# Potamoi: Accelerating Neural Rendering via a Unified Streaming Architecture

YU FENG, John Hopcropt Center, Shanghai Jiao Tong University, Shanghai, China
WEIKAI LIN, Department of Computer Science, University of Rochester, Rochester, United States
ZIHAN LIU, Shanghai Jiao Tong University, Shanghai, China
JINGWEN LENG, Department of Computer Science, Shanghai Jiao Tong University, Shanghai, China and Shanghai Qi Zhi Institute, Shanghai, China
MINYI GUO, Computer Science, Shanghai Jiao Tong University, Shanghai, China and Shanghai Qi Zhi Institute, Shanghai, China
HAN ZHAO, Shanghai Jiao Tong University, Shanghai, China
XIAOFENG HOU, Shanghai Jiao Tong University, Shanghai, China
JIERU ZHAO, Shanghai Jiao Tong University, Shanghai, China
YUHAO ZHU, University of Rochester, Rochester, United States

Neural Radiance Field (NeRF) has emerged as a promising alternative for photorealistic rendering. Despite recent algorithmic advancements, achieving real-time performance on today's resource-constrained devices remains challenging. In this article, we identify the primary bottlenecks in current NeRF algorithms and introduce a unified algorithm-architecture co-design, Potamoi, designed to accommodate various NeRF algorithms. Specifically, we introduce a runtime system featuring a plug-and-play algorithm, SpaRW, which significantly reduces the per-frame computational workload and alleviates compute inefficiencies. Furthermore, our unified streaming pipeline coupled with customized hardware support effectively tames both SRAM and DRAM inefficiencies by minimizing repetitive DRAM access and completely eliminating SRAM bank conflicts. When evaluated against a baseline utilizing a dedicated DNN accelerator, our framework demonstrates a speedup and energy reduction of 53.1× and 67.7×, respectively, all while maintaining high visual quality with less than a 1.0 dB reduction in peak signal-to-noise ratio.

CCS Concepts: • **Computer systems organization** → **Neural networks**; *Data flow architectures*; **Real-time system architecture**; • **Computing methodologies** → *Rendering;*

Y. Feng and W. Lin contributed equally to this article.

The work was partially supported by the National Key R&D Program of China under grant 2022YFB4501400, and NSFC grants 62072297 and 62222210.

Authors' Contact Information: Yu Feng, John Hopcropt Center, Shanghai Jiao Tong University, Shanghai, China; e-mail: y-feng@sjtu.edu.cn; Weikai Lin, Department of Computer Science, University of Rochester, Rochester, New York, United States; e-mail: wlin33@ur.rochester.edu; Zihan Liu, Shanghai Jiao Tong University, Shanghai, China; e-mail: altair.liu@sjtu.edu.cn; Jingwen Leng (Corresponding author), Department of Computer Science, Shanghai Jiao Tong University, Shanghai, Shanghai, China and Shanghai Qi Zhi Institute, Shanghai, Shanghai, China; e-mail: leng-jw@cs.sjtu.edu.cn; Minyi Guo, Computer Science, Shanghai Jiao Tong University, Shanghai, Shanghai, China and Shanghai Qi Zhi Institute, Shanghai, Shanghai, China; e-mail: guo-my@cs.sjtu.edu.cn; Han Zhao, Shanghai Jiao Tong University, Shanghai, China; e-mail: zhao-han@cs.sjtu.edu.cn; Xiaofeng Hou, Shanghai Jiao Tong University, Shanghai, China; e-mail: hou-xf@cs.sjtu.edu.cn; Jieru Zhao, Shanghai Jiao Tong University, Shanghai, China; e-mail: zhao-jieru@sjtu.edu.cn; Yuhao Zhu (Corresponding author), University of Rochester, Rochester, New York, United States; e-mail: yzhu@rochester.edu.



ACM 1544-3973/2024/11-ART80
https://doi.org/10.1145/3689340

Additional Key Words and Phrases: Mobile architecture, neural rendering acceleration, virtual reality

**ACM Reference Format:**
Yu Feng, Weikai Lin, Zihan Liu, Jingwen Leng, Minyi Guo, Han Zhao, Xiaofeng Hou, Jieru Zhao, and Yuhao Zhu. 2024. Potamoi: Accelerating Neural Rendering via a Unified Streaming Architecture. *ACM Trans. Arch. Code Optim.* 21, 4, Article 80 (November 2024), 25 pages. https://doi.org/10.1145/3689340

## 1 Introduction

The **Neural Radiance Field (NeRF)** model [48] revitalizes classic image-based rendering techniques [65, 69] using modern deep learning approaches, offering a compelling alternative to conventional photorealistic rendering methods like path tracing [15, 56, 57]. However, the original NeRF algorithm is notoriously slow [48], which has spurred numerous efforts to reduce the computational demands of NeRF rendering [11, 30, 51, 55, 68, 75]. Despite these efforts, achieving real-time performance on mobile devices remains unsolved. For example, on a mobile Volta GPU in NVIDIA's Xavier SoC, models like DirectVoxGO [68] can only achieve 0.8 **Frames Per Second (FPS)**. Even the latest developments, such as **3D Gaussian Splatting (3DGS)** [33], can barely reach 5 FPS for rendering an $800 \times 800$ frame on Xavier.

Although accelerating NeRF is critical, it is crucial not to over-specialize hardware for any specific model due to the fast evolution in algorithm design. Since its introduction, NeRF models have undergone significant algorithmic transformations, evolving from the original **Multilayer Perceptron (MLP)**-based designs [48] to those incorporating structured representations [51, 68, 75], and most recently, to implementations utilizing 3DGS with unstructured representations [12, 33, 38, 74]. Given the ongoing NeRF development, it is conceivable that NeRF models will continue to evolve. Therefore, system and architectural support should address the fundamental bottlenecks inherent to NeRF models as a whole rather than optimizing for artifact features specific to individual models.

This article introduces Potamoi, an algorithm-architecture co-designed approach to address fundamental inefficiencies in both computation and memory accesses inherent to a diverse set of NeRF models.

***Bottleneck Analysis.*** We first describe a general pipeline that unifies the computation flows across various NeRF models (Section 2). We then analyze the performance bottlenecks using this general pipeline. This pipeline decomposes NeRF models into three key stages: Indexing, Feature Gathering, and Feature Computation. By analyzing the workloads across different NeRF models, we identify their inherent bottlenecks in both algorithm design and underlying hardware (Section 3). Algorithmically, a NeRF model processes millions of rays, each containing hundreds of samples. Each of these samples requires executing a **General Matrix Multiply (GEMM)**-based inference, collectively imposing a significant computational load [48]. Architecturally, the GEMMs associated with ray samples require fetching pre-trained features, leading to irregular memory accesses. This, in turn, results in irregular DRAM accesses and SRAM bank conflicts.

***Algorithmic Support.*** Potamoi presents a plug-and-play extension to existing NeRF algorithms and significantly reduces the computational workload of NeRF models (Section 4). Our method, SpaRW (*sparse radiance warping*), reduces up to 88% of radiance computations by reusing ray radiances in previously computed reference frames. SpaRW operates based on the observation of *radiance proximity*: the radiance of nearby rays originating from the same physical point is approximately equivalent. It is worth noting that SpaRW is not a replacement for current NeRF models; rather, it is a plug-and-play extension that can be integrated with all NeRF models.

***Runtime Support.*** Choosing reference frames is critical to unleash the full power of SpaRW. We observe that reference frames do not have to be a frame on the camera's past trajectory: reference frames merely provide pixel data used to warp target frames. We introduce a proactive rendering runtime system (Section 4.3), which predicts optimal reference frames that are *off* the camera trajectory and thus decouples the reference frame rendering from target frame rendering. The runtime then schedules the reference and target frames to maximally overlap the rendering of the two sets of frames.

***Dataflow Optimization.*** SpaRW reduces, rather than eliminates, computations in NeRF models. The remaining NeRF computations are still bounded by irregular DRAM accesses and frequent SRAM bank conflicts. Therefore, we propose two dataflow optimizations that enhance memory efficiency during radiance computation (Section 5). First, to mitigate irregular DRAM accesses, we propose to shift NeRF from a *pixel-centric* order to a *memory-centric* order. Our memory-centric order accesses the scene representations sequentially, inherently yielding full-streaming memory accesses. Second, to tackle SRAM bank conflicts, we introduce a novel data layout strategy. Instead of the traditional feature-major data layout, which stores an entire feature vector within a single SRAM bank, we propose a channel-major layout where different channels of the same feature vector are distributed across multiple SRAM banks.

***Hardware Augmentation.*** To effectively support our two dataflow optimizations, we have enhanced the classic systolic array architecture with two significant augmentations. First, we extend the existing vector unit to support exponential computations, a necessity for the Feature Computation in 3DGS. Second, we introduce a dedicated **Gathering Unit (GU)** designed specifically to accommodate our novel data layout. This new unit is tailored to manage the channel-major layout of data across multiple SRAM banks, ensuring that on-chip bank conflicts are eliminated.

The contributions of this article are as follows:

- We introduce SpaRW, a novel algorithm that reduces up to 88% of the GEMM in NeRF by exploiting radiance similarities between nearby rays.
- We propose a unified fully streaming framework for NeRF that reduces the redundant DRAM access and eliminates SRAM bank conflicts, ensuring complete streaming DRAM accesses.
- We augment associated hardware supports on the existing accelerator to support our streaming algorithms, and make it compatible across various NeRF models with structured and unstructured representations.
- We integrate Potamoi with six state-of-the-art NeRF solutions and demonstrate that Potamoi achieves a 53.1× and speedup and 67.7× energy savings over a baseline with a dedicated **Deep Neural Network (DNN)** accelerator while maintaining less than 1.0 dB degradation in **Peak Signal-to-Noise Ratio (PSNR)**.

## 2 Background

In this section, we start with reviewing the fundamentals in NeRF (Section 2.1). Then, we propose a unified framework to express general NeRF algorithms (Section 2.2).

