ICARUS: A Lightweight Neural Plenoptic Rendering Architecture

CHAOLIN RAO, ShanghaiTech University, China
HUANGJIE YU, ShanghaiTech University, China
HAOCHUAN WAN, ShanghaiTech University, China
JINDONG ZHOU, ShanghaiTech University, China
YUEYANG ZHENG, ShanghaiTech University, China
YU MA, ShanghaiTech University, China
ANPEI CHEN, ShanghaiTech University, China
MINYE WU, KU Leuven, Belgium
BINZHE YUAN, ShanghaiTech University, China
PINGQIANG ZHOU, ShanghaiTech University, China
XIN LOU, ShanghaiTech University, China
JINGYI YU, ShanghaiTech University, China

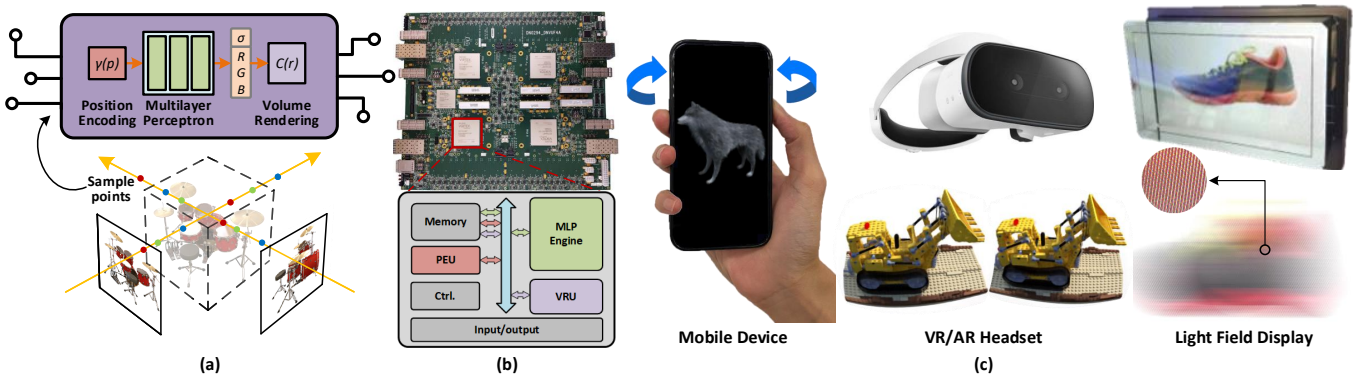


Fig. 1. We present ICARUS, a novel lightweight graphics architecture tailored for NeRF rendering with high energy efficiency. (a) Our hardware design involves operations in NeRF rendering. (b) We implement our architecture design on a FPGA platform. (c) Various applications can benefit from our architecture.

The practical deployment of Neural Radiance Field (NeRF) in rendering applications faces several challenges, with the most critical one being low rendering speed on even high-end graphic processing units (GPUs). In this paper, we present ICARUS, a novel lightweight graphics architecture tailored for NeRF rendering. Unlike GPUs using general purpose computing and memory architectures for NeRF, ICARUS executes the complete NeRF pipeline using dedicated plenoptic cores (PLCore) consisting of a positional encoding unit (PEU), a multi-layer perceptron (MLP) engine, and a volume rendering unit (VRU). A PLCore takes in positions & directions and renders the corresponding pixel colors without any intermediate data going off-chip for temporary storage and exchange, which can be time and power consuming. To implement the most expensive component of NeRF, i.e., the MLP, we transform the fully connected operations to approximated reconfigurable multiple constant multiplications (MCMs), where common subexpressions are shared across different multiplications to improve the computation efficiency. We build a prototype ICARUS using Synopsys HAPS-80 S104, an FPGA-based prototyping system for large-scale integrated circuits and systems. We evaluate the area and power consumption of a PLCore using 40nm LP CMOS process. Working at 300 MHz, a single PLCore occupies 7.59 mm^2 and consumes 309.8 mW, translating to 0.174 uJ/sample. Evaluation results show that for NeRF rendering, the energy efficiency of ICARUS is 146 times higher than GPUs. By scaling to a multi-core system, the energy-efficient ICARUS can be deployed in practical edge applications for NeRF-based rendering tasks.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Additional Key Words and Phrases: Neural radiance field (NeRF), neural rendering, light weight, energy efficient

Authors' addresses: Chaolin Rao, raocl@shanghaitech.edu.cn, ShanghaiTech University, Shanghai, China; Huangjie Yu, yuhj@shanghaitech.edu.cn, ShanghaiTech University, Shanghai, China; Haochuan Wan, wanhch@shanghaitech.edu.cn, ShanghaiTech University, Shanghai, China; Jindong Zhou, zhoujd@shanghaitech.edu.cn, ShanghaiTech University, Shanghai, China; Yueyang Zheng, zhenyy1@shanghaitech.edu.cn, ShanghaiTech University, Shanghai, China; Yu Ma, mayu@shanghaitech.edu.cn, ShanghaiTech University, Shanghai, China; Anpei Chen, chenap@shanghaitech.edu.cn, ShanghaiTech University, Shanghai, China; Minye Wu, minye.wu@kuleuven.be, KU Leuven, Leuven, Belgium; Binzhe Yuan, yuanbzh@shanghaitech.edu.cn, ShanghaiTech University, Shanghai, China; Pingqiang Zhou, zhoupq@shanghaitech.edu.cn, ShanghaiTech University, Shanghai, China; Xin Lou, louxin@shanghaitech.edu.cn, ShanghaiTech University, Shanghai, China; Jingyi Yu, yujingyi@shanghaitech.edu.cn, ShanghaiTech University, Shanghai, China.

1 INTRODUCTION

In computer graphics, advances on rendering algorithms are often trailed by their adoptions by hardware, ranging from earlier z-buffer [Catmull 1974] for depth sorting [Newell et al. 1972], to

geometry shaders [Bailey 2007] for realizing various subdivision schemes [Akenine-Möller et al. 2019; Han 2007; Kazakov 2007], and to tailored microarchitectures (RDNA2 of AMD and Ampere of NVDIA) [hardware 2020; Nvidia 2020] for accelerating ray tracing [Meister et al. 2021; Sanzharov et al. 2020]. Even with today's most powering hardware and accelerating schemes, it is still fairly easy to distinguish between a synthetic image and an actual photography of the scene. Computer vision takes a different route: take photographs of a real scene and then recover its 3D geometry, color and texture, and surface reflectance. With high quality reconstruction, simply texturing the geometry produces the realism surpassing physically simulated rendering, let alone it is much faster. However, 3D reconstruction is essentially an inverse rendering problem. Ill-posed unless with strong assumptions, reconstruction is equally if not more challenging than rendering itself.

For decades researchers from both fields have sought a "mid-ground" solution, i.e., image-based modeling and rendering or IBMR: starting from a set of source images (photographs) stored as a ray database, one can render new viewpoints by querying the database via tailored signal processing techniques. Once viewed as a prodigy, IBMR had only achieved limited successes because the plenoptic function [Adelson et al. 1991] for a static object is 5D for which even moderate sampling would yield to a ray database prohibitively large. Reducing the size of ray samples while preserving visual realism either require highly accurate proxy geometry [Buehler et al. 2001; Gortler et al. 1996] or expensive interpolation schemes [Isaksen et al. 2000] to account for occlusions and view-dependences.

The recent advent of neural plenoptic functions has ignited renewed interest on applying IBMR to real-time photo-realistic rendering [Kellnhofer et al. 2021]. The seminal work of Neural Radiance Field (NeRF) [Mildenhall et al. 2020] implicitly models the radiance field and the density of a volume with multi-layer perceptrons (MLPs) and employs a direct volume rendering function for photo-realistic view synthesis and potentially 3D reconstruction. The NeRF breakthroughs are multi-fold. On storage, deep neural net serves as a de facto compression scheme: the standard NeRF uses around 1,200,000 parameters of a total size 4.6MB. On interpolation, NeRF provides an inherent continuous representation of the plenoptic function and hence is capable of non-linear interpolation largely absence in prior art.

For practical deployment to rendering applications, NeRF still faces two challenges. On training, NeRF requires expensive per-scene optimization that takes hours to converge even on high-end GPUs. Various acceleration schemes have been developed over the past two years based on clever data structures or training strategies. Plenoxel [Yu et al. 2021a] discretizes the space into voxels with a fixed resolution and employs spherical harmonics to store attributes such as densities and colors within the voxels, reducing the training time from hours to minutes. Instant-ngp[Müller et al. 2022] leverages multi-resolution hashing to efficiently encode a scene into light-weight networks, reduce training time by another order of magnitude to seconds. Neural representations seem poised to serve an alternative to 3D geometry for both graphics and visions.

