NeRF-OPU: An FPGA-based Overlay Processor for Neural Radiance Fields

ABSTRACT

Recent research on Neural Radiance Fields (NeRF) has demonstrated the capacity of neural networks to encode complex 3D environments, enabling the photorealistic rendering of scenes from diverse perspectives. Nevertheless, traditional NeRF algorithms encounter difficulties when it comes to achieving real-time rendering, thereby highlighting the need for dedicated NeRF-specific accelerators. FPGAs serve as an ideal platform for accelerating NeRF. However, the time-consuming nature of FPGA reconfiguration when switching between different NeRF-based models poses a significant impediment. This paper proposes NeRF-OPU, an FPGA-based overlay processor for general NeRF accelerations with excellent flexibility and software-like programmability. The executable code of NeRF-based models is automatically compiled and reloaded without requiring FPGA reconfiguration. Our experimental results show that NeRF-OPU achieves on average 1.80×, 1.03× speedup over NVIDIA RTX 3060 Ti on vanilla NeRF and Instant-NGP. Compared with the previous NeRF FPGA accelerator, NeRF-OPU performs on average 2.01× speed up and 2.16× energy efficiency on Instant-NGP. To the best of our knowledge, NeRF-OPU is the first in-depth study to develop an FPGA-based processor for NeRF acceleration with competitive performance and energy efficiency.

1 INTRODUCTION

Neural Radiance Fields (NeRF) [5] is a recently proposed approach for representing complex 3D scenes using a learned continuous function, which can then be used for rendering novel views of the scene from any viewpoint. The method has received significant attention due to its ability to generate highly detailed and photorealistic renderings of complex scenes, surpassing the capabilities of traditional rendering techniques. One of the key advantages of NeRF is its ability to handle complex scenes with detailed geometry [7, 10, 12] and lighting effects [1, 9, 15, 16], which are difficult to model using traditional rendering techniques.

However, even on high-end GPUs such as the NVIDIA RTX 3090, achieving a frame rate of 30 frames per second (FPS) during the rendering of a 2-million-pixel (1920×1080) image still remains unattainable, even when employing state-of-the-art (SOTA) NeRF algorithms. This observation underscores the imperative to introduce hardware acceleration as a means to address this performance disparity.

Current SOTA Deep Neural Network (DNN) accelerators, as demonstrated by notable works [13, 14], exhibit limited capability in executing the NeRF-based models. This limitation originates from the absence of dedicated ray sampling and volume rendering modules specifically designed for NeRF within these accelerators. Consequently, there has been a concerted effort to design specialized FPGA accelerators tailored to enhance the computational efficiency of NeRF [11, 17]. [11] designed a customized accelerator for vanilla NeRF and [17] designed a customized accelerator for Instant-NGP [6]. However, they are engineered to support a singular NeRF algorithm, necessitating a time-consuming process involving logic synthesis, placement and routing when transitioning to fit a different NeRF algorithm. Due to recent advancements in NeRF algorithms, focusing solely on a single algorithm is no longer sufficient. This highlights the need for a more flexible and versatile approach in FPGA-based NeRF acceleration methodologies.

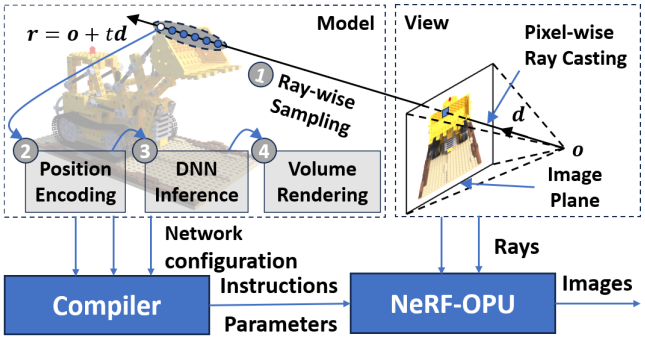

Figure 1: NeRF-OPU workflow. The compiler maps the distinct stages of NeRF into customized instructions for execution on the NeRF-OPU micro-architecture.

To address the aforementioned challenges, we propose NeRF-OPU as an FPGA-based overlay processor unit for the acceleration of NeRF-based algorithms with software programmability. NeRF-OPU comprises custom-designed instruction sets and fine-grained pipeline at the micro-architecture level. As shown in Fig. 1, the compiler, integrated into the system, accepts network architecture configurations from nerfacc [3], a NeRF acceleration toolbox based on the PyTorch deep learning framework. Afterward, the compiler maps network operations to processor modules for instruction sequence generation. The resulting instruction sequence is subsequently dispatched to NeRF-OPU for execution. Consequently, this architecture enables rapid deployment of officially published models without micro-architecture modifications. To summarize, the features of our proposed NeRF-OPU are listed as follows:

An Overlay Architecture. In this paper, we develop an overlay processor for three mainstream NeRF-based models with a dedicated micro-architecture for sampling, encoding, DNN inference, and volume rendering. To the best of our knowledge, NeRF-OPU is the first FPGA-based general processor for NeRF acceleration.

