekas, S. Zhao, K. Bala, T. Zickler, and A. Levin, “Inverse volume rendering with material dictionaries,” *ACM Transactions on Graphics (TOG)*, vol. 32, no. 6, pp. 1–13, 2013.
- [30] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen, “The lumigraph,” in *Seminal Graphics Papers: Pushing the Boundaries, Volume 2*, 2023, pp. 453–464.
- [31] C. P. Gribble and K. Ramani, “Coherent ray tracing via stream filtering,” in *2008 IEEE Symposium on Interactive Ray Tracing*. IEEE, 2008, pp. 59–66.
- [32] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu, “Kilo-noc: a heterogeneous network-on-chip architecture for scalability and service guarantees,” in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2011, pp. 401–412.
- [33] B. Guenter, M. Finch, S. Drucker, D. Tan, and J. Snyder, “Foveated 3d graphics,” *ACM transactions on Graphics (tOG)*, vol. 31, no. 6, pp. 1–10, 2012.
- [34] D. Han, J. Ryu, S. Kim, S. Kim, J. Park, and H.-J. Yoo, “Metavrain: A mobile neural 3-d rendering processor with bundle-frame-familiarity-based nerf acceleration and hybrid dnn computing,” *IEEE Journal of Solid-State Circuits*, 2023.
- [35] P. Hedman, P. P. Srinivasan, B. Mildenhall, J. T. Barron, and P. Debevec, “Baking neural radiance fields for real-time view synthesis,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 5875–5884.
- [36] J. Hladky, M. Stengel, N. Vining, B. Kerbl, and H.-P. Seidel, “Quad-stream: A quad-based scene streaming architecture for novel viewpoint reconstruction,” *ACM Transactions on Graphics (SIGGRAPH Asia’22)*, p. C32, 2022.
- [37] X. Hou, J. Liu, X. Tang, C. Li, J. Chen, L. Liang, K.-T. Cheng, and M. Guo, “Architecting efficient multi-modal aiot systems,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.
- [38] X. Hou and S. Dey, “Motion prediction and pre-rendering at the edge to enable ultra-low latency mobile 6dof experiences,” *IEEE Open Journal of the Communications Society*, vol. 1, pp. 1674–1690, 2020.
- [39] T. Hu, S. Liu, Y. Chen, T. Shen, and J. Jia, “Efficientnerf: efficient neural radiance fields,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 12902–12911.
- [40] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, “In-datacenter performance analysis of a tensor processing unit,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.
- [41] A. Kaufman, D. Cohen, and R. Yagel, “Volume graphics,” *Computer*, vol. 26, no. 7, pp. 51–64, 1993.
- [42] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis, “3d gaussian splatting for real-time radiance field rendering,” *ACM Transactions on Graphics*, vol. 42, no. 4, pp. 1–14, 2023.
- [43] D. B. Kirk and W. H. Wen-Mei, *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.
- [44] A. Knapitsch, J. Park, Q.-Y. Zhou, and V. Koltun, “Tanks and temples: Benchmarking large-scale scene reconstruction,” *ACM Transactions on Graphics*, vol. 36, no. 4, 2017.

- [45] J. Krivánek and P. Gautron, *Practical global illumination with irradiance caching*. Springer Nature, 2022.
- [46] J. Krivánek, P. Gautron, S. Pattanaik, and K. Bouatouch, “Radiance caching for efficient global illumination computation,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, no. 5, pp. 550–561, 2005.
- [47] J. Lee, K. Choi, J. Lee, S. Lee, J. Whangbo, and J. Sim, “Neurex: A case for neural rendering acceleration,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.
- [48] Y. Leng, C.-C. Chen, Q. Sun, J. Huang, and Y. Zhu, “Energy-efficient video processing for virtual reality,” in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 91–103.
- [49] M. Levoy, “Display of surfaces from volume data,” *IEEE Computer graphics and Applications*, vol. 8, no. 3, pp. 29–37, 1988.
- [50] M. Levoy and P. Hanrahan, “Light field rendering,” in *Seminal Graphics Papers: Pushing the Boundaries, Volume 2*, 2023, pp. 441–452.
- [51] C. Li, S. Li, Y. Zhao, W. Zhu, and Y. Lin, “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*, 2022, pp. 1–9.
- [52] S. Li, C. Li, W. Zhu, B. Yu, Y. Zhao, C. Wan, H. You, H. Shi, and Y. Lin, “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*, 2023, pp. 1–13.
