PlenVDB: Memory Efficient VDB-Based Radiance Fields for Fast Training and Rendering

Supplementary Material

A Overview

In Sec.B, we give more details about VDB. In Sec.C, we describe more implementation details of our PlenVDB. In Sec.D, we present experiments on more datasets. In Sec.E, we show more rendered images for visualization.

B VDB

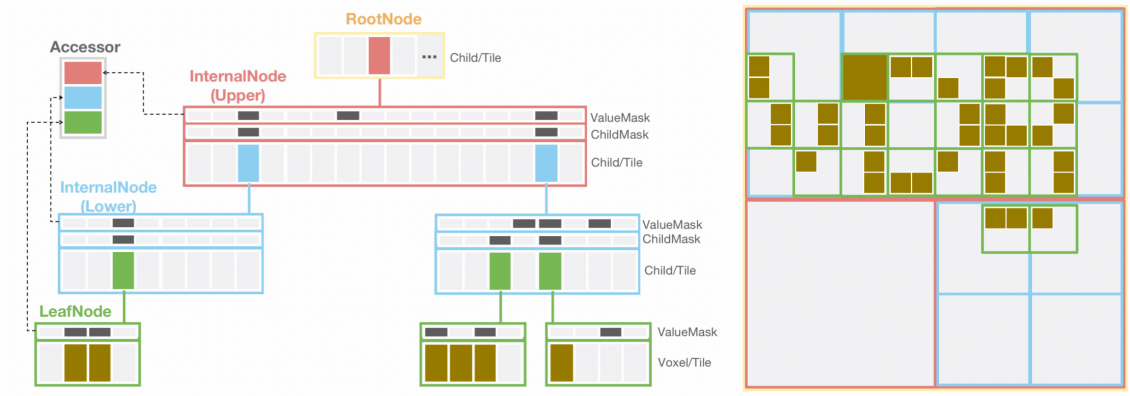

Figure 1: The 1D and 2D illustrations of OpenVDB data structure. **Left**: A VDB with one RootNode(yellow), three InternalNodes(red and blue) and three LeafNodes(green). ValueMask denotes whether it is active, and ChildMask denotes whether it is a child node or tile(gray). Only tiles and voxels(brown) have values. The Accessor has a capacity of three and will cache nodes visited during random access. The RootNode is implemented by a hash map and can be virtually infinite, while the InternalNode and LeafNode are dense. **Right**: An example of representing the word "VDB".

If interested in VDB, we refer the reader to the original paper[Mus13] for more details. Here we just introduce the main terminology and the relevant advantages of our task.

B.1 OpenVDB

OpenVDB is an open-source C++ library of the original VDB data structure. The data are stored in **voxels** and **tiles**, and a tile uses one value to represent a small dense grid where the voxels are sharing the same value and state. The **state** can be active or inactive, indicating whether the corresponding value is interesting or not. In our task, if a voxel is inactive, this coordinate will be regarded as empty space. Basically, OpenVDB is a four-layer B+ tree that consists of **LeafNodes**, **InternalNodes**, and **RootNodes**. In the left of Fig.1, from the bottom up, the fourth level is composed of LeafNodes and each LeafNode contains $8 \times 8 \times 8$ voxels. The third level is composed of InternalNodes and each contains $16 \times 16 \times 16$ LeafNodes. The second level is also composed of InternalNodes and each contains $32 \times 32 \times 32$ InternalNodes. The top-most level is only one RootNode that can accommodate virtually infinite InternalNodes, thus covering the whole coordinates in 3D space. For InternalNode, there are

two masks(**ChildMask** and **ValueMask**) that indicate if it is a child node or a tile, and if the tile value is active or inactive. For LeafNodes, only one ValueMask is available for judging the state. The illustrations are drawn in Fig.1.