On rendering, real-time performances still relies on high-end GPUs. For example, on NVDIA RTX 3090, the original NeRF renders at 0.05 frames per second (FPS) of 800×800 whereas Instant-ngp at 260 FPS with the same resolution. In contrast, latest virtual reality (VR) and augmented reality (AR) terminals, in particularly Head-Mounted Displays (HMDs), are edging towards lightweight: small in size, light in weight, low in power, and most notably, untethered. To use NeRF as the rendering engine on these terminals requires equally lightweight graphics processors with matching specs in size and power. Yet existing high-end GPUs are too bulky and power-hungry whereas Neural Processing Units (NPUs) are specifically designed for Convolutional Neural Networks (CNNs) with the focus of optimizing data (weight and activation) reusing rather than MLP-type fully connected operations.

In this paper, we present ICARUS, a novel lightweight graphics architecture tailored for NeRF rendering. Recall the GPU implementation of NeRF uses general purpose Single Instruction Multiple Data (SIMD) cores and Special Function Units (SFUs) for the computation of MLP and other steps of NeRF pipeline, along with additional core components and off-chip memory for storing and exchanging intermediate data. In contrast, ICARUS uses dedicated plenoptic cores (PLCore) as key computation components to execute the complete NeRF pipeline without off-chip data exchanges and temporary storage. Each PLCore consists of a positional encoding unit (PEU), an MLP engine, and a volume rendering unit (VRU). It takes in positions & directions and outputs the corresponding pixel colors with all computation and storage on-chip. Therefore, NeRF rendering of an image using ICARUS is parallelized with little overhead of control and off-chip memory access, which is time and power consuming. To implement the most expensive component of NeRF, i.e., the MLP, we transform the fully connected operations to approximated reconfigurable multiple constant multiplications (RMCMs). This further leads to about 1/3 hardware complexity reduction compared to conventional multiply–accumulate (MAC)-based approaches.

We demonstrate a prototype ICARUS using Synopsys HAPS-80 S104, an field programmable gate array (FPGA)-based prototyping system for large-scale integrated circuits and systems. We evaluate the area and power performance of a PLCore using 40nm LP complementary metal–oxide–semiconductor (CMOS) process. Working at 300 MHz, a single PLCore occupies 7.59 mm^2 and consumes 309.8 mW, translating to 0.174 uJ/sample. For NeRF rendering, the energy efficiency of ICARUS is 146 times higher than GPUs. We further develop a tailored instruction set for ICARUS so that it can easily scale up to form a multi-core system as well as support NeRF rendering on a variety of displays, e.g., monocular rendering on mobile devices, stereoscopy on HMDs, and multi-/many view rendering on autostereoscopic light field displays (Fig. 1). When scaled up to multi-core systems, ICARUS can achieve real-time NeRF rendering, enabling immersive viewing experiences by many participants with untethered HMDs. Finally, ICARUS also supports implicit signed distance function (SDF) evaluation, potentially useful for geometry extraction and isosurface polygonisation for subsequent geometry-oriented graphics tasks.

2 RELATED WORK

MLP-based Neural Modeling and Rendering. MLP is widely used as a key component in advanced neural modeling and rendering tasks. An early MLP-based render [Ren et al. 2015] attempts

to model light transport as a non-linear function of light source position and pixel coordinates, and the MLP allows input images to be captured under arbitrary illumination conditions, including freely moving light sources by hand. Similar to the surface light field (SLF) modeling [Wood et al. 2000], the follow-up work, Deep Surface Light Fields [Chen et al. 2018], uses an MLP network to model the radiance of surface points and learns to fill up the missing data across angles and vertices, achieving smooth view synthesis and high compression rate.

The recent of NeRF has ignited renewed interest in this direction. The core idea is to use MLPs to continuously represent scenes and form an end-to-end learning framework for scene modeling and novel view synthesis. In particular, NeRF [Mildenhall et al. 2020] implicitly models the radiance field and the density of a volume with an MLP, then uses a direct volume rendering function to synthesize novel views. It also demonstrates a hitherto unprecedented level of fidelity on a range of challenging scenes. Recently, researchers also propose various variants to adopt NeRF to dynamic scenes [Park et al. 2020, 2021; Pumarola et al. 2021], relightable rendering [Boss et al. 2021; Srinivasan et al. 2021; Zhang et al. 2021], generalized scenarios [Gu et al. 2021; Wang et al. 2021; Yu et al. 2021c], and human performer rendering [Liu et al. 2021; Peng et al. 2021], which dramatically expands the scope of MLP use.

Algorithm-level Techniques for Rendering Acceleration

Though NeRF-based methods demonstrate state-of-the-art results, they are too computationally demanding to be deployed in practical rendering applications. Recently, several algorithm-level techniques are proposed to accelerate NeRF rendering. For example, [Luo et al. 2021] leverages coarse explicit scene geometry priors to guide point sampling along with rays and hence reduce network inference computation. There is a line of methods [Garbin et al. 2021; Müller et al. 2022; Yu et al. 2021a,b] which use a hybrid representation where the space is partitioned into voxels with well-designed data structures to reduce the computational complexity of sampling fetching. Specifically, [Liu et al. 2020] uses octree in its hybrid representation to avoid redundant MLP queries in free space. [Garbin et al. 2021] compactly caches a deep radiance map from trained MLPs in voxels, preventing network inference. Similarly, [Yu et al. 2021a] and [Yu et al. 2021b] realize view-dependent effects by adopting spherical harmonics and store coefficients on a sparse voxel grid or an octree structure respectively. Although these methods significantly speed up the rendering, such volumetric representations require extensive storage. Instant-ngp [Müller et al. 2022] assembles multi-resolution scene coding for sample points from hashing tables on the fly. The coding already contains a lot of information so that Instant-ngp can utilize lightweight MLPs to decode. KiloNeRF [Reiser et al. 2021] uses thousands of tiny MLPs to represent parts of the scene so as to reduce the cost of network queries for sample points. Though efficient, these methods still use MLPs that require significant amount of computation during inference and rely on a powerful GPUs to achieve the desired rendering experience.

It should be pointed out that our proposed architecture can be considered as an accelerator for coordinate-based MLPs. It can also benefit from the aforementioned acceleration algorithms.

3 DESIGN PHILOSOPHY

3.1 Computation Analysis for NeRF-based Rendering

NeRF implicitly models a continuous plenoptic function $F_{NeRF} : (\boldsymbol{p}, \boldsymbol{d}) \rightarrow (\boldsymbol{c}, \sigma)$ with an MLP, which is trained using a set of images with known poses. By querying the MLP at a position $\boldsymbol{p} \in \mathbb{R}^3$ from a specific viewing direction $\boldsymbol{d} \in \mathbb{R}^2$, the corresponding density $\boldsymbol{\sigma}$ and view-dependent color $\boldsymbol{c}$ at that position can be recovered. To render a pixel, a number of 3D positions, denoted by $(\boldsymbol{p}_1, \boldsymbol{p}_2, ..., \boldsymbol{p}_N)$, are sampled along the ray that passes through the pixel. The plenoptic function F_{NeRF} has to be evaluated for each sample point $\boldsymbol{p}_i$ to generate the corresponding color $\boldsymbol{c}_i$ and density σ_i. Direct volume rendering [Kajiya and Von 1984] is then applied to generate the color of the pixel.

As we can calculate, to render an image with millions of pixels, the plenoptic function F_{NeRF} is evaluated N times for each pixel, translating to tens (or even hundreds) of millions of MLP queries depending on the number of N. For illustration, the original NeRF samples 192 points along each ray. To render an 800×800 image, the MLP has to be queried $800 \times 800 \times 192 = 122880000$ times. This is the reason why it takes seconds to render a single frame even on modern high-end GPUs.

The evaluations of MLP for different sample points are independent of one another. This means that they can be paralleled to improve the rendering speed, given sufficient computing resources. Parallel processing can be performed on the sample point-level or pixel (ray)-level. In addition, though not very critical in terms of computation, there are another two important steps in the NeRF-based pipeline: positional encoding which transforms positions & directions into a higher dimensional feature space and volume rendering which integrates the colors of sample points to render the final pixel color. Due to the dimension increment, the amount of data for positions and directions is expended by 20 times and 8 times (original NeRF), respectively, after positional encoding. On the contrary, volume rendering compresses the amount of data by integrating the colors and densities of all sample points to generate the RGB values of the rendering pixels.

Apart from the tremendous number of network queries, another important characteristic of NeRF is the use of MLPs instead of CNNs. For CNNs, a kernel is usually convolved with different parts of input feature maps and a specific part of a feature map is convolved with different kernels. This makes the sharing of weights and/or activations an important approach to improve the performance as well as reduce power consumption of CNN accelerators, since memory access, especially off-chip memory access, contributes the majority of delay and power consumption in these systems. While for MLPs, since they are fully connected, a weight is multiplied with only one input neuron and a neuron is multiplied with only a set of weights. Therefore, the weight and activation sharing optimization techniques for CNNs are not directly applicable to MLPs.

