Edge-NeRF: Real-Time Rendering of Instant-NGP on FPGA

ABSTRACT

Neural Radiance Fields (NeRFs) is a novel technique for 3D reconstruction of scenes and objects from a set of images. They have great potential for enhancing the immersive experience of Augmented and Virtual Reality (AR/VR) applications. One of the NeRF algorithms, Instant-NGP(iNGP), leverages multi-resolution decomposition and hash encoding to achieve state-of-the-art(SOTA) performance. However, implementation of iNGP is challenging due to the high memory and computation requirements. In this work, we first profile the iNGP algorithm and identify two bottlenecks. To address these bottlenecks, we propose *Edge-NeRF*, the first software-hardware co-design FPGA-accelerator that achieves real-time rendering on the device. Extensive experiments validate the effectiveness of Edge-NeRF, achieving 28.1× speedup and 289.3× energy efficiency over CPU and 3.9× speedup and 2.8× energy efficiency over GPU while evaluated on a Xilinx Kintex XC7K325T board.

KEYWORDS

FPGA, Hardware Accelerator, NeRF

1 INTRODUCTION

Novel View Synthesis (NVS) is a challenging task that has various applications in 3D reconstruction and Augmented and Virtual Reality (AR/VR). One of the most promising techniques for NVS is Neural Radiance Field (NeRF) [7], an algorithm that can produce photo-realistic images from arbitrary viewpoints. However, NeRF suffers from high memory and computation requirements. To address this issue, Instant-NGP(iNGP) [8] is the current SOTA NeRF algorithm that achieves fast training and rendering. Using mainstream GPUs, instantaneous training times of a few seconds and rendering speeds approaching 60 frames per second (fps) can be achieved [8], which makes it suitable for AR/VR, robotics [2], autopilots [5], and other interactive applications. Real-time rendering of NeRF on edge is highly desirable to fully unleash the potential of NeRF in the emerging interactive applications. However, even with the SOTA NeRF algorithm iNGP, real-time rendering is still difficult, with edge GPU and CPU rendering below 2fps. Therefore, a dedicated accelerator is needed. Instant-3D [4] propose an ASIC accelerator with SRAMs for training iNGP with high computational capabilities, but as for future real-world applications, the high storage requirement of large data makes accessing DRAM inevitable [8]. In this regard, FPGAs are more suitable platforms, as they offer greater adaptability and flexibility than ASICs. Furthermore, NEPHELE [6] uses Perfect Sptial Hash(PSH) [3] to speed up iNGP on the cloud. However, PSH is time-consuming and cannot be extended for multiple resolutions.

By run-time breakdown on CPU, we manage to locate two priority challenges: Interpolation of features from the embedded hash table and MLP. To efficiently tackle the bottlenecks mentioned above, we develop a software-hardware co-design FPGA-based accelerator framework for iNGP inference on edge devices, called *Edge-NeRF*. As the name states, *Edge-NeRF* stands out due to real-time rendering on edge FPGA. We now propose the following contributions in this paper:

- We propose *Edge-NeRF*, an efficient software-hardware co-design FPGA accelerator for iNGP inference. To the best of our knowledge, it is the $first$ edge FPGA-accelerator that achieves real-time rendering of iNGP.
- On the software level, we propose Density-Aware off-Set Hash (DASH),. Our DASH algorithm improves data coherence in iNGP while achieving better image quality.
- On the hardware level, we propose an FPGA-friendly Grids Projection Pipeline (GPP). Our rendering pipeline implements the reuse of hash embeddings of adjacent ray sampling points, which further reduces the bandwidth demand of iNGP.
- We evaluate *Edge-NeRF* on a Xilinx Kintex XC7K325T FPGA board and it achieves a 28.1× speedup and 289.3× energy efficiency compared with Intel Xeon Gold 5218R CPU and 3.9× speedup and 2.8× energy efficiency compared with NVIDIA Jetson Xavier NX-16G GPU.

