Booth-NeRF: An FPGA Accelerator for Instant-NGP Inference with Novel Booth-Multiplier

Zihang Ma*, Zeyu Li*, Yuanfang Wang, Yu Li, Jun Yu, and Kun Wang†
State Key Lab of ASIC & System, Fudan University, Shanghai, China
Email: *{20300750053,20300750066}@fudan.edu.cn, †kun.wang@ieee.org

***Abstract*—Instant-NGP is the state-of-the-art (SOTA) algorithm of Neural Radiance Field (NeRF) and shows great potential to be adopted in AR/VR. However, the high cost of memory and computation limits Instant-NGP's implementation on edge devices. In light of this, we propose a novel FPGA-based accelerator to reduce power consumption, called Booth-NeRF. Booth-NeRF adopts a fully-pipelined technique and is built upon the Booth algorithm. In addition, it introduces a new instruction set to accommodate Multi-Layer Perceptrons (MLPs) of different sizes, ensuring flexibility and efficiency. Moreover, we propose an FPGA-friendly multiplier architecture for matrix multiplication which is capable of performing exact or approximate multiplication using the Booth algorithm and the select-shift-add technique. Evaluations with a Xilinx Kintex XC7K325T board show that Booth-NeRF achieves 2.20× speedup and 1.31× energy efficiency compared with NVIDIA Jetson Xavier NX-16G GPU.**

***Keywords*—** FPGA, Hardware Accelerator, NeRF

I. INTRODUCTION

Neural Radiance Fields (NeRF) [1] represents the state-of-the-art (SOTA) in Novel View Synthesis (NVS), showing great potential for adoption in various applications, e.g., 3D reconstruction and Augmented- and Virtual-Reality (AR/VR). Instant-NGP [2] is recognized as the SOTA NeRF algorithm, integrating multi-resolution decomposition and hash tables encoding to reduce both training and rendering time.

However, the advantages of Instant-NGP come with increased memory and computational requirements. Instant-NGP interpolates NeRF embeddings from numerous hash tables, necessitating considerable computation even to render a single pixel. While high-end GPUs like NVIDIA RTX 3090 can achieve training times of less than one minute and rendering speeds of approximately 60 frames per second(fps), the prohibitive memory bandwidth impedes Instant-NGP's implementation on edge devices. CNN processors [3–5] exploit parallel computing to enhance efficiency by relying heavily on data regularity, but this technique is hard to accelerate Instant-NGP due to the irregular access to hash tables.

To address this challenge, recent works such as Instant-3D [6] propose an ASIC accelerator focused on training Instant-NGP with robust computational capabilities. Nevertheless, for future advancements in NeRF algorithms, adaptability and flexibility [7] are vital in design considerations. In this regard, FPGAs emerge as more suitable platforms, due to they offer greater adaptability and flexibility compared to ASICs.

To explore the probability of accelerating NeRF on edge devices, we develop an FPGA-based accelerator for Instant-NGP inference and make the following contributions:

- We analyze the profile of Instant-NGP and identify its bottlenecks in fetching features from hash tables. To optimize both memory utilization on the chip and bandwidth efficiency, we propose various hash table remapping strategies, customized for low and high resolution hash tables within the FPGA implementation. Moreover, to reduce computations, we propose a feedback mechanism for rapidly switching between different rays, enabling fine-grained parallelism in our design.
- We propose a novel FPGA-friendly approximate multiplier tailored for MLP calculations in the context of Instant-NGP. Our designed multiplier demonstrates a 16% reduction in Look-Up Tables (LUTs) utilization while maintaining comparable latency and accuracy compared with classic baseline booth-truncated multiplier.
- To the best of our knowledge, we are the first to propose an efficient edge FPGA accelerator for Instant-NGP inference, which designs an FPGA-specific architecture to reduce power consumption and latency while retaining certain adaptability and flexibility. Compared to the NVIDIA Jetson Xavier NX-16G GPU, Booth-NeRF achieves a 2.20× speedup and 1.31× energy efficiency on a Xilinx Kintex XC7K325T FPGA board.

