Supplementary Material for:
DIVeR: Real-time and Accurate Neural Radiance Fields with Deterministic Integration for Volume Rendering

Liwen Wu Jae Yong Lee Anand Bhattad Yuxiong Wang David Forsyth
University of Illinois at Urbana-Champaign
{liwenwu2, lee896, bhattad2, yxw, daf}@illinois.edu

A. Additional Implementation Details

A.1. Volume rendering approximation

Given a ray $\mathbf{r}(t) = \mathbf{x} + \mathbf{d}t$ and its intersection with the voxel grid $(t_1^{\text{in}}, t_1^{\text{out}}), \dots, (t_n^{\text{in}}, t_n^{\text{out}})$ for parameter values from eye to far end and using the notation from the main text, the volume rendering equation can be decomposed to:

$$\begin{aligned}
\hat{\mathbf{c}}(\mathbf{r}) &= \int_0^\infty e^{-\int_0^t \sigma(\mathbf{r}(\tau))d\tau} \sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t), \mathbf{d})dt. \\
&= \sum_{i=1}^n \int_{t_i^{\text{in}}}^{t_i^{\text{out}}} e^{-\int_0^t \sigma(\mathbf{r}(\tau))d\tau} \sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t))dt \\
&= \sum_{i=1}^n e^{-\sum_{j=1}^{i-1} \int_{t_j^{\text{in}}}^{t_j^{\text{out}}} \sigma(\mathbf{r}(t))dt} \\
&\times \int_{t_i^{\text{in}}}^{t_i^{\text{out}}} e^{-\int_{t_i^{\text{in}}}^t \sigma(\mathbf{r}(\tau))d\tau} \sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t))dt \\
&= \sum_{i=1}^n \prod_{j=1}^{i-1} (1-\alpha_j) \\
&\times \int_{t_i^{\text{in}}}^{t_i^{\text{out}}} e^{-\int_{t_i^{\text{in}}}^t \sigma(\mathbf{r}(\tau))d\tau} \sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t))dt.
\end{aligned} \tag{1}$$

By applying Holder's inequality to the nested integration, we have:

$$\begin{aligned}
&\int_{t_i^{\text{in}}}^{t_i^{\text{out}}} e^{-\int_{t_i^{\text{in}}}^t \sigma(\mathbf{r}(\tau))d\tau} \sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t))dt \\
&\leq \int_{t_i^{\text{in}}}^{t_i^{\text{out}}} e^{-\int_{t_i^{\text{in}}}^t \sigma(\mathbf{r}(\tau))d\tau} \sigma(\mathbf{r}(t))dt \int_{t_i^{\text{in}}}^{t_i^{\text{out}}} \mathbf{c}(\mathbf{r}(t))dt \\
&= \mathbf{c}_i \int_{t_i^{\text{in}}}^{t_i^{\text{out}}} e^{-\int_{t_i^{\text{in}}}^t \sigma(\mathbf{r}(\tau))d\tau} d\left(\int_{t_i^{\text{in}}}^t \sigma(\mathbf{r}(\tau))d\tau\right) \\
&= \mathbf{c}_i \left(-e^{-\int_{t_i^{\text{in}}}^t \sigma(\mathbf{r}(\tau))d\tau} \Big|_{t=t_i^{\text{in}}}^{t_i^{\text{out}}}\right) \\
&= \mathbf{c}_i \left(1 - e^{-\int_{t_i^{\text{in}}}^{t_i^{\text{out}}} \sigma(\mathbf{r}(\tau))d\tau}\right) \\
&= \alpha_i \mathbf{c}_i.
\end{aligned} \tag{2}$$

Therefore, we have:

$$\hat{\mathbf{c}}(\mathbf{r}) \leq \sum_{i=1}^n \prod_{j=1}^{i-1} (1-\alpha_j)\alpha_i \mathbf{c}_i. \tag{3}$$

