NeRF-XL: Scaling NeRFs with Multiple GPUs

Ruilong Li[1,2], Sanja Fidler[1], Angjoo Kanazawa[2], and Francis Williams[1]

[1]NVIDIA [2]UC Berkeley

Abstract. We present NeRF-XL, a principled method for distributing Neural Radiance Fields (NeRFs) across multiple GPUs, thus enabling the training and rendering of NeRFs with an arbitrarily large capacity. We begin by revisiting existing multi-GPU approaches, which decompose large scenes into multiple independently trained NeRFs [9, 15, 17], and identify several fundamental issues with these methods that hinder improvements in reconstruction quality as additional computational resources (GPUs) are used in training. NeRF-XL remedies these issues and enables the training and rendering of NeRFs with an arbitrary number of parameters by simply using more hardware. At the core of our method lies a novel distributed training and rendering formulation, which is mathematically equivalent to the classic single-GPU case and minimizes communication between GPUs. By unlocking NeRFs with arbitrarily large parameter counts, our approach is the first to reveal multi-GPU scaling laws for NeRFs, showing improvements in reconstruction quality with larger parameter counts and speed improvements with more GPUs. We demonstrate the effectiveness of NeRF-XL on a wide variety of datasets, including the largest open-source dataset to date, MatrixCity [5], containing 258K images covering a 25km^2 city area. Visit our webpage at https://research.nvidia.com/labs/toronto-ai/nerfxl/ for code and videos.

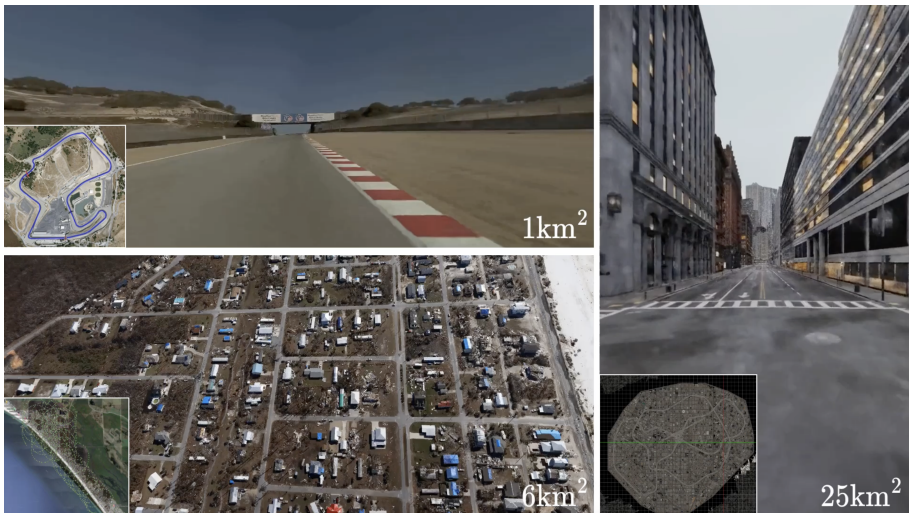


Fig. 1: Our principled multi-GPU distributed training algorithm enables scaling up NeRFs to arbitrarily-large scale.

1 Introduction

Recent advances in novel view synthesis have greatly enhanced our ability to capture Neural Radiance Fields (NeRFs), making the process significantly more accessible. These advancements enable the reconstruction of both larger scenes and finer details within a scene. Expanding the scope of a captured scene, whether by increasing the spatial scale (e.g., capturing a multi-kilometer-long cityscape) or the level of detail (e.g., scanning the blades of grass in a field), involves incorporating a greater volume of information into the NeRF for accurate reconstruction. Consequently, for scenes with high information content, the number of trainable parameters required for reconstruction may exceed the memory capacity of a single GPU.

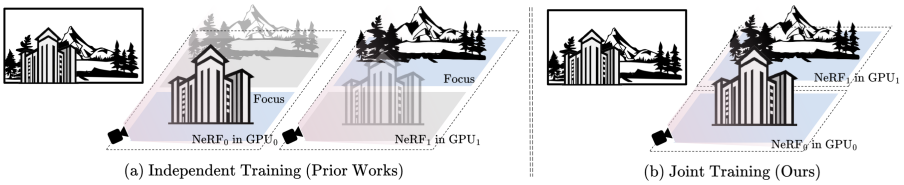
In this paper, we introduce NeRF-XL, a principled algorithm for efficiently distributing Neural Radiance Fields (NeRFs) across multiple GPUs. Our method enables the capture of high-information-content scenes, including those with large-scale and high-detail features, by simply adding more hardware resources. At its core, NeRF-XL allocates NeRF parameters across a disjoint set of spatial regions and trains them jointly across GPUs. Unlike conventional distributed training pipelines that synchronize gradients during the backward pass, our approach only requires information synchronization during the forward pass. Additionally, we drastically reduce the required data transfer between GPUs by carefully rewriting the volume rendering equation and relevant loss terms for the distributed setting. This novel rewriting enhances both training and rendering efficiency. The flexibility and scalability of our approach allows us to efficiently optimize NeRFs with an arbitrary number of parameters using multiple GPUs.

Our work contrasts with recent approaches that utilize multi-GPU algorithms to model large-scale scenes by training a set of independent NeRFs [9, 15, 17]. While these approaches require no communication between GPUs, each NeRF needs to model the entire space, including the background region. This leads to increased redundancy in the model's capacity as the number of GPUs grows. Additionally, these methods require blending NeRFs during rendering, which degrades visual quality and introduces artifacts in overlapping regions. Consequently, unlike NeRF-XL, these methods fail to achieve visual quality improvements as more model parameters (equivalent to more GPUs) are used in training.

We demonstrate the effectiveness of our method across a diverse set of captures, including street scans, drone flyovers, and object-centric videos. These range from small scenes (10m^2) to entire cities (25km^2). Our experiments show that NeRF-XL consistently achieves improved visual quality (measured by PSNR) and rendering speed as we allocate more computational resources to the optimization process. Thus, NeRF-XL enables the training of NeRFs with arbitrarily large capacity on scenes of any spatial scale and detail.

2 Related Work

Single GPU NeRFs for Large-Scale Scenes Many prior works have adapted NeRF to large-scale outdoor scenes. For example, BungeeNeRF [21] uses a



(a) Independent Training (Prior Works) (b) Joint Training (Ours)

Fig. 2: Independent Training v.s. Joint Training with multi-GPU. Training multiple NeRFs independently [9,15,18] requires each NeRF to model both the focused region and its surroundings, leading to redundancy in model's capacity. In contrast, our joint training approach utilizes non-overlapping NeRFs, thus without any redundancy.

multi-scale, coarse-to-fine pipeline to address memory constraints; Grid-guided NeRF [22] uses multiple image planes for drone-captured footage; F2-NeRF [19] introduces a space warping algorithm for efficient level-of-detail handling in a free camera trajectory capture; and UrbanNeRF [14] leverages LiDAR and segmentation maps to improve in-the-wild captures. Despite their advancements, these prior works are bounded by the computational capacity of a single GPU.

NeRFs with Multiple GPUs An alternative approach for training NeRFs on large-scale scenes is to use multiple GPUs. BlockNeRF [15], MegaNeRF [18] and SNISR [20] partition a scene into overlapping NeRFs based on camera trajectory or spatial content, and optimize each NeRF independently (one per GPU). ProgressiveNeRF [9] adopts a similar strategy but recursively optimizes one NeRF at a time with overlapped blending. While these methods overcome the memory limitations of a single GPU, each independent NeRF has to model the entire scene within a spatial region, leading to increased redudancy (in the model's capacity) and decreased visual quality as more GPUs are used in training. Furthermore, these methods must rely on depth initialization for spatial partitioning [20], or introduce overlapping between NeRFs [9,15,18], which causes visual artifacts during rendering. We provide an in-depth analysis of the problems faced by prior multi-GPU methods in the next section.

3 Revisiting Existing Approaches: Independent Training

In leveraging multiple GPUs for large-scale captures, prior research [8,9,15] has consistently employed the approach of training multiple NeRFs focusing on different spatial regions, where each NeRF is trained independently on its own GPU. *However, independently training multiple NeRFs has fundamental issues that impede visual-quality improvements with the introduction of additional resources (GPUs).* This problem is caused by three main issues described below.

Model Capacity Redundancy. The objective of training multiple independent NeRFs is to allow each NeRF to focus on a different (local) region and achieve better quality within that region than a single global model with the same capacity. Despite this intention, each NeRF is compelled to model not only its

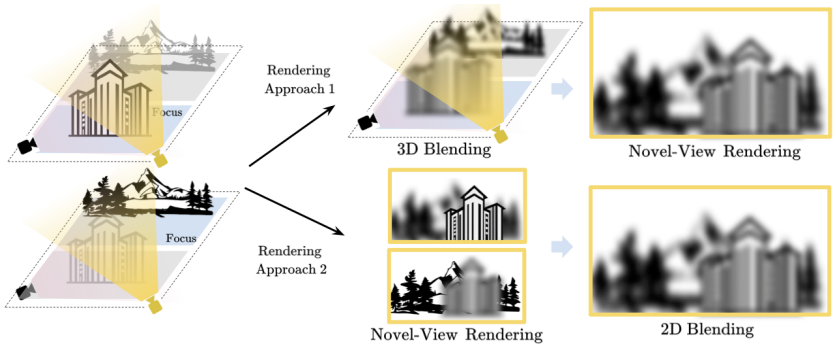


Fig. 3: Independent Training requires Blending for Novel-View Synthesis. Either blending in 2D [9, 15] or 3D [18] introduces blurriness into the rendering.

