Accelerating NeRFs: Optimizing Neural Radiance Fields with Specialized Hardware Architectures

William Shen*, Willie McClinton*
MIT CSAIL

***Abstract*—Neural Radiance Fields (NeRFs) have recently gained widespread interest not only from computer vision and graphics researchers, but also from cinematographers, visual effects artists, and roboticists. However, their high computational requirements remain a key bottleneck which limit their widespread use. Despite substantial software and algorithmic improvements, limited attention has been paid to the hardware acceleration potential of NeRFs. We aim to explore this untapped potential and conduct an in-depth profile of MLP-based NeRFs. We identify that the input activations of the fully-connected (FC) layers have an average sparsity of 65.8% due to the use of ReLUs, and the weights of ray samples for volumetric rendering have an average sparsity of 33.8%. We exploit these sparsities using an Eyeriss-based architecture with sparse optimizations, resulting in over 50% improvements in performance and energy for the MLP. Finally, we study post-training FP16 quantization on a GPU, resulting in 2.7× and 3.1× improvements in rendering speed and energy consumption, respectively. Our proposed methods demonstrate the potential for hardware acceleration to significantly speed up NeRFs, making them more accessible for a wider range of applications and low-compute devices.**

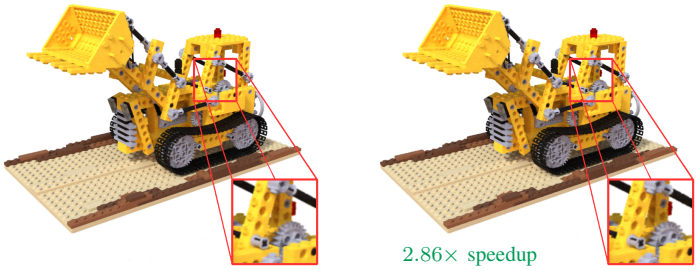
Website: https://williamshen-nz.github.io/accelerating-nerfs

I. Introduction

Neural Radiance Fields (NeRFs) have recently gained widespread interest not only from computer vision and graphics researchers, but also from cinematographers, visual effects artists, and roboticists. The intrinsic advantages of NeRFs lie in their ability to compactly represent 3D scenes compared to traditional methods such as point clouds or meshes, and render novel viewpoints that capture view-dependent lighting [11].

However, their high computational requirements remain a key bottleneck which limit their widespread use, often requiring modern GPUs with substantial VRAM. Accelerating NeRFs would enable several impactful applications including real-time rendering on low-throughput devices such as AR/VR headsets [6], integration as digital assets in game engines [9], and revolutionizing visual effects [4]. Despite remarkable progress, efforts to improve training and inference speeds have focused primarily on algorithmic approaches. The original NeRF [11] used a large Multilayer Perceptron (MLP) to model scenes, requiring up to 24 hours of training on a GPU. Subsequent research has traded off time and space by combining explicit data structures with smaller MLPs, such as multiresolution hash encodings in Instant NGP [12] and voxel grids [16]. Alternative architectures have explored improving inference speed, using thousands of tiny MLPs [15] or extracting explicit 3D representations such as polygons [3].

Our project addresses the underexplored area of the potential of hardware acceleration for NeRFs, particularly during inference. We accomplish this by 1) identifying components of NeRF that can be optimized and accelerated, 2) designing hardware architectures to address these bottlenecks, and 3) assessing the downstream impact on performance in terms of NeRF rendering quality, throughput, and energy.



(a) FP32 (PSNR = 35.13) (b) FP16 (PSNR = 32.78)

Fig. 1: **NeRF Quantization.** We compare renders from a trained NeRF using (a) its original FP32 representation, and (b) its FP16 quantized version. We observe a 2.86× rendering speedup over the Lego scene at the cost of reduced PSNR. The visual differences are not obvious unless we pixel peep.

