Supplementary Material for KiloNeRF: Speeding up Neural Radiance Fields with Thousands of Tiny MLPs

Abstract

*In this **supplementary document**, we first clarify implementation specifics in Section 1. We then give more detailed results for the experiments introduced in the main paper in Section 2. Additional experimental results to study the impact of the employed regularization techniques follow in Section 3. Finally, we showcase the challenges that NeRF-like techniques face with generalization to out-of-distribution camera poses in Section 4. The **supplementary video** shows qualitative comparisons between KiloNeRF and the NeRF baseline.*

1. Implementation Details

For KiloNeRF, we normalize position inputs such that each network receives positions in $[-1, 1]^3$. We initialize KiloNeRF such that the parameters of the individual networks are identical to each other. For the NeRF baseline, we use the PyTorch implementation from Lin Yen-Chen [3]. This implementation was verified to be numerically identical to the original TensorFlow implementation of NeRF. For NSVF, we use the implementation provided by the authors. Except for the number of training iterations for NeRF, both baselines were run with the hyperparameters advocated in the respective papers. The quantitative results of NSVF on all scenes and of NeRF on the Synthetic-NeRF dataset were taken from the respective papers. To render novel views with NSVF on our test machine equipped with a GTX 1080 Ti (11 GB VRAM), it was required to decrease the batch/chunk size for some scenes.

1.1. KiloNeRF Optimizations

A suitable implementation of KiloNeRF is crucial for achieving a good speed-up. Please note that our optimizations are mostly specific to the computation required in KiloNeRF (querying many tiny MLPs) and could not be used to accelerate the rendering of vanilla NeRFs. On top of PyTorch, we make use of custom CUDA kernels and the HPC library MAGMA since the computational primitives exposed by PyTorch are not adequate for efficiently querying many tiny MLPs.

1.1.1 Network Evaluation

In a vanilla MLP, for each linear layer, a matrix-matrix multiplication is calculated that processes a batch of B inputs in parallel. More specifically, a $B \times I$ matrix is multiplied with a $I \times O$, where I refers to the number of input features and O refers to the number of output features of the associated layer. KiloNeRF consists of many independent MLPs and therefore during evaluation for each network with index i a separate matrix-matrix multiplication needs to be calculated. In other words, for each network i a $B_i \times I$ matrix needs to be multiplied with a $I \times O$ matrix. Note how the batch size B_i depends on the network i. This is necessary since individual networks are queried for a different number of input points B_i. For instance, networks that are further away from the camera are queried for fewer points than close-by networks since the distance between points on adjacent rays increases the further away the sample points are from the camera. A straightforward possibility to implement linear layer evaluation directly in PyTorch is by calling `torch.matmul` in a loop for each network. As a consequence, a separate CUDA kernel is launched for each matrix multiplication associated with the individual networks. By default, individual CUDA kernels are executed sequentially, which would imply here that only a small fraction of the GPU's SMs are used. To increase parallelism execution of the individual CUDA kernels can be overlapped with help of CUDA streams. Nevertheless, this solution is suboptimal since there still have to be launched as many kernels as there are networks (thousands), which leads to considerable kernel launch overhead and therefore bad performance. The batched

matrix multiplication routine exposed by PyTorch (`torch.bmm`) appears like a better option since it theoretically allows that only a single CUDA kernel is launched that handles the matrix multiplications associated with all networks in parallel. However, the routine exposed by PyTorch requires that all input matrices have the same shape. Since different networks are in general queried for different numbers of points the matrices associated with different networks have different shapes (B_i is not constant). Therefore `torch.bmm` cannot be used. In contrast, MAGMA indeed provides a routine that can also handle input matrices of variable shapes and is therefore well suited for the more general case present in KiloNeRF. The inner workings of the employed MAGMA routine are described in [1]. For inference only we use a self-developed routine that fuses the entire network evaluation (consisting of Fourier feature calculation, linear layers, and activation functions) into a single CUDA kernel. This has the advantage that intermediate results (e.g. network activations) do not have to be written and read from global memory, which is especially important for our case because the ratio of arithmetic operations to memory transfers is low for multiplications of small matrices. Since intermediate results are necessary for backpropagation this strategy can only be used for inference. In our fused kernel each CUDA block is responsible for a different network and each thread is responsible for handling a single network input. The intermediate results can be fully stored in per-thread registers, which is only possible due to the small network width of 32. The kernel uses 96 registers, which implies that a maximum of 640 threads per SM can be active and that occupancy on a GPU with Compute Capability 6.1 amounts to 0.3125. Since there should be enough arithmetic operations to hide the small memory latency incurred by reading a network input and writing the final RGBσ output, a higher occupancy is not likely to lead to performance gains. Due to the small size of the individual MLPs of approximately 25 KB all parameters of a particular MLP can be loaded into shared memory, which implies that network parameters only have to be loaded once per frame from global memory into on-chip memory. **Network evaluation could be further sped up significantly by making use of Tensor Cores, which is likely to enable real-time rendering of full HD images.**