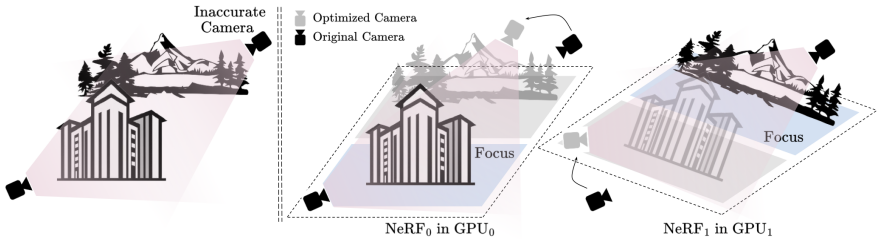


Fig. 4: Independent Training Creates Distinct Camera Optimizations. Camera optimization in NeRF can be achieved by either transforming the inaccurate camera itself or all other cameras along with the underlying 3D scene. Thus, training multiple NeRFs independently with camera optimization may lead to inconsistencies in camera corrections and scene geometry, causing more difficulties for blended rendering.

designated region but also the surrounding areas, since training rays often extend beyond the designated region as depicted in Figure 2(a). This leads to an inherent redundancy in the model's capacity since each NeRF must model both the local and surrounding regions. As a result, increasing the number of GPUs (and hence using smaller spatial regions per NeRF), increases the total redundancy in the model's capacity. For example, Mega-NeRF [18] exhibits 38%/56%/62% ray samples outside the tiled regions with 2×/4×/8× tiles on the UNIVERSITY4 capture. In contrast, our proposed method of jointly training all tiles removes the need for surrounding region modeling in each NeRF, *thereby completely eliminating redundancy*, as shown in Figure 2(b)). This feature is crucial for efficiently leveraging additional computational resources.

Blending for Rendering. When rendering independently trained NeRFs, it is often necessary to employ a blending strategy to merge the NeRFs and mitigate inconsistencies at the region boundaries. Past works typically choose local regions with a certain degree of overlap, such as 50% in Block-NeRF [15] and 15% in Mega-NeRF [18]. Two primary approaches exist for blending NeRFs during novel-view synthesis. One approach involves rendering each NeRF independently

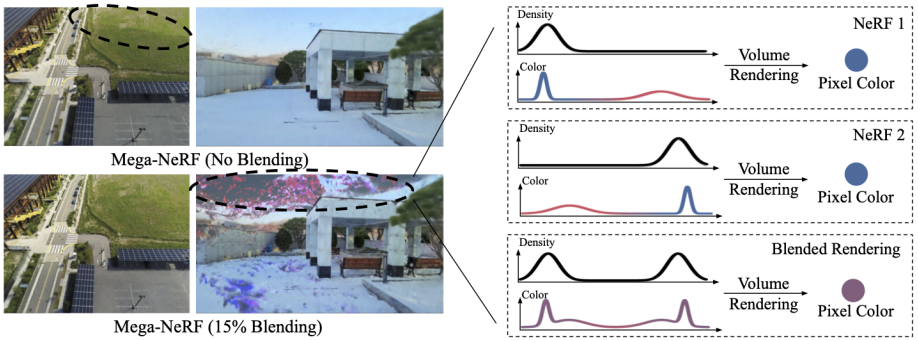


Fig. 5: Potential Artifacts Caused by 3D Blending. On the left we show Mega-NeRF results trained with 2 GPUs. At 0% overlap, boundary artifacts appear due to independent training, while at 15% overlap, severe artifacts appear due to 3D blending. On the right we illustrate the reason behind this artifact: while each independently trained NeRF renders the correct color, the blended NeRF do not guarantee correct color rendering.

and then blending the resulting images when the camera is positioned within the overlapped region (referred to as 2D blending) [9,15]. The alternative approach is to blend the color and density in 3D for ray samples within the overlapped region (referred to as 3D blending) [18]. As illustrated in Figure 3, both approaches can introduce blur into the final rendering. Moreover, blending in 3D can lead to more pronounced artifacts in rendering, due to deviations in the volume rendering equation, as demonstrated in Figure 5. In contrast, our joint training approach does not rely on any blending for rendering. In fact, our method renders the scene in the exact same way during training and inference, thereby eliminating the train-test discrepancies introduced by past methods.

Inconsistent Per-camera Embedding. In many cases, we need to account for things like white balance, auto-exposure, or inaccurate camera poses in a capture. A common approach to model these factors is by optimizing an embedding for each camera during the training process, often referred to as appearance embedding or pose embedding [6, 7, 16]. However, when training multiple NeRFs independently, each on its own GPU, the optimization process leads to independent refinements of these embeddings. This can result in inconsistent camera embeddings due to the inherently ambiguous nature of the task, as demonstrated in Figure 4. Inconsistencies in appearance embeddings across NeRFs can result in disparate underlying scene colors, while inconsistencies in camera pose embeddings can lead to variations in camera corrections and the transformation of scene geometry. These disparities introduce further difficulties when merging the tiles from multiple GPUs for rendering. Conversely, our joint training approach allows optimizing a single set of per-camera embeddings (through multi-GPU synchronization), thus completely eliminating these issues.

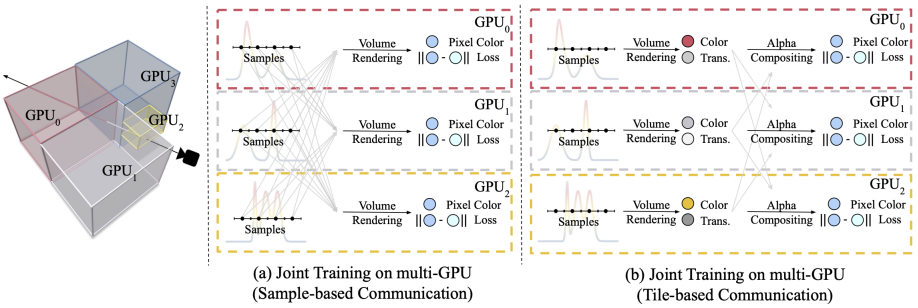


Fig. 6: Our Training Pipeline. Our method jointly trains multiple NeRFs across all GPUs, each of which covers a disjoint spatial region. The communication across GPUs only happens in the forward pass but not the backward pass (shown in gray arrows). (a) We can train this system by evaluating each NeRF to get the sample color and density, then broadcast these values to all other GPUs for a global volume rendering (§ 4.2). (b) By rewriting volume rendering equation we can dramatically reduce the data transfer to one value per-ray, thus improving efficiency (§ 4.3).

Due to the issues listed above, prior works [15,18] which train multiple independent NeRFs do not effectively harness the benefits of additional computational resources (GPUs) as they scale up, as evidenced in our experiments (§ 5). As a result, we advocate for a novel *joint* training approach. Without any heuristics, our approach gracefully enhances both visual quality and rendering speed as more GPUs are used in training, Moreover, our method reveals the multi-GPU scaling laws of NeRF for the first time.

4 Our Method: Joint Training

4.1 Background

Volume Rendering NeRFs [10] employ volume rendering to determine the pixel color through the integral equation:

$$\begin{aligned} C(t_n \to t_f) &= \int_{t_n}^{t_f} T(t_n \to t)\sigma(t)c(t)\,dt, \\ \text{where} \quad T(t_n \to t) &= \exp\left(-\int_{t_n}^{t} \sigma(s)\,ds\right). \end{aligned} \tag{1}$$

Here, $T(t_n \to t)$ represents transmittance, $\sigma(t)$ denotes density, and $c(t)$ signifies the color at position t along the ray.

Distortion Loss Initially introduced in Mip-NeRF 360 [1] and validated in subsequent works [2,4,16], this loss acts as a regularizer to address "floater" artifacts in NeRF reconstructions caused by limited training viewpoint coverage. It is

calculated along a ray as

$$\mathcal{L}_{dist}(t_n \to t_f) = \int_{t_n}^{t_f} w(t_i) w(t_j) \left| t_i - t_j \right| \, dt_i \, dt_j, \tag{2}$$

where $w(t) = T(t_n \to t)\sigma(t)$ represents the volume rendering weight for each sample along the ray. Intuitively, it penalizes floaters by encouraging density concentration in minimal, compact regions. See [1] for more details.

4.2 Non-overlapped NeRFs

A straightforward strategy to increase model capacity with multiple GPUs is to partition 3D space into tiles and allocate a NeRF for each tile. But unlike prior works [9, 15, 18] that employ overlapped NeRFs to model both tiles and their surrounding regions, our method exclusively models *non-overlapped* tiles, with each NeRF assigned to a single tile. This distinction is illustrated in Figure 2.

To render our NeRFs across multiple GPUs, we first distribute ray samples among GPUs based on the bounding box of the tiles. Notably it's important to ensure that sample intervals do not extend beyond tile boundaries to prevent overlap between samples. We subsequently query sample attributes (*i.e.* color and density) on each respective GPU. Volume rendering is then performed through a global gather operation, consolidating information across all GPUs onto a single GPU to compute the final pixel color. Since all sample intervals are non-overlapping, the scene can be rendered accurately following the volume rendering equation without the need for any blending.