II. BACKGROUNG AND MOTIVATION

A. Preliminary of Instant-NGP

To synthesize an image, we first identify a central point of projection of the camera, denoted by $\boldsymbol{o}$. Then, we cast camera rays $\boldsymbol{r}(t)=\boldsymbol{o}+t\boldsymbol{d}$ through each pixel in the image plane, where t is the distance between sampled points along the ray. For the distance of the kth sample away from the original point t_k, we compute the corresponding 3D position along the ray, denoted by $\boldsymbol{p} = \boldsymbol{r}(t_k)$, and check the occupancy grid of $\boldsymbol{p}$. If the occupancy grid of $\boldsymbol{p}$ is true, we transform the 3D position using a hash encoding function $\gamma(\boldsymbol{p})$ defined as

$$\gamma(\boldsymbol{p}) = \mathrm{Interpolate}(\mathrm{Hash(p)}), \tag{1}$$

where $Interpolate$ denotes a trilinear interpolation of adjacent vertices' features from hash tables $Hash(\boldsymbol{p})$. Unlike positional encoding using Fourier features [8], hash encoding uses multiple hash tables and trilinear interpolation to approximate

*These authors contributed equally to this work.
†Corresponding author.

high-frequency functions. For each level of resolution, the generation equation is defined as

$$N_l := \left\lfloor N_{\min} \cdot b^l \right\rfloor, \tag{2}$$

$$b := \exp\left(\frac{\ln N_{\max} - \ln N_{\min}}{L-1}\right), \tag{3}$$

$$\lfloor \mathbf{x}_l \rfloor := \lfloor \mathbf{x} \cdot N_l \rfloor, \lceil \mathbf{x}_l \rceil := \lceil \mathbf{x} \cdot N_l \rceil, \tag{4}$$

where b denotes the growth factor of resolution between the minimum resolution N_{min} and maximum resolution N_{max}, and L denotes the number of levels. The pixel count per level N_l is computed by taking the floor value of the product of two factors, the minimum resolution and the growth factor b raised to the power of the corresponding level. To determine the coordinates x_l at each level, the coordinates x undergo a scaling process using the corresponding resolution N_l.

For each level l, there exists 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 required. $\lfloor \mathbf{x}_l \rfloor$ and $\lceil \mathbf{x}_l \rceil$ span a voxel into 2^d integer vertices, where d represents the dimension. The vertices are then passed to either the hash or one-to-one mapping functions to access the hash tables. The hash function is stated as follows

$$h(\mathbf{x}) = \left(\bigoplus_{i=1}^{d} x_i \pi_i\right) \bmod T, \tag{5}$$

where π_i is a large prime number. The results are sent as addresses to hash tables to get $F \cdot 2^d$ features, where F is the feature number per vertice. The feature vectors at each vertex are subject to trilinear interpolation based on the relative position of x within its hypercube. In other words, the interpolation weight w_l is determined by the fractional part of x_l, expressed as $w_l = x_l - \lfloor x_l \rfloor$. The output density σ and three primary colors c are computed by

$$[\sigma_k, \boldsymbol{c}_k] = \mathrm{MLP}\left(\gamma(\mathbf{p}(\mathrm{t_k})); \Theta\right), \tag{6}$$

where the interpolated result $\gamma(\mathbf{p}(t_k))$ is fed as the input to an MLP with weights Θ. MLP outputs a density value σ and an RGB color $\boldsymbol{c}$. Note that the view direction d is also included as the input to the MLP, but we omit it here for simplicity.

We apply classical volume rendering to compute the pixel color $\hat{C}(\boldsymbol{r})$ for a ray $\boldsymbol{r}$ by integrating the features along the ray. This integration is a sum of the ray samples

$$\begin{aligned} \hat{C}(r) &= \sum_{i=1}^{N} T_i (1 - \exp(-\sigma_i \delta_i)) \boldsymbol{c}_i, \\ T_i &= \exp(-\sum_{j=1}^{i-1} \sigma_i \delta_i), \end{aligned} \tag{7}$$

where N denotes the number of samples; T_i denotes the accumulated transmittance which is the probability of the ray reaching $\boldsymbol{p} = \boldsymbol{r}(t_k)$ without hitting anything.

B. Sampling Strategy of Instant-NGP

Instant-NGP adopts the ray marching strategy with a sparse occupancy grid to record emptiness in space. By adopting this approach, Instant-NGP substantially decreases the number of sample points by skipping the empty grid. To accelerate the process, a technique called digital differential analyzer (DDA) is used to skip the empty grids. However, for GPUs, accessing the occupancy grid becomes the computation bottleneck due to the sparsity of the occupancy grid and irregular access to DDR [9]. While on an FPGA, the occupancy grid can be entirely stored on board, achieving extremely fast irregular access.