Through an in-depth profile of MLP-based NeRFs, we identify that the input activations of fully-connected (FC) layers exhibit an average sparsity of 65.8% due to the use of ReLU activation functions, while the weights of ray samples for volumetric rendering have an average sparsity of 33.8%. We exploit these sparsities using an Eyeriss-based architecture with sparse optimizations, resulting in 60% and 51% improvements in cycles and energy for the MLP, respectively. We also investigate post-training FP16 quantization on a GPU, achieving 2.7× and 3.1× improvements in rendering speed and energy consumption, respectively. Our insights emphasize the potential of hardware acceleration in making NeRFs more efficient and accessible, contributing to their development as a practical and scalable 3D representation for a variety of applications and low-compute devices.

II. Background

We target the original MLP architecture [11] with twelve FC layers as described in Figure 2. While recent network architectures have shown substantial speed improvements, they rely on optimized CUDA implementations and data structures [12], [16] which are infeasible to model using Timeloop and Accelergy, and are hence outside the scope of this project.

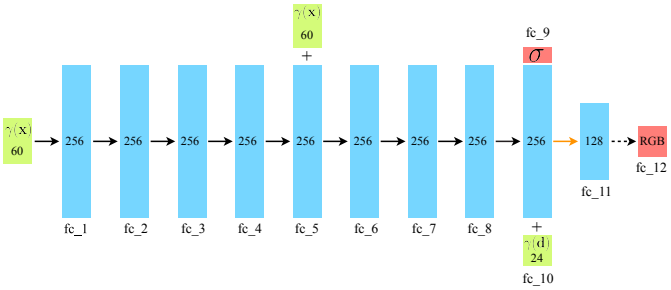


Fig. 2: **NeRF MLP.** The fully-connected (FC) architecture we use (modified from Fig. 7 in [11]). Green blocks indicate input tensors, blue indicates hidden FC layers, and red indicates output FC layers. Black arrows indicate layers with ReLU activation functions, while orange indicates no activation. The numbers indicate the dimensionality (dim) of the tensors for green blocks, and the output dim for FC layers. The σ and RGB red blocks have an output dim of 1 and 3, respectively. $\gamma(\cdot)$ represents positional-encoding, see [11] for details.

NeRFs learn a continuous representation of a 3D scene from a set of 2D RGB images with known camera intrinsics and poses. Formally, a NeRF maps a 3D position $\mathbf{x} = (x, y, z)$ and viewing direction $\mathbf{d} = (d_x, d_y, d_z)$ into a volume density using the density field $\sigma(\mathbf{x})$, and a view-dependent color using the color field $\mathbf{c}(\mathbf{x}, \mathbf{d}) = (r, g, b)$. NeRF renders a pixel by sampling N points along a ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$, where $\mathbf{o}$ is the camera center and t is the distance from the camera. The color $\hat{\mathbf{C}}(\mathbf{r})$ of the pixel is estimated using a quadrature approximation [11] to the volumetric rendering integral [10]:

$$\hat{\mathbf{C}}(\mathbf{r}) = \sum_{n=1}^{N} \underbrace{T_n(1 - \exp(-\sigma(\mathbf{x}_n)\delta_n))}_{\text{density-based weight}} \cdot \underbrace{\mathbf{c}(\mathbf{x}_n, \mathbf{d})}_{\text{color}}, \quad (1)$$

where $T_n = \exp(-\sum_{j=1}^{n-1} \sigma(\mathbf{x}_j)\delta_j)$ is the accumulated transmittance, $\mathbf{x}_n = \mathbf{r}(t_n)$ is a point sampled along ray $\mathbf{r}$, and $\delta_n = t_{n+1} - t_n$. During training, NeRF samples batches of rays from the training images and optimizes the photometric loss. During inference, NeRF renders an image by generating all the rays corresponding to the desired camera parameters, and uses volumetric rendering to estimate the color of each pixel. We refer the reader to [11] for a deep dive into NeRFs.

III. Related Work

In this work, we aim to accelerate NeRF inference by leveraging hardware architectures, a topic that has seen limited but growing interest. RT-NeRF [6] focuses on grid-based NeRFs (i.e., TensoRF [1]), and uses hybrid encodings for sparse grid embeddings to enable real-time rendering, while [19] proposes a resistive random access memory (RRAM)-based accelerator and leverages the parallel dataflow of MLP-based NeRF rendering to increase RRAM utilization. Our work is most similar to ICARUS [14], an accelerator architecture for MLP-based NeRFs which avoids off-chip data movement using custom positional encoding units, a MLP engine, and volumetric rendering units. Two papers were released after our project proposal submission. GenNeRF [5] focuses on generalizable NeRFs and replaces the transformer with *Ray-Mixers* and exploits geometric relationships across rays to maximize data reuse. Instant-3D accelerates grid-based NeRFs by decomposing the embedding grid into color and density grids, and uses a mapper to maximize SRAM reuse [8].