Training proceeds in a similar fashion to rendering, except that during the forward pass *each* GPU performs the global gather operation (*i.e.* broadcast) to obtain the information (*i.e.* color and density) from all other GPUs (illustrated as gray lines in Figure 6(a)). Then, each GPU computes the loss locally and back-propagates the gradients to its own parameters. Notably the forward pass produces the exact same loss values on every GPU, but each loss lives in a different computational graph that only differentiates with respect to its own local parameters, thus no gradient communication are required across GPUs.

Such a naive scheme is extremely simple to implement, and mathematically identical to training and rendering a NeRF represented by multiple small NeRFs [12, 13] on a single large GPU. Distributing learnable parameters and computational cost across multiple GPUs allows scaling NeRF to scenes of any size, as well as making most parts of training fully parallel (*e.g.*, network evaluation, back-propagation). Despite its simplicity and scalability in comparison to blending overlapping NeRFs in prior works [9, 15, 18], this naive approach requires synchronizing $\mathcal{O}(SK^2)$ data across GPUs, where K is the number of GPUs, and S is the average number of samples per-ray per-GPU. As the number of GPUs increases or the ray step size decreases, synchronization across GPUs quickly becomes a bottleneck. Therefore, on top of this approach, we present a sophisticated solution that significantly alleviates the burden of multi-GPU synchronization in a principled manner.

4.3 Partition Based Volume Rendering

If we consider the near-far region $[t_n \to t_f]$ consisting of N segments $[t_1 \to t_2, t_2 \to t_3, ..., t_N \to t_{N+1}]$, we can rewrite the volume-rendering integral (1) into a sum of integrals for each segment along the ray:

$$C(t_1 \to t_{N+1}) = \int_{t_1}^{t_{N+1}} T(t_1 \to t)\sigma(t)c(t)dt = \sum_{k=1}^{N} T(t_1 \to t_k)C(t_k \to t_{k+1}) \tag{3}$$

in which the transmittance $T(t_1 \to t_k)$ can be written as:

$$T(t_1 \to t_k) = \prod_{i=1}^{k-1} T(t_i \to t_{i+1}) \tag{4}$$

The above equation states that volume rendering along an entire ray is equivalent to first performing volume rendering independently within each segment, then performing alpha compositing on all the segments. We can also rewrite the accumulated weights $A(t_1 \to t_{N+1})$ and depths $D(t_1 \to t_{N+1})$ in a similar way:

$$A(t_1 \to t_{N+1}) = \int_{t_1}^{t_{N+1}} T(t_1 \to t)\sigma(t)dt = \sum_{k=1}^{N} T(t_1 \to t_k)A(t_k \to t_{k+1}) \tag{5}$$

$$D(t_1 \to t_{N+1}) = \int_{t_1}^{t_{N+1}} T(t_1 \to t)\sigma(t)tdt = \sum_{k=1}^{N} T(t_1 \to t_k)D(t_k \to t_{k+1}) \tag{6}$$

We can further rewrite the point-based integral in the distortion loss as an accumulation across segments:

$$\begin{aligned}\mathcal{L}_{dist}(t_1 \to t_{N+1}) &= \int_{t_1}^{t_{N+1}} w(t_i)w(t_j)\left|t_i - t_j\right| dt_i dt_j \\ &= 2\sum_{k=1}^{N} T(t_1 \to t_k)S(t_1 \to t_k) + \sum_{k=1}^{N} T(t_1 \to t_k)^2 \mathcal{L}_{dist}(t_k \to t_{k+1})\end{aligned} \tag{7}$$

in which the $S(t_1 \to t_k)$ is defined as:

$$S(t_1 \to t_k) = D(t_k \to t_{k+1})A(t_1 \to t_k) - A(t_k \to t_{k+1})D(t_1 \to t_k) \tag{8}$$

Intuitively, the first term $S(t_1 \to t_k)$ penalizes multiple peaks across segments (zero if only one segment has non-zero values), while the second term $\mathcal{L}_{dist}(t_k \to t_{k+1})$ penalizes multiple peaks within the same segment. This transforms the pairwise loss on all samples into a hierarchy: pairwise losses within each segment, followed by a pairwise loss on all segments. Derivations for all the above formulae are given in the appendix.

Recall that the main drawback of our naive approach was an expensive per-sample data exchange across all GPUs. The above formulae convert sample-based composition to tile-based composition. This allows us to first reduce the per-sample data into per-tile data in parallel within each GPU and exchange only the per-tile data across all GPUs for alpha compositing. This operation is cost-effective, as now the data exchange is reduced from $O(KS^2)$ to $O(S^2)$ (each GPU contains a single tile). Figure 6(b) shows an overview of our approach. § 5.4 quantifies the improvement gained from this advanced approach compared to the naive version.

In addition to the volume rendering equation and distortion loss, a wide range of loss functions commonly used in NeRF literature can be similarly rewritten to suit our multi-GPU approach. For further details, we encourage readers to refer to the appendix.

4.4 Spatial Partitioning

Our multi-GPU strategy relies on spatial partitioning, raising the question of how to create these tiles effectively. Prior works [15, 18] opt for a straightforward division of space into uniform-sized blocks within a global bounding box. While suitable for near-rectangular regions, this method proves suboptimal for free camera trajectories and can lead to unbalanced compute assignment across GPUs. As noted in [19], a free camera trajectory involves uneven training view coverage, resulting in varying capacity needs across space (*e.g.*, the regions that are far away from any camera require less capacity than regions near a camera). To achieve balanced workload among GPUs, we want to ensure each GPU runs a similar number of network evaluations (*i.e.* has a similar number of ray samples). This balance not only allocates compute resources evenly but also minimizes waiting time during multi-GPU synchronization for communicating the data, as unequal distribution can lead to suboptimal GPU utilization.

We propose an efficient partitioning scheme aimed at evenly distributing workload across GPUs. When a sparse point cloud is accessible (*e.g.*, obtained from SFM), we partition the space based on the point cloud to ensure that each tile contains a comparable number of points. This is achieved by recursively identifying the plane where the Cumulative Distribution Function (CDF) equals 0.5 for the 3D point distribution along each axis. As a result, this approach leads to approximately evenly distributed scene content across GPUs. In cases where a sparse point cloud is unavailable, indicating a lack of prior knowledge about the scene structure, we instead discretize randomly sampled training rays into 3D samples. This serves as an estimation of the scene content distribution based on the camera trajectory, enabling us to proceed with partitioning in a similar manner. This process is universally applicable to various types of captures, including street, aerial, and object-centric data, and runs very quickly in practice (typically within seconds). Please refer to the appendix for visualizations of partitioned tiles on different captures.

	Garden [1]	University4 [9]	Building [18]	Mexico Beach [3]	Laguna Seca	MatrixCity [5]
#Img	161	939	1940	2258	27695	258003
#Pix$_c$	175M	1947M	1920M	2840M	47294M	25800M
#Pix$_d$	0.84M	3.98M	-	9.63M	2819M	2007M

Table 1: Data Statistics. Our experiments are conducted on these captures from various sources, including street captures (University4, MatrixCity, Laguna Seca), aerial captures (Building, Mexico Beach) and an object-centric 360-degree capture (Garden). These data span a wide range of scales, enabling a comprehensive evaluation of the multi-GPU system. Pix$_c$ and Pix$_d$ are denoted for color pixels and depth pixels, respectively.

5 Experiments

Datasets. The crux of a multi-GPU strategy lies in its ability to consistently improve performance across all types of captures, regardless of scale, as additional resources are allocated. However, prior works typically evaluate their methods using only a single type of capture (*e.g.*, street captures in Block-NeRF, aerial captures in Mega-NeRF). In contrast, our experiments are conducted on diverse captures from various sources, including street captures (University4 [9], MatrixCity [5], Laguna Seca[1]), aerial captures (Building [18], Mexico Beach [3]) and an object-centric 360-degree capture (Garden [1]). These data also span a wide range of scales, from Garden with 161 images in a 10m^2 area, to MatrixCity with 258K images in a 25km^2 area, thereby offering a comprehensive evaluation of the multi-GPU system. Table 1 shows detailed statistics for each of these captures.

5.1 Joint Training v.s. Independent Training

In this section, we conduct a comparative analysis between our proposed approach and two prior works, Block-NeRF [15] and Mega-NeRF [18], all of which are aimed at scaling up NeRFs beyond the constraints of a single GPU. To ensure a fair evaluation solely on the multi-GPU strategies, we re-implemented each baseline alongside our method within a unified framework[2]. Each method is configured with the same NeRF representation (Instant-NGP [11]), spatial skipping acceleration structure (Occupancy Grid [11]), distortion loss [1], and multi-GPU parallel inference. This standardized setup allows us to focus on assessing the performance of different multi-GPU strategies in both training (*i.e.*, joint vs. independent [15, 18]) and rendering (*i.e.*, joint vs. 2D blending [15] vs. 3D blending [18]). For each baseline method, we adopt their default overlapping configurations, which is 15% for Mega-NeRF and 50% for Block-NeRF. All methods are trained for the same number of iterations (20K), with an equal

[1] Laguna Seca: An in-house capture of a 3.6km race track.

[2] On Building scene, our 8 GPU Mega-NeRF implementation achieves 20.8 PSNR comparing to 20.9 PSNR reported in the original paper.