2 BACKGROUNG AND MOTIVATION

2.1 Background of iNGP

Multi-Resolution. iNGP uses a multi-resolution hash encoding that maps spatial coordinates to trainable feature vectors that can be optimized in the standard flow of NeRF training. For each level of resolution, the generation equation is defined as

$$N_l := \left\lfloor N_{\min} \cdot b^l \right\rfloor, \tag{1}$$

$$b := \exp\left(\frac{\ln N_{\max} - \ln N_{\min}}{L-1}\right), \tag{2}$$

$$\lfloor \mathbf{x}_l \rfloor := \lfloor \mathbf{x} \cdot N_l \rfloor, \lceil \mathbf{x}_l \rceil := \lceil \mathbf{x} \cdot N_l \rceil, \tag{3}$$

where b is the growth factor of resolution between the minimum resolution N_{min} and maximum resolution $N_{max,}$ and

L is the number of levels. The pixel count per level N_l is obtained by taking the floor value of the product of the minimum resolution and the power of the growth factor b to the level. To compute the coordinates x_l at each level, the coordinates x are scaled by the resolution N_l.

Hash Encoding. For each level l, there is a corresponding hash table. If $(N_l + 1)^d \leqslant T$, where T is the maximum number of entries of hash tables, the mapping from vertices to hash tables is one-to-one. Otherwise, a hash function is needed. $\lfloor x_l \rfloor$ and $\lceil x_l \rceil$ form a voxel with 2^d integer vertices, where d is the dimension. The vertices are then mapped to either the hash or one-to-one mapping functions to access the hash tables. The hash function is given by

$$h(\mathbf{x}) = \left(\bigoplus_{i=1}^{d} x_i \pi_i \right) \bmod T, \tag{4}$$

where π_i is a large prime number. The outputs are used as addresses to hash tables to retrieve $F \cdot 2^d$ features, where F is the number of features per vertex. The feature vectors at each vertex are interpolated by the relative position of x within its voxel. The interpolation weight w_l is determined by the fractional part of x_l, expressed as $w_l = x_l - \lfloor x_l \rfloor$.

2.2 Observations of iNGP

We conduct run-time breakdown experiments on CPU (Intel(R) Xeon(R) Gold 5218R) and GPU (RTX-3090). The results are shown in Fig. 1. On CPU, our results show that Interpolation* [1] and MLP dominate the run-time, each taking nearly 40% of the overall time. While on GPU, the run-time is more evenly distributed. The following reasons explain the causes of these two challenges.

Hash Function. iNGP's hash achieve random data distribution. While random data distribution leads to fewer collision counts, it will also result in poor data coherency and low efficiency on edge devices. Furthermore, when two grids with similar density collide, it will also lead to aliasing and deterioration of image quality.

Limited Resources. The performance of iNGP is estimated to be restricted by limited resources on edge FPGA. Aggressive quantification and a high parallel computation scheme need to be adopted to boost hardware efficiency.

3 DENSITY-AWARE OFFSET HASH

To address the problem of hash function,NEPHELE [6] combines the Shape-from-Silhouette algorithm and perfect spatial hash (PSH) [3] to eliminate hash collisions. The hash function is stated as follows

$$h(p) = (h_0(p) + \Phi[h_1(p)]) \bmod \bar{m}, \tag{5}$$

[1]Interpolation* contains trilinear interpolation time and hash table accessing time.

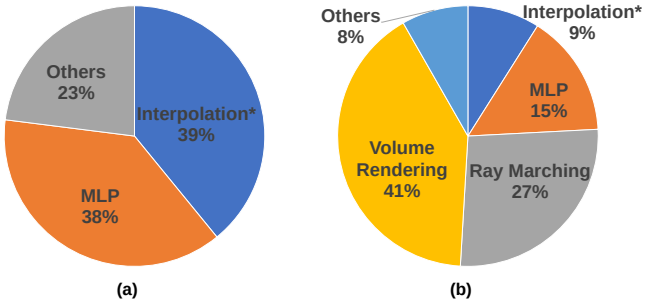

Figure 1: Profile of iNGP. (a) CPU; (b) GPU.

where position p is mapped to the hash table with the size $\bar{m}$ being the hash table size, Φ being the offset table and h_0 and h_1 being the linear transformation matrix. However, PSH is non-trivial and time-consuming, making it impractical to use for high-resolution voxel grids. Moreover, NEPHELE only uses single-resolution PSH since there are no precedent methods to combine multi-resolution hashing with PSH. Without multi-resolution hashing, one cannot utilize both the coarse and fine information in the real world, leading to subpar image quality.