Flexible Dataflow. The granularity of our instructions is optimized to ensure the generality of computation engines. Furthermore, the control mechanism based on instructions enables dynamic pipelining of operations, effectively concealing communication latency and improving overall efficiency.

Competitive Performance. Experimental results show that NeRF-OPU achieves on average 1.80×, 1.03× speedup over NVIDIA RTX 3060 Ti on vanilla NeRF and Instant-NGP. Compared with the previous NeRF FPGA accelerator, NeRF-OPU performs on average 2.01× speed up and 2.16× energy efficiency on Instant-NGP.

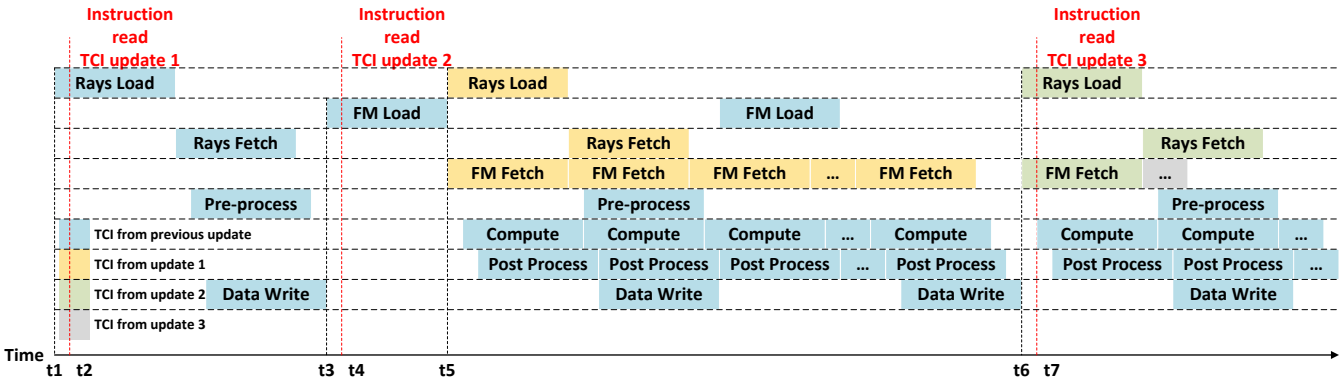

Figure 2: Instruction execution and TCI updates. TCI updates triggered by instruction reads are denoted by red lines. Each distinctively colored block delineates the execution time range of an individual triggered instruction.

2 BACKGROUND AND MOTIVATION

2.1 Preliminaries of NeRFs

In contrast to the conventional graphics rendering pipeline, which generates pixel colors based on input scene information such as physics, textures, and materials, NeRF leverages a 5D representation of the points in the scene and employs implicit MLP modeling to fit the corresponding colors and densities. NeRF's rendering can be broadly divided into four main stages: sampling, position encoding, DNN inference, and volume rendering. As shown in Fig. 1, following a ray-wise sampling procedure that bypasses empty spatial regions in the scene, the sampled 5D points undergo encoding and inference to generate color and density. Subsequently, volume rendering is tasked with the accumulation of color and density information for individual pixels, resulting in the production of the final output image.

2.2 Motivation

The profiling of NeRF execution on GPU reveals a consistent time allocation across all computation stages. In Instant-NGP, the hash encoding stage emerges as a crucial factor affecting these outcomes, as it deteriorates the GPU's proficiency with memory-bound processes. Furthermore, the implementation of the MLP requires significant computational power, which increases the overall computational requirements.

Note that both sampling and rendering also have a significant impact on rendering time. Our analysis reveals that the sampling stage's ineffective utilization of GPU resources can be attributed to an overdependence on conditional functions in sampling operations. The extended period of time it takes to resolve long scoreboards on the GPU results in an inefficient utilization of available computational resources. Moreover, numerous memory-bound element-wise operations during the rendering process significantly reduce computational efficiency on the GPU.

With respect to GPU underutilization, FPGAs have the potential to offer a solution by adapting the logic for certain calculations, thus providing performance advantages.

3 INSTRUCTION SET ARCHITECTURE

NeRF-OPU has been purposefully designed to cater to the general requirements of NeRF inference. Leveraging the foundational instruction framework of OPU [13], NeRF-OPU incorporates additional parameter configurations tailored to the sampling and rendering operations intrinsic to NeRF computations, thus enabling support for a diverse range of NeRF-based models.