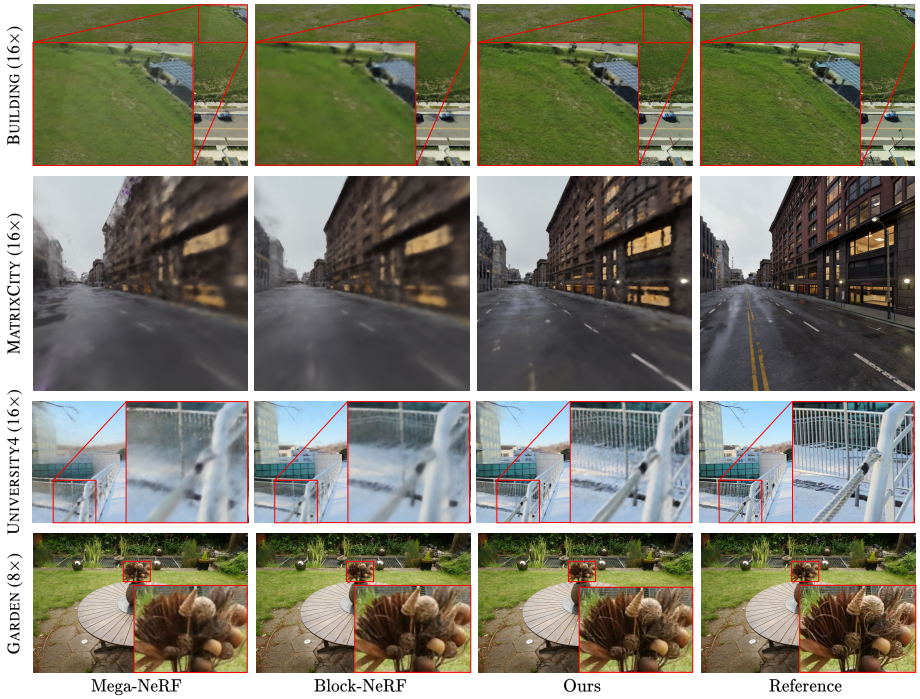


Fig. 7: Qualitative Comparison. Comparing to prior works, our method efficiently harnesses multi-GPU setups for performance improvement on all types of data.

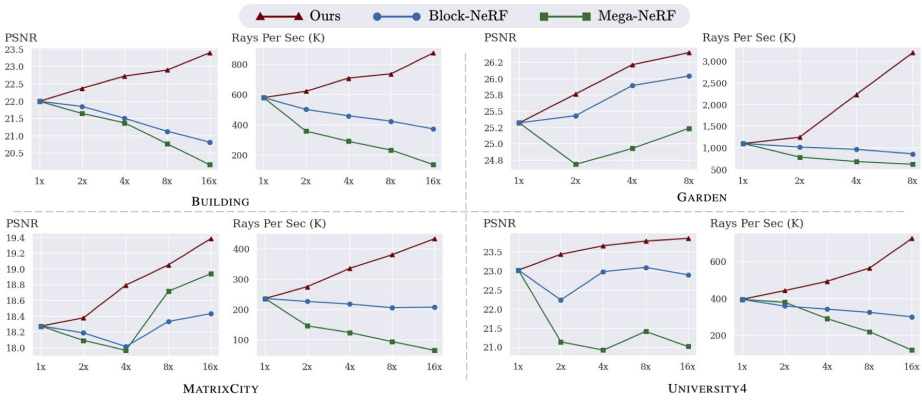


Fig. 8: Quantitative Comparison. Prior works based on independent training fails to realize performance improvements with additional GPUs, while our method enjoys improved rendering quality and speed as more resources are added to training.

number of total samples per iteration (effectively the batch size of the model). Please refer to the appendix for implementation details.

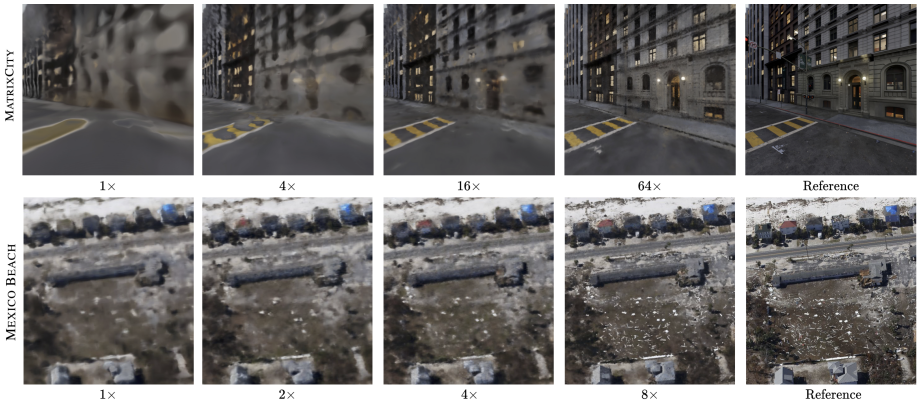


Fig. 9: Scalability of Our Approach. More GPUs allow for more learnable parameters, leading to larger model capacity with better quality.

In this section we conduct experiments on four captures, including Garden [1], Building [18], University4 [9] and MatrixCity [5], with GPU configurations ranging from 1× to 16× (multi-node). We evaluate the scalability of each method using two key metrics: Peak Signal-to-Noise Ratio (PSNR) for quality assessment and Rays Per Second for rendering speed, on the respective test sets of each capture. As illustrated in Figure 8, baseline approaches struggle to improve rendering quality with an increase in the number of GPUs, largely due to the inherent issues associated with independent training outlined in § 3. Additionally, baseline methods also fails to achieve faster rendering with additional GPUs, as they either need to evaluate duplicate pixels for 2D blending [15] or duplicate 3D samples for 3D blending [18]. In contrast, our proposed approach, employing joint training and rendering, effectively eliminates model redundancy and train-test discrepancy. Thus, it gracefully benefits from increased parameters and parallelization with additional GPUs, resulting in nearly linear improvements in both quality and rendering speed. More qualitative comparisons are shown in Figure 7.

5.2 Robustness and Scalability

We further evaluate the robustness and scalability of our approach by testing it on larger scale captures with increased GPU resources. Specifically, Figure 10 showcases our novel-view rendering results on the 1km^2 Laguna Seca with 8 GPUs, the 6km^2 Mexico Beach [3] with 8 GPUs, and the 25km^2 MatrixCity [5] with 64 GPUs. It's noteworthy that each of these captures entails billions of pixels (see Table 1), posing a significant challenge to the NeRF model's capacity due to the vast amount of information being processed.

Figure 9 presents qualitative results obtained using our approach, highlighting how the quality improves with the incorporation of more parameters through

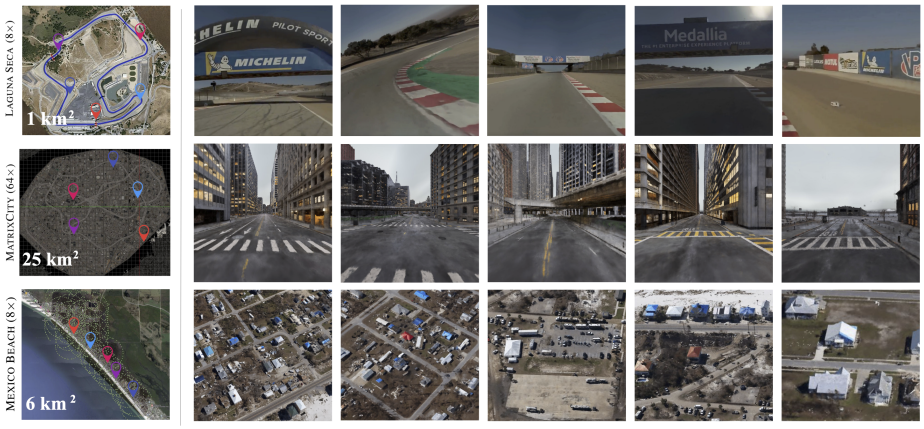


Fig. 10: More Rendering Results on Large Scale Captures. We test the robustness of our approach on larger captures with more GPUs. Please refer to the our webpage for video tours on these data.

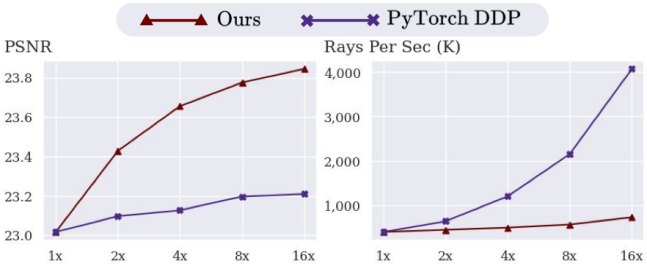


Fig. 11: Comparison with PyTorch DDP on University4. PyTorch Distributed Data Parallel (DDP) is designed for faster rendering by distributing rays across GPUs. In contrast, our approach distributes parameters across GPUs, scaling beyond the memory limits of single GPU in the cluster, and enabling larger model capacity for better quality.

the utilization of additional GPUs. Please refer to our webpage for the video rendering.

