A Memory-Efficient KinectFusion Using Octree

Ming Zeng, Fukai Zhao, Jiaxiang Zheng, and Xinguo Liu[⋆]

State Key Lab of CAD&CG, Zhejiang University, Hangzhou, China, 310058
mingzeng85@gmail.com, xgliu@cad.zju.edu.cn

Abstract. KinectFusion is a real time 3D reconstruction system based on a low-cost moving depth camera and commodity graphics hardware. It represents the reconstructed surface as a signed distance function, and stores it in uniform volumetric grids. Though the uniform grid representation has advantages for parallel computation on GPU, it requires a huge amount of GPU memory. This paper presents a memory-efficient implementation of KinectFusion. The basic idea is to design an octree-based data structure on GPU, and store the signed distance function on data nodes. Based on the octree structure, we redesign reconstruction update and surface prediction to highly utilize parallelism of GPU. In the reconstruction update step, we first perform "add nodes" operations in a level-order manner, and then update the signed distance function. In the surface prediction step, we adopt a top-down ray tracing method to estimate the surface of the scene. In our experiments, our method costs less than 10% memory of KinectFusion while still being fast. Consequently, our method can reconstruct scenes 8 times larger than the original KinectFusion on the same hardware setup.

Keywords: Octree, GPU, KinectFusion, 3D Reconstruction.

1 Introduction

3D reconstruction for real scenes is an important research area in computer vision and computer graphics. For decades, researchers have developed all kinds of methods for efficient and accurate reconstruction. One popular method reconstructs scenes by fusing depth maps from different views. Especially, Newcombe et al. [9] and Izadi et al. [6] proposed KinectFusion, which used a commodity depth camera Kinect[8] to scan and model the dense surface of a room-size (about $4m \times 4m \times 4m$) scene in realtime. The algorithm leverages the parallel computing ability of the modern GPU to track the pose of the Kinect, and fuses the depth map of each frame into a scene volume.

KinectFusion uses a uniformly divided volume, and stores the data of all voxels in this volume. In a real scene, however, large amount of space is not occupied by the object's surface, therefore KinectFusion greatly wastes GPU memory, which hinders scene reconstruction in larger scale. To solve the problem, we introduce an octree structure to efficiently store the scene data. Based on the octree, we propose an algorithm to maintain the octree and integrate depth maps online. Our method sufficiently exploits the hierarchical structure of the octree and GPU parallelism. We adopt different

[⋆] Corresponding author.

traversal manners in our method to update and trace the octree to achieve optimized performance. When updating the volume, we traverse the octree in a level-order manner to exploit GPU parallelism. When predicting the scene surface, we traverse the octree in a top-down way to skip large non-data spaces. Benefit from careful designs, our method performs better than KinectFusion both in memory and computational efficiency.

2 Related Work

This section introduces the related works on 3D reconstruction and octree construction on GPU. We only survey works most relevant to this paper.

3D Reconstruction. 3D reconstruction techniques capture scene information by RGB cameras or depth cameras, and then reconstruct the geometry of the scene.

Lots of researchers capture RGB images of a scene, then utilize structure from motion (SFM) and multi-view stereo (MVS) to locate cameras and recover a sparse point cloud of the scene [2,11]. This kind of method is time consuming and unable to obtain dense scene surface. Recently, some studies used RGB sequence to reconstruct the dense surface [12,13,10] of a small scene (about $1m \times 1m \times 1m$) in real time.

Depth cameras are able to capture a depth map of current scene on each frame. Leveraging this ability, we can align depth maps to form the whole scene surface. The most popular method for this task is Iterative Closest Point (ICP)[1]. It was widely used to register views of depth maps into a global coordinate. With ICP, the KinectFusion proposed by Newcombe et al. [9] and Izadi et al. [6] leverages depth camera and GPU to reconstruct a dense surface of a room-size (about $4m \times 4m \times 4m$) scene in real time. Based on the KinectFusion, Whelan et al. [15] proposed a system “Kintinuous” which supports modeling on unbounded regions by shifting volume and extracting meshes continuously. This system utilized KinectFusion as a building block, and extended the ability to support large scale scanning. In contrast to [15], our method improves the core parts of the KinectFusion, and it can be a building block of large scale scanning systems like Kintinuous.

Octree Construction on GPU. An octree adaptively splits the space of the scene according to the complexity of the scene to use memory efficiently [3]. Though the simplicity of its definition, it is hard to be maintained using parallelism feature of GPU due to the sparseness of its nodes [16]. Sun et al. [14] built an octree with only leaf nodes to store volume data and accelerated photon tracing based on the octree. Zhou et al. [16] constructed a whole octree structure on GPU to accelerate Poisson Reconstruction [7].