### 2.1 NeRF Fundamentals

The debate between neural rendering and traditional ray tracing (physically based rendering) [57] is a widely contested topic in graphics. Our work does not seek to settle this debate; instead, we aim to enhance the efficiency of neural rendering, making it a more compelling option.

Fundamentally, NeRF rejuvenates the classic image-based rendering techniques [65] by learning the light field [24, 41] of a scene using modern deep learning methods. NeRF does away with detailed 3D modeling and physically simulating light transport in space. It avoids the complicated

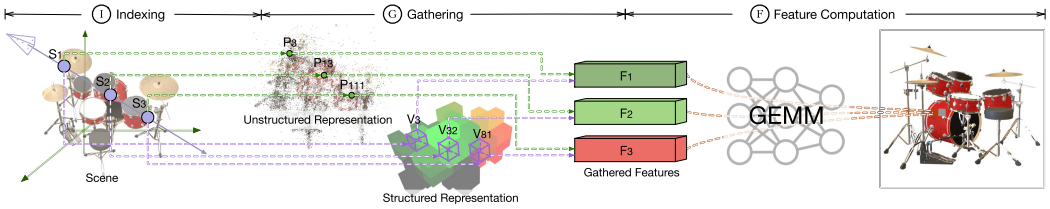



Fig. 1. The rendering pipeline of today's NeRFs consists of three stages: Indexing ($\mathcal{I}$), Feature Gathering ($\mathcal{G}$), and Feature Computation ($\mathcal{F}$). For NeRFs with structured representations, each ray first samples points ($S_1$, $S_2$, and $S_3$) along the ray direction. Each ray sample gathers and interpolates 3D features from eight vertices of the intersected voxel ($V_3$, $V_{32}$, and $V_{81}$) to obtain features ($F_1$, $F_2$, and $F_3$), as highlighted in purple. For NeRFs with unstructured representations, each ray directly intersects with Gaussian points ($P_3$, $P_{13}$, and $P_{111}$) to obtain features, as highlighted in green. Then, the features are fed into the GEMM-based computation to get the partial pixel values, as highlighted in orange. The final pixel value is summed from all partial pixel values [48].

setup in ray tracing (e.g., modeling the geometry of the scene and describing material properties). Instead, NeRF uses offline-captured images of a scene to train a differentiable model, which encodes the volume density and the light field in the scene (i.e., radiance of any ray) [24, 41]. At rendering time, given the camera pose where an image is to be rendered, the model is probed through the classic volume rendering [32, 40] to synthesize the image.

Existing NeRF algorithms can be classified into two main categories: MLP-based models with structured representations [11, 51, 68] and 3DGS with unstructured representations [12, 33, 38, 74]. Structured representations capture the geometry of the physical world using regular voxel grids, where the granularity of the voxels directly impacts the rendering quality and model size. However, unstructured representations are more flexible: the scene is represented by a point cloud, and each point is, in turn, represented by an ellipse whose shape and color are learned in training. The overall sizes of these unstructured representations are determined by two: the total number of Gaussian points and the parameter configurations of each Gaussian point.

## 2.2 General NeRF Pipeline

Despite numerous variants, we find that all prior NeRF algorithms can fit into a single computational paradigm. In this paradigm, the general NeRF pipeline consists of three stages: Indexing ($\mathcal{I}$), Feature Gathering ($\mathcal{G}$), and Feature Computation ($\mathcal{F}$), as illustrated in Figure 1.

***Indexing* ($\mathcal{I}$).** NeRF renders images in a pixel-centric order. In this order, we first generate a ray for each pixel to be rendered. Depending on the scene representation, rays have different fashions to sample points. With structured representation, each ray uniformly samples a set of points (e.g., $S_1$, $S_2$, and $S_3$ in Figure 1) along its direction. In contrast, with an unstructured representation, each ray directly interacts with specific Gaussian points it intersects. For instance, in Figure 1, the ray would directly intersect with $P_3$, $P_{13}$, and $P_{111}$ at $S_1$, $S_2$, and $S_3$, respectively. Any points that intersect with the ray contribute to the final pixel value.

***Feature Gathering* ($\mathcal{G}$).** In this step, each ray sample gathers corresponding features for Feature Computation. These features encode both the density and radiance field at the corresponding locations. With structured representation, each ray sample would calculate the ID of the voxel that contains the sample. 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 Figure 1, $S_1$ would access the eight vertex features in $V_3$. This ray sample then computes its feature by trilinearly interpolating the feature vectors of the eight vertices. With unstructured representation, each sample would directly apply the feature of the intersected Gaussian points. For instance, $S_1$ would apply the feature at $P_3$.

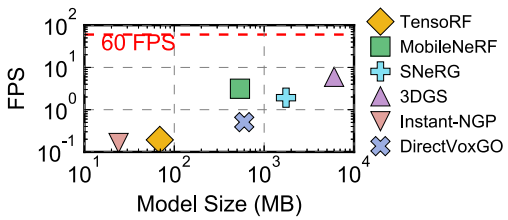



Fig. 2. Frame rate vs. model size on the Xavier SoC [53] across state-of-the-art NeRF algorithms [11, 14, 28, 33, 51, 68].

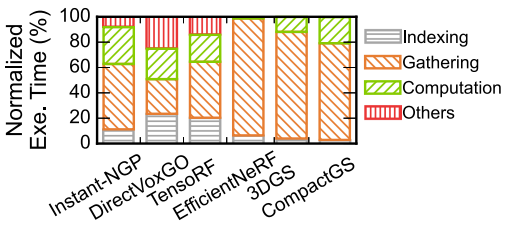



Fig. 3. Normalized execution breakdown across state-of-the-art NeRF algorithms [11, 30, 33, 38, 51, 68].

***Feature Computation*** **($\mathcal{F}$).** In Feature Computation, each ray sample passes its intermediate feature through a lightweight rendering model to obtain that sample's actual density and radiance value. Varied by designs, the major computation in the rendering model could be GEMM or other vector multiplications. For instance, Instant-NGP applies a lightweight MLP-based model to infer the radiance value while 3DGS leverages high-order spherical harmonics to approximate the radiance variation of a given point. 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.

## 3 Motivation

In this section, we first profile the existing NeRF algorithms and identify the performance bottleneck, Feature Gathering, across NeRF algorithms (Section 3.1). Next, we characterize the memory inefficiencies in Feature Gathering (Section 3.2).

### 3.1 Computation Inefficiencies

***Performance and Model Size.*** Today's NeRF models not only suffer from low performance but also present model storage challenges that could not fit in on-chip SRAMs. Figure 2 shows the distribution of model size ($x$-axis) and frame rate ($y$-axis) for six state-of-the-art NeRF models [11, 14, 28, 33, 51, 68] on a mobile Volta GPU [53]. By overlaying the 60 FPS runtime requirement, current NeRF algorithms cannot achieve real-time rendering on mobile devices.

Moreover, the size of NeRF models greatly surpasses the SRAM capacities of a typical modern mobile SoC, which leads to frequent DRAM accesses. In particular, a significant bulk of the model size is constituted by the feature vectors of the voxels, which range from approximately 10 to 1,000 MB. In contrast, the model weights in Feature Computation are relatively small, usually between 10 and 100 KB.

***Performance Bottleneck.*** Figure 3 shows the execution breakdown of different stages across six popular NeRF algorithms on a mobile Volta GPU [53]. All three stages take non-trivial execution time with Feature Gathering dominating the execution (>64% of execution time on average).

### 3.2 Memory Inefficiencies

Given the memory-intensive nature of Feature Gathering, we characterize its memory access patterns. Although Feature Gathering benefits from computational parallelism (both across different rays and between samples on a single ray), it does not translate well to memory efficiency, leading to irregular DRAM accesses and frequent on-chip SRAM bank conflicts.

***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.

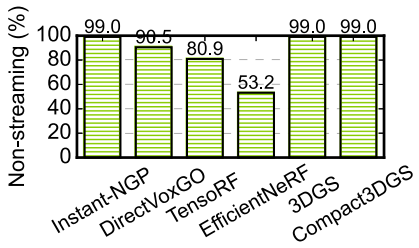



Fig. 4. Percentage of non-continuous DRAM accesses in Feature Gathering ($\mathcal{G}$).

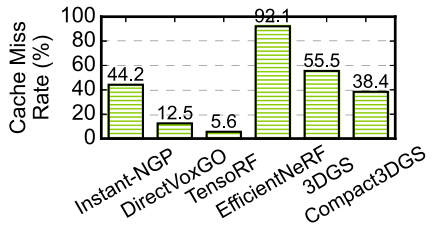



Fig. 5. On-chip memory miss rate in Feature Gathering ($\mathcal{G}$) across NeRF algorithms.

NeRF inference is parallelized across pixels as shown in Section 2.2. This pixel-centric rendering approach, however, introduces two distinct levels of memory-access irregularities: inter-ray and intra-ray irregularity. Inter-ray irregularity is from that two rays corresponding to adjacent pixels might access non-continuous memory regions. Despite originating from spatially close points, these rays may diverge as they travel through space, leading to varied memory access patterns.

Intra-ray irregularity arises when sampling points along a single camera ray access discontinuous memory regions. This occurs because the feature vectors corresponding to different ray samples may be stored at arbitrary locations in memory. As shown in Figure 1, for unstructured representations, ray samples, $S_1$ and $S_2$, intersect points, $P_2$ and $P_{13}$; for structured representations, ray samples, $S_1$ and $S_2$, intersect voxels, $V_3$ and $V_{32}$. In both scenarios, gathered features are spatially separate and stored discontinuously in DRAM. Figure 4 shows the non-streaming DRAM access in six popular NeRF algorithms. More than 87% of DRAM access is non-streaming. In particular, Instant-NGP, 3DGS, and Compact3DGS all have high non-streaming DRAM access (>99%) due to irregular memory access to their naive feature representations.