3.2 Mapping NeRF to GPUs and NPUs

For the computation of neural networks, there are two commonly used architectures: GPU and NPU. GPU, or more precisely general purpose GPU (GPGPU), is a device optimized for general purpose computing, especially for neural networks nowadays. NPU is a

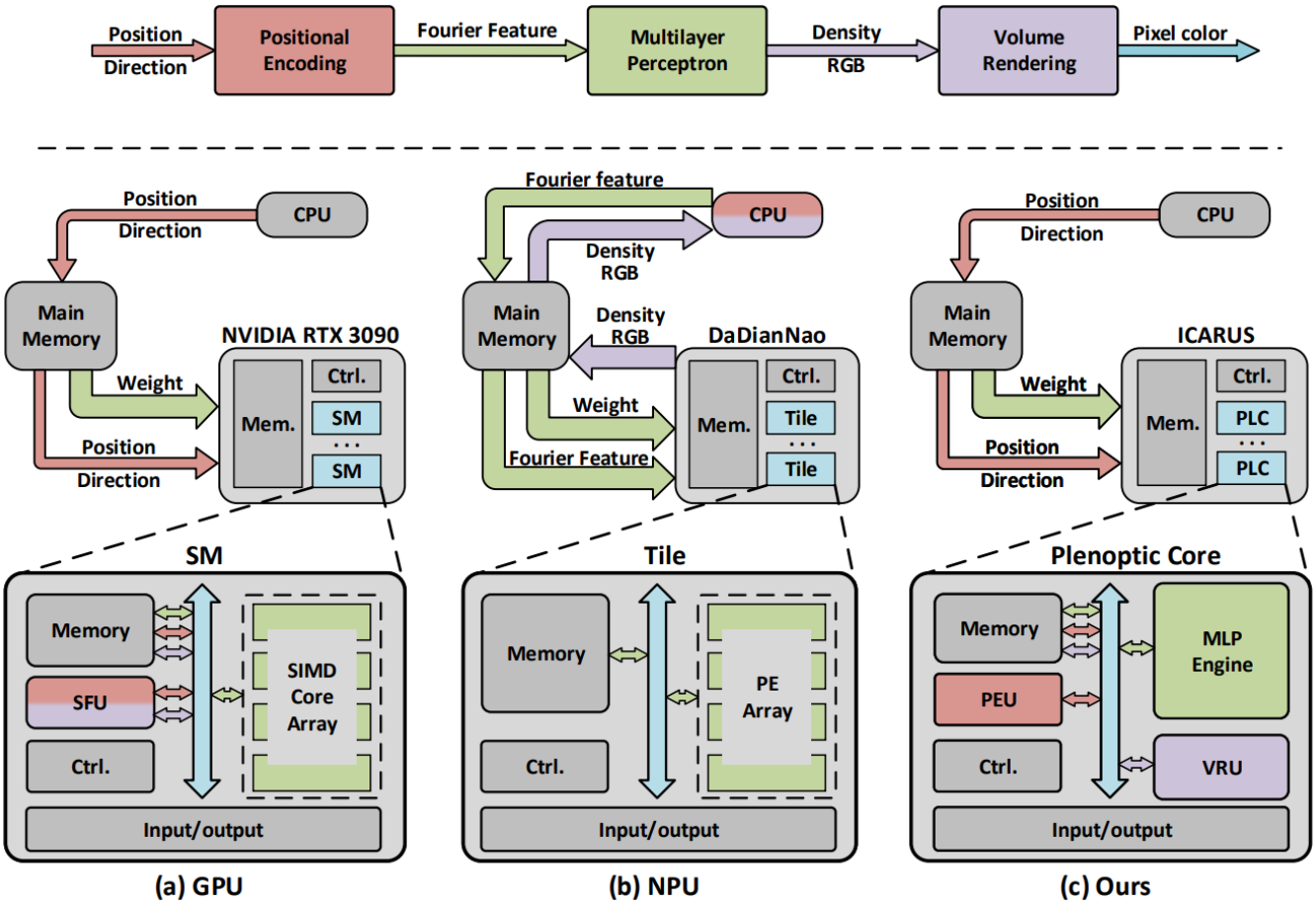


Fig. 2. Illustration of mapping NeRF to (a) GPU (b) NPU and (c) ICARUS. For GPU-based NeRF implementation, positional encoding and volume rendering are executed in the SFU, and MLP is mapped to SIMD core arrays. Intermediate data is exchanged using off-chip memory. For NPU-based NeRF implementation, positional encoding and volume rendering are executed in CPU and MLP is mapped to PE arrays. Intermediate data is also exchanged using off-chip memory. ICARUS executes the complete NeRF pipeline using dedicated plenoptic cores (PLCore) consisting of a PEU, an MLP engine, and a VRU.

type of a dedicated hardware architecture designed for neural networks, with each specific design has its own strategy of optimization, especially on data sharing [Chen et al. 2016]. Fig.2 illustrates the overview of mapping NeRF to GPGPU, NPU and the proposed ICARUS architecture.

As we can see, for GPGPU, the main computation components are streaming multiprocessors (SMs). Each SM consists of a SFU, a SIMD core array and some local memories. The SFU is a multi-core unit designed for the execution of transcendental functions such as sine, cosine and square root. The SIMD core array is responsible for the execution of massive parallel computing tasks like convolution in CNNs. As illustrated in Fig. 2(a), when running NeRF on GPGPU, positional encoding and volume rendering are executed in the SFU and MLP is mapped to the SIMD core array. Intermediate data is temporary stored and exchanged using off-chip memory, which is critical for delay and power. Moreover, though modern GPUs support certain formats of fixed-point computation, i.e., int8 and int4, floating-point is used for NeRF rendering because the above mentioned fixed-point formats are not flexible enough for NeRF.

Fig. 2(b) illustrates the mapping of NeRF to NPU architectures. As we can see, an NPU core mainly consists of a processing element (PE) array, which is optimized for convolution operations in CNNs. Moreover, memory access, especially off-chip memory access optimization based on the computation characteristics of CNNs, is one of the most crucial design considerations for NPUs. When mapping NeRF to a NPU-based system, since there is no computing unit like SFU in GPU, positional encoding and volume rendering have to be implemented using CPU. Therefore, intemediate data has to be exchanged through the main memory. Moreover, as we have mentioned, data sharing optimizations for CNNs are not directly applicable to NeRF, because the network structure is changed to MLP.

In addition to the problems mentioned above, another limitation of GPUs and NPUs for NeRF is the lact of computation capability.

A typical high-end GPU conducts tens of Tera floating-point operations (FLOPs) per second, which is not sufficient for NeRF-based rendering. Moreover, though GPUs are easy to use for neural networks, they are not energy-efficient. This limitation makes GPU unsuitable for mobile applications. NPUs, on the other hand, are usually designed for energy efficiency. But the computation capabilities of NPUs are far less than the need of NeRF-based rendering. Moreover, the energy efficiency of NPUs will be lower when running NeRF, since they are not optimized for MLPs.

3.3 Design Decisions

In this section, we present the design decisions for ICARUS. The first and most important decision is to finish the entire NeRF pipeline in a single PLCore without any intermediate data going off-chip for temporary storage and exchange. This is because as we have mentioned, off-chip memory access can be time and power consuming, and has become the bottleneck for many computation systems. To achieve this, we need to design a dedicated architecture which takes in positions & directions and outputs final pixel colors. Moreover, though NeRF-based rendering algorithms share the same general flow consisting of positional encoding, MLP and volume rendering, they may differ in specific implementation of each module. Therefore, the dedicated architecture needs to have the flexibility to support different implementations of these modules.

Another decision is to optimize the implementation of MLP for computational complexity reduction. Therefore, we use fixed-point instead of floating-point in ICARUS. It is well-known that floating-point computations are generally more complex than their fixed-point counterparts. Since NeRF is computationally demanding, it is very hard, if not impossible, to meet the performance and energy-efficiency requirement for mobile applications using floating-point computation. Apart from fixed-point computation, the fault-tolerant property of neural networks can be utilized to further reduce the computational complexity of NeRF.

The last design decision for ICARUS is to use a multi-core system to meet the performance requirement, meaning that parallel computing strategy must be carefully designed. We have mentioned that the rendering of sample points are independent of one another without any data dependency. This means that parallel computing is a practical and effective approach to improve the performance of the rendering system.