3.1 Instruction discription

To accommodate the additional sampling and rendering stages inherent to NeRF computations, we introduce supplementary C-type instructions tailored to our micro-architecture. Correspondingly, we also incorporate several U-type instructions for parameter transmission. In our instruction set architecture (ISA), we design a total of five distinct ISA instructions, which are categorized as memory access, data load, sample, compute, and post-process:

Memory Access manages data transfer between external memory and onboard memory, as well as bidirectional transfers from onboard memory to external memory. This instruction defines the relevant on-chip memory read/write states and indices. Specific memory access addresses and sizes are specified by corresponding U-type instructions.

Data Load transfers data from on-chip memory to either the sample engine or the computation engine. Specifically, operations such as data splitting/merging, copying, concatenation, and reshaping are available. The data processed by this module is subsequently conveyed to the computation engine for further processing.

Preprocess governs the operations of the ray sampling along with the encoding module, which computes sample points along rays in space and then encodes to higher dimensions.

Compute controls the operations of processing element (PE) units and the selective adder tree. A PE unit computes the dot product of two one-dimensional vectors, each of length N, with N set to 16 in NeRF-OPU. This choice accommodates spatial exploration across different MLP networks effectively.

Post Process handles non-compute-intensive operations, such as nonlinear functions and volume rendering. Nonlinear functions include operations like exponentials, ReLU, sigmoid, and other mathematical functions commonly encountered in the MLP networks of NeRF.

These specialized instructions collectively facilitate the efficient execution of NeRF computations, encompassing data manipulation, ray sampling, encoding, and post-processing stages, thereby ensuring the overall continuity and integrity of NeRF operations on our micro-architecture.

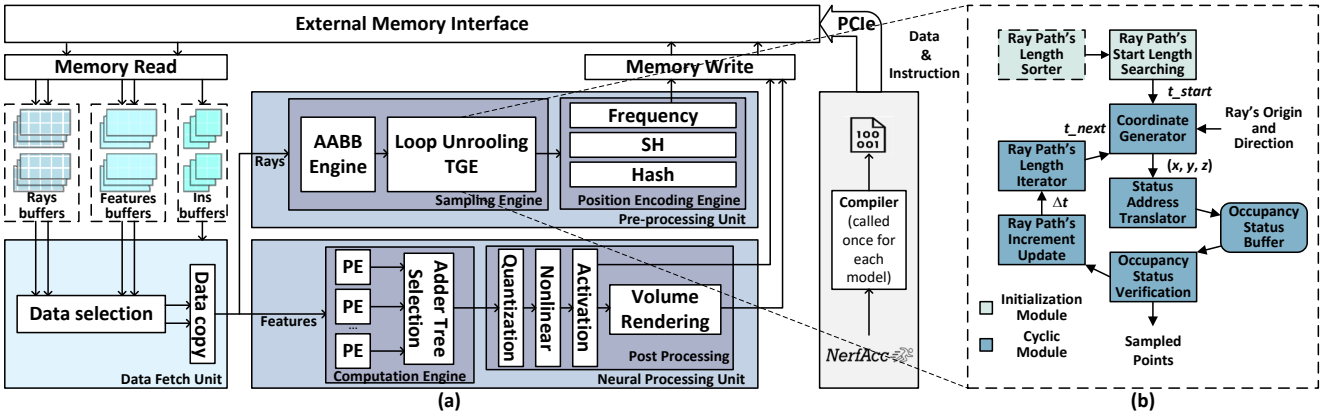

Figure 3: (a) Overview of NeRF-OPU micro-architecture; (b) The details of TGE.

3.2 Instruction execution

In the inference process of NeRF-based model, each step of operation relies on distinct preceding operations. To achieve effective instruction control, we adopt a trigger condition index (TCI) mechanism similar to that of OPU. Specifically, during runtime, instructions modify the trigger condition index to establish module launch dependencies. The use of a dependency-based execution strategy relaxes the strict sequencing of instruction sequences, allowing sufficient flexibility to accommodate time uncertainties arising from memory-related operations. This approach ensures efficient instruction control while mitigating the impact of memory-related time variability in NeRF's inference workflow. For example, Fig. 2 displays a segment of instruction execution. Several Instruction Read is performed to update TCI. The color of each instruction during execution indicates the TCI currently in use. The subsequent TCI is updated immediately following the triggering of the current TCI to guarantee that the operation will initiate based on the new TCI next time.