Given that the entire feature representations cannot be stored completely on-chip (see Figure 2), both types of irregularities would lead to repetitive redundant accesses to each voxel during rendering, which translates to redundant DRAM accesses. Given an on-chip buffer size of 2 MB with oracle replacement [53], Figure 5 presents the cache miss rates for various NeRF algorithms during Feature Gathering. The miss rate can be as high as 92% (average 41%). Practically, an even smaller on-chip buffer is expected to accommodate feature representations, further degrading rendering performance.

***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 meticulous data layout across SRAM banks [34, 78]. Conversely, the data access pattern of Feature Gathering in NeRF depends on the camera view and cannot be known offline.

Section 5.2 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 (Figure 6). Assuming a 2 MB buffer with 16 banks and 16 concurrent camera rays, the average bank conflict rate is 51%, 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 (Figure 8). Increasing the number of banks does alleviate bank conflicts. However, heavily banked SRAM designs are highly undesirable due to costly crossbars [1, 26].

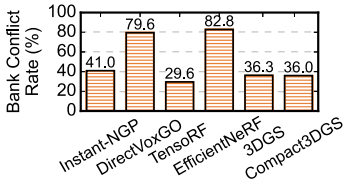



Fig. 6. SRAM bank conflict rate in Feature Gathering ($\mathcal{G}$), assuming 16 banks and 16 concurrent ray queries.

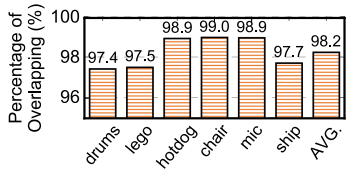



Fig. 7. The overlapping percentage across six scenes in Synthetic-NeRF [48].

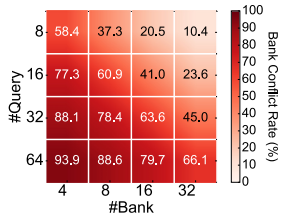



Fig. 8. Profiling SRAM bank conflicts of Instant-NGP with various banks and concurrent queries.

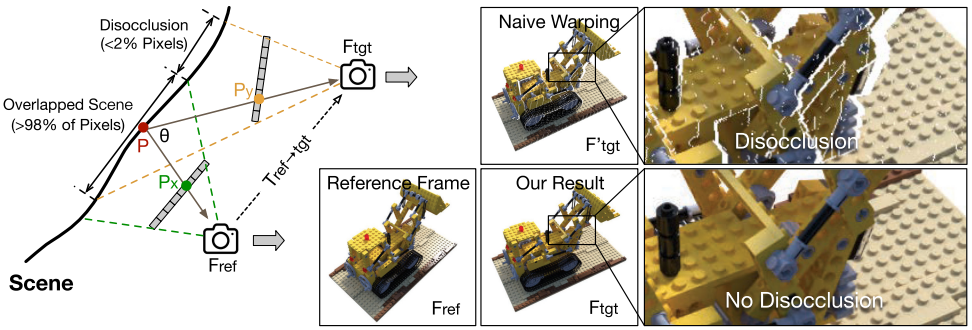



Fig. 9. Intuition of our SpaRW algorithm. The radiance of the $\overline{PP_x}$ ray can approximate the radiance of the $\overline{PP_y}$ ray if the angle $\theta$ between these two rays is sufficiently close. Leveraging this intuition, we show examples of a reference frame $F_{ref}$, a result of naive warping $F'_{tgt}$, and our result $F_{tgt}$ by SpaRW. Note that disocclusions (missing pixels) are eliminated in our result.

## 4 Sparse Radiance Warping

This section introduces SpaRW, an algorithm that exploits the radiance similarity across rays from nearby camera views. We first provide an intuition of the SpaRW algorithm (Section 4.1), followed by a description of the overall algorithm (Section 4.2). Last, we discuss a customized runtime to support SpaRW (Section 4.3).

### 4.1 Intuition

The goal of SpaRW is to reuse pixel values rendered in previous frames using a technique called *image warping*. Figure 9 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. Figure 7 characterizes the overlapping between two adjacent frames in the Synthetic-NeRF dataset [48]. More than 98% of pixels are overlapped (standard deviation: 1.7%), indicating

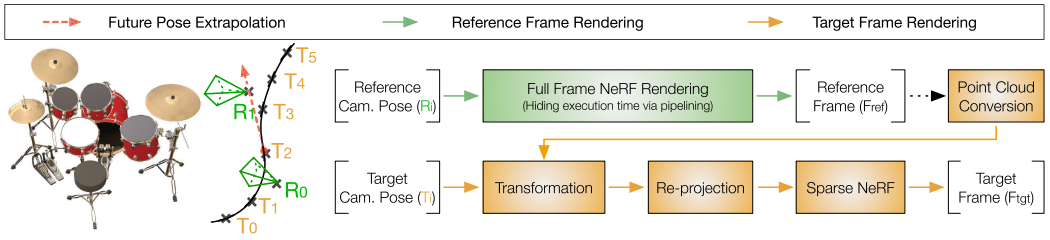



Fig. 10. An overview of the SpaRW algorithm. Only reference frames ($R_i$) undergo full-frame NeRF inference as denoted by the green path. All target frames ($T_i$) are computed using the lightweight warping operations denoted by the orange path. The reference frames are not on the camera trajectory,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

that less than 2% of pixels require re-rendering. The same conclusion holds for real-world datasets: on the Unbounded-360 [6] and Tanks and Temples [35] 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 $F_{tgt}$. Figure 9 shows the effect of a naive warping. Without recalculating the disoccluded pixels, the rendered image $F'_{tgt}$ 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.

### 4.2 Basic Algorithm

In SpaRW, there are two rendering paths, which are illustrated in Figure 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 Section 4.1 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: ❶ point cloud conversion, ❷ transformation, ❸ re-projection, and ❹ sparse NeRF rendering.

❶ 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:

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

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 a depth buffer) [63], $f$ is the camera focal length, and $[C_x, C_y]$ is the camera center. Both the focal length and camera center are part of the camera's intrinsic parameters [21].

❷ 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:

$$P_{tgt} = T_{ref \to tgt} \times P_{ref}, \tag{2}$$

where $T_{ref \to tgt}$ 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.

❸ 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 [63]:

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

where $D_{tgt}$ is the depth of all the points corresponding to the pixels in the target frame. A standard depth buffer technique (also known as a $z$-buffer) is used to maintain the correct depth of objects in 3D space from the camera perspective.

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

$$F_{tgt} = F'_{tgt} \circledast \Gamma_{sp}. \tag{4}$$

$\Gamma_{sp}$ denotes sparse NeRF rendering of disoccluded pixels, and $\circledast$ combines the warped pixels with 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, for example, 1 million points, is less than 1 ms on an NVIDIA Volta mobile GPU.

### 4.3 Proactive Rendering Runtime

In SpaRW, the rendering of a target frame depends on the existence of a reference frame. The choice of reference frames and when to render them affect both the rendering quality and the performance. Rendering reference frames on the trajectory is a common strategy in previous work [10, 20, 66, 79]. The idea is that adjacent frames are highly correlated so one can extrapolate from a previously rendered frame to render the current frame. While this approach is work-efficient since reference frames are required to be rendered regardless, it has an inherent limitation in which it poses the data dependency between reference frames and target frames. Specifically, a reference frame can only be rendered after the completion of the previous target frame. This dependency inherently forces the rendering process into a serial order, where target frames and reference frames cannot be rendered concurrently, potentially slowing down overall system performance.

Our key observation is that a reference frame can be any frame, even those that are not on the camera's trajectory, such as $R_0$ and $R_1$ in Figure 10. As long as the reference frame is close to the camera trajectory, the radiance approximation still holds. We introduce a runtime system that predicts suitable reference frames *off the trajectory*; that way, we can overlap reference frame rendering with target frame rendering. As depicted in Figure 11, our runtime system comprises three key components: a predictor, a scheduler, and a rendering engine.

***Predictor.*** We introduce a *proactive* approach to predict camera poses for *future* reference frames. The idea is that reference frames might not be the frames viewed by users. Rather, the primary functionality of reference frames is to supply pixel information for target frames. To this

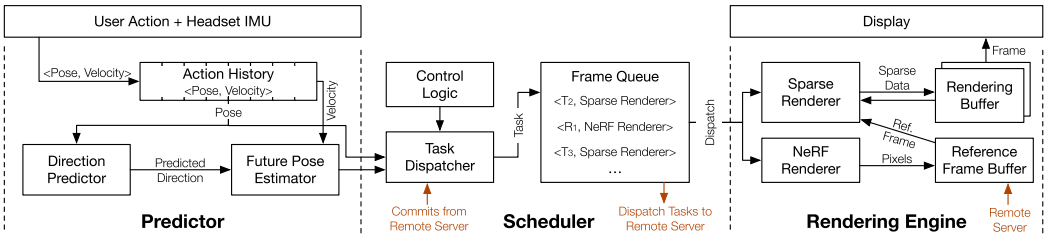



Fig. 11. Proactive rendering runtime consists of three key components: a predictor, a scheduler, and a rendering engine. The predictor proactively predicts the future camera poses for reference frames, and the predicted poses ($R_i$) are sent to the task dispatcher along with real-time camera poses ($T_i$) for target frames. The scheduler then dispatches tasks to the corresponding renderers to achieve concurrent rendering for reference and target frames in Figure 10. Our runtime also allows reference frames to be offloaded to a remote server by leveraging cloud computing resources. Once the reference frames are finished, the remote server will send them back to the reference frame buffer.

end, we propose a lightweight direction predictor that utilizes a brief history of past camera poses to predict the future direction of the current trajectory. In particular, our direction predictor takes the five previous camera poses as input and uses a Bayesian Ridge regression model [71] to estimate the upcoming camera direction. Experimental results show that our predictor maintains a low prediction error (<0.01 rad) with a negligible runtime overhead.