5.3 Comparison with PyTorch DDP

Another common approach to utilize multi-GPU for NeRF is distributing rays across GPUs, *e.g.*, PyTorch's Distributed Data Parallel (DDP). This method typically allows for larger batch sizes during training or faster rendering through increased parallelization. However, DDP necessitates that all GPUs host *all* model parameters, thus limiting the model's capacity to the memory of a single GPU. In contrast, our approach assigns each GPU to handle a distinct 3D tiled region, aiming to alleviate memory constraints and ensure optimal quality even for large-

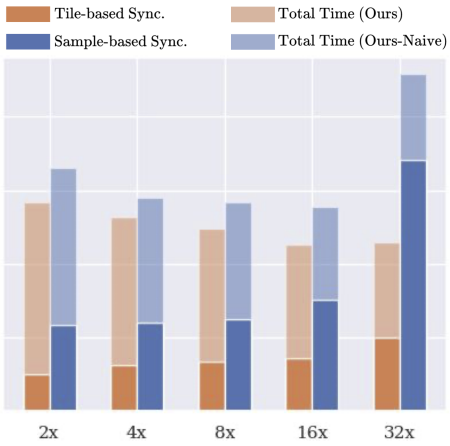


Fig. 12: Synchronization Cost on UNIVERSITY4. Our partition-based volume rendering (§ 4.3) allows tile-based communication, which is much cheaper than the naive sample-based communication (§ 4.2), thus enabling faster rendering.

scale scenes. Figure 11 illustrates a comparison between our method and DDP on the UNIVERSITY4 dataset. In this comparison, our method employs N× more parameters while DDP trains with N× more rays on N GPUs. The substantial improvement in PSNR indicates that large-scale NeRF benefits more from increased model capacity than from training more rays, a benefit uniquely enabled by our approach. However, DDP renders much faster than our approach due to the balanced workload created by parallelizing rays across GPUs. In contrast, our approach does not guarantee balanced workload distribution and consequently suffers from multi-GPU synchronization in run-time.

5.4 Multi-GPU Communication

We report the profiling results of multi-GPU communication costs on the UNIVERSITY4 capture in Figure 12. Despite achieving a reduction in communication costs by over 2× through partition-based volume rendering (tile-based vs. sample-based synchronization), multi-GPU communication remains the primary bottleneck of our system. We attribute this to imbalanced workload distribution across GPUs, which could potentially be addressed through better spatial partitioning algorithms. We leave this optimization for future exploration.

6 Conclusion and Limitation

In conclusion, we revisited the existing approaches of decomposing large-scale scenes into independently trained NeRFs, and identified significant issues that impeded the effective utilization of additional computational resources (GPUs),

thereby contradicting the core objective of leveraging multi-GPU setups to improve large-scale NeRF performance. Consequently, we introduced NeRF-XL, a principled algorithm to efficiently harness multi-GPU setups, and enhance NeRF performance at any scale by jointly training multiple non-overlapped NeRFs. Importantly, our method does not rely on any heuristics, and adheres to scaling laws for NeRF in the multi-GPU setting across various types of data.

However, our approach still has limitations. Similar to any other multi-GPU distributed setup, synchronization and communication overhead is inevitable in our joint training approach, which results in a slightly slower training speed ($1\times$-$1.5\times$) compared to baselines with independent training. Additionally, while our distributed approach is agnostic to NeRF representation in theory, we have been only experimented with a popular choice, Instant-NGP [11], that equips with hash grids and MLPs. It will be an interesting future work to apply the framework to other representations, even beyond the task of static scene novel-view synthesis.

7 Acknowledgement

This project is supported in part by IARPA DOI/IBC 140D0423C0035. We would like to thank Brent Bartlett and Tim Woodard for providing and helping with processing the Mexico Beach data.

A Implementation Details

In this section, we elaborate on the implementation intricacies of our multi-GPU NeRF representation, multi-GPU volume rendering and distortion loss, as well as the spatial partitioning strategy we employed. Additionally, we outline detailed differences between the baseline approaches and our method from the implementation perspective.

A.1 NeRF Representation

Our experiments are all conducted with the hash-grid based NeRF representation introduced in Instant-NGP [11]. In single GPU experiments, we adhere to the original configuration of Instant-NGP. This entails predicting sample density through hash encoding followed by a one-layer MLP, and predicting sample color through another two-layer MLP. The latter is conditioned on the view direction and, optionally, an appearance embedding.

In the multi-GPU joint training scenario, we assign each GPU a NeRF with its own independent hash encoding and density MLP. However, for the color MLP, we adopt a different approach. As the sample color is conditioned on the view direction and optionally an appearance embedding, which can be out of training distribution during novel-view rendering, independent color MLPs will lead to inconsistent color prediction given the same input. Thus we adopt parameter sharing across all GPUs for the color MLP utilizing Distributed Data Parallel (DDP), to ensure consistent interpretation of novel view direction and appearance embedding among all GPUs. Notably, when enabled during training, an appearance embedding of zero is utilized for rendering novel-view images. Similarly, when camera optimization is enabled, the camera pose embedding of each camera is also shared across GPUs.

While our experiments specifically focus on the Instant-NGP representation, we believe our approach can be generalized to many other NeRF representations. The Instant-NGP representation combines grid-based and MLP elements, suggesting that our multi-GPU distribution strategy is applicable to both grid-based and MLP-based NeRFs, as long as each NeRF is confined within a non-overlapping bounding box for proper integration. In essence, regardless of which representation is used, our approach simply increases its model capacity by deploying multiple instantiations with multi-GPU, while enabling parameter sharing across them. Consequently, all instantiations are conceptually united as a "single NeRF" with spatially partitioned parameters.

A.2 Multi-GPU Volume Rendering and Distortion Loss

Algorithm 1 and 2 demonstrate the implementation of multi-GPU volume rendering and distortion loss on each GPU, corresponding to the formulations presented in § 4.3 of the main paper. Both algorithms take in locally integrated data (e.g., $T_{k\to k+1}$, $C_{k\to k+1}$) within the k-th GPU, and perform a global aggregation of integrated data from all GPUs. This ensures identical outputs on

Algorithm 1: Volume Rendering on k-th GPU

Data: $T_{k\to k+1}$; $C_{k\to k+1}$; $I = \{i_1, i_2, ...i_N\}$;
Result: $C_{1\to N+1}$;
$T_{1\to i} \leftarrow 1$;
$C_{1\to N+1} \leftarrow 0$;
for $s \leftarrow 1$ **to** N **do**
  /* s-th segment is from i-th GPU. */
  $i \leftarrow I[s]$;
  if $i \neq k$ **then**
    /* Get the data from i-th GPU with auto-grad disabled. */
    $T_{i\to i+1}, C_{i\to i+1} \leftarrow \text{gather}(i)$;
  end
  /* Global composition. */
  $C_{1\to N+1}$ += $T_{1\to i} \times C_{i\to i+1}$;
  $T_{1\to i}$ *= $T_{i\to i+1}$;
end

Algorithm 2: Distortion Loss on k-th GPU

Data: $T_{k\to k+1}$; $C_{k\to k+1}$, $I = \{i_1, i_2, ...i_N\}$;
  $A_{k\to k+1}$, $D_{k\to k+1}$; $\mathcal{L}_{k\to k+1}$;
Result: $\mathcal{L}_{1\to N+1}$;
$T_{1\to i} \leftarrow 1$;
$A_{1\to i} \leftarrow 0$;
$D_{1\to i} \leftarrow 0$;
$\mathcal{L}_{1\to N+1} \leftarrow 0$;
for $s \leftarrow 1$ **to** N **do**
  /* s-th segment is from i-th GPU. */
  $i \leftarrow I[s]$;
  if $i \neq k$ **then**
    /* Get the data from i-th GPU with auto-grad disabled. */
    $T_{i\to i+1}, C_{i\to i+1} \leftarrow \text{gather}(i)$;
    $A_{i\to k+1}, D_{i\to i+1}, \mathcal{L}_{i\to i+1} \leftarrow \text{gather\_loss}(i)$;
  end
  /* Global composition. */
  $S_{1\to i} \leftarrow D_{k\to k+1} \times A_{1\to i} - A_{k\to k+1} \times D_{1\to i}$;
  $\mathcal{L}_{1\to N+1}$ += $T^2_{1\to i} \times \mathcal{L}_{i\to i+1} + T_{1\to i} \times S_{1\to i}$;
  $A_{1\to i}$ += $T_{1\to i} \times A_{i\to i+1}$;
  $D_{1\to i}$ += $T_{1\to i} \times D_{i\to i+1}$;
  $T_{1\to i}$ *= $T_{i\to i+1}$;
end

every GPU, but each lives in a distinct computational graph that is only differentiable with respect to it's respective NeRF parameters. During inference, data gathering is only required on a single GPU (e.g., 0-th GPU) to render the final image, necessitating less data transfer compared to training. While we present

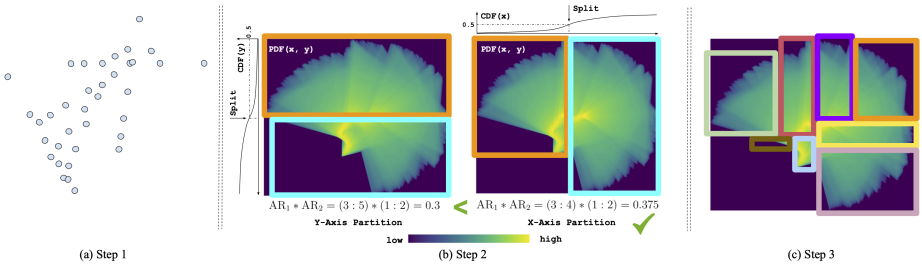


Fig. 13: Our Spatial Partitioning Approach. We partition the space such that each region contains similar amount of spatial content, which helps balance the compute across multi-GPUs. **(a)** Initially, we generate a point cloud either through Structure from Motion (SFM) or by discretizing training rays into samples. **(b)** Subsequently, we construct a Probability Density Function (PDF) for the point cloud and partition the space accordingly. This process is repeated for each axis, selecting the partitioning scheme that results in partitions as close to cubic as possible (denoted by AR_i, representing the aspect ratio for each partition). **(c)** Finally, we recursively apply step (b) for n iterations to achieve 2^n partitions.

our algorithms here as a loop over all GPUs for clarity, in practice, this is accomplished through batched asynchronous send/receive operations executed in parallel. The global composition is implemented with parallel prefix scan using the NerfAcc [4] toolbox.