4 MICRO-ARCHITECTURE

4.1 Overall Architecture

Designing a versatile and resource-efficient OPU architecture presents a significant challenge. The key objective is to create an OPU that can accommodate a wider range of NeRF-based models while ensuring ease of control, reducing design redundancies, and enhancing structural reusability. To address this challenge, we devise a parametrically customizable pre-processing module, which facilitates the generation of vectors of varying dimensions, enabling tailored adaptability to diverse requirements through the use of counters and configuration registers. In the context of the computational engine, we investigate the optimization of PE size and the parallelism of PE arrays, thereby working towards achieving an improved configuration.

As shown in figure 3, the NeRF-OPU consists of five main parts: data fetch unit, NeRF pre-processing unit, neural processing unit, data buffer, and instruction control unit. Each unit can be controlled by instructions to perform the functions defined in Section 3.1. Since most of the control flow is embedded in the instructions, the micro-architecture mainly manages the computational of NeRF, specifically NeRF pre-processing and neural processing.

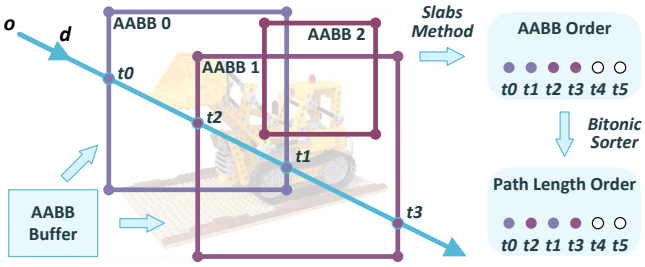

Figure 4: The details of AABB intersection detection.

4.2 NeRF Pre-processing Unit

The NeRF pre-processing unit consists of two integral components: a sampling engine and a position encoding engine, corresponding to the initial two stages of the NeRF pipeline process. The former is responsible for sampling the ray, while the latter encodes the coordinates of the sampled points.

4.2.1 **Sampling Engine(SE).** SE leverages the occupancy grid method in nerfacc to discern points on the ray that contribute to the volume rendering calculation. SE consists of an axis-aligned bounding box (AABB) engine and a traverse grid engine (TGE). The AABB engine initially samples intervals of the essential part, and subsequently, the TGE samples points within those intervals.

The AABB Engine comprises an AABB logic Engine and a sorter. Before executing the calculation, NeRF-OPU acquires the configuration of all bounding boxes of the rendering scene and stores them in a register array. As shown in Fig. 4, Upon receiving the ray input $ray = \{r|r = o + td\}$, the AABB Engine calculates the intersection point of the ray and the bounding box based on the slabs method [2]. Rays not intersecting with bounding boxes are skipped, and the AABB Engine returns the ray path's length t_i of all ray-bounding box intersections. The ray path's length t_i is subsequently sent into the sorter which employs the bitonic sorting method. Note that the labels of t_i in the AABB order are still attached to sorted t for TGE to determine in which AABB any interval $t_i t_j$ lies. As not all AABBs necessarily intersect with the given ray, for those AABBs lacking intersection with the ray, the corresponding ray path's length t_i for the intersection point is set to the maximum value. This adjustment is made to maintain uniformity in the array dimensions, accounting for scenarios where intersections occur with all rays.

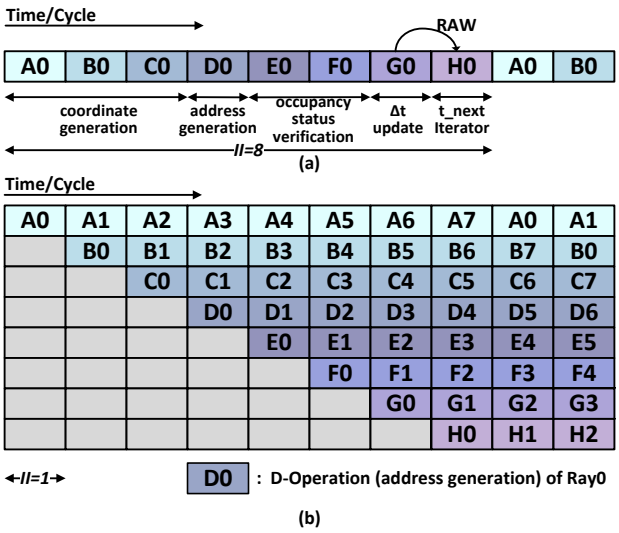

Figure 5: (a) RAW dependency of the Non-pipelined implementation of TGE; (b) Loop unrolling to compensate the loop iteration initiation interval (II) = 8.

The TGE, depicted in Fig. 3(b), receives the sorted ray path length and identifies the interval $t_i t_j$ including the near plane of the frustum. Starting from this interval, the TGE iteratively finds all sample points on the current ray. The coordinate generator utilizes the ray path's length t to generate the corresponding 3D coordinate based on the ray's origin coordinate and direction. The generated coordinate is then forwarded to the occupancy status address translator to access the occupancy status buffer. The accessed result determines whether this particular point has high density and necessitates sampling. Subsequently, the step size Δt is updated depending on whether the point is sampled. This step size Δt is then transmitted to the ray path's length t iterator for the next iteration.