4 SYSTEM DESIGN

4.1 Overall System Architecture

Fig. 3 illustrates the overall architecture of ICARUS. For a rendering system consisting of CPU, dynamic random access memory (DRAM) and ICARUS, the inputs, i.e., network model, encoding frequency and positions & directions are stored in DRAM and fed to ICARUS under the control of CPU during runtime. An on-chip network is used to dispatch input data to the destination PLCores. After loading the network model, positions & directions are continuously streamed into ICARUS for NeRF processing and final rendered pixel colors are streamed out through the data path. Inside ICARUS, the sample points of a cluster of rays, comprising a sample point batch, are processed by the same PLcore, where the whole NeRF pipeline is completed inside the PLcore without any intermediate data going off-chip for temporary storage and exchange.

The detailed structure of a PLCore is illustrated in the right half of Fig. 3. According to the NeRF pipeline, we design a PLCore consisting of a PEU, an MLP engine and a VRU, where specific rendering steps are executed by the corresponding dedicated hardware module. The PEU transforms the input positions & directions into higher dimensional feature space based on the input encoding frequencies. The encoded positions & directions are fed to the MLP engine to render the colors $\boldsymbol{c} = (r, g, b)$ and densities σ of the sample points, which are further consumed by the VRU to generate the final pixel colors. On-chip static random access memory (SRAM) blocks are used for temporary storage of weights, input positions & directions and intermediate activations during the runtime. The execution of the NeRF pipeline on a PLCore is controlled by the internal finite state machine based on the input processing instructions.

As we have mentioned, due to the volumetric scene representation of NeRF, each sample point in a ray has to go through the pipeline while the processings are independent of one another. Accordingly, each PLCore in ICARUS is designed to independently handle the NeRF processing pipeline of mapping positions & directions to pixel colors. Therefore, NeRF rendering of an image using ICARUS is parallelized with little overhead of control and off-chip memory exchange, saving tremendous amount of processing time and power. As illustrated in the left part of Fig. 3, the information of different clusters of rays are fed to different PLCores, and pixel colors for different locations of an image are generated in parallel. With such simple control, ICARUS can be easily scaled up to a multi-core system for high-performance NeRF rendering.

4.2 Positional Encoding Unit

Previous researches have revealed that vanilla feed-forward neural networks tend to learn low frequency signals [Mildenhall et al. 2020; Rahaman et al. 2019; Zhong et al. 2019]. In neural rendering tasks, a vanilla neural network takes in spatial and angular coordinates, and outputs attributes such as density, color, signed distance, etc. However, such vanilla networks often produce over-smoothed results, leading to over-blurred rendered images. One technique to overcome such difficulty is to map raw input coordinates into a relatively higher dimensional feature space through Fourier features mapping as

$$\phi(x; A) = [\cos A^T x, \sin A^T x]. \tag{1}$$

This procedure is called positional encoding in NeRF-based neural rendering. Afterward, a neural network takes in transformed features instead of raw coordinate vectors.

It is easily perceived that the selection of the frequency matrix A is critical for tuning the smoothness of network outputs. Shown in Fig. 4(a) are different features (frequency matrices) used in different neural rendering algorithms, e.g., fixed frequency for NeRF rendering, isotropic random Fourier features for encoding implicit geometries, and anisotropic random Fourier features for neural image-based rendering of implicit geometries.

In ICARUS, we develop a universal PEU for computing different kinds of feature mapping functions. Fig. 4(b) illustrates the architecture of the PEU in a PLCore. The frequency matrix A, which

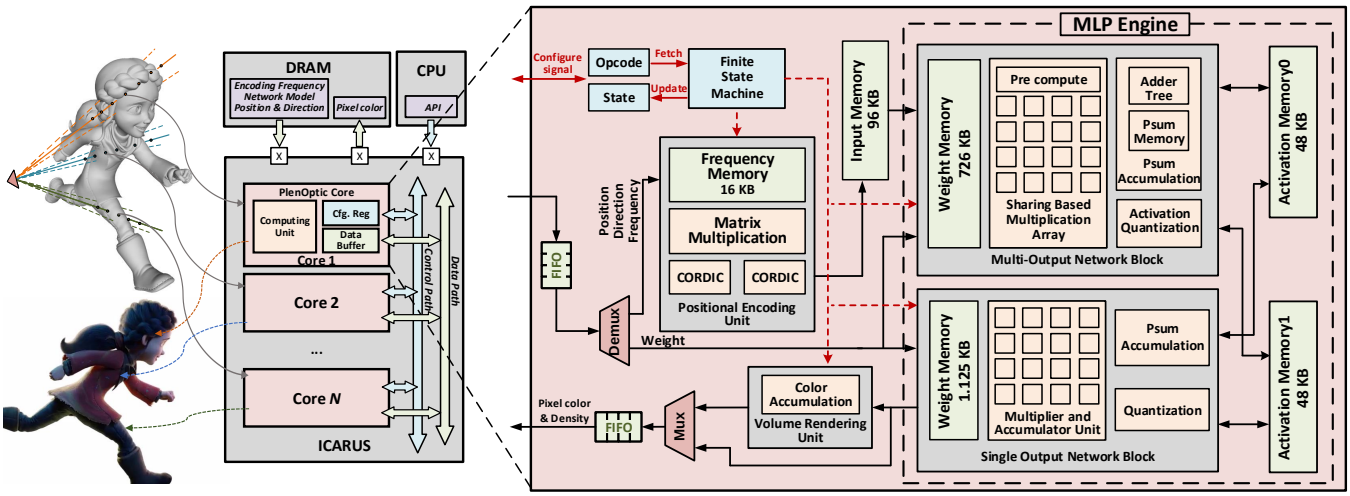


Fig. 3. Overall architecture of the proposed ICARUS. The main computation components in ICARUS is PLCore array. For NeRF rendering, a batch of sample points are processed by the same PLcore, where the whole NeRF pipeline for a ray is completed inside the PLcore, i.e., a PLCore takes in positions & directions and renders the corresponding pixel colors without any intermediate data going off-chip for temporary storage and exchange.

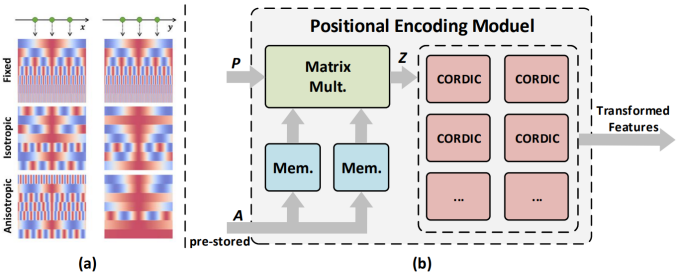


Fig. 4. (a) Three different types of frequency patterns. (b) Overall structure of the PEU. (A: Frequency matrix. P: Input positional data matrix. Z: Results of the matrix multiplication of AP)

is the same for all sample points in a particular rendering task, is stored in local memories. During the runtime, positions & directions are streamed into the matrix multiplication unit, in which they are multiplied with the frequency matrix A in a pipelined manner. The resultant intermediate vector z is further fed to the Coordinate Rotation Digital Computer (CORDIC) array to generate the final transformed features, i.e., $sin(z)$ and $cos(z)$.

We design PEU to meet the requirement of inner product operations with two different sizes, i.e., ($\mathbb{R}^3$ and $\mathbb{R}^6$), for different positional encoding algorithms. For the case of $\mathbb{R}^3$, both frequency matrix data and position & direction data are of 3-dimensional, meaning that one block memory of size 3×128 is enough to store the frequency matrix. For the inner product computation, a 3-stage cascaded MAC unit is used to perform three multiplications and two additions in serial.

For the case of $\mathbb{R}^6$, instead of doubling the size of memory to 6×128 to maintain the same performance, we use two separate 3×128 memory blocks. Since two memory blocks can be accessed independently, the normal $\mathbb{R}^3$ frequency matrix or the first half of the $\mathbb{R}^6$ frequency matrix is loaded into the first memory block, and the other memory block stores only the second half of the $\mathbb{R}^6$ frequency matrix. In $\mathbb{R}^6$ computation mode, two memory blocks work simultaneously. While in $\mathbb{R}^3$ computation mode, the second memory is set to a sleep mode for power reduction. Moreover, the inner product results go through the whole 6 stages of the cascade MAC during $\mathbb{R}^6$ computation, while in $\mathbb{R}^3$ calculation, the last 3 stages are bypassed.

Note that for positional encoding algorithms using fixed frequency, i.e., the one in NeRF, the value of the input series fed to CORDIC of one sample point are doubled one after another. Thus, several CORDIC modules can be replaced by the more efficient 'double-angle' computations to further reduce the computational complexity. Also, the trick that the two's compliment representation can exactly match the period of trigonometric functions can eliminate certain amount of operations in CORDIC implementation. Therefore, in our design, the PEUs can be configured to work in a specific NeRF mode or general positional encoding mode.