- [53] R. Liang, H. Sun, and N. Vijaykumar, “Coordx: Accelerating implicit neural representation with a split mlp architecture,” *arXiv preprint arXiv:2201.12425*, 2022.
- [54] C. Liu, S. Chen, T.-H. Tsai, B. De Salvo, and J. Gomez, “Augmented reality-the next frontier of image sensors and compute systems,” in *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65. IEEE, 2022, pp. 426–428.
- [55] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, “Nerf: Representing scenes as neural radiance fields for view synthesis,” *Communications of the ACM*, vol. 65, no. 1, pp. 99–106, 2021.
- [56] M. H. Mubarik, R. Kanungo, T. Zirr, and R. Kumar, “Hardware acceleration of neural graphics,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–12.
- [57] J. H. Mueller, P. Voglreiter, M. Dokter, T. Neff, M. Makar, M. Steinberger, and D. Schmalstieg, “Shading atlas streaming,” *ACM Transactions on Graphics (TOG)*, vol. 37, no. 6, pp. 1–16, 2018.
- [58] M. Muja and D. G. Lowe, “Flann - fast library for approximate nearest neighbors,” <https://github.com/flann-lib/flann>, commit: 1d04523268c388dabf1c0865d69e1b638c8c7d9d.
- [59] T. Müller, A. Evans, C. Schied, and A. Keller, “Instant neural graphics primitives with a multiresolution hash encoding,” *ACM Transactions on Graphics (ToG)*, vol. 41, no. 4, pp. 1–15, 2022.
- [60] T. Müller, F. Rousselie, J. Novák, and A. Keller, “Real-time neural radiance caching for path tracing,” *ACM Transactions on Graphics (TOG)*, vol. 40, no. 4, pp. 1–16, 2021.
- [61] V. Nagarajan and M. Kulkarni, “Rt-dbscan: Accelerating dbscan using ray tracing hardware,” in *Proceedings of the 38th International Parallel and Distributed Processing Symposium*, 2023.
- [62] V. Nagarajan, D. Mandarapu, and M. Kulkarni, “Rt-knns unbound: Using rt cores to accelerate unrestricted neighbor search,” in *Proceedings of the 37th International Conference on Supercomputing*, 2023, pp. 289–300.
- [63] Y. Noimark and D. Cohen-Or, “Streaming scenes to mpeg-4 video-enabled devices,” *IEEE Computer Graphics and Applications*, vol. 23, no. 1, pp. 58–64, 2003.
- [64] J. Olszewski, “Hashcc: Lightweight method to improve the quality of the camera-less nerf scene generation,” *arXiv preprint arXiv:2305.04296*, 2023.
- [65] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, “A survey of general-purpose computation on graphics hardware,” in *Computer graphics forum*, vol. 26, no. 1. Wiley Online Library, 2007, pp. 80–113.
- [66] J. Pantaleoni and D. Luebke, “Hlbvh: Hierarchical lbbvh construction for real-time ray tracing of dynamic geometry,” in *Proceedings of the Conference on High Performance Graphics*, 2010, pp. 87–95.
- [67] A. Patney, M. Salvi, J. Kim, A. Kaplanyan, C. Wyman, N. Benty, D. Luebke, and A. Lefohn, “Towards foveated rendering for gaze-tracked virtual reality,” *ACM Transactions on Graphics (TOG)*, vol. 35, no. 6, pp. 1–12, 2016.
- [68] PCL, “Spatial partitioning and search operations with octrees,” <https://pcl.readthedocs.io/projects/tutorials/en/latest/octree.html>.
- [69] M. Pharr, W. Jakob, and G. Humphreys, *Physically based rendering: From theory to implementation*. MIT Press, 2023.
- [70] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan, “Rendering complex scenes with memory-coherent ray tracing,” in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, 1997, pp. 101–108.
- [71] C. Rao, H. Yu, H. Wan, J. Zhou, Y. Zheng, M. Wu, Y. Ma, A. Chen, B. Yuan, P. Zhou *et al.*, “Icarus: A specialized architecture for neural radiance fields rendering,” *ACM Transactions on Graphics (TOG)*, vol. 41, no. 6, pp. 1–14, 2022.
- [72] S. Sarangi and B. Baas, “Deepscaletool: A tool for the accurate estimation of technology scaling in the deep-submicron era,” in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2021, pp. 1–5.