A.2. Feature integration

Assume the size of every voxel is $1 \times 1 \times 1$ and a voxel has feature vectors $\mathbf{f}_1, \dots, \mathbf{f}_8$ placed at its eight corners. In the voxel's local coordinate system, the feature value inside the voxel at $\mathbf{x} = (x, y, z)$ is then given by

$$\mathbf{f}(x, y, z) = \sum_{k=1}^8 \mathbf{f}_k \chi_k(x, y, z), \tag{4}$$

where:

$$\begin{cases}
\chi_8(x, y, z) &= xyz \\
\chi_7(x, y, z) &= (1-x)yz \\
\chi_6(x, y, z) &= x(1-y)z \\
\chi_5(x, y, z) &= (1-x)(1-y)z \\
\chi_4(x, y, z) &= xy(1-z) \\
\chi_3(x, y, z) &= (1-x)y(1-z) \\
\chi_2(x, y, z) &= x(1-y)(1-z) \\
\chi_1(x, y, z) &= (1-x)(1-y)(1-z)
\end{cases}. \tag{5}$$

Let $\mathbf{x}_0 = (x_0, y_0, z_0), \mathbf{x}_1 = (x_1, y_1, z_1)$ be the voxel's intersection with a ray at entry and exit. It defines a ray segment $\mathbf{x}(t)$:

$$\begin{aligned}
\mathbf{x}(t) &= (x(t), y(t), z(t)) \\
&= (x_0, y_0, z_0)(1-t) \\
&+ (x_1, y_1, z_1)t, t \in [0, 1].
\end{aligned} \tag{6}$$

We want the interpolation function (basis) to be normalized given fixed $\mathbf{x}_0, \mathbf{x}_1$; the normalized $\mathbf{f}(\mathbf{x}(t))$ along the ray is:

$$\begin{aligned}
\hat{\mathbf{f}}(\mathbf{x}(t)) &= \frac{\mathbf{f}(\mathbf{x}(t))}{\int_{\mathbf{x}_0}^{\mathbf{x}_1} \sum_{k=1}^8 \chi_k(\mathbf{x}(t))dt} \\
&= \frac{\mathbf{f}(\mathbf{x}(t))}{\int_0^1 \sum_{k=1}^8 \chi_k(x(t), y(t), z(t))\|\mathbf{x}'(t)\|_2 dt} \\
&= \frac{\mathbf{f}(\mathbf{x}(t))}{\|\mathbf{x}_1 - \mathbf{x}_0\|_2},
\end{aligned} \tag{7}$$

which gives normalized feature integration:

$$\begin{aligned}\int_{\mathbf{x}_0}^{\mathbf{x}_1} \hat{\mathbf{f}}(\mathbf{x}(t))dt &= \int_0^1 \frac{\mathbf{f}(x(t),y(t),z(t))}{\|\mathbf{x}_1-\mathbf{x}_0\|_2}\|\mathbf{x}'(t)\|dt \\ &= \int_0^1 \sum_{k=1}^{8} \mathbf{f}_k \chi_k(x(t),y(t),z(t))dt \\ &= \sum_{k=1}^{8} \mathbf{f}_k \int_0^1 \chi_k(x(t),y(t),z(t))dt \\ &= \sum_{k=1}^{8} \mathbf{f}_k X_k(\mathbf{x}_0,\mathbf{x}_1).\end{aligned} \tag{8}$$

The integration of each interpolation function only depends on the entry and exit that produces a polynomial. In our CUDA implementation, we factorize $X_k(\mathbf{x}_0,\mathbf{x}_1)$ as:

$$\begin{cases} X_8(\mathbf{x}_0,\mathbf{x}_1) &= \frac{2x_0y_0z_0+2x_1y_1z_1+abc}{12} \\ X_7(\mathbf{x}_0,\mathbf{x}_1) &= -X_8(\mathbf{x}_0,\mathbf{x}_1)+d \\ X_6(\mathbf{x}_0,\mathbf{x}_1) &= \frac{ac+x_0z_0+x_1z_1}{6} - X_8(\mathbf{x}_0,\mathbf{x}_1) \\ X_5(\mathbf{x}_0,\mathbf{x}_1) &= \frac{c}{2} - X_6(\mathbf{x}_0,\mathbf{x}_1) - d \\ X_4(\mathbf{x}_0,\mathbf{x}_1) &= -X_8(\mathbf{x}_0,\mathbf{x}_1)+e \\ X_3(\mathbf{x}_0,\mathbf{x}_1) &= \frac{b}{2} - X_7(\mathbf{x}_0,\mathbf{x}_1) - e \\ X_2(\mathbf{x}_0,\mathbf{x}_1) &= \frac{a}{2} - X_6(\mathbf{x}_0,\mathbf{x}_1) - e \\ X_1(\mathbf{x}_0,\mathbf{x}_1) &= 1 - \frac{a+b}{2} - X_5(\mathbf{x}_0,\mathbf{x}_1) + e \end{cases}, \tag{9}$$