4.3 MLP Engine

In the NeRF pipeline, MLP is the most computation intensive component because tens of millions of MLP query leads to tremendous amount of MAC operations. Therefore, we design a dedicated MLP engine to accelerate the computation of MLP. A typical MLP layer can be expressed as

$$\boldsymbol{y} = f(W\boldsymbol{x} + \boldsymbol{b}) \tag{2}$$

where $\boldsymbol{x}$, W, and $\boldsymbol{b}$ are the input neuron vector, weight matrix and bias vector, respectively, and f denotes the activation function. As an example, W is a 256×256 matrix in the original NeRF.

The overall architecture of the MLP engine is illustrated in Fig.3, which consists of a multi-output network block (MONB), a single output network block (SONB) and two activation memory blocks. The MONB block is responsible for the computation of hidden layers

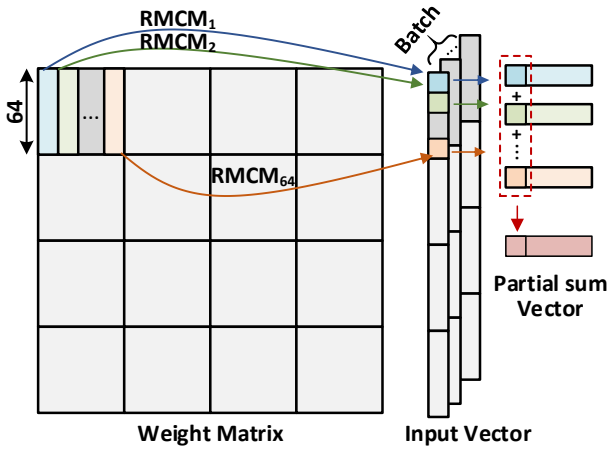


Fig. 5. Overall computation flow in the MLP engine. A large matrix-vector multiplication (MVM) for a entire layer is partitioned into several sub-MVM blocks with sizes of 64 × 64, where each MVM is implemented using 64 reconfigurable multiple constant multiplication (RMCM) blocks. The weight matrix stays stationary for a certain amount cycles and multiply with a batch of input vectors for different sample points.

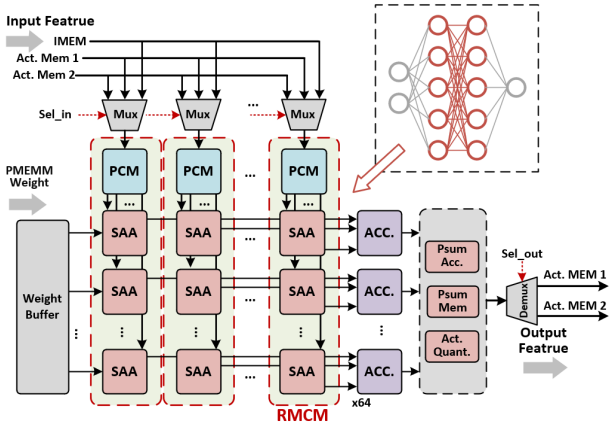


Fig. 6. Computation of MVM using a multiple output network block (MONB).

in an MLP where multiple output neurons are generated, while the SONB block is designed to compute the output layer of MLPs in NeRF, which generates the color of samples points. We design a dedicated SONB block because when mapping the output layer to MONB, most of the computation resources are wasted. This is because only a single output is generated by the output layer, it cannot be transformed to MCMs. We use MAC array to implement the fully connected operations in SONB.

Fig. 5 illustrates the overall computation flow in an MLP engine. Instead of using a large matrix-vector multiplication (MVM) block for a entire layer, we partition it into several sub-MVM blocks with sizes of 64 × 64, making it more flexible for various NeRF-based rendering tasks with different matrix size (usually multiples of 64). For the implementation of a sub-MVM, taking the top-left block as an example, we transform it into 64 shift-add based RMCM operations, where adder trees are further used to compress 64 partial results into the final resultant vector. The advantages of using RMCM are multi-fold. It is faster and more power-efficient than conventional MAC-based implementation since the common subexpressions are shared across different multipliers [Park et al. 2004]. Moreover, we further adopt the idea of batch-computing to reduce off-chip memory access for weights. As illustrated in Fig. 5, after loaded into the sub-MVM block, the weight matrix stays stationary for a certain amount cycles and multiply with a set of input vectors for different sample points. Batch-computing reduces off-chip memory access for weight matrix at the cost of a small extra on-chip memory for intermediate output vectors. In our design, a batch size of 128 sample points are used, i.e., the weight matrix stays stationary in the MLP engine for the computation of sub-MVMs of 128 sample points.

The computation of MVM in Fig. 5 using a MONB block is illustrated in Fig.6. In our design, a MONB block consists of 64 parallel RMCM blocks, where each RMCM block performs the RMCM operation shown in Fig. 5 with 64 coefficients. The multiplication results of RMCM blocks at the same vertical positions, i.e., in the same row, are summed up by the corresponding adder tree. The summation results from the adder trees are further accumulated in the following block to generate the final MVM results. In particular, three buses load inputs come from the Input Memory, Activation Memory 1 and Activation Memory 2, where each data bus transfers 64 parallel activations. In each cycle, only one of the three data bus is activated by the multiplexers under the control of the Sel_in signal. The selected data is fed to 64 pre-compute modules (PCMs), where each PCM generates 4 common subexpressions ($1x$, $3x$, $5x$ and $7x$) that shared by the following 64 select & shift-add (SSA) blocks in the same column. Each SSA calculates the approximated product between input an activation and a weight by proper shift-add operations. Along the horizontal direction, the SSA in the same row computes 64 products of the same inner product operation, which are summed up by the adder tree to generate one inner product for the MVM. Once finished, the results of MVM are sent to the activation & quantication (Act. Quant.) module for bias accumulation, activation operation and re-quantization operation. A demultiplexer (Demux) finally chooses the correct activation memory (Activation Memory 1 or Activation Memory 2) to save the output neurons.

Fig. 7(a) illustrates the working principle of the RMCM for MVM computation, which is inspired by the work in [Park et al. 2004]. The basic idea is to pre-compute a set of partial products, usually referred to as common subexpressions, and share them across different multipliers. Since common subexpressions are fixed, they can be implemented using only shift-add operations. For example, $3x$ can be realized using only one shift-add operation as

$$3x = 1x << 1 + 1x \tag{3}$$

where "<<" represents a left-shift operation. In the select & shift-add (SAA) block in Fig. 7(a), multiplexers are used to select the needed common subexpressions according to the value of input weights. The selected common subexpressions are then shifted and summed

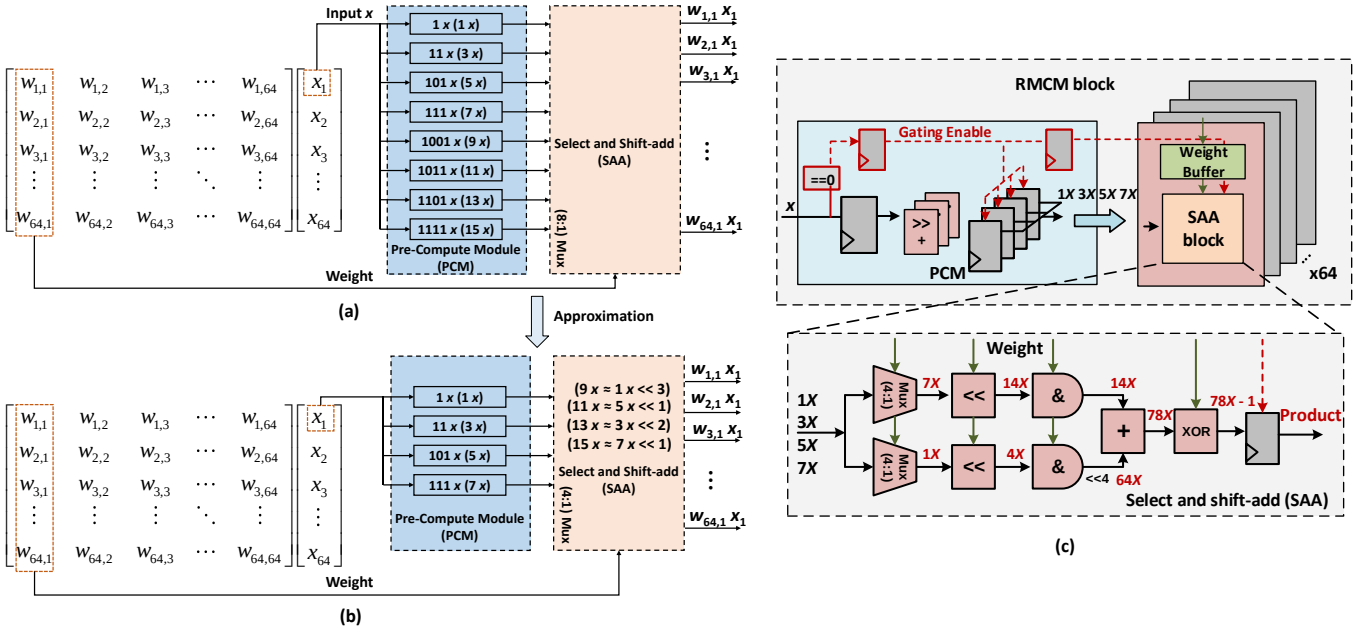


Fig. 7. Working principle of (a) RMCM and (b) approximated RMCM. (c)Detailed implementation of the RMCM block. For approximated RMCM, the second half of common subexpressions are approximated using their nearest neighbors.

up to generate the final multiplication results. Let us take a 9-bit weight $-78(1\_0100\_1110)_2$ represented in the signed-magnitude form as an example. We first divide the weight into two parts, i.e., higher part 0100 and lower part 1110. Two multiplexers in the SAA block choose $1x$ and $7x$ for the higher part and lower part, respectively. Two shifters are then used to shift $1x$ and $7x$ to $0100x(4x)$ and $1110x(14x)$, respectively. Then we shift the result of higher part to the left by 4 bits to get $0100\_0000x(64x)$. An adder finally sums up the results of higher part and lower part to produce the final result $78x$.