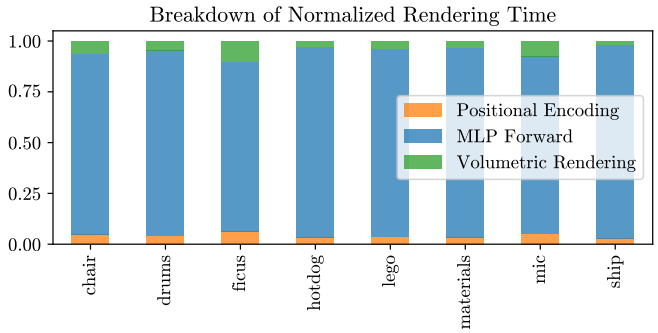


Fig. 3: **Breakdown of NeRF render time.** We present the normalized breakdown of the three main components of NeRF over the synthetic benchmark. See Section IV-A for details.

In comparison, we target MLP-based NeRF inference and perform an in-depth profile to uncover patterns which can be exploited. Different to [14], [19] which also target MLPs, we exploit the sparsity of the activations exhibited within each FC layer. We believe our work is valuable given the limited existing research, helping shed light on accleration approaches.

IV. Technical Contributions

Firstly, we conduct an extensive profile of our NeRF MLP architecture depicted in Figure 2. We then explore avenues for acceleration and demonstrate significant improvements in metrics including performance and energy.

A. NeRF Setup and Profiling

We train NeRFs on all 8 scenes of the synthetic benchmark [11] using a batch size of 1024 rays for 50000 steps. We modify the reference implementation provided by Nerfacc [7] (details in Appendix A). We evaluate our NeRFs by rendering 200 test frames on each scene in the benchmark at a 400×400 resolution. We profile the time required for 1) positional encoding of ray samples, 2) MLP forward pass, and 3) volumetric rendering. Our results in Figure 3 show that the majority of rendering time may be attributed to the MLP forward pass, with volumetric rendering coming in second. We focus our efforts on accelerating these two components.

We additionally analyze the sparsities of the input activations and weights of each fully-connected (FC) layer. We find that while the weights have zero sparsity, the input activations exhibit high sparsity due to the use of the ReLU activation (see Figure 4), with an average sparsity of 65.8% excluding `fc_1` and `fc_11`. Figure 5 shows that the density-based weights used by volumetric rendering (see Eq. 1) also exhibit an average sparsity of 33.8%. Recall NeRF renders a pixel by sampling multiple points along a ray; many points have zero density as they represent free space and can be skipped.

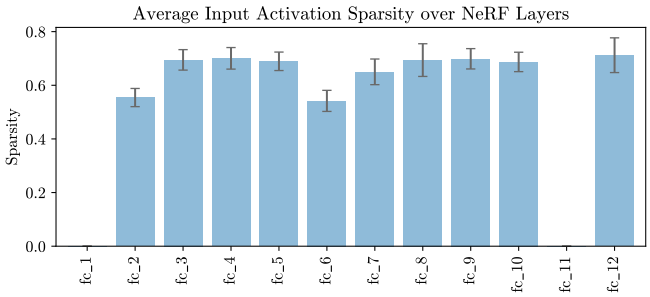


Fig. 4: **Activation Sparsity of NeRF MLP Layers.** We show the average and standard deviation of the input activation sparsities of the FC layers averaged over renders on the synthetic benchmark [11]. `fc_1` receives position-encoded inputs, while `fc_11` receives the outputs from `fc_10` which has no activation function. See Figure 8 for per-scene breakdown.