1.1.2 Input Reordering

Both the MAGMA routine for matrix multiplication and the self-developed CUDA kernel require that input points are sorted according to the index i of the network that is responsible for that point. As mentioned above for each network a $B_i \times I$ matrix $\mathbf{X}_i$ is multiplied with a weight matrix $\mathbf{W}_i$ with shape $I \times O$. Here it is required that the matrix $\mathbf{X}_i$ is stored densely. However, the unprocessed result from the preceding sampling point generation step is the unordered $B \times 6$ matrix $\tilde{\mathbf{X}}$, where $B = \sum_{i=1}^{B} B_i$ and $I = 6$ because of the 3-dimensional position and direction inputs. With each input $\tilde{\mathbf{X}}_j$ ($j = 1, \ldots, B$) there is associated a network with index i_j that is responsible for that input. We use thrust to perform a key-value sort where the keys are $i_1, \ldots, i_B$ and the values are the row vectors of $\tilde{\mathbf{X}}$. The result consists of the dense matrices $\mathbf{X}_i$. To further decrease the non-negligible overhead introduced by this sorting step, we do not directly reorder the input matrix where each row consists of 6 floats. Memory traffic can be decreased by reordering instead a list of integers, where each element corresponds to an index from which position and direction can be calculated.

2. Detailed Results

Quantitative results for each scene are given in Tables 2, 3, 4 and 5. Qualitative results of all scenes that were not shown in the main paper are given in Fig. 3, Fig. 4, Fig. 5 and Fig. 6. KiloNeRF matches the quality of the baselines, while being significantly faster.

3. Impact of Distillation and L_2 Regularization

We performed a large-scale ablation study on all 25 scenes to determine which impact distillation and L_2 regularization have on quality. In Table 1, we can see that both techniques lead to an improvement in terms of PSNR, SSIM and LPIPS. Note that due to distillation mainly artifacts in free space are reduced. These artifacts are striking to the human eye as can be seen in Fig. 1, but do not have a strong influence on quality metrics, since only a small amount of pixels are affected.

Method	PSNR ↑	SSIM ↑	LPIPS ↓
KiloNeRF w/o Distillation and L_2 Regularization	29.54	0.932	0.051
KiloNeRF w/o Distillation	30.44	0.940	0.048
KiloNeRF	**30.66**	**0.945**	**0.043**

Table 1: **Large-scale Ablation Study.** Both distillation and L_2 Regularization have a positive effect on quality metrics.