Once the camera direction is predicted, we use the position of the last two rendered frames, $T_1$ and $T_2$, to calculate the current velocity $v$ at $T_2$, $v = \frac{T_2 - T_1}{\Delta t}$, where $\Delta t$ represents the interval between two consecutive frames. We then calculate the pose of the reference frame, $R_1 = T_2 + v \times t_r$, $t_r = \frac{N}{2}\Delta t$, where $N$ is the number of target frames that share the same reference frame (i.e., four in this example as $T_2$–$T_5$ share $R_1$). $\Delta t$ represents the interval between two consecutive frames. Using $\frac{N}{2}$ allows the reference frame to be roughly at the center of, and thus increase the overlap with, its target frames.

***Scheduler.*** Both predicted and current camera poses are sent to the task dispatcher, which inserts rendering tasks into the frame queue. Depending on the type of camera pose, the rendering tasks will be dispatched to the corresponding renderers. For instance, a predicted camera pose ($R_1$) would invoke a full-frame rendering task to the NeRF renderer while a target camera pose $T_3$ will be dispatched to the sparse renderer *simultaneously*. Our strategy can effectively hide the latency of reference frame rendering by overlapping with rendering multiple target frames. Our experiment shows that a single reference frame can effectively support up to 30 target frames with minimal loss in visual quality (Section 8.2).

Meanwhile, a control logic continuously monitors the task scheduling to guarantee two properties. First, the control logic would check the data dependencies among the rendering tasks so that no target frame rendering would be dispatched without its corresponding reference frame rendered. Second, if the angle ($\theta$) subtended by a ray in the reference frame and its corresponding ray in the target frame exceeds a pre-defined threshold ($\phi$), the control logic would stop the frame warping process and initiate a full-frame NeRF rendering task instead. This ensures that rendering quality is guaranteed with no significant delays.

***Rendering Engine.*** Once a rendering task is dispatched, the rendering engine activates a corresponding renderer based on the task type. For example, a reference frame rendering task is directed to the NeRF renderer, which activates full-frame NeRF rendering to generate a reference frame and stores it in the reference frame buffer. Here, we maintain a dedicated reference buffer, enabling the sparse renderer and the NeRF renderer to execute concurrently without mutual

interference. Meanwhile, leveraging the previously generated reference frame, the sparse renderer performs sparse NeRF rendering along with other lightweight computations shown in Figure 10. To facilitate efficient processing, the rendering buffer is double-buffered, allowing it to hold some intermediate data during the execution of SpaRW.

***Support Remote Rendering.*** Our runtime system also supports offloading the heavy full-frame NeRF rendering to a remote server and receives the rendered result from the remote server as highlighted in Figure 11. The end of Section 7 describes the rationale of remote rendering, and Section 8.3 quantifies the advantage of SpaRW in remote rendering.

## 5 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 Section 3.2. This section first describes an algorithmic optimization that eliminates redundant DRAM accesses and guarantees fully streaming DRAM accesses (Section 5.1). We then discuss a new on-chip data layout that eliminates SRAM bank conflicts (Section 5.2).

### 5.1 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 feature vectors (either voxel features in structured representations or point features in unstructured representations) in the NeRF model. However, this buffer is generally too small (1–3 MB) to hold all feature vectors (10–1,000 MB). Additionally, there is a dedicated on-chip buffer to store NeRF model weights; these weights are generally small (10–100 KB).

***Memory-Centric Rendering.*** 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, feature vectors 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 feature vector is read only once and the DRAM accesses to the feature vectors are fully streaming. Memory-centric rendering incurs no additional storage overhead. The vertex features are stored in memory *as is*, without duplication. What is being reordered is the feature *accessing* order. Figure 12 illustrates this idea and how the three stages in NeRF are changed accordingly.

***Indexing ($\mathcal{I}$).*** We first group spatially close feature vectors into "macro voxels" (MVoxels). For instance, with structured representation, in Figure 12, we can combine every $2 \times 2$ voxels into one MVoxel; with unstructured representation, we group all point features confined within one MVoxel. All the data in a MVoxel is loaded to the SRAM together when the MVoxel is loaded. The intention here is that, via offline analysis, we guarantee that the overall data stored in one MVoxel is smaller than the on-chip buffer size. To better exploit memory locality, features within one MVoxel are stored continuously in the DRAM, the same as MVoxels.

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 is used here to overlap

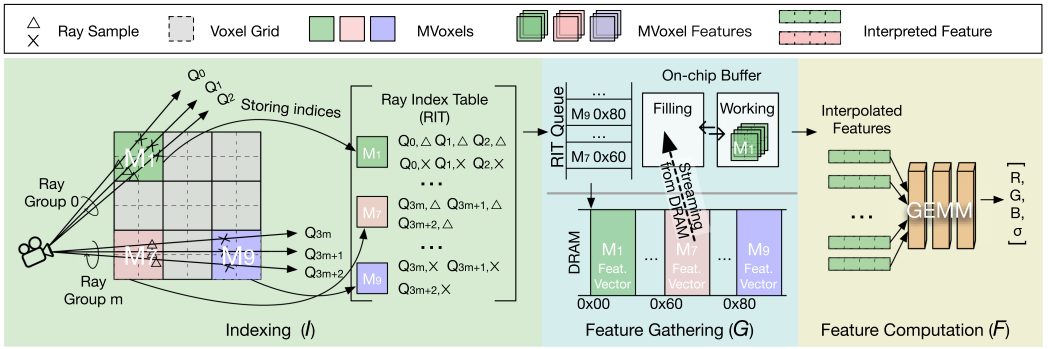



Fig. 12. Fully streaming NeRF rendering algorithm for structured representations [11, 51, 68]. We first group all the voxels into MVoxels, which are continuously stored in the DRAM. The RIT records, for each MVoxel, the IDs of ray samples whose features reside in the MVoxel. During Feature Gathering ($\mathcal{G}$), 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.

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

In practice, we first group a set of nearby rays into a ray group. For each ray group, we identify MVoxel IDs that all ray samples in the ray group would index to, from the near-camera point to the far distance. Based on the indexed MVoxel IDs, these ray samples are gathered into a RIT. The Feature Computation is also performed starting from the near-camera MVoxel, thus the early ray termination in the general NeRF rendering still applies.

***Accommodating Unstructured Representation.*** The naive approach to adapting an unstructured representation is, at Indexing ($\mathcal{I}$), to uniformly partition the entire scene space into the same-size MVoxels while ensuring that the point features within one MVoxel can fit in the on-chip buffer. However, this uniform partitioning often leads to significant under-utilization of the on-chip buffer. The root cause is that Gaussian points are not evenly distributed across the space, so there would be many MVoxels that contain no Gaussian points.

To improve the utilization of the on-chip buffer, we merge adjacent MVoxels with low point densities and process them together. Figure 13 illustrates our idea. For instance, given that the four purple MVoxels (at the bottom right) are sparser than their neighboring MVoxels, we group the them into one single, larger MVoxel $M_9$. In our implementation, we balance the complexity and efficiency and only groups of $2 \times 2 \times 2$ MVoxels into a larger MVoxel if the combined size of the point features within these eight MVoxels does not exceed the capacity of the on-chip buffer. The remaining two operations, Feature Gathering and Feature Computation, are kept the same.

Coupled with variable-sized MVoxels, we use an octree structure to facilitate traversing and processing these MVoxels. As Figure 13 illustrates, starting from the root node, the tree traversal would first visit one of its subtree nodes and perform a depth-first search until a leaf node is reached. A lead node represents an MVoxel. The possible rays would perform Gaussian point-ray intersections to check which Gaussian points would intersect with the rays and store the results in the RIT. This process would iterate until all MVoxels are visited.

Note that we group low-density MVoxels into larger MVoxels (eight subtree nodes into one single node), resulting in an octree that is not perfectly balanced. However, this unbalanced octree leads to more efficient use of the on-chip buffer. Figure 14 shows the point-per-MVoxel distribution before and after merging. Post-merging, the average number of points per MVoxel

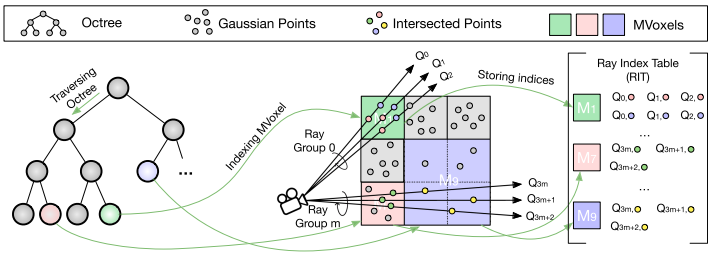



Fig. 13. Modified Indexing ($\mathcal{I}$) stage for unstructured representations [12, 33, 74]. We merge adjacent low-density MVoxels for better on-chip buffer utilization. An offline-built octree is used to help MVoxel traversal.

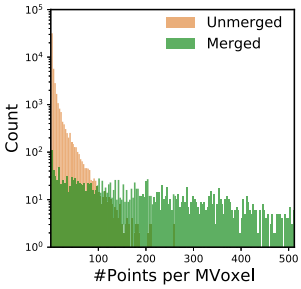



Fig. 14. Distribution of points per MVoxel before and after merging. Post-merging octree improves on-chip buffer utilization.

improves significantly from 7.2 to 172.8, indicating a more compact and memory-efficient structure. Section 8.3 will further illustrate the performance benefits of the merged octree.

***Accommodating Hierarchical Data Encodings.*** Some NeRF algorithms, instead of storing voxel features directly, use hierarchical data structures such as hierarchical voxel grid [51], hashing [55], and factorized tensor [11] to encode.