- [73] D. Seyb, P.-P. Sloan, A. Silvennoinen, M. Iwanicki, and W. Jarosz, “The design and evolution of the überbake light baking system,” *ACM Transactions on Graphics (TOG)*, vol. 39, no. 4, pp. 150–1, 2020.
- [74] P. Shirley, M. Ashikhmin, and S. Marschner, *Fundamentals of computer graphics*. AK Peters/CRC Press, 2009.
- [75] K. Shkurko, T. Grant, D. Kopta, I. Mallett, C. Yuksel, and E. Brunvand, “Dual streaming for hardware-accelerated ray tracing,” in *Proceedings of High Performance Graphics*, 2017, pp. 1–11.
- [76] H.-Y. Shum, S.-C. Chan, and S. B. Kang, *Image-based rendering*. Springer Science & Business Media, 2008.
- [77] Z. Song, F. Wu, X. Liu, J. Ke, N. Jing, and X. Liang, “Vr-dann: Real-time video recognition via decoder-assisted neural network acceleration,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 698–710.
- [78] A. Stillmaker and B. Baas, “Scaling equations for the accurate prediction of cmos device performance from 180 nm to 7 nm,” *Integration*, vol. 58, pp. 74–81, 2017.
- [79] C. Sun, M. Sun, and H.-T. Chen, “Direct voxel grid optimization: Superfast convergence for radiance fields reconstruction,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 5459–5469.
- [80] R. Szeliski, “Chapter 14 image-based rendering,” in *Computer vision: algorithms and applications*. Springer Nature, 2022.
- [81] E. Teler and D. Lischinski, “Streaming of complex 3d scenes for remote walkthroughs,” in *Computer Graphics Forum*, vol. 20, no. 3. Wiley Online Library, 2001, pp. 17–25.
- [82] Q. Wang, Z. Wang, K. Genova, P. P. Srinivasan, H. Zhou, J. T. Barron, R. Martin-Brualla, N. Snavely, and T. Funkhouser, “Ibrnet: Learning multi-view image-based rendering,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 4690–4699.
- [83] G. J. Ward, F. M. Rubinstein, and R. D. Clear, “A ray tracing solution for diffuse interreflection,” in *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, 1988, pp. 85–92.
- [84] T. Wu, Y.-J. Yuan, L.-X. Zhang, J. Yang, Y.-P. Cao, L.-Q. Yan, and L. Gao, “Recent advances in 3d gaussian splatting,” *arXiv preprint arXiv:2403.11134*, 2024.
- [85] L. Xiao, S. Nouri, M. Chapman, A. Fix, D. Lanman, and A. Kaplanyan, “Neural supersampling for real-time rendering,” *ACM Transactions on Graphics (TOG)*, vol. 39, no. 4, pp. 142–1, 2020.
- [86] Z. Ying, S. Zhao, H. Zhang, C. S. Mishra, S. Bhuyan, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das, “Exploiting frame similarity for efficient inference on edge devices,” in *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2022, pp. 1073–1084.
- [87] A. Yu, R. Li, M. Tancik, H. Li, R. Ng, and A. Kanazawa, “Plenoctrees for real-time rendering of neural radiance fields,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 5752–5761.
- [88] S. Zhao, H. Zhang, S. Bhuyan, C. S. Mishra, Z. Ying, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das, “Déjà view: Spatio-temporal compute reuse for ‘energy-efficient 360 vr video streaming,’” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 241–253.
- [89] S. Zhao, H. Zhang, C. S. Mishra, S. Bhuyan, Z. Ying, M. T. Kandemir, A. Sivasubramaniam, and C. Das, “Holoar: On-the-fly optimization of 3d holographic processing for augmented reality,” in *MICRO-54:*

- 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 494–506.
- [90] Y. Zhou, M. Yang, C. Guo, J. Leng, Y. Liang, Q. Chen, M. Guo, and Y. Zhu, “Characterizing and demystifying the implicit convolution algorithm on commercial matrix-multiplication accelerators,” in *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2021, pp. 214–225.
 - [91] Y. Zhu, “Rtnn: accelerating neighbor search using hardware ray tracing,” in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022, pp. 76–89.
 - [92] Y. Zhu, A. Samajdar, M. Mattina, and P. Whatmough, “Euphrates: Algorithm-soc co-design for low-power mobile continuous vision,” *arXiv preprint arXiv:1803.11232*, 2018.