where:

$$\begin{cases} a &= x_0+x_1 \\ b &= y_0+y_1 \\ c &= z_0+z_1 \\ d &= \frac{bc+y_0z_0+y_1+z_1}{6} \\ e &= \frac{ab+x_0y_0+x_1y_1}{6} \end{cases}. \tag{10}$$

A.3. Real-time ray-voxel intersection

For each ray, we first find its closest hit on the voxel grid, and then ray marching to a fixed number of voxels in each iteration of MLP evaluation. For closest hit calculation, we build an octree from the occupancy map as a list of 3D arrays by max pooling and traverse in the octree to speed up the calculation. For ray marching after the closest hit, we do not use the octree and use the algorithm described in [1].

A.4. Real-time MLP evaluation

Because weights and biases of the decoder MLP are globally shared, we upload them to CUDA constant memory to speed up the memory read. Additionally, we refactor two linear layers in the MLP to reduce calculations. We use the DIVeR32 decoder architecture for the illustration, which can be easily extended to DIVeR64.

Pre-multiplication of the first layer: Because there is no activation (ReLU) between the integrated feature and the first layer of the MLP, the weight of the first layer can be pre-multiplied to the feature vectors. Given the weight and bias of the first linear layer as $\mathbf{W}_1, \mathbf{b}_1$, the first layer's output $\mathbf{e}_1$ (without activation) is:

$$\begin{aligned}\mathbf{e}_1 &= \mathbf{W}_1 \int_{\mathbf{x}_0}^{\mathbf{x}_1} \hat{\mathbf{f}}(\mathbf{x}(t))dt + \mathbf{b}_1 \\ &= \mathbf{W}_1 \sum_{k=1}^{8} \mathbf{f}_k \mathbf{X}_k(\mathbf{x}_0,\mathbf{x}_1) + \mathbf{b}_1 \\ &= \sum_{k=1}^{8} \mathbf{f}'_k \mathbf{X}_k(\mathbf{x}_0,\mathbf{x}_1) + \mathbf{b}_1 \\ &= \int_{\mathbf{x}_0}^{\mathbf{x}_1} \hat{\mathbf{f}}'(\mathbf{x}(t))dt + \mathbf{b}_1,\end{aligned} \tag{11}$$

where:

$$\mathbf{f}'_k = \mathbf{W}_1\mathbf{f}_k. \tag{12}$$

By pre-multiplying the weight to each feature vector after the training and using Eq. 11 during inference time, the operation needed for evaluating the first layer is reduced to a vector add.

Composition of the third and fourth layers: Similarly, the hidden feature $\mathbf{h}_3$ of the third layer is not mapped with ReLU, such that weights in the third and fourth layers can be composited. Let $\mathbf{W}_3, \mathbf{b}_3$ denote the weight and bias of the third layer, and $\mathbf{W}_4, \mathbf{b}_4$ denote the weight and bias of the fourth layer. Given the hidden feature of the second layer $\mathbf{h}_2$ and the positional encoded viewing direction $\gamma(\mathbf{d})$, we have:

$$\begin{aligned}\begin{bmatrix}\sigma \\ \mathbf{h}_3\end{bmatrix} &= \mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3 \\ &= \begin{bmatrix}\mathbf{W}_3^{\sigma} \\ \mathbf{W}_3^{\mathbf{h}}\end{bmatrix}\mathbf{h}_2 + \begin{bmatrix}\mathbf{b}_3^{\sigma} \\ \mathbf{b}_3^{\mathbf{h}}\end{bmatrix}\end{aligned} \tag{13}$$