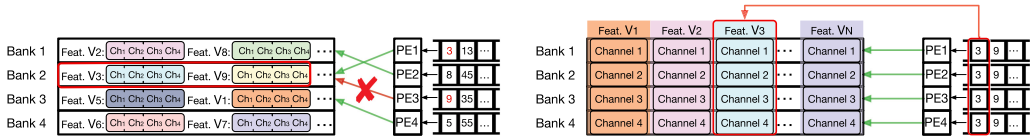
To accommodate our fully streaming dataflow with these hierarchical data structures, we first partition 3D voxels at each level into MVoxel grids. For hash grid data structures, each hash entry is an MVoxel rather than one single voxel feature so that DRAM streaming within one MVoxel can be guaranteed. During Feature Gathering, we group all rays into small ray groups (see Figure 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) dataflow. This reversion happens in, for instance, Instant-NGP [51] from level 5 (out of eight levels) onward. As a result, about half of the DRAM traffic on Instant-NGP is non-streaming (which is faithfully captured in evaluation).

## 5.2 Bank Conflict Free Interleaving

With fully streaming DRAM accesses, the inefficiency shifts to the on-chip SRAM, which experiences frequent bank conflicts (see Figure 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 [34, 78], the SRAM access pattern of ray samples is known only at the runtime, because the exact ray samples depend on the runtime camera pose information.

The reason behind bank conflicts in Feature Gathering has to do with the way feature vectors are laid out in SRAM; Figure 15(a) illustrates this point. State-of-the-art NeRF accelerators [36, 43] 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 that 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 Figure 15(b), where different channels of the same feature vector are spread across banks. For instance, bank 1 stores




(a) The feature vector layout in existing NeRF accelerators, where vertex features are spread across SRAM banks, but all the channels in the same feature vector are stored in the same bank.

(b) Our data layout spreads channels of a feature vector across different banks.

Fig. 15. A comparison between the original feature vector layout (a) and our data layout (b). The example has four banks and four concurrent PEs. In (a), 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 (b) eliminates bank conflicts by (1) spreading channels across banks and (2) having each PE collect a particular channel across different ray samples.

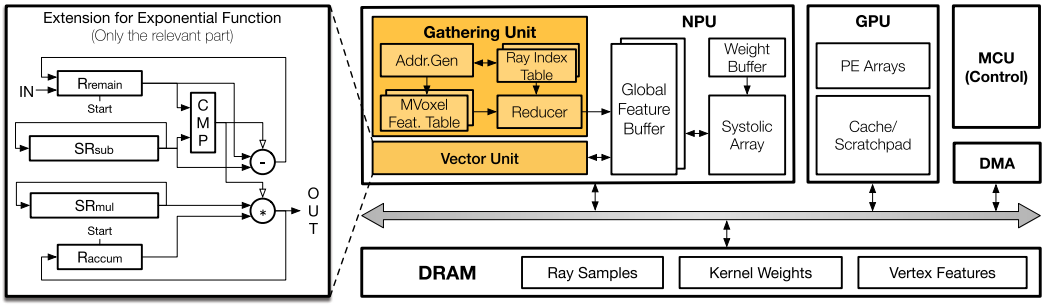



Fig. 16. The SoC architecture. The uncolored portion is the baseline architecture, and we augment a standard systolic array-based NPU with a GU and a vector unit, which are colored. The GU executes Feature Gathering ($\mathcal{G}$), and the MAC array executes Feature Computation ($\mathcal{F}$). The GPU executes the rest. The vector unit supports element-wise updates such as ReLU, and exponential operation.

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. In other words, each PE is dedicated to a specific bank. For instance, in Figure 15(b), the four PEs are collecting the four channels of the same feature vector 3 (required by one ray sample).

## 6 Hardware Support

***Architecture Overview.*** We extend a standard NPU architecture to support the two memory optimizations. Figure 16 shows the SoC architecture, in which we augment the NPU with the new GU and modify the vector unit to support exponential computation for 3DGS. 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 ($\mathcal{F}$). The GPU executes the rest of the computations—for example, Ray Indexing ($\mathcal{I}$) and the first three steps in SpaRW for the target frames. For 3DGS with unstructured representations, *Sorting* and *Spherical-Harmonics-to-Color* are also executed on the GPU.

***Vector Unit.*** To support Gaussian splatting operation within 3DGS algorithms, we augment the vector unit to include exponential functionality. We introduce a convergence method to

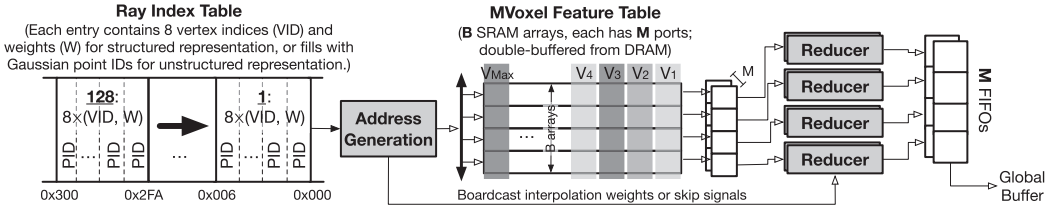



Fig. 17. The GU. Each RIT entry stores the information of a ray sample. For structured representations, each entry has eight vertex IDs (VIDs) as well as the corresponding weights (W) for trilinear interpolation. For unstructured representations, each entry contains a single ID of the intersected Gaussian point (PID). The address generation logic uses the VID/PIDs to access the corresponding features from the MFT, which stores the features of a MVoxel using the channel-major layout. The VFT has $B$ individual SRAM arrays, each of which has $M$ ports, supporting retrieving features for $M$ ray samples simultaneously. Trilinear interpolation is supported by $B \times M$ number of reducers to calculate the final features for $M$ ray samples. The weights are broadcast to the reducers. A FIFO holds the feature results before writing to the global feature buffer in Figure 16.

iteratively compute the exponential function [7]. Here, we focus on explaining the computation of exponential values within the range of −1 to 1. For values beyond this interval, they still can be computed using our extension by scaling down using powers of 2.

The key idea of this approach is to decompose the power $x$ into a sequence $\{\ln(1 + \alpha_i \times 2^{-i})\}$, where $\alpha_i \in \{0, 1\}$ and $i \in [0, M)$. This can be done by comparing the residual value in Register $R_{\text{remain}}$ with the first element in Shift Register $SR_{\text{sub}}$, and enabling their subtraction (see Figure 16). Meanwhile, at each iteration, the first element of Shift Register $SR_{\text{sub}}$ cycles to the end, while the subtraction result is written back into $R_{\text{remain}}$. The outcome from the comparator serves as a flag to enable the multiplication between the value in $R_{\text{accum}}$ and the first element in Shift Register $SR_{\text{mul}}$ (containing $\{(1 + \alpha_i \times 2^{-i}), i \in [0, M)\}$). Once the multiplication is done, the result is also written back to register $R_{\text{accum}}$, and the first element in Shift Register $SR_{\text{mul}}$ would cycle to the end. Following $M$ cycles, the final exponential value would be obtained.

***Gathering Unit.*** The GU architecture is detailed in Figure 17. This GU includes a dedicated buffer for RIT and a double-buffered **MVoxel Feature Table (MFT)** for streaming MVoxels from DRAM, as discussed in Section 5.1. The data layout for the MFT, optimized to eliminate bank conflicts, is outlined in Section 5.2. After the RIT entries are loaded, the address generation logic in the GU processes each entry to compute the necessary addresses for retrieving the corresponding vertex or Gaussian point features from the MFT.

The MFT, free of bank conflicts, is configured as $B$ individual SRAM arrays without a crossbar to reduce area overhead. All channels of a feature vector are accessed simultaneously, enabling a one-cycle read per vertex feature. For structured representations that require eight vertices per voxel, this translates to eight cycles to access all features for a ray sample. Conversely, unstructured representations only require a single cycle per entry. Each SRAM bank is designed with $M$ ports, allowing parallel reading and processing of $M$ ray samples.

The GU uses $B \times M$ reducers to perform trilinear interpolation to calculate feature vectors for each ray sample. A skip-reduction flag enables bypassing trilinear interpolation for 3DGS. Once processed, results are stored in the Feature FIFO and then transferred 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 GU) 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 RIT through the DMA to the NPU.

## 7 Experimental Setup

***Hardware Details.*** The NPU is a systolic array based DNN accelerator, which has a $24 \times 24$ MAC array, where each MAC unit mimics the design of that in the TPU [31]. Each PE consists of two 16-bit input registers, a 16-bit fixed-point MAC unit with a 32-bit accumulator register, and simple trivial control logic. The NPU also consists of a vector unit, which can parallelize element-wise updates such as ReLU, exponential operations. 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. Depending on the NeRF algorithms, 6 KB can store 128 entries (each accommodates eight vertices) for structured representation or 1,536 Gaussian point indices for unstructured representation. The MFT is also double-buffered, with 32 KB each (organized as $B = 32$ banks each with $M = 2$ ports), which can store a MVoxel (8×8×8 points) with 32 channels. When the channel size of a feature is greater than 32, we partition the features into segments along the channel direction and load each segment sequentially.

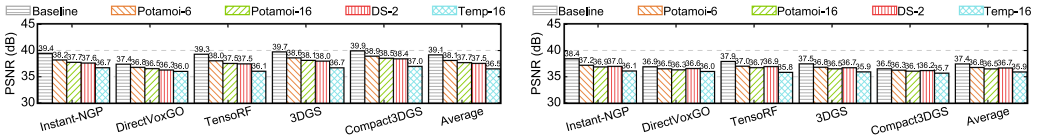
***Experimental Methodology.*** We directly time the GPU execution as well as the kernel launch on the mobile Volta GPU on NVIDIA's Xavier SoC [53]. The GPU power is directly measured using the built-in power sensing circuitry. We synthesize, place, and route the datapath of our design using an EDA flow consisting of Synopsys and Cadence tools with the TSMC 16 nm FinFET technology and scale the results to 12 nm using the DeepScaleTool [62, 67] so that the results can be comparable with the mobile Volta GPU on NVIDIA's Xavier SoC in 12 nm node [53].