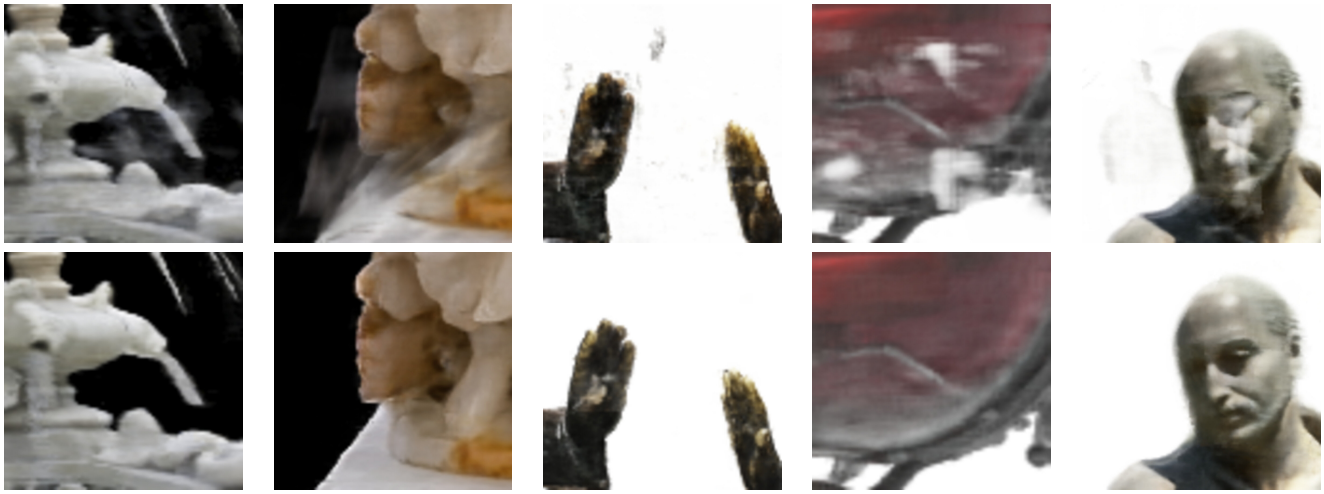

Figure 1: **Importance of Distillation.** Without distillation cloud-like artifacts in empty space emerge (top row). The proposed training strategy mitigates this issue (bottom row)

4. Out-of-distribution Generalization

In general, since NeRF models are trained on a per-scene basis, rendering a novel view that requires querying for view directions that are not part of the training distribution yields poor results. More concretely, envision an object of which only photographs on a circle around that object are available. Synthesizing a novel view with NeRF from any point on this circle works well since it is ensured that the model is only queried for viewing directions that are inside the training distribution. It is also possible to zoom in or out as long as the camera stays on the plane on which the circle lies. However, if we leave that plane e.g. by increasing camera elevation, results are getting significantly worse as the model is queried for view directions outside the training distribution. Since a NeRF model is trained on a per-scene basis, there is no knowledge about material properties available and conclusively colors for out-of-distribution view directions cannot be inferred. With that in mind let us take a look at the novel views synthesized by NeRF respectively KiloNeRF from Fig. 2. For that scene, all training images lie approximately on a circle. In particular, the training set does not contain any images from high viewpoints. The images shown in Fig. 2 are rendered from such an out-of-distribution viewpoint. As a result, both NeRF and KiloNeRF predict colors poorly. In KiloNeRF's case, an aggravating factor is that networks are independent and hence each network is biased in a different way. A possible solution is thus to combine KiloNeRF with a method that learns inductive biases from multiple scenes like for instance [2].

(a) Out-of-distribution camera pose (b) NeRF (c) KiloNeRF

Figure 2: **Out-of-distribution generalization.** Novel views are synthesized with NeRF (b) and KiloNeRF (c) from an out-of-distribution camera pose that is visualized in (a), where gray cameras correspond to the training images and the red camera corresponds to the rendered pose. Both methods have problems dealing with this situation. In KiloNeRF's case, the independence of the networks additionally might cause that each network is biased towards its local training distribution, which results in block-like artifacts.