Since OpenVDB has fixed depth as 4, random access can be very fast(on average constant time). Additionally, OpenVDB designs an **Accessor** for high performance sequential access which enables fast neighboring nodes queries. For example, in Fig.1, after visiting the leftmost voxel, the Accessor caches all visited nodes. While querying the adjacent voxel, the OpenVDB checks the nodes cached in the Accessor bottom-up. Because of spatial proximity, the first check to Accessor usually hits with a large probability, thus dispensing with another top-down tree traversal.

B.2 NanoVDB

Despite the performance advantages of OpenVDB, it does not support GPUs. To address this limitation, NanoVDB offers C++ and C99 implementation of a VDB tree that runs on both CPUs and GPUs. The only drawback is that the topology is assumed to be static. But it is also due to this assumption, the VDB tree can be serialized into a contiguous block of memory, contributing to better performance on access and storage(See Fig.2).

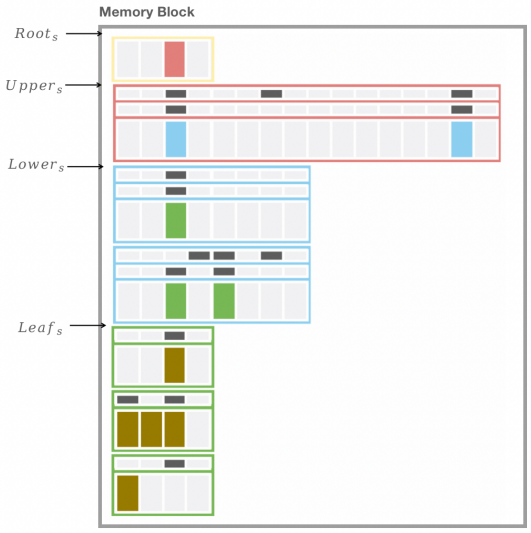

Figure 2: The illustration of NanoVDB. Compared with OpenVDB, NanoVDB serialized all nodes into a contiguous memory block. As the topology is assumed to be static, the size of the memory block can be pre-calculated from the OpenVDB data structure. To locate different types of nodes, there are four pointers $Root_s, Upper_s, Lower_s, Leaf_s$ pointing to the first(source) node of each layer, respectively.

C Implementation Details

Here we review the PlenVDB in Fig.3. We encapsulate the PlenVDB with three general VDBs for training: **DataVDB**, **GradVDB** and **(Adam)OptVDB**. And only DataVDB is used for rendering.

C.1 Backward

In *backward* pass, the gradients calculated from the loss function and the chain law will be accumulated into the GradVDB. We call this approach Coordinate-driven Modification, which is described in Alg.1. While NanoVDB does not provide us with APIs to modify voxel values in VDB directly, the only way is to get the corresponding non-const voxel pointer and apply *SetValue* function. Given an index coordinate, we can access the non-const leaf pointer with the corresponding offset by a coordinate-driven method in Alg.1.