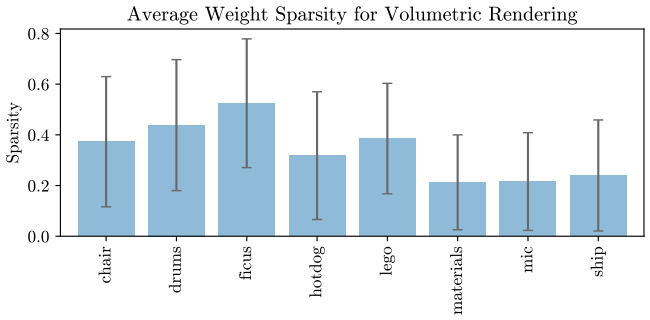


Fig. 5: **Volumetric rendering weight sparsity.** We show the average and standard deviation of the sparsity of the weights in the ray samples for volumetric rendering. We find that sparsity is scene dependent, and the overall mean sparsity is 33.8%.

	Energy (μJ)	Cycles	EDP (J $*$ cycle)	Area (mm^2)
Eyeriss	650.93	568192	3.70e8	16.51
w/ Compression only	416.01	568192	2.36e8	10.89
w/ Skipping only	386.28	307616	1.19e8	10.89
w/ Gating only	336.59	307616	1.04e8	10.25
w/ Skipping	316.71	227813	7.22e7	10.89
w/ Gating	313.05	307611	9.63e7	10.25

TABLE I: **Eyeriss - Activation Sparsity.** We show the results of Eyeriss with several ablations to exploit the activation sparsities of the FC layers. 'w/ Skipping' and 'w/ Gating' use both onchip compression and skipping and gating, respectively.

Overall, our NeRF profile unveiled valuable insights into computational costs and sparsity patterns, guiding our choice of components to accelerate in the following sections.

B. Exploiting Activation Sparsity

The sparsity exhibited by the activations of the FC layers result in ineffectual computations which we can exploit at the hardware level. We experiment with Eyeriss, a convolutional neural network (CNN) accelerator that reduces energy consumption while maximizing performance [2]. Although Eyeriss was designed for CNNs, we use it for NeRFs by mapping the FC layers to 1×1 convolutions. The accelerator achieves improved performance and energy through a row stationary dataflow that maps workloads onto a 2D grid of 168 PEs which process simultaneously to maximize data reuse and reduce off-chip accesses. Eyeriss also supports sparse optimizations which are beneficial for our use case including onchip compression and decompression of sparse input and output activations to and from the on-chip global buffer. It can avoid ineffectual computations by either gating for keeping hardware idle, or skipping to the next effectual computation.

We use the average sparsity across scenes as input to Sparseloop [18] and Accelergy [17] with a batch size of 128 ray samples to find mappings and estimate energies for our sparse designs, as the sparsities are relatively consistent across the benchmark (Figure 8). Following Sparseloop naming conventions, we use Uncompressed Offset Pairs-Run Length Encoding (UOP-RLE) compression of the sparse activations. This leads to lower memory requirements and ideally energy consumption. We use Timeloop [13] to evaluate a dense baseline Eyeriss design without sparse optimizations.

Table I presents our results across the baseline Eyeriss and various ablations that exploit sparse optimizations including onchip compression, skipping and gating. We find that compression alone significantly reduces the energy by 36.1% through reducing the amount of data that needs to be moved around. The improvements from using skipping or gating only lead to better results by avoiding ineffectual computations, as demonstrated by the reduction in cycles by 45.9%. The overhead of the additional hardware required by skipping is shown by the 50μJ increase in energy required by skipping versus gating only. We observe a further reduction in the cycles and energy when we combine sparse compression with skipping, with a 60% and 51.3% reduction in cycles and energy compared to dense Eyeriss, respectively. This combination results in the lowest EDP and potentially increases the throughput as fewer cycles need to be processed within a given period of time. In fact, there is a 25.9% decrease in cycles compared to skipping only, demonstrating that the sparse compression allows us to efficiency encode the pattern of zeros in a manner that is amenable for the underlying skipping hardware and PE grid.