Hardware complexity is reduced by RMCM since the pre-compute block can be shared by all the 64 multipliers. Moreover, since neural networks are naturally fault-tolerant, we further adopt the idea of approximation to the RMCM block for further hardware complexity reduction. As illustrated in Fig. 7(b), the second half of the pre-compute common subexpressions, i.e., $9X$, $11X$, $13X$ and $15X$, are omitted, and the values are approximated with their nearest neighbours. Since the number of common subexpressions is reduced from 8 to 4, the original (8:1) multiplexer is replaced by a (4:1), leading to a significant complexity reduction for circuit implementation. In the meantime, the error introduced by this approximation is small (maximum error is 1/9 of the original multiplication result), which can be further compensated during the training process. Fig. 8 shows two images rendered using (a) the original NeRF and (b) NeRF implemented using approximated RMCM. As we can see, there is no observable visual difference. The peak signal-to-noise ratio (PSNR) is as high as 48.24. By the introducing of approximation, a significant amount of hardware complexity, about 1/3, can be saved according to our evaluation.

(a) Ground Truth (b) Approximation

Fig. 8. Two images rendered using (a) the original NeRF and (b) NeRF implemented using approximated RMCM.

Fig. 7(c) illustrates the implementation details of the RMCM block. The pre-compute module generates the four common subexpressions, which are consumed by the following 64 SAA block. A comparator in PCM checks if the input is a zero or not, and gate the products (for zero input) for power-saving. A SAA block takes in the common subexpressions and generates the final multiplication result with proper selection and shift-add operations.

Compared with hidden layers, the computation of output layer is much simpler. In order to keep the same data from Activation Memory 1 and Activation Memory 2 to reduce the complexity of reading control, we implement 64 multiplication units in SONB. The data flow in SONB is similar to MONB except that the multiplications in SONB are implemented using general multipliers instead of RMCM blocks. An output first-in-first-out (FIFO) buffer the final results of the network before writing them back into the external DRAM.

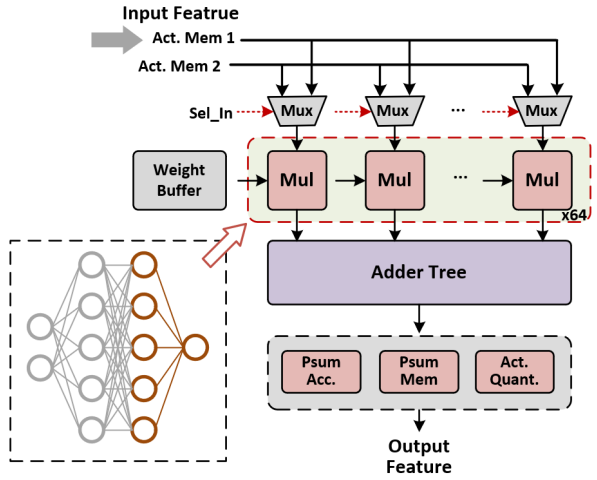


Fig. 9. Detailed structure of the single output network block (SONB). It is used to compute the output layer of an MLP.

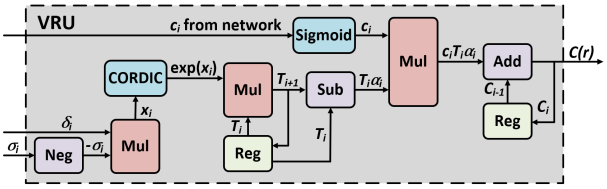


Fig. 10. Detailed structure of the volume rendering unit (VRU).

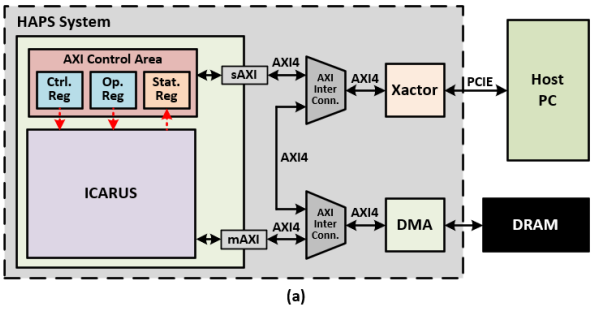
4.4 Volume Rendering Unit

In ICARUS, a VRU is implemented to render the final pixel colors. The main motivation of using a dedicated module instead of using CPU for volume rendering is to reduce the off-chip memory access, which is time and power consuming. Since colors and densities of all sample points in a ray are compressed to the color of a pixel after volume rendering, data exchange between ICARUS and main memory is significantly reduced. Take the original NeRF as an example. To perform the volume rendering in CPU, we need to output the colors $\boldsymbol{c} = (r, g, b)$ and densities σ of 192 sample points for each ray. While after volume rendering, the amount of data is reduced to the color $\boldsymbol{c} = (r, g, b)$ of one pixel, which is $3/(192 \times 4) = 1/256$ of the original.

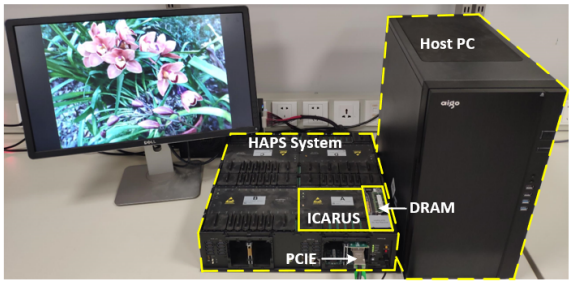
The equation of classic volume rendering by Max [Max 1995] is given by

$$\begin{aligned} C(r) &= \sum_{i=1}^{N} T_i(1 - \exp(x_i))c_i, \\ T_i &= \exp(\sum_{j=1}^{i-1} x_j), x_i = -\sigma_i\delta_i \end{aligned} \tag{4}$$

where $\delta_i = t_{i+1} - t_i$ is the distance between adjacent samples, $C(r)$ is the color of one ray we estimated by sample points, σ is the volume density calculated by network and c_i is the color of each sample point that goes through the activation function of sigmoid. We can



(a)



(b)

Fig. 11. (a) Block diagram and (b) system setup of the FPGA-based prototype.

rewrite the formula (4) to

$$\begin{aligned} C(r) &= \sum_{i=1}^{N} (T_i - T_{i+1})c_i, \\ T_{i+1} &= \prod_{j=1}^{i-1} \exp(x_j) \cdot \exp(x_i) = T_i \cdot \exp(x_i). \end{aligned} \tag{5}$$

Fig.10 illustrates the architecture of the VRU, which implements the rendering equation (5). In the VRU, we use a CORDIC module to calculate the exponential of x_i. The saved T_i is used in two calculations: 1) the multiplication with $\exp(x_i)$ to get T_{i+1} and 2) the subtraction with T_{i+1} to generate $T_i \cdot \alpha_i$, where $\alpha_i = 1 - \exp(x_i)$. After finishing these two operations, T_i is replaced by T_{i+1} for the calculation in next accumulation. Once we have the coefficient of c_i, we multiply it with c_i to get the weighted color C_i of one sample point. The accumulations proceeds until we get the color of one pixel.

5 RESULTS

To validate our rendering system, we implement a single core version of ICARUS on Synopsys HAPS-80 S104 system and also synthesis with 40LP CMOS process. In this section, we first describe how we map the rendering architecture in the HAPS system. Then we compare the rendering performance of NeRF with the GPU implementation. Our system also support MLP-based rendering tasks. Therefore, we implement an surface light field algorithm on the prototype system. On the last, we will analyse and discuss the system performance on our ASIC version base on these two algorithm comparing to GPU.