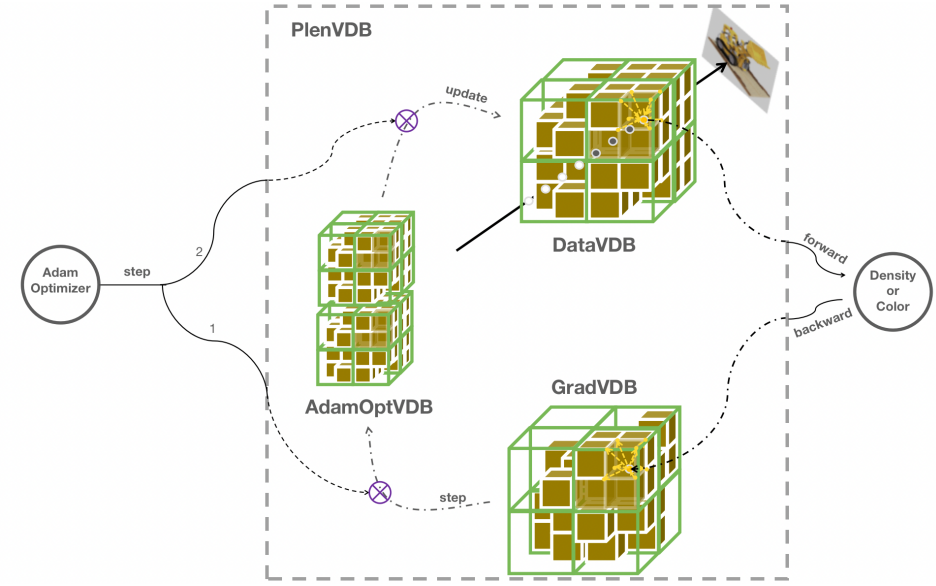

Figure 3: The overview of the PlenVDB.

Algorithm 1 Coordinate-driven Modification

Given the GradVDB G_{vdb}, coordinate c, new value v and the first non-const leaf pointer $Leaf_s$
$acc \leftarrow \text{getAccessor}(G_{vdb})$
Do $acc.getValue(c)$ to cache nodes
if the leaf including c is cached **then**
  $CLeaf_d \leftarrow acc.getLeafNode()$
  $det \leftarrow PtrDiff(CLeaf_d, const(Leaf_s))$
  $Leaf_d \leftarrow PtrAdd(Leaf_s, det)$
  $offset \leftarrow Leaf_d.CoordTooffset(c)$
  $Leaf_d.setValueOnly(offset, v)$
end if

C.2 Step

In one *step* of the optimizer, PlenVDB will first update the OptVDB by GradVDB, and then modify the data in DataVDB. Different from the modification method in *backward*, we determine the total number of voxels based on the number of LeafNodes, and then update each location by using Coordinate-free Modification, which is described in Alg.2.

D Additional Datasets

In this section, we evaluate our PlenVDB on three additional inward-facing datasets. For fairness, we choose DVGO as our baseline and keep all the hyper-parameters the same.

NSVF-Synthetic [LGL+20] consists of eight scenes, and each scene has 200 views for training and 100 views for testing. Every image has 800×800 resolution. **BlendedMVS** [YLL+20] consists of four scenes, and each scene has one-eight views for testing. Every image has 768×576 resolution. **Deepvoxels** [STH+19] consists of four simple Lambertian objects, and each scene has 479 views for training and 1000 views for testing. Every image has 512×512 resolution.

The comparisons on the training time, rendering time(FPS), model size and image quality(PSNR) are listed in Tab.1, Tab.2 and Tab.3. We find that for some scenes, our PlenVDB achieves slightly better PSNR, while the others are worse. The reason is that for smaller model size, we set voxels masked by the **MaskGrid** to the background value (See Fig.4). Here the **MaskGrid** is same as the DVGO's, which is used to pre-filter uninteresting sampled points. But the points on the boundaries of the MaskGrid are difficult to handle: they will influence the trilinear interpolated values of the

Algorithm 2 Coordinate-free Modification

Given the first non-const leaf pointer $Leaf_s$, new value v and the number of LeafNodes $nLeafCount$

Choose $n \in [0, nLeafCount - 1]$
$nleaf \leftarrow n >> 9$
$nvox \leftarrow n \ \& \ 512$
$Leaf_d \leftarrow Leaf_s + nleaf$
if $leaf_d.isActive(nvox)$ **then**
  $Leaf_d.setValueOnly(nvox, v)$
end if