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 (four channels) according to its datasheet [46]. The DRAM energy is obtained using Micron System Power Calculators [47]. 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.*** Potamoi introduces minimal area overhead with GU augmentation. The major overhead is from 44 K SRAM introduced from the RIT buffer and VFT buffer. The additional area overhead (0.048 $mm^2$) compared to the baseline NPU is less than 2.5%, in which the Potamoi-specific portion is almost negligible compared to the entire SoC area, such as 350 $mm^2$ for NVIDIA Xavier [54] and 108 $mm^2$ for Apple A15 [4]. 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.*** Potamoi can accommodate arbitrary NeRF algorithms. To demonstrate the flexibility of Potamoi, we evaluate three different NeRF algorithms: Instant-NGP [51], DirectVoxGO [68], TensoRF [11], 3DGS [33], and Compact-3DGS [38], with varying model size-computation tradeoffs (see Figure 2). We evaluate algorithm quality on PSNR.

***Datasets.*** We use the Synthetic-NeRF dataset [48], which includes eight different synthetic scenes, for our evaluations. Additionally, to show a broader applicability of Potamoi, we use two real-world datasets, Unbounded-360 [6] (Bonsai trace) and Tanks and Temples [35] (Ignatius trace), to assess SpaRW under real-world conditions. Although real-world datasets lack corresponding scene meshes, generating meshes from real-world images is a well-mature field (photogrammetry).




(a) Quality evaluation on the Synthetic-NeRF dataset (b) Quality evaluation on real-world datasets

Fig. 18. Image quality comparison. Potamoi-6 and Potamoi-16 use a warping window size (i.e., the number of target frames a reference frame is used for) of 6 and 16, respectively.

For mesh generation, we use Agisoft Metashape [2], a well-established photogrammetry software. To avoid background discrepancies, we initially segment the backgrounds from all images and focus on generating meshes solely for the foreground scenes. We show that Potamoi achieves high accuracy even with these imperfect meshes in Section 8.1.

***Baseline.*** Our baseline is the SoC in Figure 16 without the GU. It executes Ray Indexing ($\mathcal{I}$) and Feature Gathering ($\mathcal{G}$) on the GPU and Feature Computation ($\mathcal{F}$) on the NPU for all frames.

***Variants.*** We evaluate three variants of Potamoi to decouple the contribution proposed in our article:

- 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.
- Potamoi: The full version of Potamoi, 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 stand-alone device with the hardware described previously.
- *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 [44].

## 8 Evaluation

We first demonstrate that Potamoi achieves quality levels comparable to the baseline (Section 8.1). Next, we discuss the effectiveness of Potamoi on real-world datasets (Section 8.2.) We then show the speedup and energy reduction compared to the baseline hardware incorporating a dedicated DNN accelerator (Section 8.3). Finally, we show that Potamoi achieves better speedups compared to prior NeRF accelerators (Section 8.4).

### 8.1 Rendering Quality

Figure 18 shows the rendering quality of applying our SpaRW algorithm to NeRF algorithms on both the Synthetic-NeRF dataset (see Figure 18(a)) and real-world scenes (see Figure 18(b)). We use a *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

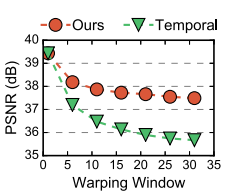



Fig. 19. The accuracy sensitivity to window size between PSNR and the warping-based method.

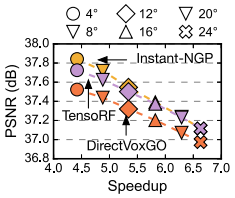



Fig. 20. Speedup and PSNR of Potamoi-16 under different warping thresholds $\phi$ on the 1 FPS sequence.

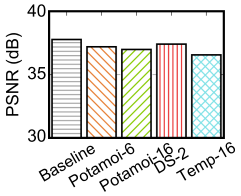



Fig. 21. PSNR comparison on the Tanks and Temples dataset in sparse sequences.

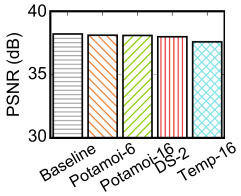



Fig. 22. PSNR comparison on the Tanks and Temples dataset in dense sequences.

rendering and then upsamples 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, Potamoi-6 retains an average PSNR drop within 1.0 dB compared to the original algorithms. Despite Potamoi-16 dropping the average quality by 1.4 dB, it still has better quality compared against DS-2 and Temp-16 on the Synthetic-NeRF dataset. Temp-16 is the worst because it warps from previous frames and accumulates errors. The quality of Potamoi-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 Section 8.2.

### 8.2 Robustness

Figure 19 shows the comparison between two methods: *reference-based warping* and *temporal-based warping*. Here, we use the Synthetic-NeRF dataset and evaluate the quality under different warping window sizes using Instant-NGP [51]. As shown in Figure 19, temporal-based warping drops its visual quality rapidly as the warping window size increases. At a warping window of 6, the quality of temporal-based warping is already lower than DS-2 (37.60 dB). In contrast, reference frame based warping can still maintain competitive visual quality even with a large warping window size of 31.

We use the *Ignatius* scene in the Tanks and Temples dataset to discuss the effectiveness and limitations of SpaRW on real-world scenes. Figure 21 shows the results. Both Potamoi-6 and Potamoi-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., $\theta$ in Figure 9 is too large), so the radiance approximation does not hold well for a 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 Potamoi 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 Figure 22. In this more realistic scenario, Potamoi-16 has little quality loss over the baseline and has a similar quality compared to DS-2 but is about 4× 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 Section 4.3 can be used to mitigate the rendering quality loss. Figure 20 shows the speedup and PSNR of Potamoi-16 across different warping thresholds on the challenging 1 FPS sequence of *Ignatius*. As $\phi$ reduces toward the left, the quality increases, since fewer pixels are warped and more pixels are NeRF rendered, which also means that the performance reduces. At a threshold of 4°, SpaRW has a quality drop within 0.1 dB and a speedup of 4.3×.

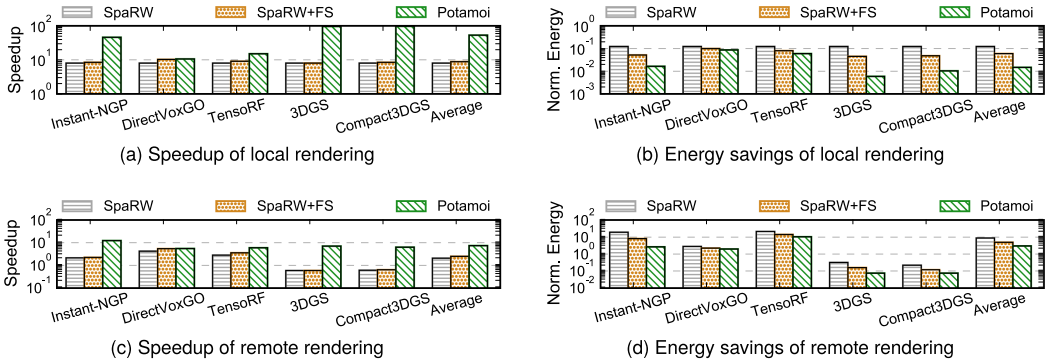



Fig. 23. End-to-end speedup and normalized energy of our variants over the baseline with a GPU and an NPU. We evaluate two application scenarios: local rendering and remote rendering. All values are normalized to the baseline.

### 8.3 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 of which are common in VR. Unless stated otherwise, we use a warping window of 16 for our evaluation.

***Local Rendering.*** Figure 23(c) 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 1.1× speedup and 2.1× energy saving under the same hardware configuration.

Two factors help SpaRW + FS improve upon the baseline. First, the SpaRW algorithm reduces the amount of full-frame NeRF computation. Second, fully streaming NeRF rendering reduces redundant DRAM accesses. With GU hardware support, Potamoi further boosts the speedup and energy saving to 53.1× and 67.7×, respectively. For instance, Potamoi achieves only 9.33 FPS on Instant-NGP, whereas Instant-NGP achieves merely 0.20 FPS on the baseline hardware. However, 3DGS achieves more than 1,000 FPS since the baseline 3DGS is already fast.

***Remote Rendering.*** One factor that prevents Potamoi from achieving higher speedup is resource contention: even though algorithmically reference frame and target frame rendering can be overlapped, they compete for the same NPU and GPU resources. With additional resources on a remote machine, Potamoi further boosts the performance.

Figure 23(d) 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 2.1× speedup against the baseline, whereas SpaRW + FS achieves a 2.5× speedup by applying fully streaming NeRF rendering. Note that for 3DGS and Compact-3DGS, SpaRW and SpaRW + FS slow down the overall performance due to the computational gap between the remote GPU and local SoC. With GU hardware support, Potamoi further improves the speedup to 7.5×. Even for 3DGS and Compact-3DGS, Potamoi can achieve 6.0× and 7.1× speedup. 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 Potamoi.

Notably, the baseline in this scenario consumes lower energy than all three variants of Potamoi. This is because when all the computations are offloaded to the remote GPU in the baseline, the

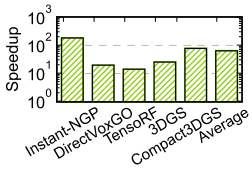



Fig. 24. Speedup of Feature Gathering.

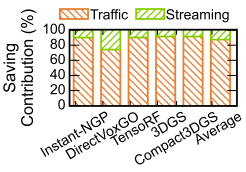



Fig. 25. Memory energy saving contribution.

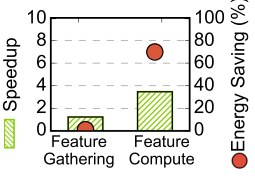



Fig. 26. Energy reduction from merged octree data structure.

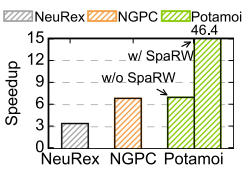



Fig. 27. Performance comparison against prior works.

main energy consumption of the local device is wireless communication. Transferring one frame consumes almost 3.0× lower energy than rendering a frame in Potamoi.

