Appendices

A. Implementation Details

A.1. Voxel to Hash Grid

When we convert a voxel grid to hash grid, the main challenge is hash collisions. In normal Instant-NGP [4] scenario, two colliding points means that they are not spatially close while they yield features from the same location in the hash table when queried with. At one certain feature g_m in the hash table, let the all colliding points on it be $P = \{p_1, p_2, \ldots, p_L\}$. We weight all these points by learning importance. In fact, the influence of p_l on g_m has already been weighted with the gradient during back propagation. So, reweighting is not necessary and we shall treat every p_l evenly. As a result, for estimating g_m from P, we choose to average all points in P.

The algorithm. Here we give an algorithm to convert a standard voxel grid into an Instant-NGP-compatible hash grid. A voxel grid can be represented in sparse form $\{C, F\}$, where $C = \{c_1, c_2, \ldots, c_K\}$ is the set of valid coordinates and $F = \{f_1, f_2, \ldots, f_K\}$ are corresponding features. Given a voxel grid with features $\mathcal{V} = \{C, F\}$ and an empty hash table $\mathcal{T} = \{g_1, g_2, \ldots, g_M\}$, in which g_m are hash grid features, we transfer the values through the following steps. First, we assign each hash grid feature point its ordinal position in flatten sequence. In this step, we use $\sqrt[p]{M}$-ary number system encoding (in pratice, we choose $p = 4$) to avoid data overflow especially for half precision environment. Then we query the hash grid with C to get the corresponding serial indexes I. With I, each K feature f_k in $\mathcal{V}$ is scattered to its corresponding g_{i_k}, and mean reduction is applied to produce the final g_{i_k}. This process is also decribed in Alg. 1.

Algorithm 1 Voxel to Hash Grid.

Input: $\mathcal{V} = \{C, F\}$, an voxel grid with features; $\mathcal{T} = \{g_1, g_2, \ldots, g_M\}$, an empty hash grid
Output: $\mathcal{T}'$, the hash grid filled with features from $\mathcal{V}$

$M \leftarrow |\mathcal{T}|$
$R \leftarrow [1, 2, \ldots, \sqrt[p]{M}]$
$\mathcal{T}[:p] \leftarrow \texttt{MeshGrid}(R, R, R, R)$
$I \leftarrow \mathcal{T}(C)$
$\mathcal{T}' \leftarrow \texttt{ScatterMean}(I, F)$
return $\mathcal{T}'$

A.2. Backward for Ray Marching

Ray marching is implemented with CUDA in Instant-NGP for better performance. However, since classic NeRF [3] optimization does not require gradients in this process, the compute graph truncates here, preventing us from optimizing poses.

Mathematical analysis. Suppose we cast a bunch of rays $\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots$, where $\boldsymbol{r}_r = (\boldsymbol{o}_r, \boldsymbol{d}_r)$. And for each ray $\boldsymbol{r}_r$, we sample a sequence $t_{r1}, t_{r2}, \ldots$, resulting in position and direction sequences $\boldsymbol{x}_{r1}, \boldsymbol{x}_{r2}, \ldots$ and $\boldsymbol{d}_{r1}, \boldsymbol{d}_{r2}, \ldots$, where $\boldsymbol{x}_{ri} = \boldsymbol{r}_r(t_{ri}) = \boldsymbol{o}_r + t_{ri}\boldsymbol{d}_r$, $\boldsymbol{d}_{ri} = \boldsymbol{d}_r$. Then we consider the moment when gradients for each $\boldsymbol{x}_{ri}$ and $\boldsymbol{d}_{ri}$ are ready. We calculate gradients for $\boldsymbol{o}_r$ and $\boldsymbol{d}_r$ through

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{o}_r} = \sum_i \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}_{ri}} \cdot \frac{\partial \boldsymbol{x}_{ri}}{\partial \boldsymbol{o}_r} = \sum_i \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}_{ri}}, \tag{10}$$

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \boldsymbol{d}_r} &= \sum_i \left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}_{ri}} \cdot \frac{\partial \boldsymbol{x}_{ri}}{\partial \boldsymbol{d}_r} + \frac{\partial \mathcal{L}}{\partial \boldsymbol{d}_{ri}} \cdot \frac{\partial \boldsymbol{d}_{ri}}{\partial \boldsymbol{d}_r} \right) \\ &= \sum_i \left(t_{ri} \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}_{ri}} + \frac{\partial \mathcal{L}}{\partial \boldsymbol{d}_{ri}} \right), \end{aligned} \tag{11}$$

where $\mathcal{L}$ is the objective function.

Implementation. From Eq. 10, 11, to smoothly calculate $\frac{\partial \mathcal{L}}{\partial \boldsymbol{o}_r}$ and $\frac{\partial \mathcal{L}}{\partial \boldsymbol{d}_r}$, we need for every r and i

- $\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}_{ri}}$ and $\frac{\partial \mathcal{L}}{\partial \boldsymbol{d}_{ri}}$, which is maintained by PyTorch [6];
- value of $t_r i$;
- the correspondence between from the sampled point $(\boldsymbol{x}_{ri}, \boldsymbol{d}_{ri})$ to the ray it belongs to.

For both $t_r i$ and the correspondence , we modify the CUDA kernel to record t_{ri} and $r_{ri} \triangleq r$ synchronized with the generation of $\boldsymbol{o}_{ri}$ and $\boldsymbol{d}_{ri}$. Finally, the gradients will be calculated in parallel for each r and i and then accumulated to their corresponding $\frac{\partial \mathcal{L}}{\partial \boldsymbol{o}_r}$, $\frac{\partial \mathcal{L}}{\partial \boldsymbol{d}_r}$ with scatter sum according to r_{ri}.