C. Storing Strategy of Hash Tables

As shown in Instant-3D [6], memory access to the hash tables stored in DDR is the most severe bottleneck of running Instant-NGP on GPU mainly due to the large storage space and irregular access. Through our observations during inference, we note that the occupancy grid's inherent sparsity leads to a significant portion of the parameters within the hash tables remaining inaccessible, owing to the corresponding vertices inside empty grids. For example, when considering a hyperparameter configuration of $L = 16$, $F = 2$, $Nmin = 16$, $Nmax = 1024$, $T = 2^{19}$, only less than 1.5% of the parameters for $l \leqslant 8$ hash tables can be used and less than 10% of the parameters for $l \leqslant 11$ hash tables can be used during the rendering process.

D. Early Ray Termination

Incorporated within its rendering pipeline, Instant-NGP utilizes early ray termination, in which the volume rendering process ceases when the accumulated transparency T falls below a predefined threshold value. In an ideal scenario, when volume rendering is terminated, a signal should be promptly dispatched to halt all previous computing phases, including sampling, hash encoding, and MLP computations. However, in the case of GPU-based implementations of Instant-NGP, this transition is not instantaneous, as the GPU only verifies the rays' aliveness after completing a batch of computations. Consequently, a portion of the sample points, on average 15.7%, remains unused by the volume rendering process on the synthetic NeRF dataset. Addressing this issue of computation waste during early ray termination represents a critical aspect for optimizing the inference speed of Instant-NGP.

III. HARDWARE DESIGN OPTIMIZATION

A. Hash Tables Remapping

Based on our observation of storage utility in Section II-C, we propose different storing strategies between hash tables of low-resolution and high-resolution to reduce data usage and tackle the limited DDR bandwidth problem.

To efficiently handle hash tables of low resolution, we map the value of hash function $h(\mathbf{x})$ to another smaller value table as the address of the feature. We choose Fibonacci Hashing for the map function due to the continuity brought by the original hash function in Instant-NGP [2]. To resolve potential hash collisions introduced by this mapping during feature retrieval,