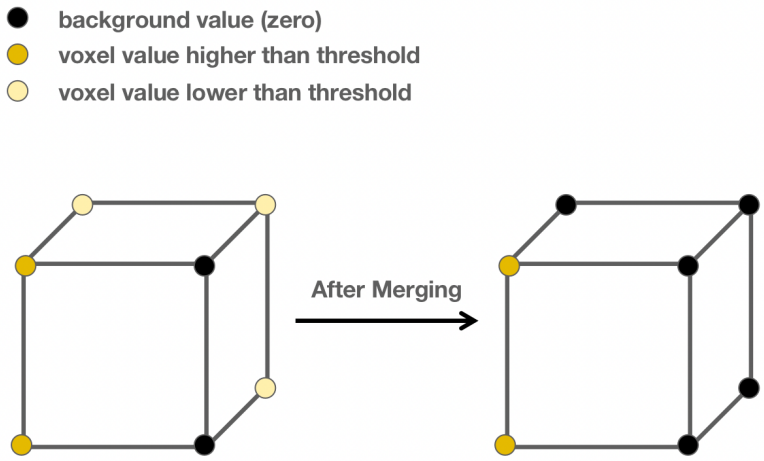

Figure 4: The information loss caused by the merging operation. Note that the background value is higher than the values in the MaskGrid.

sampled points. Therefore, setting them to background value may cause information loss, or make the model more robust.

Table 1: Experimental Results on *NSVF-Synthetic*.

Metrics	Methods	Bike	Lifestyle	Palace	Robot	Spaceship	Steamtrain	Toad	Wineholder
Training Time ↓	DVGO	4:01	4:12	4:03	4:39	7:13	3:21	4:00	3:58
	PlenVDB	9:53	11:12	13:44	11:33	13:39	10:54	11:57	10:52
FPS ↑	DVGO	5	4	4	5	5	5	5	5
	PlenVDB	39	20	17	42	22	25	39	24
Model Size ↓	DVGO	201	203	200	201	201	203	201	201
	PlenVDB	11	13	23	12	20	28	30	10
PSNR ↑	DVGO	38.15	33.72	34.31	36.27	37.54	36.44	32.98	30.22
	PlenVDB	38.08	33.72	34.30	36.25	37.53	36.42	32.99	30.20

E Additional Rendered Images

Though we do not focus on the improvement of rendered quality, our method reconstructs more details than the PlenOxels and PlenOctree, and has comparable quality with DVGO (See Fig.5 and Fig.6). Moreover, we present the rendered images of the other three datasets in Fig.7, Fig.8, Fig.9 and Fig.10.

Table 2: Experimental Results on *BlendedMVS*.

Metrics	Methods	Character	Fountain	Jade	Statues
Training Time ↓	DVGO	4:19	4:37	4:29	4:27
	PlenVDB	12:05	13:24	12:46	12:52
FPS↑	DVGO	7.13	5.30	5.12	5.55
	PlenVDB	26.91	18.16	13.77	16.93
Model Size ↓	DVGO	203	202	201	202
	PlenVDB	26	15	26	17
PSNR↑	DVGO	30.26	28.38	27.70	26.12
	PlenVDB	30.27	28.37	27.69	26.17

Table 3: Experimental Results on *DeepVoxels*.

Metrics	Methods	armchair	cube	vase	greek
Training Time↓	DVGO	4:41	4:24	4:54	4:31
	PlenVDB	10:44	10:07	11:24	11:57
FPS↑	DVGO	12	12	12	13
	PlenVDB	41	50	50	50
Model Size↓	DVGO	202	202	203	203
	PlenVDB	29	56	34	46
PSNR↑	DVGO	48.36	43.25	41.85	48.53
	PlenVDB	48.32	43.23	41.87	48.55

References

[LGL+20] Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. Neural sparse voxel fields. *NeurIPS*, 2020.

[Mus13] Ken Museth. Vdb: High-resolution sparse volumes with dynamic topology. *ACM Trans. Graph.*, 2013.

[STH+19] Vincent Sitzmann, Justus Thies, Felix Heide, Matthias Nießner, Gordon Wetzstein, and Michael Zollhöfer. Deepvoxels: Learning persistent 3d feature embeddings. In *CVPR*, 2019.

[YLL+20] Yao Yao, Zixin Luo, Shiwei Li, Jingyang Zhang, Yufan Ren, Lei Zhou, Tian Fang, and Long Quan. Blendedmvs: A large-scale dataset for generalized multi-view stereo networks. *CVPR*, 2020.