3 Overview

The overviews of KinectFusion and our method are shown in Figure 1 left. The KinectFusion contains four main stages: Surface Measurement, Camera Pose Estimation, Reconstruction Update, and Surface Prediction (see [9,6]). In our method, we adopt the similar flowchart of KinectFusion. However, to overcome the large memory consumption due to the uniform voxel in KinectFusion, we introduce an octree structure to compactly organize the scene data in our method. Based on the octree, we design new algorithms for **Reconstruction Update** and **Surface Prediction** to utilize GPU parallelism.

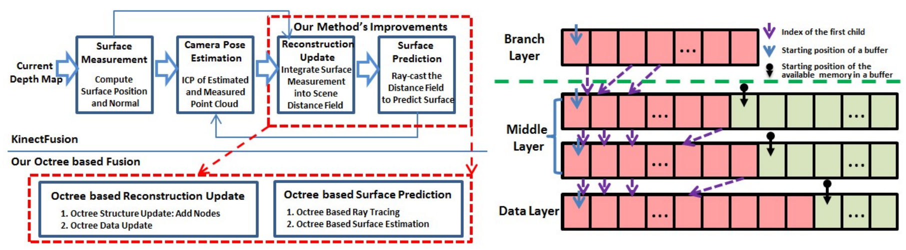

Current Depth Map
Surface Measurement
Compute Surface Position and Normal
Camera Pose Estimation
ICP of Estimated and Measured Point Cloud
Our Method's Improvements
Reconstruction Update
Integrate Surface Measurement into Scene Distance Field
Surface Prediction
Ray-cast the Distance Field to Predict Surface
KinectFusion
Our Octree based Fusion
Octree based Reconstruction Update
1. Octree Structure Update: Add Nodes
2. Octree Data Update
Octree based Surface Prediction
1. Octree Based Ray Tracing
2. Octree Based Surface Estimation
Branch Layer
Middle Layer
Data Layer
Index of the first child
Starting position of a buffer
Starting position of the available memory in a buffer

Fig. 1. Overview and Data Structure of our method. Left: overview of KinectFusion and improvements of our method. The red dash rectangle indicates the improved parts. Right: illustration of the octree structure.

In the following sections, we first introduce the octree structure in Section 4, followed by Reconstruction Update and Surface Prediction. In Section 7, we describe implementation details and experimental results, after which we give the conclusion.

4 Octree Structure

Our octree structure is stored in arrays of different layers. One array corresponds to one layer of the octree. As illustrated in Figure 1 right, there are three kinds of layers in our octree structure: Branch Layer, Middle Layer and Data Layer. All possible nodes in the branch layer are allocated, and can be randomly accessed. Nodes of other layers are not fully allocated, which can only be visited by links betweens adjacent layers.

- **Branch Layer:** The branch layer contains all nodes in its layer. Each node of the branch layer stores the index $idChild$ of the first child. If the node doesn't have children nodes, $idChild$ is set to -1.
- **Middle Layer:** Nodes in middle layers record both $idChild$ and **shuffled** xyz **key** [16]. The shuffled xyz key of a node at depth D is defined as a bit string: $x_1y_1z_1x_2y_2z_2...x_Dy_Dz_D$, where each 3-bit code $x_iy_iz_i$ encodes one of the eight subregions at depth i. $idChild$ can be used to traverse an octree in the depth-first manner, while shuffled xyz key can be utilized for the level-order traversal.
- **Data Layer:** Each node in this layer stores its shuffled xyz key and data of the scene [9,6]: Signed Distance Field SDF and weight w.

In the data structure, the branch layer is the starting layer, and it is not necessarily the root layer. We use a symbol $OT(T, L)$ to represent an octree with a branch layer at depth T, and data layer at depth L. Accordingly, $OF(T, L)$ represents our algorithm based on $OT(T, L)$. We also denote our algorithm with a n-depth octree as OFn. To represent the octree structure, we use following arrays to organize above data:

- $NodeBuf$ is a pre-allocated two-dimensional array to store nodes of each level. The ith node at level L is stored in $NodeBuf[L][i]$.
- $IdStart$ is a pre-allocated one-dimensional array to record the current starting index of the available buffer of each level, as black arrows shown in right of Figure 1. The current starting index of level L is recorded in $IdStart[L]$.