Our results demonstrate the clear benefit of exploiting the sparsity of the activations in the FC layers at a hardware-level. We achieve significant gains in energy and cycles by avoiding ineffectual computations and using sparse encodings to compress the activations. However, it is also important to note that skipping and gating introduce additional complexity to the hardware. These overheads may trump potential improvements when the activations exhibit low sparsity. We found this was the case for `fc_1` which has near-zero activation sparsity.

C. Accelerating Volumetric Rendering

Volumetric rendering is fundamentally a weighted sum of the density-based weights and RGB values, as shown in Equation 1. This can be represented as a dot product between

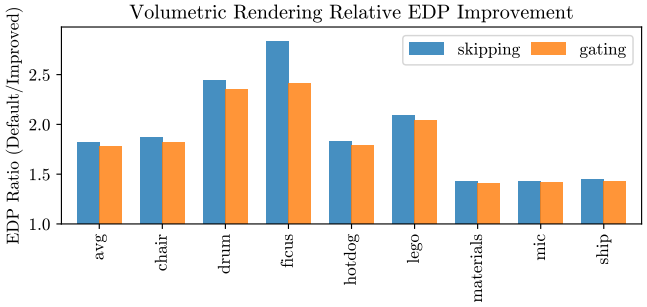


Fig. 6: Volumetric Rendering EDP Improvement by scene.

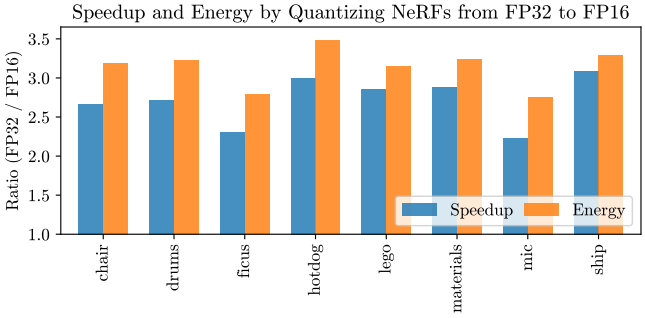


Fig. 7: Speedup and Energy from Quantization to FP16.

	Energy (μJ)	Cycles	EDP (J $*$ cycle)	Area (mm^2)
Eyeriss	0.04	16	6.24e-07	1.38
w/ Skipping	0.03	11	3.42e-07	0.91
w/ Gating	0.03	11	3.50e-07	0.85

TABLE II: **Eyeriss - Volumetric Rendering.** Eyeriss results with sparse optimizations for dot products.

	PSNR (FP32)	**PSNR (FP16)**	**Decrease** (%)
chair	35.11	33.91	3.42%
drums	26.25	25.93	1.19%
ficus	32.93	32.06	2.62%
hotdog	36.93	36.24	1.87%
lego	33.72	32.74	2.90%
materials	30.34	29.86	1.58%
mic	34.59	33.52	3.10%
ship	29.85	29.43	1.39%

TABLE III: **Effect of quantization on PSNR.** We compare the PSNR averaged over 200 images rendered for each scene on the original NeRF (FP32) and its quantized FP16 version.

a vector containing the weights and a vector of color values. Figure 5 shows that these density-based weights exhibit scene-dependent sparsity, with an overall mean of 33.8%. Similar to exploiting activation sparsity, we investigate how we can leverage the weight sparsity to accelerate volumetric rendering.

We use the Eyeriss setup described previously in Section IV-B, and extend it to compute dot products. Observe that dot products are equivalent to 1×1 convolutions with a filter size equivalent to the size of the vector. Hence, we can seamlessly adapt Eyeriss to efficiently handle volumetric rendering. We evaluate the impact of onchip compression with skipping or gating, as these produced the best results in Section IV-B.

We present our results on a batch size of 128 ray samples in Table II, and the per-scene relative EDP improvement compared to the dense Eyeriss baseline in Figure 6. We observe significant improvements in energy and performance, as demonstrated by the near 50% reductions in energy-delay product. Unsurprisingly, there is a strong positive correlation between the EDP improvement and weight sparsity – the higher the sparsity, the better the improvement. While our results are promising, we may expect the improvements from sparse optimizations to diminish as more intelligent ray sampling strategies are developed to further reduce the number of samples with zero weight and hence decrease sparsity.