3.1 Multi-Resolution Offset Hash

To address the issues mentioned above, we propose the Density-Aware offSet Hash (DASH) strategy. DASH strategy adopts the same offset hash technique in PSH [3] for base resolution. For hash function in multi resolutions, we propose a simple but efficient method by simply adding a shift., as shown in Fig. 2(a). The distance $shift$ and ratio $scale$ are calculated as

$$shift = x_l \% scale, \tag{6}$$

$$scale = \lceil \frac{N_l}{N_{base}} \rceil, \tag{7}$$

where x_l is the coordinate of the voxel grid. N_l and N_{base} represent the resolution in lth level and base level resolution respectively. Thereby the DASH is computed by

$$h(\mathrm{x_l}) = h_0(\mathrm{x_{base}}) + \Phi\left(h_1(\mathrm{x_{base}})\right) + shift. \tag{8}$$

In DASH, all features inside base level grid are adjacent to each others, leading to superior data coherency.

In the DASH algorithm, a voxel grid occupies all the adjacent slots within a distance of $scale$ in the high-resolution hash table. Therefore, it is reasonable to distribute the voxel grids with similar densities as far as possible, ideally

$$distance(\mathrm{h(x1)}, \mathrm{h(x2)}) \geqslant scale, \tag{9}$$

where $x1$ and $x2$ have similar density. To demonstrate this, we present a three-resolution DASH in Fig. 2(b), where $b = 2$ and the voxel grids in the base level grids are divided into three types according to their densities' magnitude. Red,

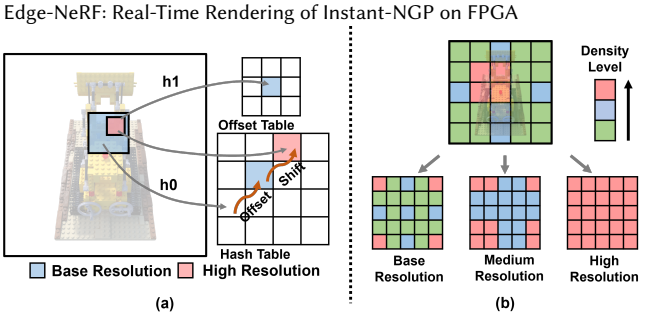

Figure 2: Proposed DASH Algorithm. (a) Multi-Resolution Offset Hash; (b) Three-Resolution DASH.

blue, and green slots represent high-, medium-, and low-density values respectively. Clearly, there is no hash collision between grids with the same density level according to the hash mapping condition presented in Fig. 2. DASH alleviates the aliasing effect caused by the hash collision between two grids with similar density.

3.2 Density-Aware Construction of Offset Table

Throughout the training process of iNGP, the features within the hash table undergo the most substantial updates from the grid exhibiting the highest gradients. As shown in Eq.10

$$\begin{aligned} \frac{\partial \hat{C}(r)}{\partial c_i} &= T_i(1-\exp(-\sigma_i\delta_i)), \\ gradient &\propto (1-\exp(-\sigma_{grid}\delta_{grid})), \end{aligned} \tag{10}$$

where $\frac{\partial \hat{C}(r)}{\partial c_i}$ represents the updating gradient. The construction of the offset table adheres to the principles established in DASH, similar to PSH. Initially, the offset table undergoes sorting based on the summation of gradients associated with its respective grids. Subsequently, a greedy algorithm is employed for the systematic construction of the offset table. To be specific, our approach involves the initial exploration of offset values that map grids to vacant slots within the hash table. These slots are specifically identified by addresses divisible by $S_{max} = \lceil N_{max}/N_{base} \rceil$. In cases where these designated slots are occupied, a sequential attempt is made for slots with addresses divisible by $S_{max}/2$ and subsequently in a similar fashion. The details are shown in Algorithm 1.