$$\begin{aligned}\mathbf{e}_4 &= \mathbf{W}_4 \begin{bmatrix}\gamma(\mathbf{d}) \\ \mathbf{h}_3\end{bmatrix} + \mathbf{b}_4 \\ &= \begin{bmatrix}\mathbf{W}_4^{\mathbf{d}} & \mathbf{W}_4^{\mathbf{h}}\end{bmatrix}\begin{bmatrix}\gamma(\mathbf{d}) \\ \mathbf{h}_3\end{bmatrix} + \mathbf{b}_4 \\ &= \mathbf{W}_4^{\mathbf{d}}\gamma(\mathbf{d}) + \mathbf{W}_4^{\mathbf{h}}\mathbf{h}_3 + \mathbf{b}_4 \\ &= \mathbf{W}_4^{\mathbf{d}}\gamma(\mathbf{d}) + \mathbf{W}_4^{\mathbf{h}}(\mathbf{W}_3^{\mathbf{h}}\mathbf{h}_2 + \mathbf{b}_3^{\mathbf{h}}) + \mathbf{b}_4 \\ &= \mathbf{W}_4^{\mathbf{d}}\gamma(\mathbf{d}) + (\mathbf{W}_4^{\mathbf{h}}\mathbf{W}_3^{\mathbf{h}})\mathbf{h}_2 + (\mathbf{W}_4^{\mathbf{h}}\mathbf{b}_3^{\mathbf{h}} + \mathbf{b}_4).\end{aligned} \tag{14}$$

Therefore, the density σ and hidden feature of the fourth layer $\mathbf{e}_4$ (without activation) could be directly calculated from $\gamma(\mathbf{d})$ and $\mathbf{h}_2$ without evaluating $\mathbf{h}_3$:

$$\sigma = \mathbf{W}_3^{\sigma}\mathbf{h}_2 + \mathbf{b}_3^{\sigma} \tag{15}$$

$$\mathbf{e}_4 = \mathbf{W}_4^{\mathbf{d}}\gamma(\mathbf{d}) + \mathbf{W}'_4\mathbf{h}_2 + \mathbf{b}'_4 \tag{16}$$

where:

$$\mathbf{W}'_4 = \mathbf{W}_4^{\mathbf{h}}\mathbf{W}_3^{\mathbf{h}} \text{ and } \mathbf{b}'_4 = \mathbf{W}_4^{\mathbf{h}}\mathbf{b}_3^{\mathbf{h}} + \mathbf{b}_4, \tag{17}$$

which avoids one 32×32 matrix multiplication and one 32 dimension vector add.

A.5. Object swapping

We use two cuboids to mark the objects to be swapped and run k-mean clustering for each region to get the fine segmentation. Feature vectors that belong to the largest cluster are treated as the background; the rest of the features are treated as the foreground objects to be swapped. In the hot-dog scene, we use 12 clusters.

B. Experiment Details

In Tab. 1, we show the per-scene rendering quality comparison on the NeRF-synthetic dataset for all the baselines we compared with (offline, real-time pre-trained, and real-time applications). Tab. 2 shows the per-scene offline rendering quality on the Tanks and Temple and BlendedMVS datasets, and Tab. 3 shows the per-scene real-time performance on the NeRF-synthetic dataset. For ablation on the network architecture, we also show the per-scene performance and rendering quality in Tab. 4.

References

[1] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics*, 1987. 2

PSNR ↑	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship	*Mean*
NeRF	33.00	25.01	30.13	36.18	32.54	29.62	32.91	28.65	31.00
JaxNeRF	33.88	25.08	30.15	36.91	33.24	30.03	34.52	29.07	31.64
AutoInt	25.60	20.78	22.47	32.33	25.09	25.90	28.10	24.15	25.55
NSVF	33.19	25.18	31.23	37.14	32.29	32.68	34.27	27.93	31.74
NeRF-SH	3.98	25.17	30.72	36.75	32.77	29.95	34.04	29.21	31.57
JaxNeRF+	35.35	25.65	32.77	37.58	35.35	30.29	36.52	30.48	33.00
PlenOctrees	34.66	25.31	30.79	36.79	32.95	29.76	33.97	29.42	31.71
SNeRG	33.24	24.57	29.32	34.33	33.82	27.21	32.60	27.97	30.38
FastNeRF	32.32	23.75	27.79	34.72	32.28	28.89	31.77	27.69	29.97
KiloNeRF	-	-	-	-	-	-	-	-	31.00
DIVeR64	34.34	25.39	31.77	36.83	35.52	29.63	34.58	30.50	32.32
DIVeR32	34.10	25.40	32.03	36.50	35.27	29.25	34.56	30.17	32.16
DIVeR32(RT)	34.09	25.40	32.02	36.35	35.17	29.24	34.53	30.14	32.12