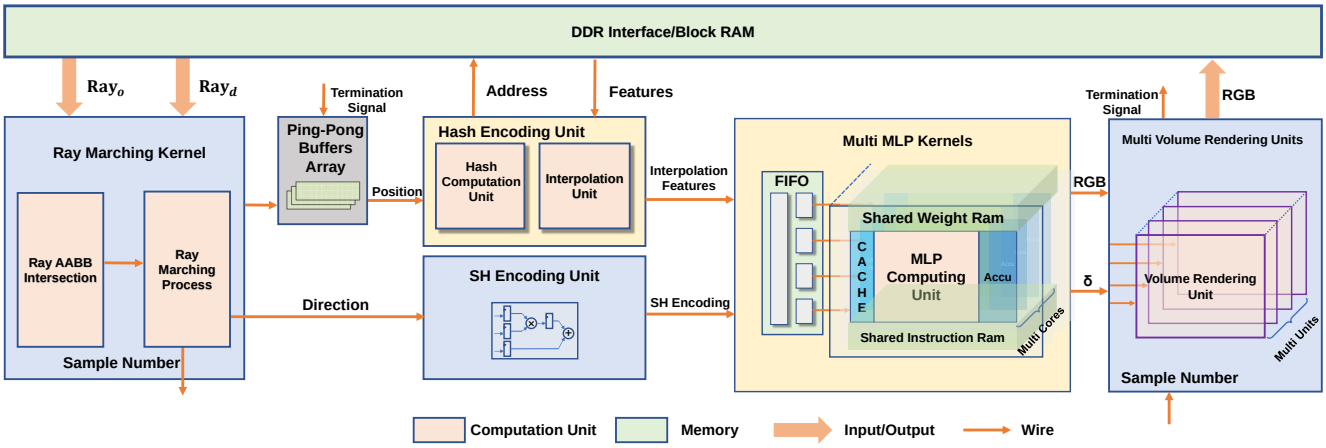

Fig. 1: Overview of our proposed Booth-NeRF accelerator

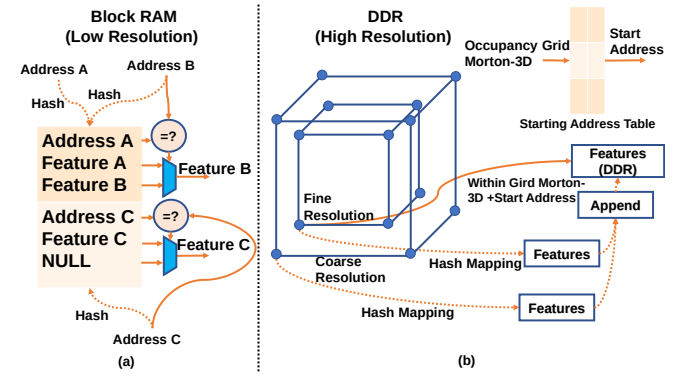

Fig. 2: Hash tables remapping strategy. (a) Storage method of low-resolution hash tables (stored in Block RAM). (b) Storage method of high-resolution hash tables (stored in DDR).

we store the original key along with its corresponding feature, as depicted in Fig. 2. The corresponding feature is selected by checking whether the original keys match the computed keys. By implementing this mapping scheme, we are able to store low-resolution hash tables on board, resulting in a substantial reduction in data storage memory requirements. Our experiment indicates that when the size of the value table is twice the number of accessible vertice features, the final collision probability of our mapping fluctuates around 1‰, resulting in negligible degradation in image quality.

For high-resolution hash tables that exhibit high utility, DDR memory is required, which prioritizes access speed over storage space. To do so, we transform the mapping relation from hash to one-to-one Morton-3D mapping, which is an encoding strategy that offers coherent memory addresses and is also utilized by Instant-NGP. Because the highest-resolution vertices offer the finest data granularity, we integrate the hash tables by appending the features of the highest-resolution vertices to those of their corresponding low-resolution vertices.

Furthermore, to optimize storage usage, we only store the features of vertices that fall within full occupancy grids. To facilitate efficient access and retrieval, we maintain a grid starting address table in DDR. Consequently, when a vertex's position shifts from one grid to another, the starting address table is accessed to determine the new starting address. These data rearrangements significantly improve data coherence between different resolutions and adjacent vertices.

B. Feedback Mechanism

We introduce a novel feedback mechanism to achieve further computational efficiency in the rendering process by implementing early ray termination, as discussed in Section II-D. This mechanism is activated to respond when the rendering process is terminated, triggering the transmission of a feedback signal to the ping-pong buffers, and instructing them to change the buffer which means a transition to the next ray. A new rendering process starts and the entire computing unit's FIFOs and cache are flushed. As a result of this feedback mechanism, significant reductions in MLP computations are realized when compared to those performed by conventional GPUs.

IV. HARDWARE ARCHITECTURE

A. Instant-NGP Accelerator Architecture

The proposed accelerator architecture is depicted in Fig. 1. The accelerator consists of five units: namely ray marching kernel (RMK), hash encoding unit (HEU), spherical harmonic encoding unit (SHEU), multi-MLP cores (MMC), and volume rendering unit (VRU). SHEU is implemented using multipliers following the spherical harmonic equation and VRU is similar to that in ICARUS [10]. Other units will be described in the following subsections. The accelerator performs four steps

1) Sample coordinates with RMK.
2) Interpolate hash-embedded features and apply spherical harmonic encoding with HEU and SHEU.
3) Caculate $\mathbf{c}$ and σ with MMC.
4) Perform volume rendering and feedback ray termination signal with VRU.

B. Ray Marching Kernal

The Ray Marching Kernel implements the sampling of rays (the concept is described in detail in Section II-B). This kernel is composed of two essential components. The first component

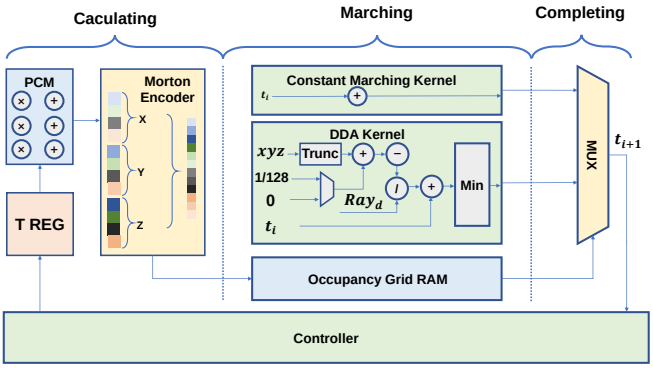

Fig. 3: Ray Marching Kernel design

is the Ray Axis-Aligned Bounding Box (AABB) Intersection module, which leverages the AABB algorithm to determine the range of rays contained within the scene's bounding box.