Algorithm 1: Construction of offset table

Data: density_grids , sorted_slots, I_max, S_max
Result: An offset table with assigned values

```
for slot in sorted_slots do
    grids ← density_grids that map to slot
    max_grid ← argmax(grids' gradients)
    for I_current in range(I_max) do
        hash_slot ← find_empty_slot(S_max)
        if hash_slot is None then
            S_max ← S_max/2
            hash_slot ← find_empty_slot(S_max)
        end
        offset ← hash_slot − max_grid
        hash_slots ← offset_hash(grids, offset)
        penalty ← sum(collide_grids' gradients *
          collide_slots' gridients)
        collision ← check_collision(offset)
    end
    offset_table[slot] ← offset with min_penalty
    hash_slots ← offset_hash(grids, min_offset)
    hash_slots' gradients += grids' gradients
end
```

4 ARCHITECTURE OVERVIEW

The proposed accelerator architecture is depicted in Fig. 3. We propose an efficient Grid Projection Pipeline(GPP) based on DASH. The GPP is executed through the following steps.

Initialization. The host sorts the density grid cells according to their distance from the camera. and sends the sorted grid coordinates to the FPGA board via the Peripheral Component Interconnect Express (PCIE) protocol.

Offset Hash. Hash module implement the DASH algorithm we propose. The coordinates undergo the Offset Hash module to extract features from DDR and store them in the BRAMs buffers.

Rasterization. The coordinates are also sent to the Rasterization module to determine rays that pass through the density grids. At the same time, rays' *RGB* and σ values will also be stored in a pixel buffer from DDR.

Ray Marching Kernel. The *Ray Marching Kernel* is composed of two modules. One is the *Ray Axis-Aligned Bounding Box (AABB) Intersection* module which determines the fringe of the bounding box by the AABB algorithm. The other is the *Ray Marching Process* module, which calculates the distance from the ray original point t and the locations of sample points. The Ray Marching Kernel samples uniformly in the density grids.

Hash Encoding. The sample points and ray directions will be sent to the two Encoding Units, the *Hash Encoding* and the *Spherical Harmonic (SH) Encoding*. *SH Encoding* is implemented using multipliers and adders following the spherical harmonic equation and the Hash Encoding calculates the $shift$ in Eq. 6.

MLP Cores. *MLP Cores* utilizes multiple *MLP Computing* to obtain the *RGB* and σ values of the sample points, with interpolated features and spherical harmonic(SH) encoded