Loop Unrooling TGE. Profiling of mainstream NeRF-based models on the GPU reveals that the sampling process is a computational bottleneck, consuming approximately 22% of the computation time on average. This necessitates the acceleration of the sampling process. However, parallelizing the TGE leads to excessive consumption of computational resources and on-chip memory due to the high resource utilization of a single TGE. Additionally, a pipelined pattern TGE encounters a RAW conflict, as the ray path's length iterator must wait for the step size Δt to be updated.

Therefore, we develop a loop-unrolling TGE. As shown in Fig. 5(a), we first add 2 additional pipeline stages at the coordinate generator, and 1 additional pipeline stage at the occupancy status check, and design a loop iteration initiation interval(II) = 8 pipelined TGE, after which we insert the other 7 sets of rays for each step in it. We set a group of pipeline states indicating tasks with computational resources in the TGE, and a *BUSY* signal indicating when the cluster of 8 rays has completed processing. Once the previous 8 rays have been sampled, the subsequent 8 rays are input for a new round of calculations. This yields an II=1 pipeline structure with shared computational units. Compared to running 8 original TGEs in parallel, our approach saves 49 DSPs and 58% of the LUT resources, without any additional register or memory overhead.

4.2.2 **Position Encoding Engine.** The Position Encoding Engine processes the output of the SE and engages a designated encoding module to encode the acquired data according to specified instructions. In this paper, to accommodate vanilla NeRF, D-NeRF [8], and Instant-NGP, a total of three encoding formats are adapted, frequency encoding, spherical harmonic encoding, and hash encoding.

To accommodate vector generation in different dimensions, both frequency encoding and spherical harmonic encoding use different specialized loopable arithmetic units. For frequency encoding, the approach detailed in DIF-LUT [4] is employed to approximate sine and cosine within $(-\pi, \pi]$. In instances where the input extends beyond this range, a shift by 2π is applied to bring it within the specified interval. Spherical harmonic encoding, conversely, entails the Cartesian expansion of the spherical harmonics into polynomials in 3D coordinates. The spherical harmonic encoding engine is engineered to accommodate degrees up to 5. To align with the SE sampling results, 8 frequency encoding engines, and 8 spherical harmonic encoding engines are implemented.

Unlike the previously mentioned frequency encoding and spherical harmonic encoding, the hash encoding engine is implemented in a pipelined architecture due to the inherent limitation that reading a hash table cannot be performed in parallel. Hash tables are stored in the on-board URAM to enable a streamlined pipelining process.

4.3 Neural Processing Unit

The neural processing unit comprises a computation engine and a post processing engine, corresponding to the last two stages of the NeRF pipeline process.

Computation Engine. The computation process of NeRF entails a substantial number of matrix-vector multiplications utilizing distinct MLP layer sizes across separate layers and networks. We deploy 256 PEs along with 16 16-input selective adder trees for selective summation of every 16 PE outputs, to achieve maximum functional coverage and resource utilization. In every PE, we incorporate 16 multipliers and a 16-input adder tree. Based on instructions, the computation unit can distribute the outputs from PEs to the corresponding input of the selected adder tree. As a result, the computation engine can efficiently handle matrix-vector multiplication where the vector's length is 32, 64, 128, and 256.

Every 16-input selective adder tree contains 8 register chains and a Selector. The register chain permits simultaneous output of the result from arbitrary levels of adders. Thus, the selector can activate specific dimensions of the result based on the instruction. Then, the activated values are either temporarily stored or sent to the activation function. Additionally, for certain irregular matrix-vector multiplication sizes (*e.g.* [27, 128]), we add the minimum number of non-zero elements to meet one of the previously mentioned sizes.

Post Processing Engine. The post processing engine performs data quantization, nonlinear function, activation function, and volume rendering function. Since data quantization, nonlinear function, and activation function are conducted once for one DNN layer, and volume rendering is similarly only executed once for one DNN inference, they are concatenated with the memory write module to reduce extra on-chip data movement.