The second component is a pipelined module to step through each sampling point along a ray. As shown in Fig. 3, the second part comprises three stages

1) Calculating: At the initial stage, the Position Calculating Module (PCM) computes the coordinate of the sampling point using the equation $\boldsymbol{r}(t)=\boldsymbol{o}+t\boldsymbol{d}$. Subsequently, the computed coordinates are passed to the Morton 3D encoder. The hardware-based implementation of the Morton 3D encoder offers greater simplicity compared to the software-based approach, employing a straightforward concatenation instead of shift and add operations.
2) Marching: The second stage executes concurrent processes involving $t+dt_{\text{constant}}$ and $t+dt_{\text{DDA}}$ computations. Here, dt_{constant} corresponds to the constant marching step length, while dt_{DDA} represents the step length determined by the Digital Differential Analyzer (DDA).
3) Completing: In the final stage, the value obtained from the occupancy grid in the preceding stage dictates the next sampling point starts from whether $t + dt_{\text{constant}}$ or $t + dt_{\text{DDA}}$. If the retrieved occupancy grid value is 0, the step length is set to dt_{DDA}, and the Controller module receives a signal to flush the previous pipeline stages and assign the register of t to $t + dt_{\text{DDA}}$, thus efficiently skipping empty grid regions. Otherwise, the register of t is updated merely by $t + dt_{\text{constant}}$.

C. Hash Encoding Unit

Upon observation of Eq. 4 and 5, we can efficiently compute the hash equation using only two Multiply-Accumulate (MAC) operations. Specifically, the process requires multiplying $\lfloor \mathbf{x}_l \rfloor$ with π_i and subsequently adding the resulting product to π_i, thereby generating the terms in Eq. 5. This approach reduces computational overhead and enhances the efficiency of the hashing process. By employing two hash table ports implemented using block RAM and DDR, the concurrent transmission of features from both vertices becomes feasible. Consequently, the interpolation phase benefits from a streamlined pipelining process, as depicted in Fig. 4.

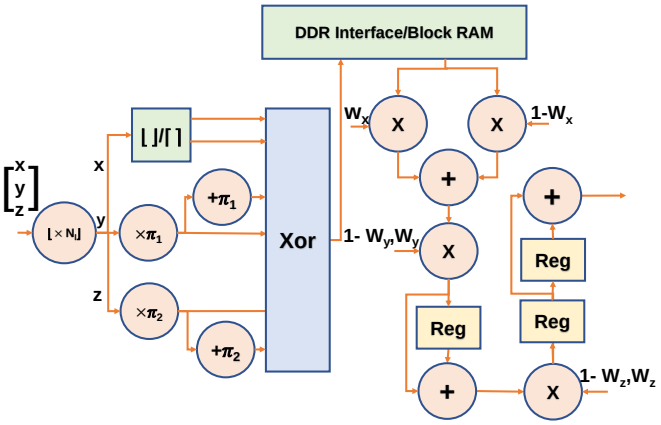

Fig. 4: The architecture of Hash Encoding Unit

D. Multi MLP Cores

Instant-NGP uses different sizes of MLP for different applications [2]. Therefore, we design a dedicated MLP core with a novel instruction set to be used for any size of MLP.

Core architecture. As shown in Fig. 1, the Multi MLP cores consist of multiple processors, each having a cache, an MLP computing unit, and an addressable accumulator whose output is fused with an activation unit (ReLU). The weights and instructions of the MLP are stored in RAM and are shared by all MLP cores. These cores can process multiple samples in parallel at the same time.