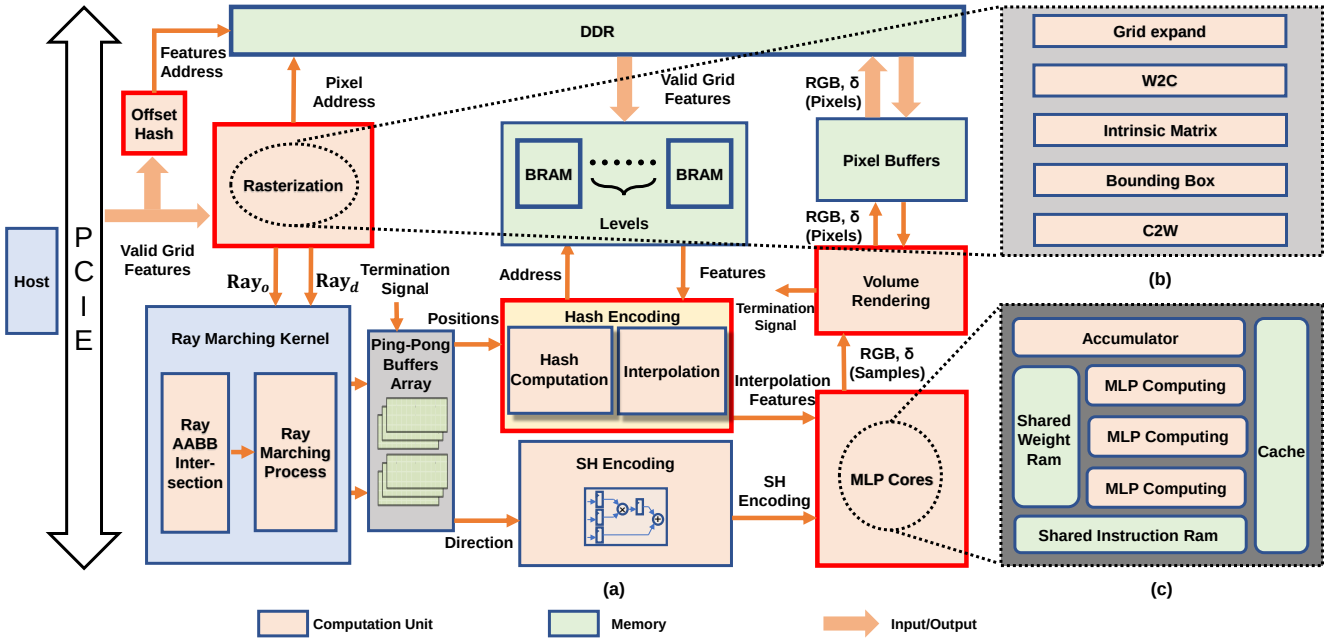

Figure 3: Overview of *Edge-NeRF* architecture. The module underscored with a red highlighted border signifies the central focus of the architecture. (a) Data flow of *Edge-NeRF*; (b) Detailed structure of *Rasterization*; (c) Detailed structure of *MLP Cores*.

rays' directions as inputs. Besides, a simple but efficient instruction set achieves fine-grained control over MLP Cores.

Volume Rendering. *Volume Rendering* perform volume rendering using the *RGB* and σ values stored in the pixels buffer as initial parameters and *RGB* and σ from MLP as inputs. When all the density grids are traversed, the final rendered image is obtained.

5 HARDWARE DESIGN

In this section, we present our proposed accelerator Edge-NeRF modules elucidated in detail as follows.

5.1 Rasterization

As depicted in Fig. 3, the *Rasterization* follows the algorithm of classical rasterization. First, the grid points are expanded to encompass the eight vertices of the respective grid cells. Then, these eight points undergo transformation to the image plane through the world-to-camera (w2c) and intrinsic matrix transformations. By finding the minimum and maximum values of the coordinates of these eight points, we can obtain the bounding box. For the sake of simplicity, it is assumed that all pixel points within the bounding box are valid, and their corresponding pixel values, *RGB*s, and σs are extracted out. Finally, the origin and direction of the rays are calculated by using the camera-to-world (c2w) matrix.

5.2 Offset Hash and Hash Encoding

The hash function is bifurcated into two distinct components: the offset hash and the shift hash. They are implemented by the *offset hash* module and the *hash computation* unit respectively. Moreover, the hash computation unit incorporates a one-to-one mapping strategy for the low-resolution hash table, ensuring that their respective features are comprehensively stored on board.

5.3 MLP Cores

According to our preliminary, MLP computation on edge FPGA becomes the secondary bottleneck that vastly obstructs the high throughput of the accelerator. Our strategy is through high-parallel computation. We propose *MLP Cores* to accelerate the matrix computation of MLP. We also introduce a simple but efficient instruction set to achieve fine-grained control over MLP considering that iNGP uses different sizes of MLP for different applications. .

Core Composition. As illustrated in Fig. 3, the Multi MLP cores comprise multiple processors, each equipped with a cache, an MLP computing unit, and an addressable accumulator. The output of the accumulator is fused with an activation unit (ReLU) as well as requantification unit. The weights and instructions of the MLP are stored in RAM and are shared among all MLP cores.

Table 1: The instruction set of MLP unit (several opcodes are saved for further expansion of the instruction set)

	Functional description	Opcode	address1	address2	address3
load cache	load outside value to cache	000	cache address	null	null
MVM with ReLU	perform MVM, when finished simultaneously ReLU the result in accumulator to the same address as accumulator address in the cache, achieving layer fusion	001	cache address	accumulator address	weight RAM address
MVM	MVM the weight matrix stored in weight RAM and vector stored in cache and accumulate the results in accumulator	010	cache address	accumulator address	weight RAM address
wait	wait so that the pipeline can finish	011	null	null	null
read accumulator	read from the accumulator	100	null	accumulator address	null
load accumulator bias	load bias to accumulator	101	null	accumulator address	bias RAM address

Instruction Set. Table 1 provides a comprehensive overview of the instruction set details. For each layer in the MLP, the corresponding matrices are partitioned into smaller square matrices and similarly, the input and outputs of MLP neurons are segmented into smaller vectors. Within the cache and accumulator, each address stores a vector. The execution of each MVM instruction is pipelined.