Figure 5: Qualitative Results on test views from *NeRF-Synthetic* dataset.

Figure 6: Qualitative Results on test views from *NeRF-Synthetic* dataset.

Figure 7: Qualitative Results on test views from *NVSF-Synthetic* dataset.

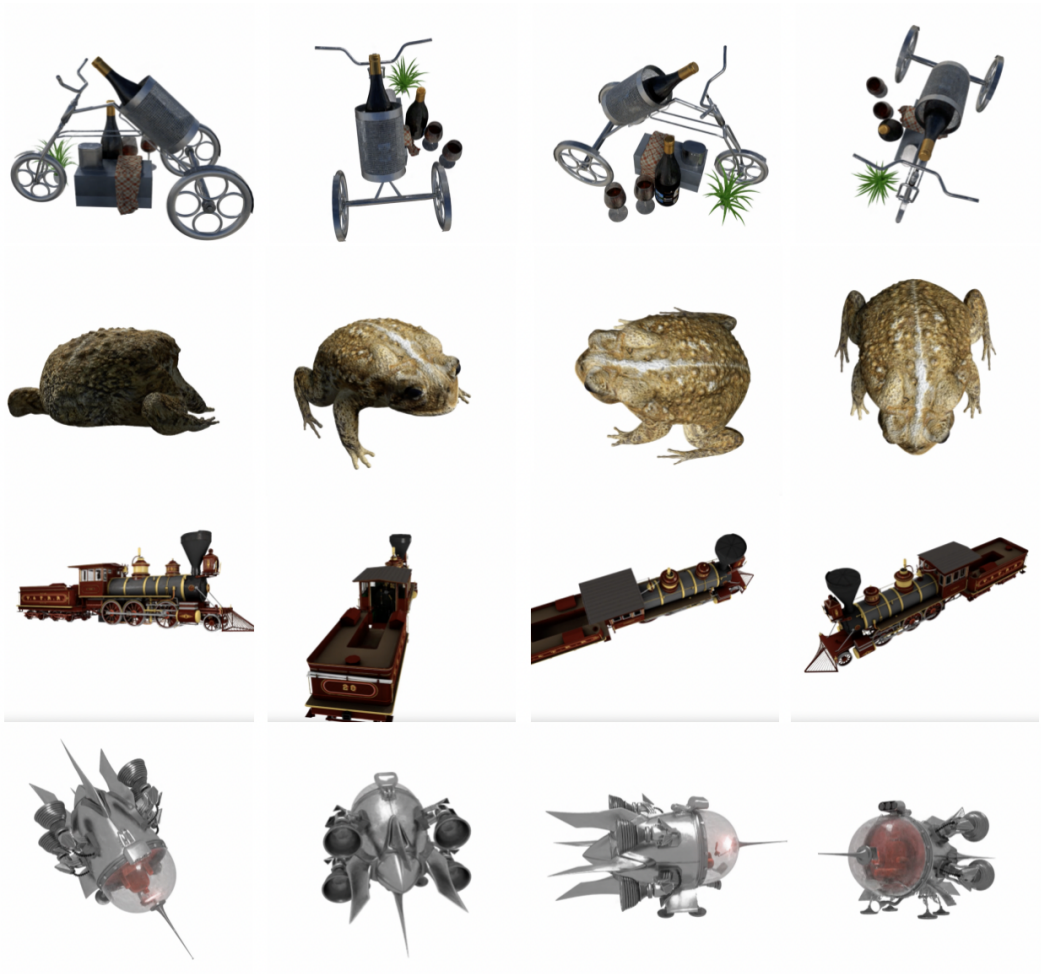

Figure 8: Qualitative Results on test views from *NVSF-Synthetic* dataset.

Figure 9: Qualitative Results on test views from *Blended-MVS* dataset.

Figure 10: Qualitative Results on test views from *DeepVoxels* dataset.