D. Quantization from FP32 to FP16

We explore post-training half-precision (FP16) quantization with no additional fine-tuning to further improve the energy efficiency and rendering speed of NeRFs. We evaluate the FP32 and FP16 NeRFs using PyTorch on a NVIDIA RTX 3090 GPU, and estimate energy by multiplying the render time by the average power draw of the GPU during the render for a given scene. Note that recent NVIDIA GPUs have tensor cores which accelerate FP16 matrix multiplies. To make our project feasible within the limited time, evaluation of quantization on specialized hardware architectures is left as future work.

We present the improvements in overall speed and energy in Figure 7, and a detailed breakdown in Appendix D. The average power draw was 320.7W for FP32 and 277.5W for FP16. We compare the quality of the rendered images with the peak-signal-to-noise ratio (PSNR), which measures how much noise there is compared to the ground truth (lower is worse). Table III shows the reduction in PSNR due to quantization. Across all scenes, we observe a 2.26% decrease in PSNR and significant $2.72\times$ and $3.14\times$ improvements in the speed and energy, respectively. While PSNR is marginally decreased, the qualitative effects of this are difficult to observe unless we pixel peep, as shown in Figure 1 and the videos on our website (click here). Another benefit of FP16 quantization is that the model size is reduced by half from 2.4MB to 1.2MB.

Overall, we believe that the improved model size, rendering time, and energy consumption offset the loss in PSNR, which may be especially valuable in low-compute environments. Thus, this warrants further investigation on custom hardware architectures and potential quantization to INT8.

V. Conclusion

We investigated the potential of hardware acceleration for MLP-based NeRFs. Our experiments showed an Eyeriss-based architecture with sparse optimizations achieves significant performance and energy improvements, allowing us to accelerate the MLP and volumetric rendering. Post-training FP16 quantization further gains in render speed and energy. Though our work targets MLP-based NeRFs, our findings may generalize to other models that use MLPs in some form. Future work includes designing hardware that is flexible for a variety of NeRF architectures, and developing a customized accelerator for NeRFs through an algorithm-hardware co-design.

ACKNOWLEDGEMENTS

We thank Jaeyeon Won for his advice and assistance on this project, and the authors of Nerfacc [7] for making their NeRF implementation and software accelerator open source.

REFERENCES

[1] A. Chen, Z. Xu, A. Geiger, J. Yu, and H. Su, "Tensorf: Tensorial radiance fields," in *European Conference on Computer Vision (ECCV)*, 2022.

[2] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, 2017.

[3] Z. Chen, T. Funkhouser, P. Hedman, and A. Tagliasacchi, "Mobilenerf: Exploiting the polygon rasterization pipeline for efficient neural field rendering on mobile architectures," *arXiv preprint arXiv:2208.00277*, 2022.

[4] Corridor Crew, "Why this is the future of imagery (and nobody knows it yet)." [Online]. Available: https://www.youtube.com/watch?v=YX5AoaWrowY

[5] Y. Fu, Z. Ye, J. Yuan, S. Zhang, S. Li, H. You *et al.*, "Gen-nerf: Efficient and generalizable neural radiance fields via algorithm-hardware co-design," *arXiv preprint arXiv:2304.11842*, 2023.

[6] 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," 2022.

[7] R. Li, H. Gao, M. Tancik, and A. Kanazawa, "Nerfacc: Efficient sampling accelerates nerfs." *To Be Updated*, 2023.

[8] S. Li, C. Li, W. Zhu, C. Wan, H. You, H. Shi *et al.*, "Instant-3d: Instant neural radiance field training towards on-device ar/vr 3d reconstruction," *arXiv preprint arXiv:2304.12467*, 2023.

[9] Luma AI, "Luma unreal engine plugin." [Online]. Available: https://docs.lumalabs.ai/9DdnisfQaLN1sn

[10] N. L. Max, "Optical models for direct volume rendering," *IEEE Trans. Vis. Comput. Graph.*, vol. 1, pp. 99–108, 1995.

[11] 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," in *ECCV*, 2020.

[12] T. Müller, A. Evans, C. Schied, and A. Keller, "Instant neural graphics primitives with a multiresolution hash encoding," *ACM Trans. Graph.*, 2022.