5.4 Volume Rendering

The design of the *Volume Rendering* is depicted as Fig. 4. Volume Rendering contains multiplier computations (MUL), exponent (EXP), accumulation (ACC), subtraction(SUB), and comparison (COM) computations. COM delivers a termination signal when a ray's transmittance falls below a specified threshold. Upon receipt of this termination signal, ping-pong buffers array discard the corresponding ray and switch to the next.

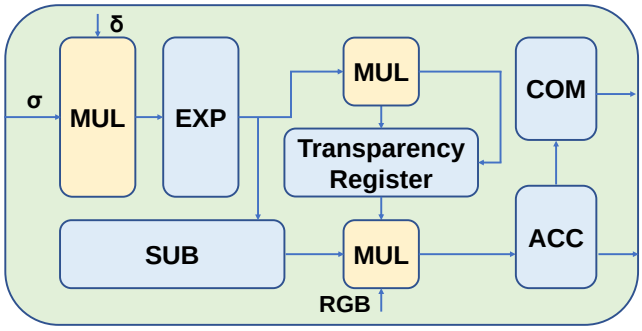

Figure 4: The structure of Volume Rendering.

6 EXPERIMENTS

6.1 Experimental Setup

We implement our proposed accelerator design on Xilinx's Kintex XC7K325T board using Verilog in Vivado design suite 2022.2 and test it on synthetic NeRF datasets. We use the training data and code from taichi-nerf [9], an iNGP implementation based on the taichi framework. The rendered image resolution for the tests was 800×800. For configuration, we use the default in iNGP [8] and for DASH, the offset table size is 41. The MLP cores can compute 16×16 matrix and 16×1 vector MVM operation, and each core has 256 multipliers.

6.2 Algorithm Experiment

6.2.1 *Performance on CPU and GPU.*

We train the original iNGP for 1000 epochs and use its density grid to build an offset table. Furthermore, we terminate the algorithm when all offset table's slots that are divisible by $N_{max}/2$ is built to save time. The $I_{max} = 100$ and for the scene Lego, building the offset table takes about 10 seconds. We compare our result with the original iNGP and multi-resolutions offset hash with the offset table being randomly built. Our experiment shows that even with randomly built offset table, the image quality is similar to iNGP's. With our DASH algorithm, we also achieve less interpolation time and faster rendering speed as well as better image quality compared to iNGP on CPU. The result is shown in Table. 2.

6.3 Hardware Experiment

6.3.1 *Quantum Aware Training.*

Given that MLP constitutes the most computation-intensive aspect of iNGP, we employ Quantum Aware Training (QAT) to quantize the MLP parameters. We observe that the range of MLP layers' inputs is typically 8 times larger than the range of MLP layer's weight, therefore we quantify the inputs and weights into 8-bits and 5-bits respectively. After training the model in FP32, we quantify it and use QAT to fine-tune the model for another 500$\tilde{2}$000 epochs, depending on the scenes.

Additionally, the ray marching as well as spherical harmonic encoding parameters are quantified to 16-bit during training.

6.3.2 *Image Quality.*

We render 200 images corresponding to different poses for each dataset and compare them with those rendered by GPU. The average PSNR is shown in Table 3.

6.3.3 *Resource Utilization, Energy, and Rendering Time Analysis.*

Table 5 summarizes the FPGA resource utilization in our implementation. Considering the constraints of FPGA resources, our accelerator can incorporate four MLP cores

Table 2: Peak signal-to-noise ratio (PSNR), Structure Similarity Index Measure (SSIM), rendering speed, and interpolation time of iNGP implementation, random-built offset table implementation, and our DASH implementation with scene Lego (higher value represents better rendering quality)

	iNGP	Random Offset	Ours
PSNR	34.75	34.63	35.04
SSIM	0.9759	0.9753	0.9767
Rendering Speed (GPU)	18.99 fps	20.24 fps	19.82 fps
Rendering Time (CPU)	2.90 s	2.43 s	2.36 s
Interpolation* (GPU)	4.485×10^{-3}	4.645×10^{-3}	4.659×10^{-3}
Interpolation* (CPU)	1.132s	0.86s	0.85s