Algorithm 1. Add Nodes for the Octree at Level L

```
1: //Step1: Predict whether a node needs further split
2: for Node $O_i$ at Level $L$ in parallel do
3:   if $O_i$ in view frustum
4:     $v_g \leftarrow$ PosFromNode$(O_i, L)$
5:     $v \leftarrow$ WorldCoord2CamCoord$(v_g)$
6:     $p \leftarrow$ perspective project vertex v
7:     $dir \leftarrow v_g - v_{cam}$
8:     $sdf = \| v_{cam} - v_g \| - D(p, dir)$
9:     if IsSplit$(O_i, sdf)$ and $O_i$ has no child
10:      $ScanIn[i] = 1$
11:    else
12:      $ScanIn[i] = 0$
13:    endif
14:  endif
15: end for

16: //Step 2: Scan the split flag array
17: $(ScanOut, nNewNodeCount) \leftarrow$ Scan$(ScanIn, +)$

18: //Step 3: Assign child index and compute $xyz$ key
19: for $O_i$ at level $L$ in parallel
20:   if $ScanIn[i] == 1$
21:     $idx = IdStart[L+1] + (ScanOut[i] << 3)$
22:     $NodeBuf[L][i].idChild = idx$
23:     $key = O_i.xyzkey$
24:     for k=0 to 7 do
25:       $NodeBuf[L+1][idx+k].xyzkey = (key << 3) \mid k$
26:     end for
27:   end if
28: end for

29: //Step 4: Update $IdStart$
30: $IdStart[L+1] += 8 * nNewNodeCount$
```

5 Reconstruction Update Based on Octree

There are two operations in the step of reconstruction update: add nodes for new scene data, and SDF data update.

5.1 Add Nodes for Octree

To add new nodes for the octree, we adopt a top-down level-order manner to traverse the octree. We split nodes and assign children indices level by level, starting from the branch layer and moving towards the data layer, one level at a time. For level L, the pseudo code of the procedure is listed in Algorithm 1.

At the first step, we predict whether a node needs further split in parallel. For the node in the view frustum, we calculate the signed distance sdf from its center to the current depth map (Line 4 to 8). Then, given sdf, we use the function **IsSplit** (described later) to predict node splits and mark 1/0 in an auxiliary array $ScanIn$. At step 2, we take Parallel Prefix Sum (**Scan**) [5] with the add operation "$+$" on $ScanIn$ to compute the unique id of new split nodes. At step 3, we assign index of its first child to the node to be split. At this step, we also figure out and store the shuffled xyz key of each new child node according to the shuffled xyz key of its parent node and its relative position in all eight children. Finally at step 4, we update the index of the first available node in the memory pool of the child layer.

In the function **IsSplit**, we assume the scene near the node is approximately a plane. We denote distance between center of the node and scene surface as sdf, diagonal length of the node as d, then distances from all points within the node to the surface must be in $[sdf - d/2, sdf + d/2]$. Take the further consideration that the distance field stored in scene volume is in $[-U, U]$, U is the maximal truncation value. To ensure a correct split prediction, the node split iff $[sdf - d/2, sdf + d/2]$ intersects with $[-U, U]$, i.e. $sdf \in [-U - d/2, U + d/2]$.

5.2 Update SDF for Octree

After updating the structure of the octree, the current depth map should be integrated into the scene volume to update signed distance function. We only need to update the

Algorithm 2 Surface Prediction

```
1: for each pixel $u$ in imaging plane in parallel do
2:    $Ray_{dir} \leftarrow$ direction of the ray
3:    $g \leftarrow$ first voxel along ray dir in the finest level
4:    $Ray_{step} \leftarrow 0$
5:    while voxel $g$ within volume bounds do
6:       //determine the marching step
7:       $xyzkey \leftarrow$ Grid2Key($g$)
8:       $l \leftarrow$ level of the branch layer
9:       $node \leftarrow$ Key2Node($xyzkey, l, NodeBuf, null$)
10:      while $node$ has children do
11:         $l++$
12:         $node \leftarrow$ Key2Node($xyzkey, l, NodeBuf, node$)
13:      end while
14:      $Ray_{step} \leftarrow$ NextStepLen($g, Ray_{dir}, node$)

15:      //march forward
16:      $g_{prev} \leftarrow g$
17:      $g+ = Ray_{dir} \cdot Ray_{step}$
18:      //estimate position and normal
19:      if $l$ == level of the finest layer then
20:         if zero crossing from $g_{prev}$ to $g$ then
21:            $P \leftarrow$ estimate surface position
22:            $N \leftarrow$ estimate surface normal
23:            break
24:         end if
25:      end if
26:   end while
27: end for
```

nodes at the data layer to integrate the depth map. We adopt the similar integration algorithm as the method in [6] except that we compute positions of data nodes from shuffled xyz code while [6] directly compute them from their grid indices.