***Feature Gathering* ($\mathcal{G}$).** Figure 24 demonstrates the speedup and energy reduction brought by GU compared to the GPU execution. Overall, our GU achieves 64.1× 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. Potamoi eliminates the irregular accesses entirely. Coupled with the GU, we achieve a 182.4× speedup on Instant-NGP.

***Memory Saving Contribution.*** Not only does Potamoi eliminate non-streaming DRAM access, it also reduces the overall DRAM traffic. Figure 25 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, Potamoi effectively improves the reuse of each MVoxel, 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 (see Figure 24).

***Merged Octree.*** Figure 26 shows the speedup and energy savings achieved by merging the octree for unstructured representations as discussed in Section 5.1. Merging low-density MVoxels achieves speedup on Feature Gathering and Feature Computation stages by 1.3× and 3.5×, respectively. This shows that merged octree improves the resource utilization of both the GU and systolic array. Regarding energy efficiency, merging octree results in a 70.1% energy reduction of Feature Computation, which is dominated by the computation of the systolic array. Conversely, a mere 1% of energy is saved in Feature Gathering, which is largely driven by memory operations. Note that octree merging is performed offline, thus incurring no runtime overhead. The average memory overhead of octree is less than 200 KB, which is much smaller compared to the overall NeRF model size.

### 8.4 Comparison with Prior Works

We also compare against two prior NeRF accelerators: NeuRex [36] and NGPC [49]. Notably, both of these accelerators are tailored to one particular NeRF algorithm, Instant-NGP, whereas Potamoi generally applies to any NeRF algorithms.

Figure 27 compares Potamoi with the two accelerators on Instant-NGP. All values are normalized to the GPU baseline. We use the data reported in the NeuRex paper[1] and implement the NGPC

[1]The 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 speedup of NeuRex over Xavier used in Figure 27.

architecture based on the paper's description. For a fair comparison, we configure Potamoi to use the same amount of PEs ($24 \times 24$) as NGPC; NeuRex has a higher PE count ($32 \times 32$) 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, Potamoi without the SpaRW algorithm demonstrates a 2.0× speedup over NeuRex. The speedup against NeuRex is attributed to hardware augmentation of GU in Potamoi, 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). Potamoi 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 a fully streaming rendering algorithm, our on-chip SRAM size is only 32 KB. With our SpaRW algorithm, Potamoi boosts the speedup to 16.4× and 8.2× against NeuRex and NGPC, respectively.

## 9 Related Work

***NeRF Acceleration.*** NeRF rendering has drawn considerable attention in the past 2 years. Recent works have proposed several accelerators for NeRF algorithms [18, 22, 27, 36, 42, 43, 49, 61]. However, prior designs are tailored to individual NeRF algorithms—one accelerator for one algorithm. For example, Instant-3D [43] and NeuRex [36] accelerate the training and inference of Instant-NGP [51], respectively. NGPC [49] accelerates a range of neural graphic algorithms with similar hierarchical feature encodings. ICARUS [61] and RT-NeRF [42], however, design specialized architecture to speed up NeRF [48] and TensoRF [11], respectively. Meanwhile, Gen-NeRF [22] accelerates IBRNet [73] for novel view synthesis. Despite that recently Cicero [18] and GSCore [37] proposed acceleration frameworks for structured and unstructured representations, Potamoi proposes a fully streaming computing framework that is generally applicable to a range of existing NeRF algorithms with minimal hardware augmentation.

***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 [3, 8, 25, 58, 64].

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 runtime, 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.

***Dataflow Framework.*** Dataflow optimization for DNN algorithms is a well-established research area. Various techniques have been proposed to improve computational efficiency and reduce off-chip data traffic via reordering, tiling, and binding [13, 59, 60]. Recently, studies have aimed to fuse DNN operations to enhance data reusability and decrease data traffic across DNN layers [76, 77]. However, these studies focus on the dataflow scheduling of regular computations (e.g., GEMM), where optimal dataflow scheduling can be determined offline due to static data access patterns. Although there are approaches designed for irregular computations [16, 17, 19, 45, 72], they are not suitable for NeRFs.

Potamoi addresses the unique challenges of NeRF rendering, where data access patterns can only be determined prior to rendering. By shifting from the conventional pixel-centric rendering

to memory-centric rendering, Potamoi proposes a streaming framework that supports coarse-grained streaming and operation-level pipelining, effectively leveraging the unique computational characteristics of NeRF algorithms.

***Real-Time VR Rendering.*** To achieve real-time VR rendering, prior works have leaned on image-based rendering or remote rendering [9, 29, 39, 50, 70]. Some systems directly stream rendered videos to clients, but they often suffer from bandwidth limits and require efficient video encoding [52]. Other works transmit one frame that can be reused to render multiple viewpoints, but they often require complex encoding methods to store texture and geometric information to address disocclusions [29, 50]. By leveraging the computation characteristics of NeRF, Potamoi proposes a straightforward yet effective approach to resolve disocclusions, achieving photorealistic visual quality.

## 10 Conclusion

Potamoi presents an interesting idea, radiance warping, to reduce the overall pixel rendering by NeRF. This heuristic is built upon the assumption that either the object surface is non-diffuse or the angular disparity between two corresponding pairs of warping pixels is sufficiently small. A more principal future direction is to use a lightweight DNN model to learn the material properties [5, 23], such as bidirectional reflectance distribution function and bidirectional subsurface scattering function [57], to approximate the warped pixel values instead.

Nevertheless, we reduced more than 95% of the MLP computation in NeRF by warping radiances computed in previous frames with less than 1 dB PSNR loss. We also showed two memory optimizations that transform the computation orders in NeRF and address the issues in both DRAM and SRAM accessing. Collectively, we demonstrated more than an order of magnitude speedup and energy savings over a mobile Volta GPU.

## Acknowledgment

We thank anonymous reviewers from TACO for their comments.

## References

[1] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K. Jha. 2009. GARNET: A detailed on-chip network model inside a full-system simulator. In *Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 33–42.

[2] Agisoft. 2024. Agisoft Metashape. Retrieved August 27, 2024 from https://www.agisoft.com/

[3] Timo Aila and Tero Karras. 2010. Architecture considerations for tracing incoherent rays. In *Proceedings of the Conference on High Performance Graphics*. 113–122.

[4] Apple. 2021. Apple A15 Die Shot and Annotation—IP Block Area Analysis. Retrieved August 27, 2024 from https://www.semianalysis.com/p/apple-a15-die-shot-and-annotation

[5] Dejan Azinovic, Tzu-Mao Li, Anton Kaplanyan, and Matthias Nießner. 2019. Inverse path tracing for joint material and lighting estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2447–2456.

[6] 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 2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR'22)*.

[7] Parhami Behrooz. 2000. *Computer Arithmetic: Algorithms and Hardware Designs. Oxford University Press* 19 (2000), 512583–512585.

[8] Jacco Bikker. 2012. Improving data locality for efficient in-core path tracing. *Computer Graphics Forum* 31 (2012), 1936–1947.

[9] Kevin Boos, David Chu, and Eduardo Cuervo. 2016. Flashback: Immersive virtual reality on mobile devices via rendering memoization. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. 291–304.

[10] Mark Buckler, Philip Bedoukian, Suren Jayasuriya, and Adrian Sampson. 2018. EVA$^2$: Exploiting temporal redundancy in live computer vision. In *Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE, 533–546.

[11] Anpei Chen, Zexiang Xu, Andreas Geiger, Jingyi Yu, and Hao Su. 2022. TensoRF: Tensorial radiance fields. In *Proceedings of the European Conference on Computer Vision*. 333–350.
[12] Guikun Chen and Wenguan Wang. 2024. A survey on 3D Gaussian splatting. *arXiv preprint arXiv:2401.03890* (2024).
[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 578–594.
[14] 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.
[15] Yangdong Deng, Yufei Ni, Zonghui Li, Shuai Mu, and Wenjun Zhang. 2017. Toward real-time ray tracing: A survey on hardware acceleration and microarchitecture techniques. *ACM Computing Surveys* 50, 4 (2017), 1–41.
[16] Yu Feng, Gunnar Hammonds, Yiming Gan, and Yuhao Zhu. 2022. Crescent: taming memory irregularities for accelerating deep point cloud analytics. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 962–977.
[17] Yu Feng, Shaoshan Liu, and Yuhao Zhu. 2020. Real-time spatio-temporal LiDAR point cloud compression. In *Proceedings of the 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'20)*. IEEE, 10766–10773.
[18] Yu Feng, Zihan Liu, Jingwen Leng, Minyi Guo, and Yuhao Zhu. 2024. Cicero: Addressing algorithmic and architectural bottlenecks in neural rendering by radiance warping and memory optimizations. *arXiv preprint arXiv:2404.11852* (2024).
[19] Yu Feng, Boyuan Tian, Tiancheng Xu, Paul Whatmough, and Yuhao Zhu. 2020. Mesorasi: Architecture support for point cloud analytics via delayed-aggregation. In *Proceedings of the 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'20)*. IEEE, 1037–1050.
[20] Yu Feng, Paul Whatmough, and Yuhao Zhu. 2019. ASV: Accelerated stereo vision system. In *Proceedings of the 2019 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*. 643–656.
[21] David A. Forsyth and Jean Ponce. 2002. *Computer Vision: A Modern Approach*. Prentice Hall.
[22] 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.
[23] Ioannis Gkioulekas, Shuang Zhao, Kavita Bala, Todd Zickler, and Anat Levin. 2013. Inverse volume rendering with material dictionaries. *ACM Transactions on Graphics* 32, 6 (2013), 1–13.
[24] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. 2023. The lumigraph. In *Seminal Graphics Papers: Pushing the Boundaries, Volume 2*. ACM, New York, NY, 453–464.
[25] Christiaan P. Gribble and Karthik Ramani. 2008. Coherent ray tracing via stream filtering. In *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing*. IEEE, 59–66.
[26] Boris Grot, Joel Hestness, Stephen W. Keckler, and Onur Mutlu. 2011. Kilo-NOC: A heterogeneous network-on-chip architecture for scalability and service guarantees. In *Proceedings of the 2011 38th Annual International Symposium on Computer Architecture (ISCA'11)*. IEEE, 401–412.
[27] Donghyeon Han, Junha Ryu, Sangyeob Kim, Sangjin Kim, Jongjun Park, and Hoi-Jun Yoo. 2023. 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*. Published Online, July 13, 2023.
[28] 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.
[29] Jozef Hladky, Michael Stengel, Nicholas Vining, Bernhard Kerbl, and Hans-Peter Seidel. 2022. QuadStream: A quad-Bbased scene streaming architecture for novel viewpoint reconstruction. *ACM Transactions on Graphics* 41, 6 (2022), Article 233, 13 pages.
[30] Tao Hu, Shu Liu, Yilun Chen, Tiancheng Shen, and Jiaya Jia. 2022. EfficientNeRF: Efficient neural radiance fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR'22)*. 12902–12911.
[31] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*.
[32] Arie Kaufman, Daniel Cohen, and Roni Yagel. 1993. Volume graphics. *Computer* 26, 7 (1993), 51–64.
[33] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 2023. 3D Gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics* 42, 4 (2023), 1–14.
[34] David B. Kirk and W. Hwu Wen-Mei. 2016. *Programming Massively Parallel Processors: A Hands-On Approach*. Morgan Kaufmann.