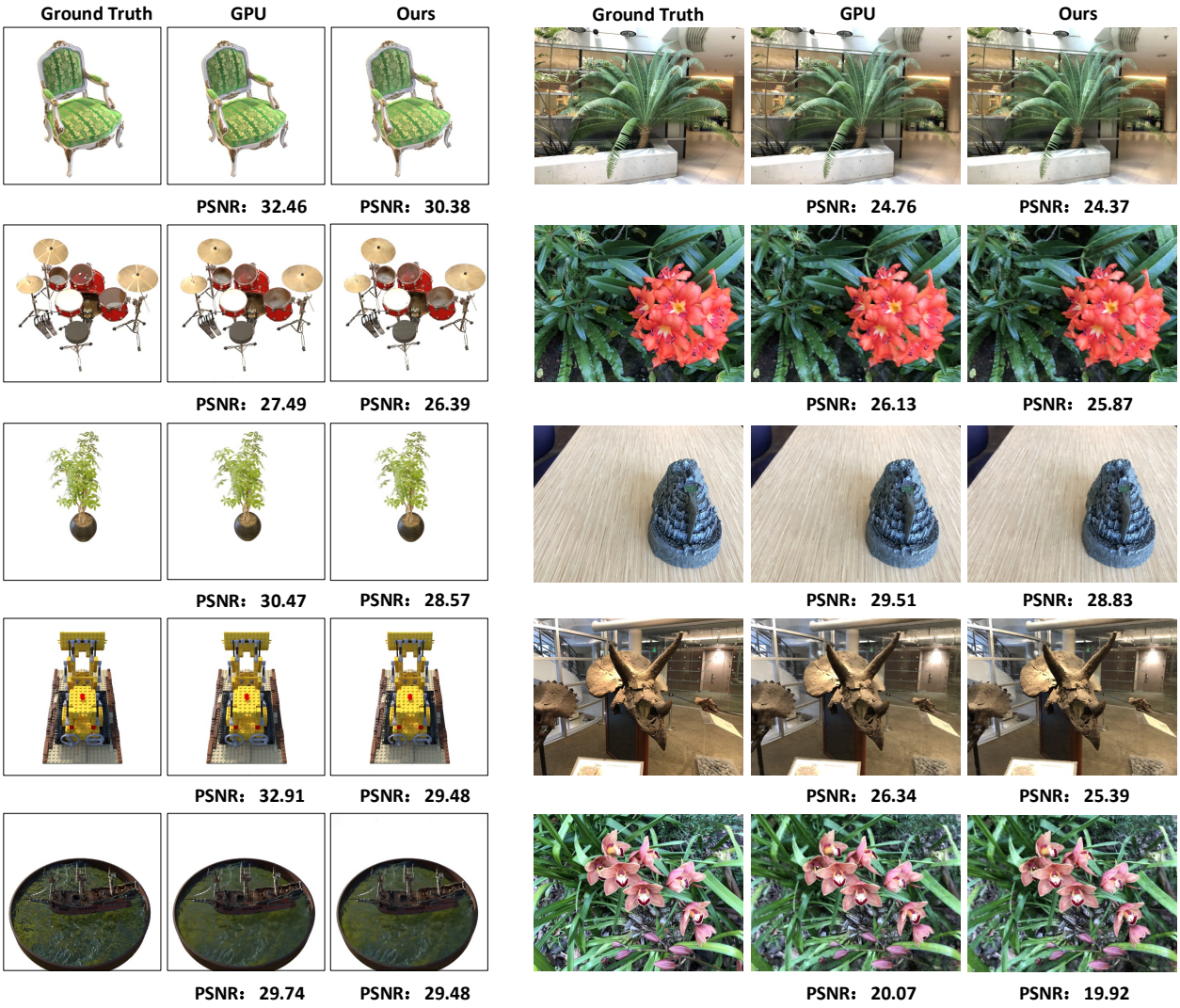


Fig. 12. Comparison of GPU and ICARUS on NeRF. PSNR is below each result.

5.1 FPGA Prototype

Fig. 11(a) shows the block diagram of the prototype system, consisting of a ICARUS co-processor for NeRF rendering algorithms, a host PC for system control and a DRAM for data exchange between ICARUS and PC. The host PC connects to ICARUS through PCIe, where a Synopsys Xactor is used to decode the PCIe stream to AXI format. There are three addressable registers in the prototype system where two of them are write-only registers for saving processing instructions and control signals and the other one is a read-only register for indicating the state of the system. The system setup is shown in Fig. 11(b). After loading the network model, the host PC send inputs and instructions to the DRAM. The ICARUS co-processor reads the inputs and processing instructions, and executes the NeRF pipeline to render the pixel colors. The pixel colors are sent back to the host PC to generate the final rendered image.

NeRF rendering using ICARUS To validate the rendering quality of ICARUS, we execute NeRF on the FPGA-based prototype system.

Specifically, for every pixel to render, we adopted a two-pass sampling strategy: first generate 64 uniformly distributed samples within the visible range, calculate density distribution along the pixel ray, finally generate another 128 samples that are more close to the surface of the object.

In Fig. 12, we compare the rendering results of GPU and our prototype ICARUS. We also include ground truth images for reference.

We run comprehensive experiments including both synthetic and real scenes that are presented in previous works. As illustrated,

Table 1. Comparison of NeRF-based implementation

Reference	SNeRG	PlenOctree	NSVF	JaxNeRF		Instant NGP	Ours
Platform	GPU				TPU	GPU	ASIC
	Radeon Pro 5500M	Nvidia Tesla V100			TPU v2	RTX 3090	HLMC
Process (nm)	7	12			16	8	40
Frequency (Mhz)	1300	1245			700	1395	300
Throgphput (FPS)	84	167.68	0.815	0.096	0.044	194.4	0.029
Power (W)	85	300			280	350	0.3092
Samples per ray	1	-	-	192	192	128	96
Energy Efficiency (uJ/sample) ↓	1.581	-	-	25.431	51.787	0.022	0.174
Memory Usage (MB)	1771.52	1976.32	16	5		7560	0.9605

Fig. 13. Comparison of GPU and ICARUS on the SLF rendering task.

there are practically no visual differences between our results and those by GPU under all scenes. In the 'ficus' and 'fern' scene, we faithfully preserve fine leaf edges. Likewise, in the 'ship' scene, we reproduce the reflectance of water and the thin mast of the ship. These experiments reveal the fact that fixed-point computation is appropriate for rendering tasks and validate the precision of our computing pipeline. They further demonstrate that our hardware is a sensible substitution of GPU in the domain of neural radiance field rendering.

SLF rendering using ICARUS An alternate neural rendering technique is SLF. An SLF is a collection of all light rays and their radiances that emit from the surface of an object in all directions. Similar with NeRF, an SLF is compactly encoded in a fully-connected neural network. To demonstrate the applicability of ICARUS to other MLP-based neural rendering algorithms, we implement an SLF framework using the prototype ICARUS. Fig. 13 shows comparison of GPU and ICARUS on the SLF rendering task. As we can see, ICARUS is capable of generating similar results to GPU, where the PSNRs are almost the same.

5.2 ASIC Evaluation

To evaluate the power and energy performance of the proposed ICARUS, we implement a single core ICARUS system using HLMC 40nm LP CMOS process. The design is synthesized using Synopsys DesignCompilier, and power is evaluated using Synopsys PrimePower based on real stimulus. Table 1 presents the rendering performance comparison of different method on different platforms. Since there is no other dedicated hardware design for NeRF, we compare the performance of our design with other GPU and TPU implementations. Among the all the compared results, only JaxNeRF and ICARUS implements the NeRF algorithm, while other methods apply algorithm-level optimizations (we discussed in related work) to speed up the rendering. As we can calculate, the energy efficiency, in terms of uJ/sample, of ICARUS is 146 times better than the GPU version of JaxNeRF and 297 times better than the TPU version of JaxNeRF. For other implementations such as PlenOctree and SNeRG, though they are fast for rendering, a lot of memory is consumed for the storage of explicit representation of scenes. Moreover, we need to emphasize that the implementation results of ICARUS in Table 1 are based on the 40nm process, which is about 3~4 generation older than that used for modern GPUs and TPUs. By using more advanced CMOS technologies, the power and energy performance can be much better.

6 DISCUSSION

Experiments demonstrate the ability of the proposed lightweight graphics architecture to render images under similar quality to other coordinate-based neural rendering methods running on GPUs. Having dedicated circuits and systems designs, our architecture delivers two orders of magnitude improvement in energy efficiency compared to conventional GPUs on neural rendering tasks, e.g., NeRF. This feature makes our architecture more friendly for mobile devices, such as HMDs in AR/VR applications, smartphones, and 3D displays. Among these scenarios, small size, light weight, low power consumption, and untethered experience are consumer pain points. People can view photo-realistic neural assets on these devices freely and fluidly with our architecture when playing games or online shopping.