Table 3: Peak signal-to-noise ratio (PSNR) of iNGP implementation with GPU and *Edge-NeRF* (higher value represents better rendering quality)

	Lego	Chair	Mic	Materials	Hotdog	Ship
GPU(FP32)	34.76	34.84	35.10	29.81	36.76	29.98
Ours(INT8)	32.41	33.13	33.47	27.35	34.72	28.64

Table 4: Performance comparison of different hardware

	NVIDIA Jetson Xavier NX	CPU (Intel Xeon Gold)	Ours (325T)
Frequency (MHz)	800	2100	160
Bit-width (bits)	FP32	FP32	INT8
Frames per second (fps)	2.45	0.34	**9.56**
Power (W)	**8.695**	125	12.15
DDR	LPDDR4	DDR4	DDR3
Technology	12nm	14nm	28nm

constructed using DSPs and fourteen MLP core utilizing LUTs. For DSPs, DSP packing technique [1] is used to fit more MLP cores.

Table 4 compares our implementation with SOTA edge GPU iNGP as well as CPU implementations. The rendering time specified in Table 4 was evaluated under the image resolution of 800×800. We used jetson-stats to measure GPU power consumption. Our results demonstrate that our accelerator achieves $28.1\times$ speedup and $289.3\times$ energy efficiency compared with Intel Xeon Gold 5218R CPU and $3.9\times$ speedup and $2.8\times$ energy efficiency compared with NVIDIA Jetson Xavier NX-16G GPU despite suffering from inferior DDR and technology.

Table 5: FPGA resource utilization on XC7K325T

	LUT	FF	BRAM	DSP
Ours	152304(75.0%)	89316(21.9%)	283(63.6%)	780(92.8%)

7 CONCLUSION

In this paper, we analyze the profile of iNGP, design an DASH algorithm that improve performance as well as image quality, a Grid Projection Pipeline and propose an FPGA-based accelerator for the iNGP algorithm. With the DASH algorithm and GPP pipeline, we manage to achieve a data reuse between adjacent sample points and significantly alleviate bandwidth requirement. With dedicated acceleration modules, our accelerator achieves on average a $3.90\times$ speedup and $2.8\times$ energy efficiency over the NVIDIA Jetson Xavier NX-16G. We hope our work can open up an exciting perspective towards more hardware architectures dedicated to NeRF and also other 3D reconstruction solutions.

REFERENCES

[1] Yao Fu, Ephrem Wu, Ashish Sirasao, Sedny Attia, Kamran Khan, and Ralph Wittig. 2016. Deep learning with int8 optimization on xilinx devices. *White Paper* (2016).

[2] Justin Kerr, Letian Fu, Huang Huang, Yahav Avigal, Matthew Tancik, Jeffrey Ichnowski, Angjoo Kanazawa, and Ken Goldberg. 2022. Evo-nerf: Evolving nerf for sequential robot grasping of transparent objects. In *6th Annual Conference on Robot Learning*.

[3] Sylvain Lefebvre and Hugues Hoppe. 2006. Perfect spatial hashing. *ACM Transactions on Graphics (TOG)* 25, 3 (2006), 579–588.

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

[5] Zhuopeng Li, Lu Li, and Jianke Zhu. 2023. Read: Large-scale neural scene rendering for autonomous driving. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 1522–1529.

[6] Haimin Luo, Siyuan Zhang, Fuqiang Zhao, Haotian Jing, Penghao Wang, Zhenxiao Yu, Dongxue Yan, Junran Ding, Boyuan Zhang, Qiang Hu, et al. 2023. NEPHELE: A Neural Platform for Highly Realistic Cloud Radiance Rendering. *arXiv preprint arXiv:2303.04086* (2023).

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

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

[9] taichi.graphics. 2023. taichi-nerf. https://github.com/taichi-dev/taichi-nerfs.