Volume rendering. The output of MLP (*i.e.* color $\boldsymbol{c}$ and density σ) needs to be calculated according to the volume rendering function:

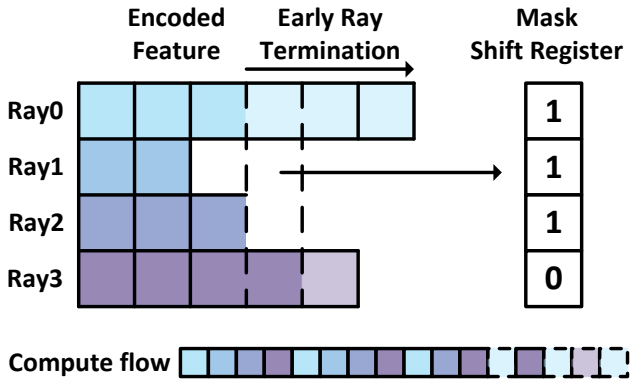

Figure 6: Implementation of early ray termination using a mask shift register. The mask shift register is configured with the binary sequence "1110" by the fourth column of encoded features, resulting in the early termination of ray0, while ray1 and ray2 exhibit an absence of further encoded features.

$\hat{C}(r) = \sum_{i=1}^{N} T_i(1 - \exp(-\sigma_i\delta_i))\boldsymbol{c}_i$, where $T_i = \exp(-\sum_{j=1}^{i-1} \sigma_i\delta_i)$, $\delta_i = t_{i+1} - t_i$, $\boldsymbol{c}_i$ denotes the color, σ_i denotes the volume density, and N indicates the total number of samples.

The volume rendering unit within the post processing engine receives a fixed batch of MLP output in each iteration, executing volume rendering through the following processes: 1) obtaining σ_i from the activation function output and calculating the corresponding $\exp(-\sigma_i\delta_i)$; 2) multiplying the stored cumulative product T_i with $\exp(-\sigma_{i+1}\delta_{i+1})$ in the order of ray path lengths on the same ray and storing the new partial product in the same register array; 3) repeating the previous two steps until all points in the batch have completed the computation, using the stored T_i in the register array to multiply with the corresponding $\boldsymbol{c}_i$ and summing up to obtain the cumulative color. Once all features in the feature buffer are consumed, all cumulative colors are written back to external memory for storage.

Early ray termination. Leveraging a fine-grained data flow, we incorporate the early ray termination strategy into our computational workflow. This strategy involves terminating the computation for a sampling ray if the accumulated density T_i falls below a certain threshold, as the subsequent point on the ray is deemed to contribute little to the final color. This approach significantly reduces computational overhead. As depicted in Fig. 6, a mask shift register is employed to indicate whether a particular ray has completed computation. If all points sampled on a ray have undergone DNN inference or are identified as terminable through early ray termination, the mask corresponding to that ray is set to 1. In each iteration of the data fetch module, the mask shift register determines the address offset for the next point based on a lookup table, enabling the skipping of rays that have completed computation.

5 COMPILER

We engineer a compiler to serve as the bridge between NeRF-based models and the inference process within our custom-designed architecture. In contrast to OPU [13], NeRF predominantly utilizes MLP for the representation of 3D scenes. This design choice obviates the necessity for network reformulation optimization procedures. Thus, the complication process is divided into three stages. Initially, the compiler takes the configuration parameters of the NeRF-based model as input, allowing for the generation of a high-level intermediate representation (IR) code that represents the NeRF-based model's information. This includes the configuration of the sampling strategy in the sampling estimator, as well as the encoding type and MLP network in the radiance field. Subsequently, the compiler executes a mapping from the high-level IR to the low-level IR, optimizing the execution efficiency on the NeRF-OPU micro-architecture. This refinement process specifically focuses on optimizing the vector dimensions aligned with the computational engine. Finally, we apply a direct mapping from low-level IR to the instructions mentioned in Section 3.1. In contrast to model switching achieved through FPGA reconfiguration, the OPU-ISA-based compiler has exhibited a remarkable reduction in time consumption.

6 EXPERIMENTS

6.1 Experimental settings

We evaluate the performance of our NeRF-OPU by implementing it on Xilinx's Virtex UltraScale+ FPGA board Alveo U200 using Verilog in Vivado design suite 2020.2. The resource utilization is shown in Table 1. We set the clock frequency to 250 MHz for our performance comparison. NeRF-based models are defined using PyTorch with the nerfacc library [3], and weights are quantized to 16-bit fixed-point.

Datasets and benchmarks. We comprehensively evaluate our NeRF-OPU using the mainstream NeRF-based models: vanilla NeRF, D-NeRF, and Instant-NGP. We followed the default setting in Instant-NGP and the same setting as nerfacc in vanilla NeRF and D-NeRF. Different encoding types (frequency encoding, spherical harmonic encoding, and hash encoding) and MLP sizes are covered. The assessments for vanilla NeRF and Instant-NGP are performed using NeRF-synthetic datasets, whereas D-NeRF undergo evaluation with D-NeRF datasets. For vanilla NeRF and Instant-NGP, we render 200 images corresponding to various poses for each dataset, while for D-NeRF, 20 images are rendered for different poses and timestamps in each dataset. The resolution of the rendered images is set at 800*800 pixels.