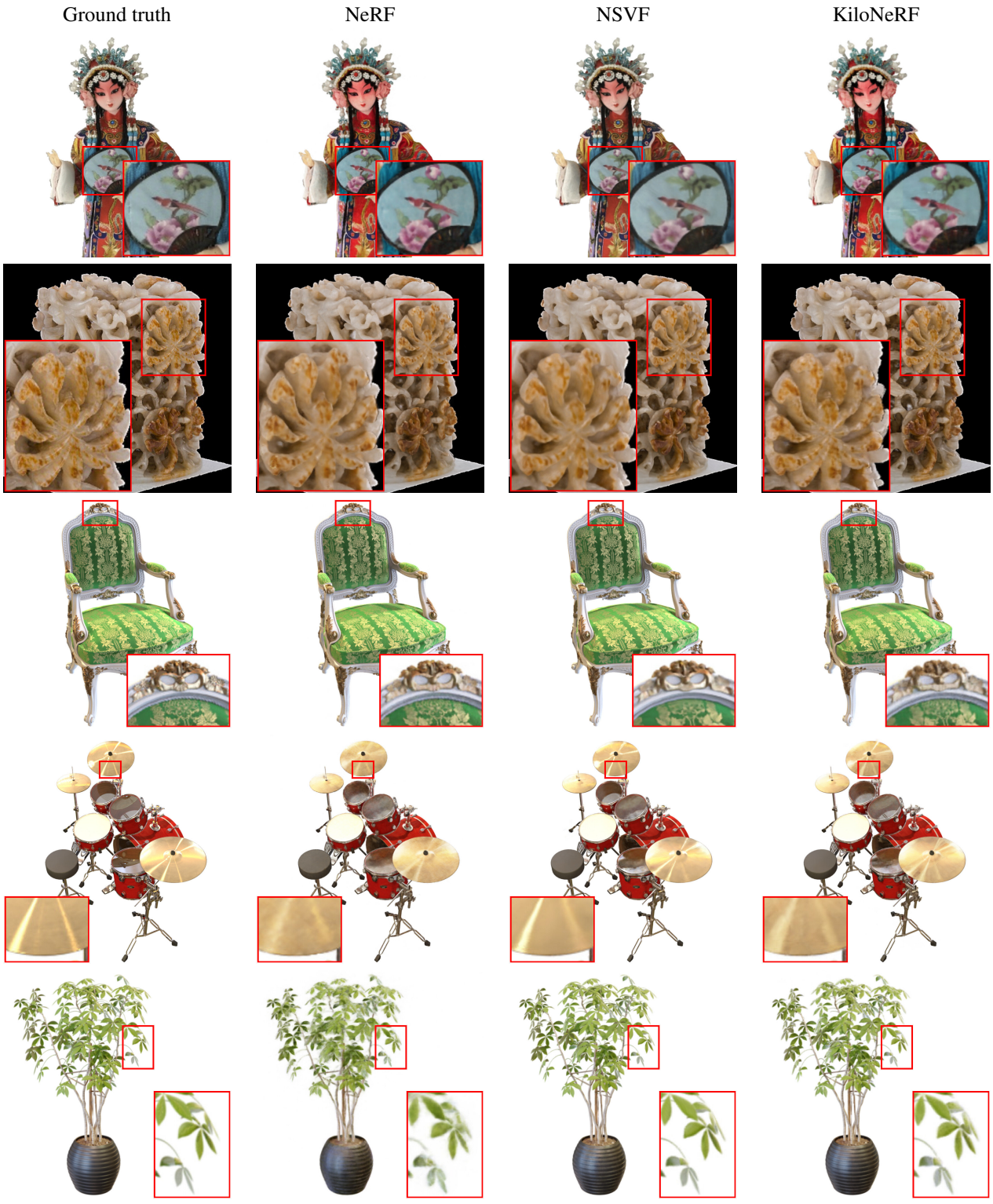

Figure 3: Qualitative results for the scenes Character (BlendedMVS), Jade (BlendedMVS), Chair (Synthetic NeRF), Drums (Synthetic NeRF) and Ficus (Synthetic NeRF)

Figure 4: Qualitative results for the scenes Hotdog (Synthetic NeRF), Materials (Synthetic NeRF), Mic (Synthetic NeRF), Bike (Synthetic NSVF) and Lifestyle (Synthetic NSVF)