Multipliers. We propose a booth-based reconfigurable multiple constant multipliers (booth-RMCM) for an FPGA-friendly multiplier design. RMCM is used by ICARUS [10] to build matrix-vector multiplication (MVM) units for MLP acceleration. RMCM reduces hardware costs by sharing common subexpressions across different multipliers. In ICARUS, a shift-add technique is used to generate shared subexpressions of the multiplicand. For example, $7x = (1x << 3) - 1x$ realizes $7x$ using only one shift-add operation. With this technique, all odd partial products (OPPs) smaller than $8x$ can be generated with no more than one shift and add operation. All products smaller or equal to $8x$ can be generated from the OPPs by using only one shift operation. However, ICARUS uses a simple bisection method to produce a partial product. For a 9 bit multiplier, ICARUS separates it into upper 4 bits and lower 4 bits, and a sign bit, thus all partial products smaller than 16x are needed, while partial products larger than 9x usually cannot be provided with a single shift operation from OPPs..

To overcome this limitation, we utilize the radix-8 booth algorithm as shown below (y being the multiplier)

$$\begin{aligned} Y =&(4 \times y_{n-2} + 2 \times y_{n-3} + y_{n-4} + y_{n-5} - \\ &8 \times y_{n-1}) \times 2^{n-4} \\ &+(4 \times y_{n-6} + 2 \times y_{n-7} + y_{n-8} + y_{n-9} - \\ &8 \times y_{n-5}) \times 2^{n-8} \\ &+ \cdots\cdots, \end{aligned} \tag{8}$$

where the multiplicand is separated into the sum of partial products. The partial product is within the range of $(-8x, 8x)$

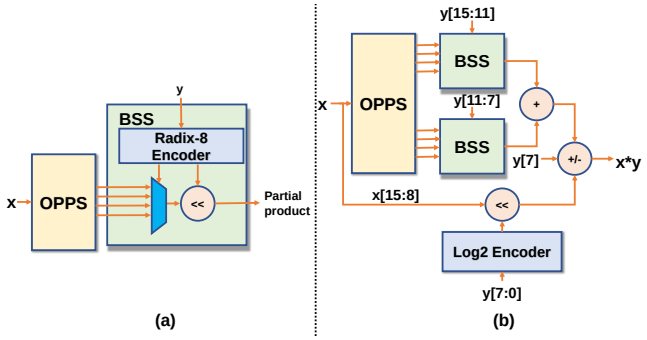

Fig. 5: (a) Booth-RMCM (b) Approximate multiplier

for the radix-8 booth algorithm and can be generated with only select and shift operations from OPPs. We call this process booth-controlled select shift (BSS). Radix-8 encoders realize the function inside the parenthesis in Eq. 8. Radix-8 encoder is FPGA-friendly because it can be directly mapped to LUT5. The structure diagram of booth RMCM is shown in Fig. 5 (a).

Approximate multipliers. Since the NeRF neural networks are very fault-tolerant, we develop an approximate multiplier based on the booth-RMCM architecture. With 16-bit multiplication, the numbers can be separated into the upper half (a and c) and the lower half (b and d). The basic approximate equation is

$$\begin{aligned}
&\left(\boldsymbol{a}\times 2^8+\boldsymbol{b}\right)\left(\boldsymbol{c}\times 2^8+\boldsymbol{d}\right)\\
&=\boldsymbol{ac}\times 2^{16}+\boldsymbol{bc}\times 2^8+\boldsymbol{ad}\times 2^8+\boldsymbol{bd}\\
&\approx\begin{cases}
\left(\boldsymbol{a}\times 2^8+\boldsymbol{b}\right)\times\boldsymbol{c}\times 2^8+(\boldsymbol{a}<<\boldsymbol{dq})\times 2^8, & \text{if}(\boldsymbol{d}<2^7)\\
\left(\boldsymbol{a}\times 2^8+\boldsymbol{b}\right)\times(\boldsymbol{c}+1)\times 2^8- & \\
(\boldsymbol{a}<<\boldsymbol{dq'})\times 2^8-\boldsymbol{b}\times 2^8, & \text{if}(\boldsymbol{d}>2^7)
\end{cases}
\end{aligned} \tag{9}$$

where dq denotes the positive quantization part of the variable q, while dq' represents the reverse quantization process of the variable q

$$\boldsymbol{dq}=\lfloor\log_2(\boldsymbol{d})\rfloor \text{ and } \boldsymbol{dq'}=\lfloor\log_2\left(2^8-\boldsymbol{d}\right)\rfloor. \tag{10}$$

The hardware structure that implements this approximate method is shown in Fig. 5 (b). We can see from Eq. 8 that there is a redundant bit y_{-1} for an 8-bit y. Instead of setting it to zero, we can plug in the highest bit of d to achieve the transformation from c to $c+1$. To achieve the transformation between addition and subtraction, we use a configurable adder/subtractor module. The $\log_2$ encoder is implemented using look-up tables. Again, the highest bit of d is plugged into the module. The $b\times 2^8$ part is omitted because it is much smaller than the error incurred by the log decoder.