6.2 Performance comparison with GPU

Compared with GPU, the average PSNR of the rendered image is shown in Table 2 and Table 3, indicating an average decline of 1.2 dB. This reduction primarily results from the impacts of quantization and nonlinear function approximation. As shown in Fig. 7, in the case of vanilla NeRF, D-NeRF, Instant-NGP, NeRF-OPU achieves on average 1.80×, 0.85×, 1.03× speedup over NVIDIA RTX 3060 Ti, which has similar Peak TOPS with Alveo U200. Notably, for D-NeRF, this performance inefficiency derives from its inadequacy for early

Table 1: Comparision of FPGA resource utilization between customized accelerator [17] of Instant-NGP and NeRF-OPU.

	board	LUT	FF	BRAM	DSP	URAM
[17]	Alveo U250	128309	551658	N/A	9874	800
NeRF-OPU	Alveo U200	460125	427063	1741.5	4501	894

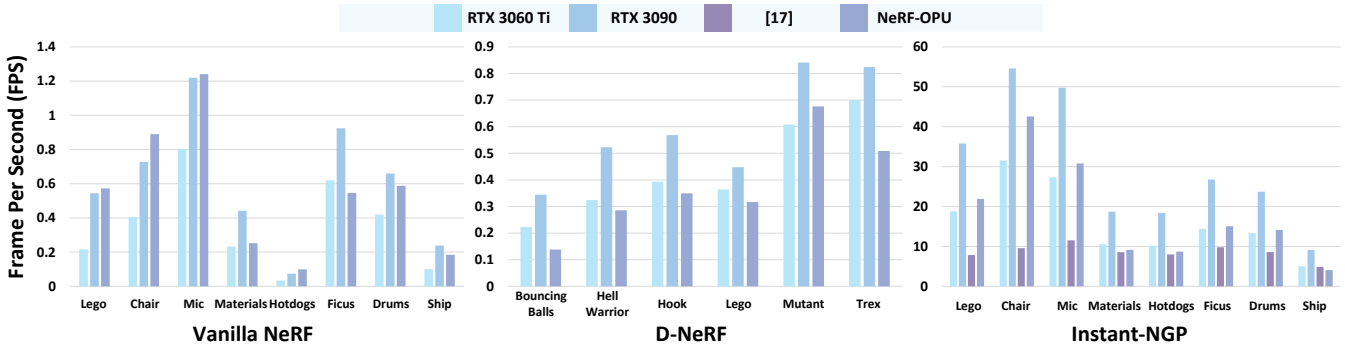

Figure 7: Performance Comparison between GPU, customized FPGA accelerator, and NeRF-OPU on 3 benchmarks.

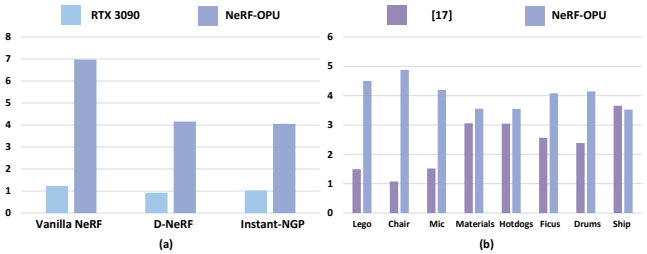

Figure 8: (a) Energy efficiency of NVIDIA RTX 3090 and NeRF-OPU on 3 benchmarks; (b) Energy efficiency of NeRF-OPU and [17] on Instant-NGP (both normalized over NVIDIA RTX 3060 Ti).

Table 2: PSNR↑ of static NeRF with GPU, customized FPGA accelerator, and NeRF-OPU.

	benchmarks	Lego	Chair	Mic	Materials	Hotdog	Ficus	Drums	Ship
GPU	Vanilla NeRF	33.53	33.06	33.71	29.65	35.59	32.50	25.41	28.13
	Instant-NGP	35.79	35.67	36.68	29.64	37.38	33.74	25.46	30.34
[17]	Instant-NGP	33.57	32.55	33.67	27.85	34.86	31.06	25.07	29.01
NeRF-OPU	Vanilla NeRF	32.10	32.04	32.71	28.68	34.27	31.14	24.35	27.13
	Instant-NGP	34.65	34.44	35.38	27.98	36.36	31.65	24.45	29.20

Table 3: PSNR↑ of dynamic NeRF with GPU and NeRF-OPU.

	benchmarks	bouncing balls	hell warrior	hook	lego	mutant	trex
GPU	D-NeRF	39.49	25.58	31.86	24.32	35.55	32.33
NeRF-OPU		37.48	26.42	30.05	23.24	32.39	31.66

ray termination. The power consumption of NeRF-OPU is 44.9W. With respect to power consumption shown in Fig. 8(a), NeRF-OPU demonstrates an energy efficiency of 6.97×, 4.16×, 4.05× compared to NVIDIA RTX 3060 Ti, and 5.59×, 4.57×, 3.95× compared to the NVIDIA RTX 3090 for vanilla NeRF, D-NeRF, and Instant-NGP.