[35] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. 2017. Tanks and temples: Benchmarking large-scale scene reconstruction. *ACM Transactions on Graphics* 36, 4 (2017), Article 78, 13 pages.
[36] 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.
[37] Junseo Lee, Seokwon Lee, Jungi Lee, Junyong Park, and Jaewoong Sim. 2024. GSCore: Efficient radiance field rendering via architectural support for 3D Gaussian splatting. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Vol. 3. 497–511.
[38] Joo Chan Lee, Daniel Rho, Xiangyu Sun, Jong Hwan Ko, and Eunbyung Park. 2023. Compact 3D Gaussian representation for radiance field. *arXiv preprint arXiv:2311.13681* (2023).
[39] Yue Leng, Chi-Chun Chen, Qiuyue Sun, Jian Huang, and Yuhao Zhu. 2019. Energy-efficient video processing for virtual reality. In *Proceedings of the 46th International Symposium on Computer Architecture*. 91–103.
[40] Marc Levoy. 1988. Display of surfaces from volume data. *IEEE Computer Graphics and Applications* 8, 3 (1988), 29–37.
[41] Marc Levoy and Pat Hanrahan. 2023. Light field rendering. In *Seminal Graphics Papers: Pushing the Boundaries, Volume 2*. ACM, New York, NY, 441–452.
[42] 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.
[43] 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.
[44] Chiao Liu, Song Chen, Tsung-Hsun Tsai, Barbara De Salvo, and Jorge Gomez. 2022. Augmented reality—The next frontier of image sensors and compute systems. In *Proceedings of the 2022 IEEE International Solid-State Circuits Conference (ISSCC'22)*, Vol. 65. IEEE, 426–428.
[45] Zihan Liu, Wentao Ni, Jingwen Leng, Yu Feng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. 2024. JUNO: Optimizing high-dimensional approximate nearest neighbour search with sparsity-aware algorithm and ray-tracing core mapping. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Vol. 2. 549–565.
[46] Micron. 2014. Micron 178-Ball, Single-Channel Mobile LPDDR3 SDRAM Features. https://www.farnell.com/datasheets/3761299.pdf
[47] Micron. 2022. Micron System Power Calculators. Retrieved August 27, 2024 from https://www.micron.com/support/tools-and-utilities/power-calc
[48] 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. *Communications of the ACM* 65, 1 (2021), 99–106.
[49] 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.
[50] Joerg H. Mueller, Philip Voglreiter, Mark Dokter, Thomas Neff, Mina Makar, Markus Steinberger, and Dieter Schmalstieg. 2018. Shading atlas streaming. *ACM Transactions on Graphics* 37, 6 (2018), 1–16.
[51] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. 2022. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Transactions on Graphics* 41, 4 (2022), 1–15.
[52] Yuval Noimark and Daniel Cohen-Or. 2003. Streaming scenes to MPEG-4 video-enabled devices. *IEEE Computer Graphics and Applications* 23, 1 (2003), 58–64.
[53] NVIDIA. 2018. NVIDIA Reveals Xavier SOC Details. Retrieved August 27, 2024 from https://www.forbes.com/sites/moorinsights/2018/08/24/nvidia-reveals-xavier-soc-details/amp/
[54] NVIDIA. 2018. NVIDIA's Xavier System-on-Chip, HotChips 30. Retrieved August 27, 2024 from https://fuse.wikichip.org/news/1618/hot-chips-30-nvidia-xavier-soc/
[55] Jan Olszewski. 2023. HashCC: Lightweight method to improve the quality of the camera-less NeRF scene generation. *arXiv preprint arXiv:2305.04296* (2023).
[56] Jacopo Pantaleoni and David Luebke. 2010. HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics*. 87–95.
[57] Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2023. *Physically Based Rendering: From Theory to Implementation*. MIT Press.
[58] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. 1997. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*. 101–108.
[59] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A reconfigurable architecture for parallel patterns. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 389–402.

[60] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* 48, 6 (2013), 519–530.

[61] Chaolin Rao, Huangjie Yu, Haochuan Wan, Jindong Zhou, Yueyang Zheng, Minye Wu, Yu Ma, Anpei Chen, Binzhe Yuan, Pingqiang Zhou, Xin Lou, and Jingyi Yu. 2022. ICARUS: A specialized architecture for neural radiance fields rendering. *ACM Transactions on Graphics* 41, 6 (2022), 1–14.

[62] Satyabrata Sarangi and Bevan Baas. 2021. DeepScaleTool: A tool for the accurate estimation of technology scaling in the deep-submicron era. In *Proceedings of the 2021 IEEE International Symposium on Circuits and Systems (ISCAS'21)*. IEEE, 1–5.

[63] Peter Shirley, Michael Ashikhmin, and Steve Marschner. 2009. *Fundamentals of Computer Graphics*. AK Peters/CRC Press.

[64] Konstantin Shkurko, Tim Grant, Daniel Kopta, Ian Mallett, Cem Yuksel, and Erik Brunvand. 2017. Dual streaming for hardware-accelerated ray tracing. In *Proceedings of High Performance Graphics*. 1–11.

[65] Heung-Yeung Shum, Shing-Chow Chan, and Sing Bing Kang. 2008. *Image-Based Rendering*. Springer Science & Business Media.

[66] Zhuoran Song, Feiyang Wu, Xueyuan Liu, Jing Ke, Naifeng Jing, and Xiaoyao Liang. 2020. VR-DANN: Real-time video recognition via decoder-assisted neural network acceleration. In *Proceedings of the 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'20)*. IEEE, 698–710.

[67] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration* 58 (2017), 74–81.

[68] Cheng Sun, Min Sun, and Hwann-Tzong Chen. 2022. Direct voxel grid optimization: Super-fast convergence for radiance fields reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR'22)*. 5459–5469.

[69] Richard Szeliski. 2022. Image-based rendering. In *Computer Vision: Algorithms and Applications* (2nd ed.). Springer Nature, 681–722.

[70] Eyal Teler and Dani Lischinski. 2001. Streaming of complex 3D scenes for remote walkthroughs. *Computer Graphics Forum* 20 (2001), 17–25.

[71] Michael E. Tipping. 2001. Sparse Bayesian learning and the relevance vector machine. *Journal of Machine Learning Research* 1 (June 2001), 211–244.

[72] Hanrui Wang, Zhekai Zhang, and Song Han. 2021. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA'21)*. IEEE, 97–110.

[73] Qianqian Wang, Zhicheng Wang, Kyle Genova, Pratul P. Srinivasan, Howard Zhou, Jonathan T. Barron, Ricardo Martin-Brualla, Noah Snavely, and Thomas Funkhouser. 2021. IBRNet: Learning multi-view image-based rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR'21)*. 4690–4699.

[74] Tong Wu, Yu-Jie Yuan, Ling-Xiao Zhang, Jie Yang, Yan-Pei Cao, Ling-Qi Yan, and Lin Gao. 2024. Recent advances in 3D Gaussian splatting. *arXiv preprint arXiv:2403.11134* (2024).

[75] 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.

[76] Size Zheng, Siyuan Chen, Siyuan Gao, Liancheng Jia, Guangyu Sun, Runsheng Wang, and Yun Liang. 2023. TileFlow: A framework for modeling fusion dataflow via tree-based analysis. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1271–1288.

[77] Size Zheng, Siyuan Chen, Peidi Song, Renze Chen, Xiuhong Li, Shengen Yan, Dahua Lin, Jingwen Leng, and Yun Liang. 2023. Chimera: An analytical optimizing framework for effective compute-intensive operators fusion. In *Proceedings of the 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA'23)*. IEEE, 1113–1126.

[78] Yangjie Zhou, Mengtian Yang, Cong Guo, Jingwen Leng, Yun Liang, Quan Chen, Minyi Guo, and Yuhao Zhu. 2021. Characterizing and demystifying the implicit convolution algorithm on commercial matrix-multiplication accelerators. In *Proceedings of the 2021 IEEE International Symposium on Workload Characterization (IISWC'21)*. IEEE, 214–225.

[79] Yuhao Zhu, Anand Samajdar, Matthew Mattina, and Paul Whatmough. 2018. Euphrates: Algorithm-SoC co-design for low-power mobile continuous vision. *arXiv preprint arXiv:1803.11232* (2018).

Received 20 May 2024; revised 10 July 2024; accepted 7 August 2024