Instruction set. Table I shows the details of the instruction set. All matrices corresponding to each layer in MLP are split into smaller square matrices and stored inside the weight RAM. The outputs of neurons of MLP are also divided into smaller vectors with the same dimension as square matrices. Each address in the cache and accumulator stores a vector. Each MVM instruction takes several clocks to finish (the multipliers and adder trees are pipelined to increase frequency). Multiple MVM instructions can be issued in series. The computation flow is shown as follow

1) When the inputs for a layer are ready: MVM the layer matrix square by square from left to right, moving to the next row when the end arrives. Repeat the process until all the MVM instructions for the layer are issued.
2) When some inputs of the layer are not ready: MVM the layer matrix square by square from top to bottom corresponding to the columns that multiply valid vector.
3) When two MVM instructions can be issued at the same time: issue the MVM instruction corresponding to the layer closer to the MLP input.

V. EXPERIMENTS

A. 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 [11], an Instant-NGP implementation based on the taichi framework, and quantize the parameters to 16-bit. The rendered image resolution for the tests was 800×800. For configuration, $L=16, F=2, N_{min}=16, N_{max}=1024$, and $T=2^{19}$. The size of MLP is the default size in the original Instant-NGP experiment [2]. The MLP cores can compute 16×16 matrix and 16×1 vector MVM operation. Each core has 256 multipliers.

B. 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 II, revealing only minor discernible differences between them. The PSNR values indicate that our approach achieves results comparable to those obtained using a GPU. We also compare our proposed accelerator with a version that utilizes approximate multipliers and found that the approximate strategy has little impact on the PSNR values of rendered images.

C. Resource Utilization, Energy, and Rendering Time Analysis

Table IV summarizes the FPGA resource utilization in our implementation. Considering the constraints of FPGA resources, the exact accelerator can incorporate two MLP cores constructed using DSPs and one MLP core utilizing LUTs. The approximate accelerator can leverage two MLP cores built with DSPs and seven MLP cores implemented with LUTs. This approach leads to a substantial increase in the number of MLP cores by approximate computation. Furthermore, we divide the resolution into $l\leqslant 10$ and $l>10$ for the hash tables remapping strategy mentioned in Section III-A. Table III compares our implementation with SOTA edge GPU Instant-NGP implementations. The rendering time specified in Table III 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 a speedup of 2.20× over NVIDIA Jetson Xavier NX and 1.31× energy efficiency despite suffering from inferior DDR and technology.

TABLE I: 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

TABLE II: Peak signal-to-noise ratio (PSNR) of Instant-NGP implementation with GPU and our accelerator (higher value represents better rendering quality)

	Lego	Chair	Mic	Materials	Hotdog	Ficus	Drums
GPU(FP32)	35.13	34.75	34.92	29.14	36.52	32.89	25.77
Ours(INT16)	34.24	34.56	34.57	28.73	36.20	32.54	25.47
Ours(INT16) (approximate)	33.83	34.34	34.59	28.61	35.43	32.52	25.51

TABLE III: Performance comparison of different hardware

	NVIDIA Jetson Xavier NX	Ours (325T)	Ours(appr) (325T)
Frequency (MHz)	800	160	180
Bit-width (bits)	FP32	INT16	INT16
Frames per second (fps)	2.45	1.79	**5.38**
Power (W)	**8.695**	10.55	14.67
DDR	LPDDR4	DDR3	DDR3
Technology	12nm	28nm	28nm

TABLE IV: FPGA resource utilization on XC7K325T

	LUT	FF	BRAM	DSP
Ours	120938(59.3%)	87679 (21.5%)	381(85.6%)	787(93.7%)
Ours(appr)	184120(90.2%)	158662(38.9%)	385(86.5%)	787(93.7%)

TABLE V: Comparison with different multiplier designs

	ERROR MEAN	MAX ERROR	ERROR PROB	MIN ERROR	LUT	Latency
Ours	174632	1048448	0.94	1	168	7ns
TBM-16	185605	461487	0.99	64	200	7ns

D. Evalution of Our Multiplier

We conduct an evaluation of our approximate multiplier by comparing the synthesized Look-Up Table (LUT) resources and accuracy with the truncated Booth multiplier with vbl=16 (TBM-16) [12], a multiplier that is generally considered as a baseline for comparing the Booth multipliers [13]. This evaluation was performed using Vivado 2022.2. The results are shown in Table V. Under similar accuracy, our design utilizes about 16% less LUTs than the TBM-16 design.