6.3 Performance comparison with FPGA Accelerators

To the best of our knowledge, existing FPGA accelerators are tailored specifically for certain NeRF-based models. In light of this, we benchmark NeRF-OPU against customized accelerators to highlight its general applicability and performance. Table 1 shows that we utilized only 46% of the DSP of [17]. Illustrated in Table 2, the use of higher bit quantization and improved nonlinear function approximation in NeRF-OPU results in on average a 0.8dB improvement in PSNR compared to [17]. As depicted in Figure 7 and 8(b), for Instant-NGP, NeRF-OPU achieves an average 2.01× speedup and 2.16 × energy efficiency compared to [17]. These improvements occur mainly because the implementation of [17] does not use early ray termination, which can significantly increase rendering speed.

7 CONCLUSION

In this paper, we propose NeRF-OPU, to the best of our knowledge, the first FPGA-based overlay processor for general NeRF accelerations with excellent flexibility and software-like programmability. Our experimental results show that NeRF-OPU achieves on average 1.80×, 1.03× speedup and 6.97×, 4.05× energy efficiency over NVIDIA RTX 3060 Ti on vanilla NeRF and Instant-NGP. Compared with the previous NeRF FPGA accelerator, NeRF-OPU performs on average 2.01× speed up and 2.16× energy efficiency on Instant-NGP.

REFERENCES

[1] Sai Bi et al. 2020. Neural reflectance fields for appearance acquisition. *arXiv preprint arXiv:2008.03824* (2020).
[2] Timothy L. Kay et al. 1986. Ray Tracing Complex Scenes. *SIGGRAPH Comput. Graph.* 20, 4 (1986), 10 pages.
[3] Ruilong Li et al. 2022. Nerfacc: A general nerf acceleration toolbox. *arXiv preprint arXiv:2210.04847* (2022).
[4] Yang Liu et al. 2023. DIF-LUT: A Simple Yet Scalable Approximation for Non-Linear Activation Function on FPGA. In *33rd FPL*. IEEE Computer Society, 322–326.
[5] Ben Mildenhall et al. 2021. Nerf: Representing scenes as neural radiance fields for view synthesis. *Commun. ACM* 65, 1 (2021), 99–106.
[6] Thomas Müller et al. 2022. Instant neural graphics primitives with a multiresolution hash encoding. *ACM ToG* 41, 4 (2022), 1–15.
[7] Sida Peng et al. 2021. Animatable neural radiance fields for modeling dynamic human bodies. In *ICCV*. 14314–14323.
[8] Albert Pumarola et al. 2021. D-nerf: Neural radiance fields for dynamic scenes. In *CVPR*. 10318–10327.
[9] Pratul P Srinivasan et al. 2021. Nerv: Neural reflectance and visibility fields for relighting and view synthesis. In *CVPR*. 7495–7504.
[10] Peng Wang et al. 2021. Neus: Learning neural implicit surfaces by volume rendering for multi-view reconstruction. *arXiv preprint arXiv:2106.10689* (2021).
[11] Yuanfang Wang et al. 2023. Moth: A Hardware Accelerator for Neural Radiance Field Inference on FPGA. In *31st FCCM*. IEEE, 227–227.
[12] Lior Yariv et al. 2021. Volume rendering of neural implicit surfaces. *NIPS* 34 (2021), 4805–4815.
[13] Yunxuan Yu et al. 2019. OPU: An FPGA-based overlay processor for convolutional neural networks. *IEEE VLSI* 28, 1 (2019), 35–47.
[14] Yunxuan Yu et al. 2020. Light-OPU: An FPGA-based overlay processor for lightweight convolutional neural networks. In *FPGA*. 122–132.
[15] Kai Zhang et al. 2021. Physg: Inverse rendering with spherical gaussians for physics-based material editing and relighting. In *CVPR*. 5453–5462.
[16] Xiuming Zhang et al. 2021. Nerfactor: Neural factorization of shape and reflectance under an unknown illumination. *ACM ToG* 40, 6 (2021), 1–18.
[17] Baoze Zhao et al. 2023. A Novel Hardware Accelerator of NeRF Based on Xilinx UltraScale and UltraScale+ FPGA. In *33rd FPL*. IEEE Computer Society, 197–203.