SSIM ↑	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship	*Mean*
NeRF	0.967	0.925	0.964	0.974	0.961	0.949	0.980	0.856	0.947
JaxNeRF	0.974	0.927	0.967	0.979	0.968	0.952	0.987	0.865	0.952
AutoInt	0.928	0.861	0.898	0.974	0.900	0.930	0.948	0.852	0.911
NSVF	0.968	0.931	0.973	0.980	0.960	0.973	0.987	0.854	0.953
NeRF-SH	0.974	0.927	0.968	0.978	0.966	0.951	0.985	0.866	0.952
JaxNeRF+	0.982	0.936	0.980	0.983	0.979	0.956	0.991	0.887	0.962
PlenOctrees	0.981	0.933	0.970	0.982	0.971	0.955	0.987	0.884	0.958
SNeRG	0.975	0.929	0.967	0.971	0.973	0.938	0.982	0.865	0.950
FastNeRF	0.966	0.913	0.954	0.973	0.964	0.947	0.977	0.805	0.941
KiloNeRF	-	-	-	-	-	-	-	-	0.950
DIVeR64	0.978	0.933	0.975	0.981	0.980	0.951	0.987	0.893	0.960
DIVeR32	0.977	0.932	0.977	0.979	0.979	0.946	0.987	0.886	0.958
DIVeR32(RT)	0.977	0.932	0.977	0.978	0.978	0.946	0.987	0.885	0.958

LPIPS ↓	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship	*Mean*
NeRF	0.046	0.091	0.044	0.121	0.050	0.063	0.028	0.206	0.081
JaxNeRF	0.027	0.070	0.033	0.030	0.030	0.048	0.013	0.156	0.051
AutoInt	0.141	0.224	0.148	0.080	0.175	0.136	0.131	0.323	0.170
NSVF	0.043	0.069	0.017	0.025	0.029	0.021	0.010	0.162	0.047
NeRF-SH	0.037	0.087	0.039	0.041	0.041	0.060	0.021	0.177	0.063
JaxNeRF+	0.017	0.057	0.018	0.022	0.017	0.041	0.008	0.123	0.038
PlenOctree	0.022	0.076	0.038	0.032	0.034	0.059	0.017	0.144	0.053
SNeRG	0.025	0.061	0.028	0.043	0.022	0.052	0.016	0.156	0.050
FastNeRF	0.032	0.083	0.031	0.031	0.022	0.034	0.022	0.192	0.053
KiloNeRF	-	-	-	-	-	-	-	-	0.030
DIVeR64	0.014	0.057	0.020	0.017	0.010	0.032	0.010	0.093	0.032
DIVeR32	0.014	0.058	0.020	0.019	0.010	0.035	0.011	0.102	0.034
DIVeR32(RT)	0.014	0.058	0.020	0.019	0.010	0.034	0.011	0.100	0.033

Table 1. **Rendering quality on the NeRF-synthetic dataset**.

	PSNR ↑										
	Barn	Caterpillar	Family	Ignatius	Truck	*Mean*	Jade	Fountain	Char	Statues	*Mean*
NeRF	24.05	23.75	30.29	25.43	25.36	25.78	21.65	25.59	25.87	23.48	24.15
JaxNeRF	27.39	25.24	32.47	27.95	26.66	27.94	-	-	-	-	-
NSVF	27.16	26.44	33.58	27.91	26.92	28.40	26.96	27.73	27.95	24.97	26.90
DIVeR64	27.31	25.64	33.40	27.80	26.74	28.18	26.52	28.30	28.81	25.36	27.25