[13] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *ISPASS*, 2019.

[14] C. Rao, H. Yu, H. Wan, J. Zhou, Y. Zheng, M. Wu, Y. Ma, A. Chen, B. Yuan, P. Zhou, X. Lou, and J. Yu, "Icarus: A specialized architecture for neural radiance fields rendering," *ACM Trans. Graph.*, 2022.

[15] C. Reiser, S. Peng, Y. Liao, and A. Geiger, "Kilonerf: Speeding up neural radiance fields with thousands of tiny mlps," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021.

[16] C. Sun, M. Sun, and H. Chen, "Direct voxel grid optimization: Super-fast convergence for radiance fields reconstruction," in *CVPR*, 2022.

[17] Y. N. Wu, J. S. Emer, and V. Sze, "Accelergy: An architecture-level energy estimation methodology for accelerator designs," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–8.

[18] Y. N. Wu, P.-A. Tsai, A. Parashar, V. Sze, and J. S. Emer, "Sparseloop: An analytical approach to sparse tensor accelerator modeling," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 1377–1395.

[19] Y. Zheng, C. Rao, H. Wan, Y. Zhou, P. Zhou, J. Yu, and X. Lou, "An rram-based neural radiance field processor," in *2022 IEEE 35th International System-on-Chip Conference (SOCC)*, 2022, pp. 1–5.

APPENDIX

A. Implementation Details

NeRF Implementation. We modified the reference NeRF MLP implementation provided by Nerfacc [7], which includes a CUDA-based software accelerator for supporting skipping of ray samples with zero density. This is achieved via an occupancy grid transmittance estimator first introduced in Instant-NGP [12]. We excluded this software accelerator time from our analysis, as it is only used for reducing the number of ray samples. This does not alter the validity of our analysis or results, as we expect the distribution of the workload to be similar if not only more exploitable when using the original coarse-to-fine ray sampling strategy from [11].

Problem Specification for Timeloop and Sparseloop. We convert the twelve fully-connected layers in our NeRF architecture (Figure 2) using the pytorch2timeloop library. We specify the densities (i.e., $1-$ sparsities) of the input activations, weights, and output activations of each layer using the values computed from our NeRF profile over the synthetic benchmark. Recall the FC layer weights have zero sparsity.

B. Input Activation Sparsity

Figure 8 depicts the input activation sparsity of the fully-connected layers in trained NeRFs over several scenes. We observe that while the sparsity of the layers fluctuates depending on the scene, the overall sparsity is relatively consistent.

C. Volumetric Rendering

Table IV depicts the per-scene results over the synthetic benchmark from using a Eyeriss-based architecture with onchip sparse compression and gating. This architecture has an area of 0.91mm^2.

	Energy (μJ)	Cycles	EDP (J $*$ cycle)
chair	0.03	11	3.33e-07
drum	0.03	9	2.55e-07
ficus	0.03	8	2.20e-07
hotdog	0.03	11	3.40e-07
lego	0.03	10	2.98e-07
materials	0.03	13	4.37e-07
mic	0.03	13	4.35e-07
ship	0.03	13	4.30e-07

TABLE IV: Volumetric rendering for each scene over the synthetic benchmark with an Eyeriss architecture using onchip compression and gating for dot-product computations.

D. Quantization

	Render Time (s)		**Energy (kJ)**	
	FP32	FP16	FP32	FP16
chair	116.75	43.74	37.68	11.80
drums	121.70	44.75	39.30	12.15
ficus	60.63	26.21	18.38	6.58
hotdog	243.52	81.12	82.49	23.72
lego	154.57	54.09	49.98	15.89
materials	148.09	51.45	47.55	14.69
mic	59.07	26.52	17.53	6.36
ship	354.27	114.59	119.35	36.19

TABLE V: We show the breakdown of render time and energy for FP32 and FP16 quantized NeRF.