A.3 Spatial Partitioning

Figure 13 illustrate our partition scheme. Initially, we create a point cloud from either SFM or by discretizing random training rays into samples, forming a Probability Density Function (PDF) for the distribution. By computing the Cumulative Distribution Function (CDF) along each axis x, y, z in 3D space, we identify the candidate planes where the CDF equals 0.5, signifying an optimal separation that evenly divides the space into two partitions. To create nearly cubic partitions, we choose the plane yielding partitions whose aspect ratios are as close to 1 as possible. We apply this process recursively within each partition, generating a power-of-two number of tile $(2\times, 4\times, 8\times, ...)$ for our distributed NeRFs. Figure 14 provides a visualization on the partitions we get on various captures.

A.4 Baselines

Below, we detail the implementation variances between the baseline approaches (Block-NeRF [15], Mega-NeRF [18]) and our method, all of which are integrated into the same system with identical configurations, including NeRF representation, spatial skipping acceleration structure, distortion loss, and multi-GPU parallel inference.

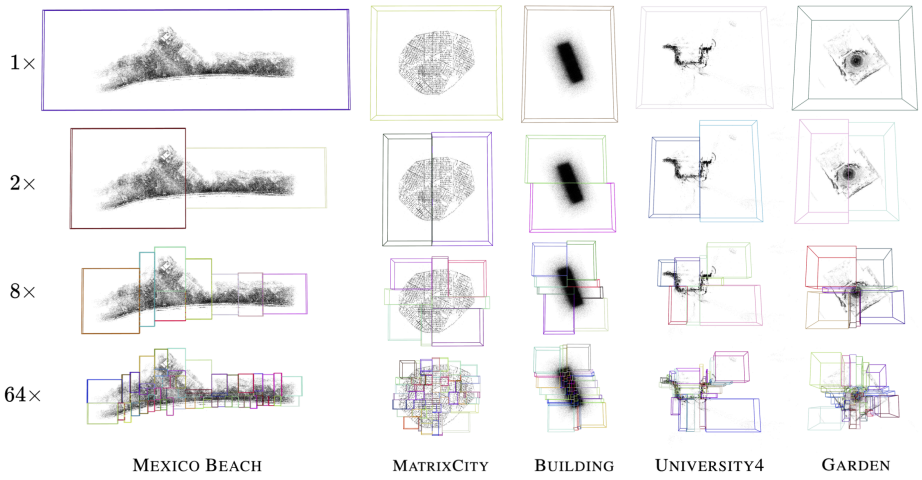


Fig. 14: Our Spatial Partitioning Results. Our partitioning approach is versatile and applicable to various types of captures, including drone footage (Mexico Beach, Building), street capture (MatrixCity, University4), and 360-degree object-centric capture (Garden). For all captures except Building, partitioning is based on SFM sparse points. However, due to the absence of point cloud data, the Building capture is partitioned based on discretizing training rays into samples. Our partition strategy ensures an even distribution of points across tiles, thereby naturally demonstrating a level-of-detail property. This means that regions with more content or those more frequently captured will have finer bounding boxes, enhancing granularity where necessary.

Ours. Our method partitions the space into non-overlapping tiles, with each NeRF assigned to a specific tile. All NeRFs are jointly trained using all training rays.

Mega-NeRF. The original Mega-NeRF paper employs uniform spatial partitioning, which may be suboptimal for free-trajectory captures [19]. To ensure a fair comparison, we apply the same spatial partitioning scheme as our approach. However, Mega-NeRF trains each NeRF independently, necessitating background modeling for every NeRF. To address this, we employ the scene contraction method from Mip-NeRF 360 [1], enabling each NeRF to focus on its assigned spatial region while also modeling the background. Consistent with the original approach, we utilize only the rays that intersect with the bounding box of each tile during NeRF training.

Block-NeRF. Instead of partitioning the space into tiles, the original Block-NeRF paper partitions it based on training cameras. Specifically, it divides the training data by grouping nearby cameras with some overlap, resulting in each NeRF focusing on a different spatial region during independent training. Since background modeling is required for each NeRF, and it is non-trivial to assign

each NeRF a distinct bounding box from a group of cameras, scene contraction does not apply. Therefore, we utilize the same bounding box covering the entire region for all independently trained NeRFs.

B Math

Here we first provide derivations of the proposed partitioned volume rendering described in § 4.3 in the main paper. Then we show that not only the volume rendering equation and distortion loss can be properly distributed across GPUs, there is a family of functions can be distributed in the same way.

B.1 Derivations of Partitioned Volume Rendering

Here we provide the full derivations of the equations in § 4.3, where we turn the volume rendering and distortion loss in the region of $[t_n \to t_f]$ into the regions of N segments $[t_1 \to t_2, t_2 \to t_3, ..., t_N \to t_{N+1}]$.

Transmittance Firstly, given that

$$T(t_1 \to t) = \exp\left(-\int_{t_1}^{t} \sigma(s)ds\right) \tag{9}$$

We can easily see that

$$\begin{aligned} T(t_1 \to t) &= \exp\left(-\int_{t_1}^{t} \sigma(s)ds\right) \\ &= \exp\left(-\int_{t_1}^{t_k} \sigma(s)ds - \int_{t_k}^{t} \sigma(s)ds\right) \\ &= \exp\left(-\int_{t_1}^{t_k} \sigma(s)ds\right) \exp\left(-\int_{t_k}^{t} \sigma(s)ds\right) \\ &= T(t_1 \to t_k)T(t_k \to t) \end{aligned} \tag{10}$$

And similarly:

$$\begin{aligned} T(t_1 \to t_k) &= \exp\left(-\int_{t_1}^{t_k} \sigma(s)ds\right) \\ &= \exp\left(-\sum_{k=1}^{k-1}\int_{t_k}^{t_{k+1}} \sigma(s)ds\right) \\ &= \prod_{k=1}^{k-1} \exp\left(-\int_{t_k}^{t_{k+1}} \sigma(s)ds\right) \\ &= \prod_{k=1}^{k-1} T(t_k \to t_{k+1}) \end{aligned} \tag{11}$$

which is the *Equation 4* in the main paper.

Accumulated Colors, Weights and Depths Given the volume rendering equation that accumulates colors $c(t)$ within the range of $[t_n \to t_f]$ along the ray:

$$C(t_n \to t_f) = \int_{t_n}^{t_f} T(t_n \to t)\sigma(t)c(t)dt \tag{12}$$

We can derive the partitioned version of it (*Equation 4* in the main paper) with the help of $T(t_1 \to t) = T(t_1 \to t_k)T(t_k \to t)$ that we just derived in Equation 10:

$$\begin{aligned} C(t_1 \to t_{N+1}) &= \int_{t_1}^{t_{N+1}} T(t_1 \to t)\sigma(t)c(t)dt \\ &= \sum_{k=1}^{N}\left[\int_{t_k}^{t_{k+1}} T(t_1 \to t)\sigma(t)c(t)dt\right] \\ &= \sum_{k=1}^{N}\left[\int_{t_k}^{t_{k+1}} T(t_1 \to t_k)T(t_k \to t)\sigma(t)c(t)dt\right] \\ &= \sum_{k=1}^{N}\left[T(t_1 \to t_k)\int_{t_k}^{t_{k+1}} T(t_k \to t)\sigma(t)c(t)dt\right] \\ &= \sum_{k=1}^{N} T(t_1 \to t_k)C(t_k \to t_{k+1}) \end{aligned} \tag{13}$$

The accumulated weights and depths (*Equation 5 and 6* in the main paper) can be derived in similar ways with the accumulated colors, thus we omit their derivations here.

Distortion Loss The original distortion loss has the form of[3]

$$\mathcal{L}_{dist}(t_1 \to t_{N+1}) = \iint_{t_1}^{t_{N+1}} w(t_1 \to u) w(t_1 \to v) \left| u - v \right| du dv \tag{14}$$

in which $w(t_1 \to u) = T(t_1 \to u)\sigma(u)$ represents the volume rendering weight for the sample at location u along the ray.