Our system design can not only serve as an individual accelerator, but also be integrated into existing GPU pipelines to enhance the ability to render neural assets. At the hardware level, the streaming

multiprocessor(SM) can integrate our architecture the same way as RT (Ray Tracing) cores. In this way, the SM controller can directly control our designed components to process neural rendering tasks to improve SMs. Alternatively, our design can also work as an independent unit like SMs. The GPU controller can dispatch instructions and data to our design unit to process neural rendering tasks. Furthermore, by doing this, some priors, e.g., depth maps, generated from the traditional rasterization pipeline can be efficiently passed to our architecture with low latency, thus aiding in the acceleration of neural rendering.

REFERENCES

Edward H Adelson, James R Bergen, et al. 1991. *The plenoptic function and the elements of early vision*. Vol. 2. Vision and Modeling Group, Media Laboratory, Massachusetts Institute of ....

Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. 2019. *Real-time rendering*. Crc Press.

Mike Bailey. 2007. Glsl geometry shaders. *Oregon State University* (2007).

Mark Boss, Raphael Braun, Varun Jampani, Jonathan T Barron, Ce Liu, and Hendrik Lensch. 2021. Nerd: Neural reflectance decomposition from image collections. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 12684–12694.

Chris Buehler, Michael Bosse, Leonard McMillan, Steven Gortler, and Michael Cohen. 2001. Unstructured lumigraph rendering. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. 425–432.

Edwin Earl Catmull. 1974. *A subdivision algorithm for computer display of curved surfaces*. The University of Utah.

Anpei Chen, Minye Wu, Yingliang Zhang, Nianyi Li, Jie Lu, Shenghua Gao, and Jingyi Yu. 2018. Deep surface light fields. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1, 1 (2018), 1–17.

Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 367–379.

Stephan J Garbin, Marek Kowalski, Matthew Johnson, Jamie Shotton, and Julien Valentin. 2021. Fastnerf: High-fidelity neural rendering at 200fps. *arXiv preprint arXiv:2103.10380* (2021).

Steven J Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F Cohen. 1996. The lumigraph. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. 43–54.

Jiatao Gu, Lingjie Liu, Peng Wang, and Christian Theobalt. 2021. Stylenerf: A style-based 3d-aware generator for high-resolution image synthesis. *arXiv preprint arXiv:2110.08985* (2021).

Dongsoo Han. 2007. Tessellating and Rendering Bezier/B-Spline/NURBS Curves and Surfaces using Geometry Shader in GPU. *University of Pennsylvania* (2007).

Tom's hardware. 2020. *AMD to Introduce New Next-Gen RDNA GPUs in 2020, Not a Typical 'Refresh' of Navi*. http://https://www.tomshardware.com/news/amds-navi-to-be-refreshed-with-next-gen-rdna-architecture-in-2020

Aaron Isaksen, Leonard McMillan, and Steven J Gortler. 2000. Dynamically reparameterized light fields. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 297–306.

J. T. Kajiya and H. B. Von. 1984. Ray tracing volume densities" computer graphics 18. (1984).

Maxim Kazakov. 2007. Catmull-Clark subdivision for geometry shaders. In *Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*. 77–84.

Petr Kellnhofer, Lars C Jebe, Andrew Jones, Ryan Spicer, Kari Pulli, and Gordon Wetzstein. 2021. Neural lumigraph rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4287–4297.

Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. 2020. Neural sparse voxel fields. *arXiv preprint arXiv:2007.11571* (2020).

Yuan Liu, Sida Peng, Lingjie Liu, Qianqian Wang, Peng Wang, Christian Theobalt, Xiaowei Zhou, and Wenping Wang. 2021. Neural Rays for Occlusion-aware Image-based Rendering. *arXiv preprint arXiv:2107.13421* (2021).

Haimin Luo, Anpei Chen, Qixuan Zhang, Bai Pang, Minye Wu, Lan Xu, and Jingyi Yu. 2021. Convolutional Neural Opacity Radiance Fields. *arXiv preprint arXiv:2104.01772* (2021).

N. Max. 1995. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics* 1, 2 (1995), 99–108. https://doi.org/10.1109/2945.468400

Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J Doyle, Michael Guthe, and Jiří Bittner. 2021. A Survey on Bounding Volume Hierarchies for Ray Tracing. In *Computer Graphics Forum*, Vol. 40. Wiley Online Library, 683–712.

Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. 2020. Nerf: Representing scenes as neural radiance fields for view synthesis. In *European conference on computer vision*. Springer, 405–421.

Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. 2022. Instant Neural Graphics Primitives with a Multiresolution Hash Encoding. *arXiv preprint arXiv:2201.05989* (2022).

Martin E Newell, RG Newell, and Tom L Sancha. 1972. A solution to the hidden surface problem. In *Proceedings of the ACM annual conference-Volume 1*. 443–450.

Nvidia. 2020. *NVIDIA Ampere Architecture*. https://www.nvidia.com/en-us/data-center/ampere-architecture/

Jongsun Park, Woopyo Jeong, Hamid Mahmoodi-Meimand, Yongtao Wang, Hunsoo Choo, and Kaushik Roy. 2004. Computation sharing programmable FIR filter for low-power and high-performance applications. *IEEE Journal of Solid-State Circuits* 39, 2 (feb 2004), 348–357. https://doi.org/10.1109/JSSC.2003.821785

Keunhong Park, Utkarsh Sinha, Jonathan T Barron, Sofien Bouaziz, Dan B Goldman, Steven M Seitz, and Ricardo Martin-Brualla. 2020. Deformable neural radiance fields. *arXiv preprint arXiv:2011.12948* (2020).

Keunhong Park, Utkarsh Sinha, Peter Hedman, Jonathan T Barron, Sofien Bouaziz, Dan B Goldman, Ricardo Martin-Brualla, and Steven M Seitz. 2021. Hypernerf: A higher-dimensional representation for topologically varying neural radiance fields. *arXiv preprint arXiv:2106.13228* (2021).

Sida Peng, Junting Dong, Qianqian Wang, Shangzhan Zhang, Qing Shuai, Hujun Bao, and Xiaowei Zhou. 2021. Animatable Neural Radiance Fields for Human Body Modeling. *arXiv preprint arXiv:2105.02872* (2021).

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

Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred Hamprecht, Yoshua Bengio, and Aaron Courville. 2019. On the spectral bias of neural networks. In *International Conference on Machine Learning*. PMLR, 5301–5310.

Christian Reiser, Songyou Peng, Yiyi Liao, and Andreas Geiger. 2021. KiloNeRF: Speeding up Neural Radiance Fields with Thousands of Tiny MLPs. *arXiv preprint arXiv:2103.13744* (2021).

Peiran Ren, Yue Dong, Stephen Lin, Xin Tong, and Baining Guo. 2015. Image based relighting using neural networks. *ACM Transactions on Graphics (ToG)* 34, 4 (2015), 1–12.

VV Sanzharov, Vladimir A Frolov, and Vladimir A Galaktionov. 2020. Survey of Nvidia RTX Technology. *Programming and Computer Software* 46, 4 (2020), 297–304.

Pratul P Srinivasan, Boyang Deng, Xiuming Zhang, Matthew Tancik, Ben Mildenhall, and Jonathan T Barron. 2021. Nerv: Neural reflectance and visibility fields for relighting and view synthesis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 7495–7504.

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*. 4690–4699.

Daniel N Wood, Daniel I Azuma, Ken Aldinger, Brian Curless, Tom Duchamp, David H Salesin, and Werner Stuetzle. 2000. Surface light fields for 3D photography. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 287–296.

Alex Yu, Sara Fridovich-Keil, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. 2021a. Plenoxels: Radiance Fields without Neural Networks. *arXiv preprint arXiv:2112.05131* (2021).

Alex Yu, Ruilong Li, Matthew Tancik, Hao Li, Ren Ng, and Angjoo Kanazawa. 2021b. Plenoctrees for real-time rendering of neural radiance fields. *arXiv preprint arXiv:2103.14024* (2021).

Alex Yu, Vickie Ye, Matthew Tancik, and Angjoo Kanazawa. 2021c. pixelnerf: Neural radiance fields from one or few images. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4578–4587.

Xiuming Zhang, Pratul P Srinivasan, Boyang Deng, Paul Debevec, William T Freeman, and Jonathan T Barron. 2021. NeRFactor: Neural Factorization of Shape and Reflectance Under an Unknown Illumination. *arXiv preprint arXiv:2106.01970* (2021).

Ellen D Zhong, Tristan Bepler, Joseph H Davis, and Bonnie Berger. 2019. Reconstructing continuous distributions of 3D protein structure from cryo-EM images. *arXiv preprint arXiv:1909.05215* (2019).