VI. CONCLUSION

In this paper, we analyze the sampling process of Instant-NGP, design an FPGA-friendly approximate multiplier, and propose an FPGA-based accelerator for the Instant-NGP algorithm. With the feedback mechanism and novel storage scheme, we manage to achieve a nearly seamless transition from ray termination to the next ray and significantly improve storage utility. With dedicated acceleration modules, our accelerator achieves on average a 2.20× speedup and 1.31× 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.

VII. ACKNOWLEDGEMENT

This work was financially supported by National Key Research and Development Program of China under Grant 2021YFA1003602, Shanghai Pujiang Program under Grant 22PJD003, FDUROP (Fudan Undergraduate Research Opportunities Program)(22816) and National Undergraduate Training Program on Innovation and Entrepreneurship grant(202310246168).

REFERENCES

[1] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "Nerf: Representing scenes as neural radiance fields for view synthesis," *Communications of the ACM*, vol. 65, no. 1, pp. 99–106, 2021.

[2] T. Müller, A. Evans, C. Schied, and A. Keller, "Instant neural graphics primitives with a multiresolution hash encoding," *ACM Transactions on Graphics*, vol. 41, no. 4, pp. 1–15, 2022.

[3] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He, "Opu: An fpga-based overlay processor for convolutional neural networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 1, pp. 35–47, 2019.

[4] Y. Yu, T. Zhao, K. Wang, and L. He, "Light-opu: An fpga-based overlay processor for lightweight convolutional neural networks," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 122–132.

[5] Y. Yu, T. Zhao, M. Wang, K. Wang, and L. He, "Uni-opu: An fpga-based uniform accelerator for convolutional and transposed convolutional networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 7, pp. 1545–1556, 2020.

[6] S. Li, C. Li, W. Zhu, B. Yu, Y. Zhao, C. Wan, H. You, H. Shi, and Y. Lin, "Instant-3d: Instant neural radiance field training towards on-device ar/vr 3d reconstruction," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

[7] Y. Wang, Y. Li, H. Zhang, J. Yu, and K. Wang, "Moth: A hardware accelerator for neural radiance field inference on fpga," in *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2023, pp. 227–227.

[8] M. Tancik, P. Srinivasan, B. Mildenhall, S. Fridovich-Keil, N. Raghavan, U. Singhal, R. Ramamoorthi, J. Barron, and R. Ng, "Fourier features let networks learn high frequency functions in low dimensional domains," *Advances in Neural Information Processing Systems*, vol. 33, pp. 7537–7547, 2020.

[9] C. Li, S. Li, Y. Zhao, W. Zhu, and Y. Lin, "Rt-nerf: Real-time on-device neural radiance fields towards immersive ar/vr rendering," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022, pp. 1–9.

[10] C. Rao, H. Yu, H. Wan, J. Zhou, Y. Zheng, M. Wu, Y. Ma, A. Chen, B. Yuan, P. Zhou *et al.*, "Icarus: A specialized architecture for neural radiance fields rendering," *ACM Transactions on Graphics*, vol. 41, no. 6, pp. 1–14, 2022.

[11] taichi.graphics, "taichi-nerf," https://github.com/taichi-dev/taichi-nerfs, 2023.

[12] F. Farshchi, M. S. Abrishami, and S. M. Fakhraie, "New approximate multiplier for low power digital signal processing," in *Proceedings of the 17th CSI International Symposium on Computer Architecture & Digital Systems*. IEEE, 2013, pp. 25–30.

[13] H. Jiang, F. J. H. Santiago, H. Mo, L. Liu, and J. Han, "Approximate arithmetic circuits: A survey, characterization, and recent applications," *Proceedings of the IEEE*, vol. PP, no. 99, pp. 1–28, 2020.