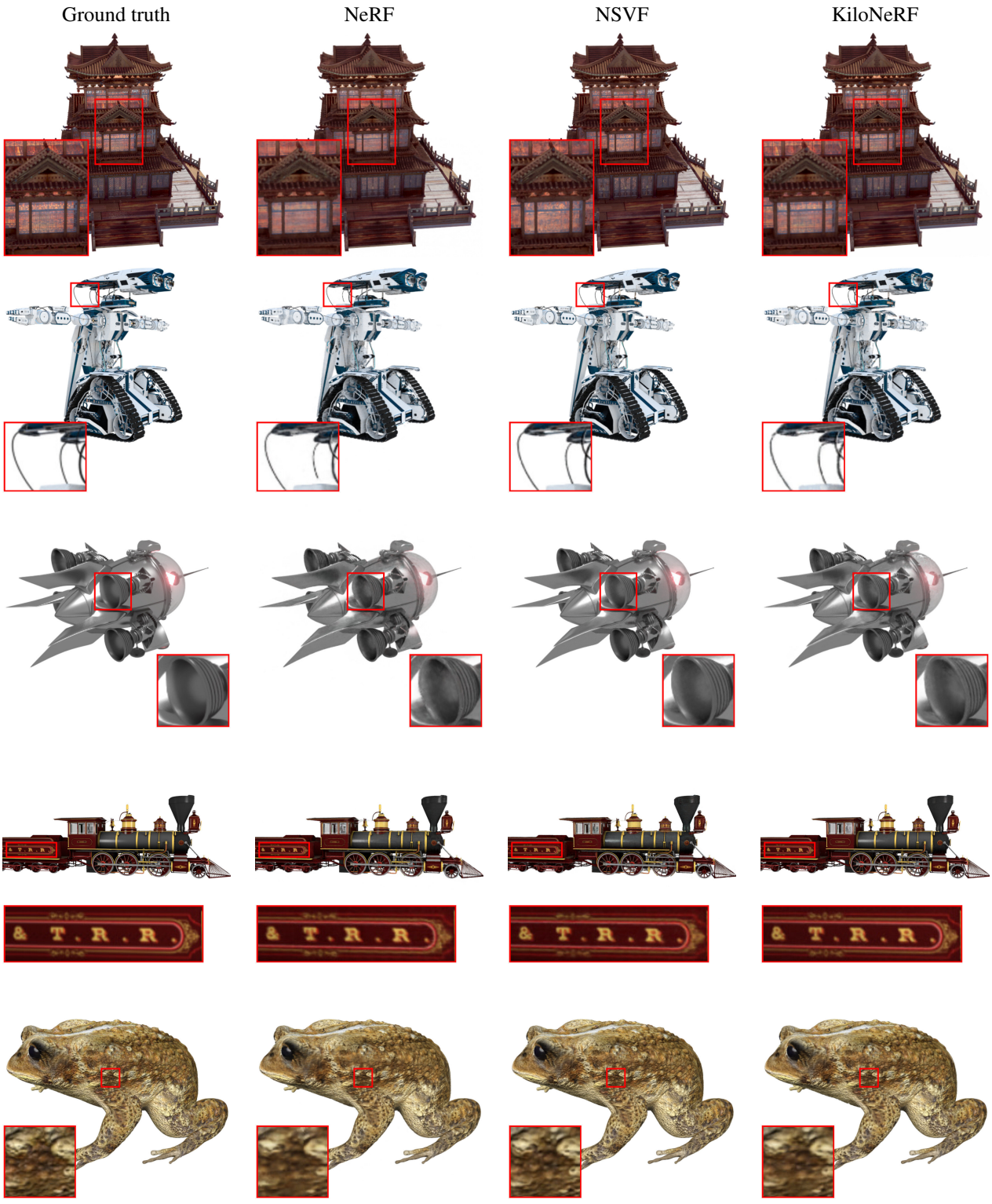

Figure 5: Qualitative results for the scenes Palace (Synthetic NSVF), Robot (Synthetic NSVF), Spaceship (Synthetic NSVF), Steamtrain (Synthetic NSVF) and Toad (Synthetic NSVF)