	SSIM ↑										
	Barn	Caterpillar	Family	Ignatius	Truck	*Mean*	Jade	Fountain	Char	Statues	*Mean*
NeRF	0.750	0.860	0.932	0.920	0.860	0.864	0.750	0.860	0.900	0.800	0.828
JaxNeRF	0.842	0.892	0.951	0.940	0.896	0.904	-	-	-	-	-
NSVF	0.832	0.900	0.954	0.930	0.895	0.900	0.901	0.913	0.921	0.858	0.898
DIVeR64	0.850	0.903	0.960	0.941	0.904	0.912	0.900	0.918	0.948	0.873	0.910

	LPIPS ↓										
	Barn	Caterpillar	Family	Ignatius	Truck	*Mean*	Jade	Fountain	Char	Statues	*Mean*
NeRF	0.395	0.196	0.098	0.111	0.192	0.198	0.264	0.149	0.149	0.206	0.192
JaxNeRF	0.286	0.189	0.092	0.102	0.173	0.168	-	-	-	-	-
NSVF	0.307	0.141	0.063	0.106	0.148	0.153	0.094	0.113	0.074	0.171	0.113
DIVeR64	0.209	0.121	0.050	0.082	0.119	0.116	0.076	0.069	0.037	0.110	0.073

Table 2. **Rendering quality on the Tanks & Temple and BlendedMVS datasets**.

	FPS ↑								
	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship	*Range*
PlenOctrees	143	78	23	15	45	13	76	10	76±66
SNeRG	-	-	-	-	-	-	-	-	98±37
FastNeRF	-	-	-	-	-	-	-	-	-
KiloNeRF	40	-	-	-	40	-	-	16	28±12
DIVeR32(RT)	59	40	39	44	67	29	66	27	47±20

	MB ↓								
	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship	*Mean*
PlenOctrees	832	1239	1792	2683	2068	3686	443	2693	1930
SNeRG	-	-	-	-	-	-	-	-	84
FastNeRF	-	-	-	-	-	-	-	-	-
KiloNeRF	204	-	-	-	108	-	-	173	161
DIVeR32(RT)	55	56	47	84	64	62	24	151	68

	GPU GB ↓								
	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship	*Range*
PlenOctrees	0.94	1.34	1.87	2.73	2.19	3.70	0.56	2.74	1.65±1.09
SNeRG	-	-	-	-	-	-	-	-	1.73±1.48
FastNeRF	-	-	-	-	-	-	-	-	-
KiloNeRF	1.94	-	-	-	1.41	-	-	1.78	1.68±0.27
DIVeR32(RT)	1.04	1.04	1.03	1.06	1.04	1.04	1.01	1.13	1.07±0.06

Table 3. **Performance of real-time applications on the NeRF-synthetic dataset**.

PSNR ↑										
N	Decoder	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship	*Mean*
256	DIVeR64(RT)	34.35	25.38	31.76	36.76	35.49	29.61	34.57	30.48	32.30
256	DIVeR32(RT)	34.09	25.40	32.02	36.35	35.17	29.24	34.53	30.14	32.12
128	DIVeR64(RT)	31.98	24.74	30.12	35.54	32.57	28.96	32.15	29.02	30.63
128	DIVeR32(RT)	31.54	24.75	30.25	35.42	32.61	28.82	31.97	28.80	30.52

FPS ↑										
N	Decoder	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship	*Range*
256	DIVeR64(RT)	31	25	18	19	28	16	35	17	26±9
256	DIVeR32(RT)	59	40	39	44	67	29	66	27	47±20
128	DIVeR64(RT)	57	38	29	33	41	28	53	17	37±20
128	DIVeR32(RT)	108	82	61	84	99	67	119	45	82±37

MB ↓										
N	Decoder	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship	*Mean*
256	DIVeR64(RT)	55	42	49	80	64	62	24	118	62
256	DIVeR32(RT)	55	56	47	84	64	62	24	151	68
128	DIVeR64(RT)	9.2	8.2	8.5	15	12	9.6	4.7	30	12
128	DIVeR32(RT)	9.7	8.9	9.3	16	13	9.8	4.5	28	12

Table 4. **Architecture ablation on the NeRF-synthetic dataset**.