First, we can break the double integral $\iint_{t_1}^{t_{N+1}}$ into two terms, $\int_{t_1}^{t_{N+1}} \int_{t_1}^{u}$ and $\int_{t_1}^{t_{N+1}} \int_{t_1}^{v}$, in which the first term covers all $u \geq v$ and the second term covers all $v \geq u$:

$$\begin{aligned} \mathcal{L}_{dist}(t_1 \to t_{N+1}) &= \iint_{t_1}^{t_{N+1}} w(t_1 \to u) w(t_1 \to v) \left| u - v \right| du dv \\ &= \int_{t_1}^{t_{N+1}} w(t_1 \to u) \left[\int_{t_1}^{u} w(t_1 \to v) \left(u - v \right) dv \right] du \\ &\quad + \int_{t_1}^{t_{N+1}} w(t_1 \to v) \left[\int_{t_1}^{v} w(t_1 \to u) \left(v - u \right) du \right] dv \end{aligned} \tag{15}$$

Since u and v are symmetric notations, it is evident that the two terms above are equal. Thus:

$$\begin{aligned} \mathcal{L}_{dist}(t_1 \to t_{N+1}) &= 2 \int_{t_1}^{t_{N+1}} w(t_1 \to u) z(t_1 \to u) du \\ \text{where} \quad z(t_1 \to u) &= \int_{t_1}^{u} w(t_1 \to v) \left(u - v \right) dv \end{aligned} \tag{16}$$

Assumes the sample u belongs to the k-th segment (i.e., $t_k \leq u \leq t_{k+1}$), then we can break the above term $z(t_1 \to u)$ into the integral up to t_k plus the integral

[3] The distortion loss in this supplemental material has slightly different notations comparing to the equation we have in the main paper. We substitute t_k with u and t_j with v to reduce confusion in the following derivations.

from t_k to u:

$$\begin{aligned} z(t_1 \to u) &= \int_{t_1}^{u} w(t_1 \to v)\,(u - v)\,dv \\ &= \int_{t_1}^{t_k} w(t_1 \to v)\,(u - v)\,dv + \int_{t_k}^{u} w(t_1 \to v)\,(u - v)\,dv \\ &= z(t_1 \to t_k) + T(t_1 \to t_k) \cdot z(t_k \to u) \end{aligned} \tag{17}$$

Substituting the term $z(t_1 \to u)$ in Equation 16 with Equation 17, we now have the distortion loss in two terms:

$$\begin{aligned} \mathcal{L}_{dist}(t_1 \to t_{N+1}) = 2 &\int_{t_1}^{t_{N+1}} w(t_1 \to u) z(t_1 \to t_k) du \\ &+ 2 \cdot T(t_1 \to t_k) \cdot \int_{t_1}^{t_{N+1}} w(t_1 \to u) z(t_k \to u) du \end{aligned} \tag{18}$$

Then we break the integral $\int_{t_1}^{t_{N+1}}$ along the entire ray into the summation of integral on each segment $[t_k, t_{k+1}]$:

$$\begin{aligned} \mathcal{L}_{dist}(t_1 \to t_{N+1}) &= 2 \sum_{k=1}^{N} (\mathcal{L}_1 + T(t_1 \to t_k) \cdot \mathcal{L}_2) \\ \text{where} \quad \mathcal{L}_1 &= \int_{t_k}^{t_{k+1}} w(t_1 \to u) z(t_1 \to t_k) du \\ \mathcal{L}_2 &= \int_{t_k}^{t_{k+1}} w(t_1 \to u) z(t_k \to t_u) du \end{aligned} \tag{19}$$

Focusing on $\mathcal{L}_1$, with the help of Equation 16 we have:

$$\begin{aligned}
\mathcal{L}_1 &= \int_{t_k}^{t_{k+1}} w(t_1 \nrightarrow u) z(t_1 \nrightarrow t_k) du \\
&= T(t_1 \nrightarrow t_k) \cdot \int_{t_k}^{t_{k+1}} w(t_1 \nrightarrow u) \left[\int_{t_k}^{t_k} w(t_1 \nrightarrow v) \, (u - v) \, dv \right] du \\
&= T(t_1 \nrightarrow t_k) \cdot \left[\int_{t_k}^{t_{k+1}} w(t_1 \nrightarrow u) u du \right] \cdot \left[\int_{t_1}^{t_k} w(t_1 \nrightarrow v) dv \right] \\
&\quad - T(t_1 \nrightarrow t_k) \cdot \left[\int_{t_k}^{t_{k+1}} w(t_1 \nrightarrow u) du \right] \cdot \left[\int_{t_1}^{t_k} w(t_1 \nrightarrow v) v dv \right]
\end{aligned} \tag{20}$$

Recall that the accumulated weights and depths (*Equation 5 and 6* in the main paper) have the formulations of:

$$A(t_1 \nrightarrow t_{N+1}) = \int_{t_1}^{t_{N+1}} w(t_1 \nrightarrow u) du \tag{21}$$

$$D(t_1 \nrightarrow t_{N+1}) = \int_{t_1}^{t_{N+1}} w(t_1 \nrightarrow u) u du \tag{22}$$

Substituting the terms in Equation 20 with the above two equations we get the *Equation 9* in the main paper:

$$\mathcal{L}_1 = T(t_1 \nrightarrow t_k) S(t_1 \nrightarrow t_k) \tag{23}$$

in which the $S(t_1 \nrightarrow t_k)$ is defined as:

$$S(t_1 \nrightarrow t_k) = D(t_k \nrightarrow t_{k+1}) A(t_1 \nrightarrow t_k) - A(t_k \nrightarrow t_{k+1}) D(t_1 \nrightarrow t_k) \tag{24}$$

Now focusing on $\mathcal{L}_2$, we can see that:

$$\begin{aligned}
\mathcal{L}_2 &= \int_{t_k}^{t_{k+1}} w(t_1 \nrightarrow u) z(t_k \nrightarrow t_u) du \\
&= T(t_1 \nrightarrow t_k) \int_{t_k}^{t_{k+1}} w(t_k \nrightarrow u) z(t_k \nrightarrow t_u) du \\
&= T(t_1 \nrightarrow t_k) \cdot \frac{\mathcal{L}_{dist}(t_k \nrightarrow t_{k+1})}{2}
\end{aligned} \tag{25}$$

Putting together the above Equations 19, 23, 25, we then have derived the *Equation 7* in the main paper:

$$\mathcal{L}_{dist}(t_1 \to t_{N+1}) = 2\sum_{k=1}^{N} T(t_1 \to t_k)S(t_1 \to t_k) + \sum_{k=1}^{N} T(t_1 \to t_k)^2 \mathcal{L}_{dist}(t_k \to t_{k+1}) \tag{26}$$

B.2 General Derivations for More NeRF-related Functions

In this section we prove that there is a family of integral functions defined in the range of $[t_1, t_{N+1}]$ can be rewritten into the form of sum product on the integrals of each individual segment $[t_k, t_{k+1}]$, which makes them suitable to be distributed across multiple GPUs using our approach (*i.e.* first calculate each segment independently within each GPU, then accumulate only per-ray data across GPUs).

$$\mathcal{F}(t_1 \to t_{N+1}) = \sum_{i=1}^{N}(\prod_{j=1}^{N} H_{ij}(t_j \to t_{j+1})) \tag{27}$$

which we will call them *breakable* integrals.

A simple example of a breakable integral is:

$$\mathcal{F}(t_1 \to t_{N+1}) = \int_{t_1}^{t_{N+1}} f(t)dt = \sum_{i=1}^{N} \mathcal{F}(t_i \to t_{i+1}) \tag{28}$$

which corresponds to $H_{ij} = \begin{bmatrix} \mathcal{F} & & \\ & \ddots & \\ & & \mathcal{F} \end{bmatrix}$.

Properties and Proofs A breakable integral has several nice properties:

Property 1 A breakable integral multiplies, adds or subtracts a breakable integral is still a breakable integral.

Property 2 If $\mathcal{F}(t_1 \to t_{N+1})$ is a breakable integral, then $\mathcal{A}(t_1 \to t_{N+1})$ is also a breakable integral when:

$$\mathcal{A}(t_1 \to t_{N+1}) = \int_{t_1}^{t_{N+1}} \mathcal{F}(t_1 \to t) f(t)dt \tag{29}$$

Proof:

$$\mathcal{A}(t_1 \to t_{N+1}) = \int_{t_1}^{t_{N+1}} \mathcal{F}(t_1 \to t) f(t)dt = \sum_{k=1}^{N} \int_{t_k}^{t_{k+1}} \mathcal{F}(t_1 \to t) f(t)dt \tag{30}$$

In which we have

$$\begin{aligned}&\mathcal{F}(t_1 \twoheadrightarrow t)\\&= \sum_{i=1}^{k-1}(\prod_{j=1}^{k-1} H_{ij}(t_j \twoheadrightarrow t_{j+1}) \cdot H_{ik}(t_k \twoheadrightarrow t)) + \prod_{j=1}^{k-1} H_{kj}(t_j \twoheadrightarrow t_{j+1}) \cdot H_{kj}(t_k \twoheadrightarrow t)\\&= S_1 + S_2\end{aligned} \tag{31}$$

Focusing on S_1, we have

$$\begin{aligned}\mathcal{A}_1(t_1 \twoheadrightarrow t_{N+1}) &= \sum_{k=1}^{N} \int_{t_k}^{t_{k+1}} S_1 f(t) dt\\&= \sum_{k=1}^{N} \int_{t_k}^{t_{k+1}} \left[\sum_{i=1}^{k-1}(\prod_{j=1}^{k-1} H_{ij}(t_j \twoheadrightarrow t_{j+1}) \cdot H_{ik}(t_k \twoheadrightarrow t))\right] f(t) dt\\&= \sum_{k=1}^{N} \sum_{i=1}^{k-1}(\prod_{j=1}^{k-1} H_{ij}(t_j \twoheadrightarrow t_{j+1}) \cdot \left[\int_{t_k}^{t_{k+1}} H_{ik}(t_k \twoheadrightarrow t) f(t) dt\right])\\&= \sum_{k=1}^{N} \sum_{i=1}^{k-1}(\prod_{j=1}^{k-1} H_{ij}(t_j \twoheadrightarrow t_{j+1}) \cdot \hat{H}_{ik}(t_k \twoheadrightarrow t_{k+1}))\end{aligned} \tag{32}$$