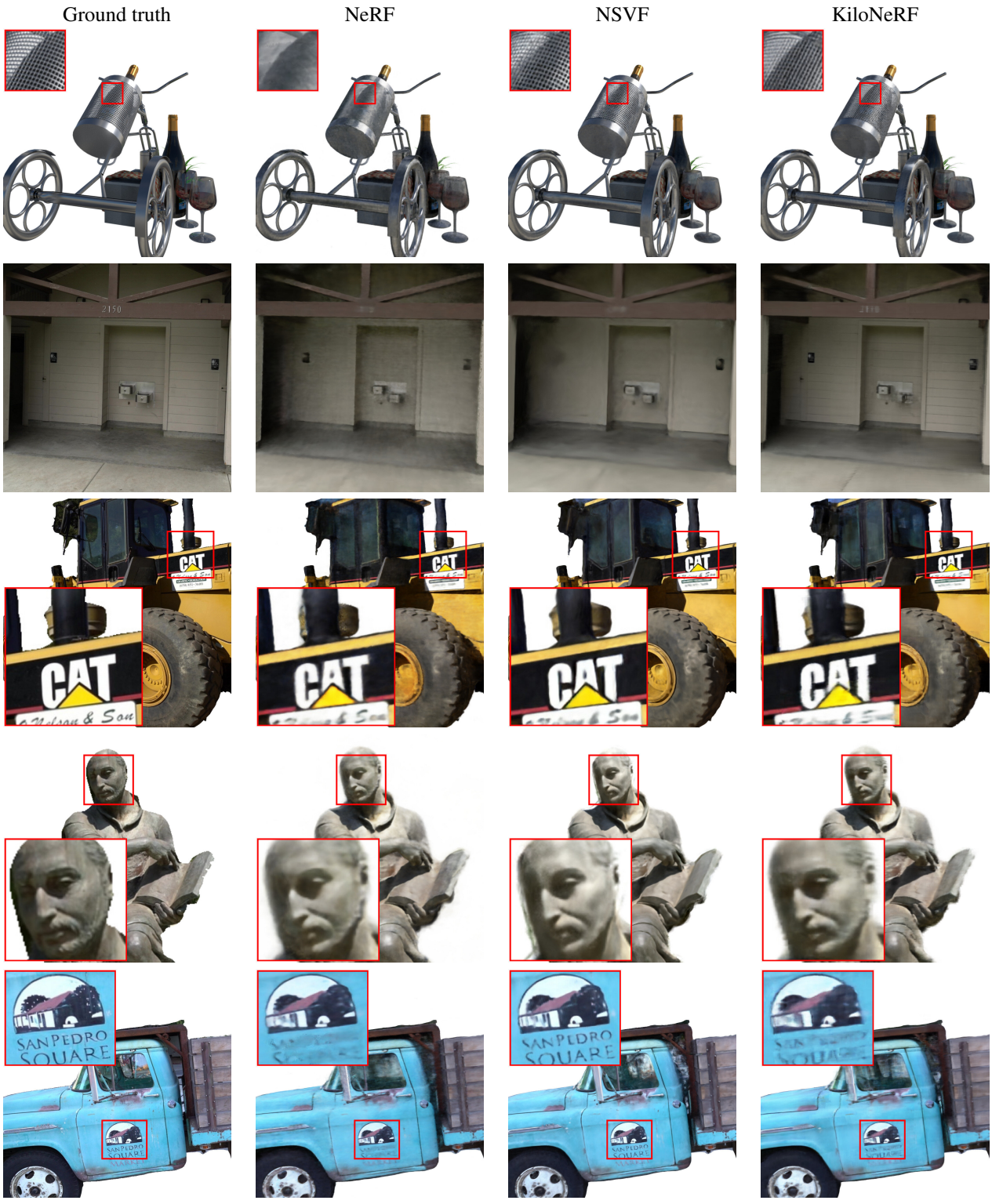

Figure 6: Qualitative results for the scenes Wineholder (Synthetic NSVF), Barn (Tanks & Temples), Caterpillar (Tanks & Temples), Ignatius (Tanks & Temples) and Truck (Tanks & Temples)

768 × 576		Character	Fountain	Jade	Statues
PSNR ↑	NeRF	29.43	28.04	26.52	**25.17**
	NSVF	27.95	27.73	26.96	24.97
	KiloNeRF	**29.44**	**28.50**	**27.14**	24.49
SSIM ↑	NeRF	**0.95**	0.91	0.89	0.87
	NSVF	0.92	0.91	0.90	0.86
	KiloNeRF	**0.95**	**0.93**	**0.91**	**0.88**
LPIPS ↓	NeRF	**0.03**	0.07	0.08	0.09
	NSVF	0.07	0.11	0.09	0.17
	KiloNeRF	0.04	**0.06**	**0.06**	**0.08**
Render time	NeRF	37266	37266	37266	37266
in ms ↓	NSVF	2291	5863	5633	3806
	KiloNeRF	**17**	**38**	**42**	**22**
Speedup	NSVF	16	6	7	10
over NeRF ↑	KiloNeRF	**2150**	**989**	**892**	**1715**

Table 2: **BlendedMVS**

800 × 800		Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship
PSNR ↑	NeRF	33.00	25.01	30.13	36.18	32.54	29.62	32.91	28.65
	NSVF	**33.19**	25.18	**31.23**	**37.14**	32.29	**32.68**	**34.27**	27.93
	KiloNeRF	32.91	**25.25**	29.76	35.56	**33.02**	29.20	33.06	**29.23**
SSIM ↑	NeRF	**0.97**	**0.93**	0.96	0.97	0.96	0.95	0.98	0.86
	NSVF	**0.97**	**0.93**	**0.97**	**0.98**	0.96	**0.97**	**0.99**	0.85
	KiloNeRF	**0.97**	**0.93**	**0.97**	**0.98**	**0.97**	0.95	0.98	**0.88**
LPIPS ↓	NeRF	0.05	0.09	0.04	0.12	0.05	0.06	0.03	0.21
	NSVF	0.04	0.07	**0.02**	0.03	0.03	**0.02**	**0.01**	0.16
	KiloNeRF	**0.02**	**0.05**	**0.02**	**0.02**	**0.02**	**0.02**	**0.01**	**0.08**
Render time	NeRF	56185	56185	56185	56185	56185	56185	56185	56185
in ms ↓	NSVF	2492	4130	3990	5599	4231	5521	2060	6730
	KiloNeRF	**18**	**25**	**19**	**29**	**22**	**29**	**22**	**43**
Speedup	NSVF	23	14	14	10	13	10	27	8
over NeRF ↑	KiloNeRF	**3181**	**2221**	**2954**	**1926**	**2548**	**1954**	**2523**	**1295**