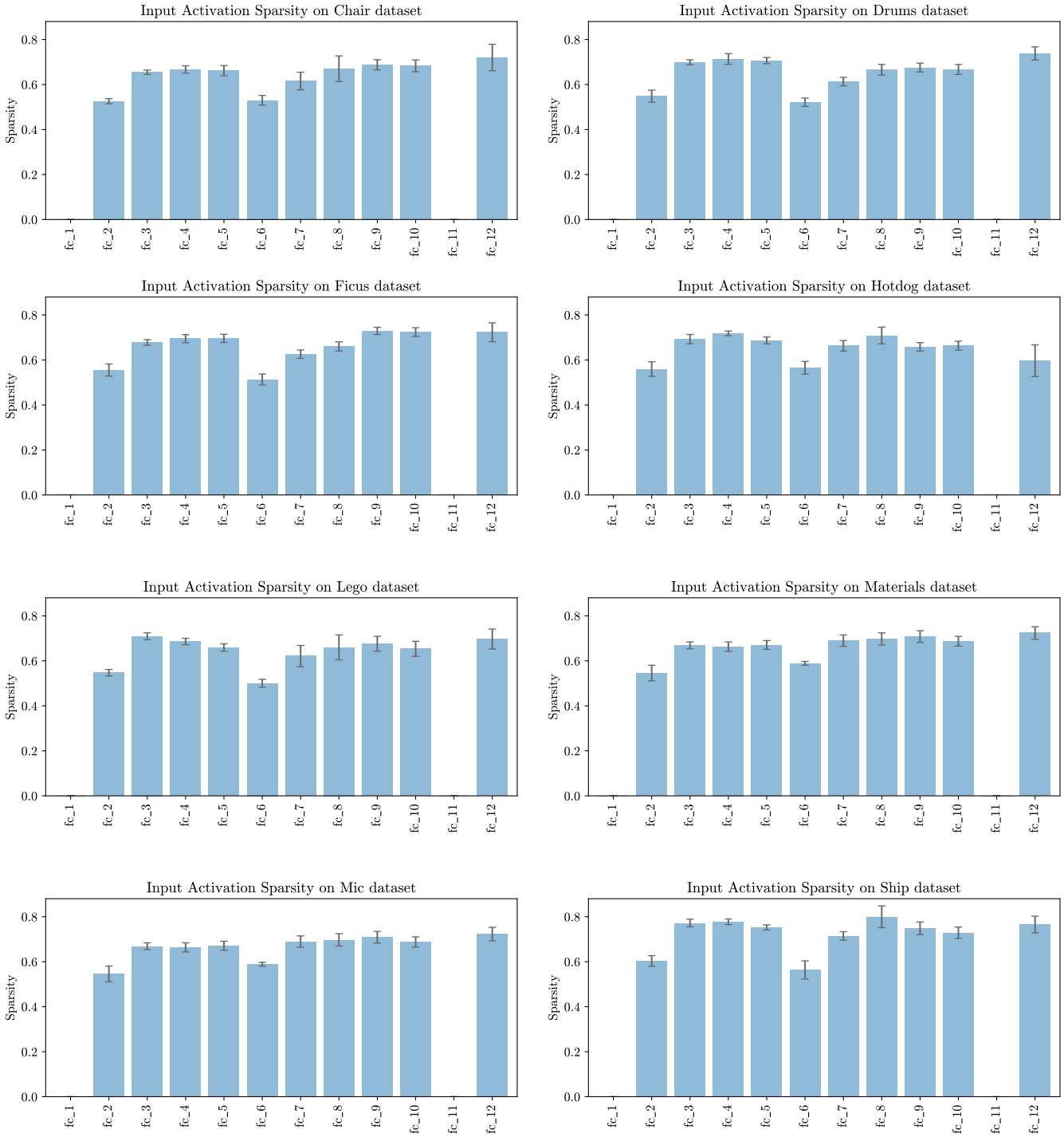


Fig. 8: **Activation Sparsity of NeRF MLP Layers.** We depict the average and standard deviation of the input activation sparsities of the fully-connected (FC) layers of NeRFs trained on the synthetic benchmark [11]. See Figure 2 for the MLP architecture. By rendering 200 test images per scene, we find an overall average sparsity of 65.8% in the FC layers, excluding `fc_1` and `fc_11`. Note that `fc_1` receives the position-encoded ray samples while `fc_11` receives the output from `fc_10`.