Focusing on S_2, we have

$$\begin{aligned}\mathcal{A}_2(t_1 \twoheadrightarrow t_{N+1}) &= \sum_{k=1}^{N} \int_{t_k}^{t_{k+1}} S_2 f(t) dt\\&= \sum_{k=1}^{N} \int_{t_k}^{t_{k+1}} \left[\prod_{j=1}^{k-1} H_{kj}(t_j \twoheadrightarrow t_{j+1}) \cdot H_{kj}(t_k \twoheadrightarrow t)\right] f(t) dt\\&= \sum_{k=1}^{N} \prod_{j=1}^{k-1} H_{kj}(t_j \twoheadrightarrow t_{j+1}) \cdot \left[\int_{t_k}^{t_{k+1}} H_{kj}(t_k \twoheadrightarrow t) f(t) dt\right]\\&= \sum_{k=1}^{N} \prod_{j=1}^{k-1} H_{kj}(t_j \twoheadrightarrow t_{j+1}) \cdot \hat{H}_{kj}(t_k \twoheadrightarrow t_{k+1})\end{aligned} \tag{33}$$

Thus $\mathcal{A}(t_1 \twoheadrightarrow t_{N+1}) = \mathcal{A}_1(t_1 \twoheadrightarrow t_{N+1}) + \mathcal{A}_2(t_1 \twoheadrightarrow t_{N+1})$ also belongs to the family of breakable integrals.

Examples Here we show both the volume rendering equation and distortion loss can be trivially proved as breakable integrals using the above properties.

Transmittance The *transmittance* belongs to this family because:

$$T(t_1 \twoheadrightarrow t_{N+1}) = \exp\left(-\int_{t_1}^{t_{N+1}} \sigma(t) dt\right) = \prod_{i=1}^{N} T(t_i \twoheadrightarrow t_{i+1}) \tag{34}$$

Volume Rendering The volume rendering has the formulation of:

$$C(t_1 \to t_{N+1}) = \int_{t_1}^{t_{N+1}} T(t_1 \to t)\sigma(t)c(t)dt \tag{35}$$

Given that the transmittance $T(t_1 \to t)$ belongs to this family, and the property 2 of this family, we know that volume rendering equation also belongs to this family.

Distortion Loss The distortion loss has the formulation of:

$$\begin{aligned}
\mathcal{L}_{dist}(t_1 \to t_{N+1}) &= \iint_{t_1}^{t_{N+1}} w(t_1 \to u)w(t_1 \to v)\,|u-v|\,dudv \\
&= 2\int_{t_1}^{t_{N+1}} w(t_1 \to u)\left[\int_{t_1}^{u} w(t_1 \to v)\,(u-v)\,dv\right]du \\
&= 2\int_{t_1}^{t_{N+1}} T(t_1 \to u)\sigma(u)\left[\int_{t_1}^{u} T(t_1 \to v)\sigma(v)\,(u-v)\,dv\right]du \\
&= 2\int_{t_1}^{t_{N+1}} T(t_1 \to u)\sigma(u)S(t_1 \to u)du
\end{aligned} \tag{36}$$

Similar to the volume rendering equation, from property 2 we know that the formula within the bracket belongs to this family:

$$S(t_1 \to u) = \int_{t_1}^{u} T(t_1 \to v)\sigma(v)\,(u-v)\,dv \tag{37}$$

From property 1 we know $T(t_1 \to u)S(t_1 \to u)$ also belongs to this family. Then apply property 2 again we can see that $\mathcal{L}_{dist}(t_1 \to t_{N+1})$ also belongs to this family.

References

1. Barron, J.T., Mildenhall, B., Verbin, D., Srinivasan, P.P., Hedman, P.: Mip-nerf 360: Unbounded anti-aliased neural radiance fields. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 5470–5479 (2022) 6, 7, 10, 12, 19
2. Barron, J.T., Mildenhall, B., Verbin, D., Srinivasan, P.P., Hedman, P.: Zip-nerf: Anti-aliased grid-based neural radiance fields. In: Proceedings of the IEEE/CVF International Conference on Computer Vision. pp. 19697–19705 (2023) 6
3. CivilAirPatrol: Hurrican michael imageries. http://fema-cap-imagery.s3-website-us-east-1.amazonaws.com/Others/2018_10_FL_Hurricane-Michael/ 10, 12

4. Li, R., Gao, H., Tancik, M., Kanazawa, A.: Nerfacc: Efficient sampling accelerates nerfs. In: Proceedings of the IEEE/CVF International Conference on Computer Vision. pp. 18537–18546 (2023) 6, 18
5. Li, Y., Jiang, L., Xu, L., Xiangli, Y., Wang, Z., Lin, D., Dai, B.: Matrixcity: A large-scale city dataset for city-scale neural rendering and beyond. In: Proceedings of the IEEE/CVF International Conference on Computer Vision. pp. 3205–3215 (2023) 1, 10, 12
6. Lin, C.H., Ma, W.C., Torralba, A., Lucey, S.: Barf: Bundle-adjusting neural radiance fields. In: Proceedings of the IEEE/CVF International Conference on Computer Vision. pp. 5741–5751 (2021) 5
7. Martin-Brualla, R., Radwan, N., Sajjadi, M.S., Barron, J.T., Dosovitskiy, A., Duckworth, D.: Nerf in the wild: Neural radiance fields for unconstrained photo collections. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 7210–7219 (2021) 5
8. Meng, Q., Chen, A., Luo, H., Wu, M., Su, H., Xu, L., He, X., Yu, J.: Gnerf: Gan-based neural radiance field without posed camera. In: Proceedings of the IEEE/CVF International Conference on Computer Vision. pp. 6351–6361 (2021) 3
9. Meuleman, A., Liu, Y.L., Gao, C., Huang, J.B., Kim, C., Kim, M.H., Kopf, J.: Progressively optimized local radiance fields for robust view synthesis. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 16539–16548 (2023) 1, 2, 3, 4, 5, 7, 10, 12
10. Mildenhall, B., Srinivasan, P.P., Tancik, M., Barron, J.T., Ramamoorthi, R., Ng, R.: Nerf: Representing scenes as neural radiance fields for view synthesis. Communications of the ACM **65**(1), 99–106 (2021) 6
11. Müller, T., Evans, A., Schied, C., Keller, A.: Instant neural graphics primitives with a multiresolution hash encoding. ACM Transactions on Graphics (ToG) **41**(4), 1–15 (2022) 10, 15, 16
12. Rebain, D., Jiang, W., Yazdani, S., Li, K., Yi, K.M., Tagliasacchi, A.: Derf: Decomposed radiance fields. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 14153–14161 (2021) 7
13. Reiser, C., Peng, S., Liao, Y., Geiger, A.: Kilonerf: Speeding up neural radiance fields with thousands of tiny mlps. In: Proceedings of the IEEE/CVF International Conference on Computer Vision. pp. 14335–14345 (2021) 7
14. Rematas, K., Liu, A., Srinivasan, P.P., Barron, J.T., Tagliasacchi, A., Funkhouser, T., Ferrari, V.: Urban radiance fields. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 12932–12942 (2022) 3
15. Tancik, M., Casser, V., Yan, X., Pradhan, S., Mildenhall, B., Srinivasan, P., Barron, J.T., Kretzschmar, H.: Block-NeRF: Scalable large scene neural view synthesis. arXiv (2022) 1, 2, 3, 4, 5, 6, 7, 9, 10, 12, 18
16. Tancik, M., Weber, E., Ng, E., Li, R., Yi, B., Wang, T., Kristoffersen, A., Austin, J., Salahi, K., Ahuja, A., et al.: Nerfstudio: A modular framework for neural radiance field development. In: ACM SIGGRAPH 2023 Conference Proceedings. pp. 1–12 (2023) 5, 6
17. Turki, H., Ramanan, D., Satyanarayanan, M.: Mega-nerf: Scalable construction of large-scale nerfs for virtual fly-throughs. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). pp. 12922–12931 (June 2022) 1, 2
18. Turki, H., Ramanan, D., Satyanarayanan, M.: Mega-nerf: Scalable construction of large-scale nerfs for virtual fly-throughs. In: Proceedings of the IEEE/CVF Con-

ference on Computer Vision and Pattern Recognition. pp. 12922–12931 (2022) 3, 4, 5, 6, 7, 9, 10, 12, 18
19. Wang, P., Liu, Y., Chen, Z., Liu, L., Liu, Z., Komura, T., Theobalt, C., Wang, W.: F2-nerf: Fast neural radiance field training with free camera trajectories. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 4150–4159 (2023) 3, 9, 19
20. Wu, X., Xu, J., Zhu, Z., Bao, H., Huang, Q., Tompkin, J., Xu, W.: Scalable neural indoor scene rendering. ACM transactions on graphics **41**(4) (2022) 3
21. Xiangli, Y., Xu, L., Pan, X., Zhao, N., Rao, A., Theobalt, C., Dai, B., Lin, D.: Bungeenerf: Progressive neural radiance field for extreme multi-scale scene rendering. In: European conference on computer vision. pp. 106–122. Springer (2022) 2
22. Xu, L., Xiangli, Y., Peng, S., Pan, X., Zhao, N., Theobalt, C., Dai, B., Lin, D.: Grid-guided neural radiance fields for large urban scenes. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 8296–8306 (2023) 3