Table 3: **Synthetic NeRF**

800 × 800		Bike	Lifestyle	Palace	Robot	Spaceship	Steamtrain	Toad	Wineholder
PSNR ↑	NeRF	25.17	32.26	33.57	33.57	34.66	33.42	30.77	28.97
	NSVF	**37.75**	**34.60**	34.05	**35.24**	**39.00**	**35.13**	**33.25**	**32.04**
	KiloNeRF	35.49	33.15	**34.42**	32.93	36.48	33.36	31.41	29.72
SSIM ↑	NeRF	0.87	0.95	0.95	0.95	0.98	0.98	0.94	0.94
	NSVF	**0.99**	**0.97**	**0.97**	**0.99**	**0.99**	**0.99**	**0.97**	**0.97**
	KiloNeRF	**0.99**	**0.97**	0.96	0.98	**0.99**	0.98	0.95	0.95
LPIPS ↓	NeRF	0.09	0.03	**0.02**	0.02	0.02	0.05	0.05	0.05
	NSVF	**0.00**	**0.02**	**0.02**	**0.01**	**0.01**	**0.01**	**0.03**	**0.02**
	KiloNeRF	0.01	**0.02**	**0.02**	0.02	**0.01**	**0.01**	0.04	0.03
Render time	NeRF	56185	56185	56185	56185	56185	56185	56185	56185
in ms ↓	NSVF	2520	23102	16929	2482	9023	10278	3264	16381
	KiloNeRF	**23**	**35**	**31**	**21**	**26**	**21**	**21**	**29**
Speedup	NSVF	22	2	3	23	6	5	17	3
over NeRF ↑	KiloNeRF	**2438**	**1586**	**1804**	**2739**	**2149**	**2670**	**2625**	**1958**

Table 4: **Synthetic NSVF**

1920 × 1080		Barn	Caterpillar	Family	Ignatius	Truck
PSNR ↑	NeRF	27.71	25.87	33.33	27.79	26.92
	NSVF	27.16	**26.44**	33.58	27.91	26.92
	KiloNeRF	**27.81**	25.61	**33.65**	**27.92**	**27.04**
SSIM ↑	NeRF	**0.85**	0.89	0.95	**0.94**	0.89
	NSVF	0.82	**0.90**	0.95	0.93	**0.90**
	KiloNeRF	**0.85**	**0.90**	**0.96**	**0.94**	**0.90**
LPIPS ↓	NeRF	0.19	0.12	0.05	0.08	0.11
	NSVF	0.31	0.14	0.06	0.11	0.15
	KiloNeRF	**0.16**	**0.10**	**0.04**	**0.06**	**0.10**
Render time	NeRF	182671	182671	182671	182671	182671
in ms ↓	NSVF	25142	17244	9314	8072	18715
	KiloNeRF	**136**	**80**	**70**	**56**	**114**
Speedup	NSVF	7	11	20	23	10
over NeRF ↑	KiloNeRF	**1341**	**2275**	**2598**	**3288**	**1606**

Table 5: **Tanks & Temples**

References

[1] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. Novel hpc techniques to batch execution of many variable size blas computations on gpus. In *International Conference on Supercomputing*, 2017. 2

[2] Matthew Tancik, Ben Mildenhall, Terrance Wang, Divi Schmidt, Pratul P. Srinivasan, Jonathan T. Barron, and Ren Ng. Learned initializations for optimizing coordinate-based neural representations. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2021. 4

[3] Lin Yen-Chen. PyTorchNeRF: a PyTorch implementation of NeRF, 2020. 1