6 Surface Prediction Based on Octree

After structure and data update of the octree, like [9,6], a ray tracer is taken to ray-cast the scene volume and estimate the scene surface. Algorithm 2 lists the pseudo code. Each ray marches forward until crossing the object's surface. Line 6 to 14 determine the current amount of finest steps to march forward. In Line 7, **Grid2Key** figures out the $xyzkey$. Then in Line 8 to 13, with $xyzkey$, we adopt a top-down way to find the most compact octree node holding the position g. The most compact node for g means no surface data is in the node, so the ray can "bravely" marches across this most compact node. So in Line 14, we compute the marching step according to the depth (level) of the most compact node by the function **NextStepLen**. After marching forward, if the ray crosses surface, we estimate the position and normal of the hitting points between the ray and the object's surface.

The position and normal of the intersection point are estimated in a trilinear interpolation way [9,6]. Normals of points around two sides of the zero-crossing surface can be calculated by forward/backward differences according to the relative position of the node of its siblings.

7 Implementation and Experiments

7.1 Implementation Details

We have implemented the whole pipeline of our algorithm in C++ with Nvidia CUDA, and test it on a desktop computer with an Intel Core2 Duo E7400 2.80GH CPU (use one core) and a Nvidia GeForce GTX480 graphics card. For the operator **Scan**, we used the implementation provided in the highly optimized GPU library CUDPP [4].

All data data structures are stored in the global memory. For a layer at depth D of an $OT(T, L)$, the node count N of the GPU memory buffer is as follows:

$$N = \begin{cases} 2^{3D}, & D \leq T \\ \mu \cdot 2^{3D}, & D > T \end{cases} \tag{1}$$

where μ is proportion to all possible nodes at its depth. In our current implementation, μ is set to 0.05 for depth no deeper than 9, and 0.03 for depth 10.

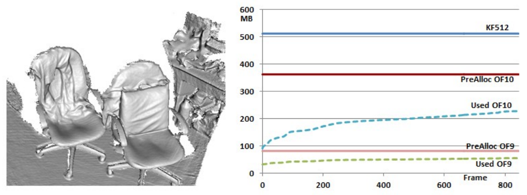
7.2 Experiments

This section compares our method with KinectFusion from three aspects: memory consumption, computation time, and reconstruction for large scene.

Memory Comparison. We compare memory consumption between KinectFusion and our method on a depth map sequence "chairs" (Figure 2). For KinectFusion, we adopt the 512^3 resolution (KF512), and for our method, we test both 9-depth (OF9) and 10-depth octree (OF10). In this experiment, branch layers of both OF9 and OF10 are set at depth 7. Figure 2 middle shows the memory consumptions on each frame of "chairs" with KF512, OF9, and OF10. KF512 cost 512 MB memory constantly, while OF9 and OF10 increase the memory as the camera scans new parts of the scene. At the beginning, OF9 and OF10 pre-allocate 81.6MB and 362.6MB, respectively. Finally, OF9 uses 55.6M, and OF10 uses 227.4MB. With 512^3 resolution, memory consumption of OF9 is $81.6/512 \approx 15.9\%$ of KF512. With 1024^3 resolution, KinectFusion will consume 4GB GPU memory, which is ≥ 10 times larger than that of our method. Figure 2 right gives details of memory consumption of each layer.

Time Comparison. We show time comparisons on a test scene in Figure 3, where OF(7, 9) processes a frame in about 19ms, and OF(7, 10) processes a frame less than 25ms. They are both faster than KF512. OF(7, 9) gains about $2\times$ speed up on the improved parts than KF512. This is because KF512 needs to update its all 512^3 voxels, while our method only updates existing nodes, and it can skip large spaces on the ray-cast stage.

Large Scale Scene. Our octree based method efficiently uses memory, and OF10 can reach a maximal 1024^3 resolution, which is able to robustly track the camera in a large scale scene and reconstruct it. We scan an office in a $8m \times 8m \times 8m$ bounding box using OF10, which captures about 3800 frames of depth maps. The scene data costs 299MB in 363MB pre-allocated GPU memory, and the extracted mesh from the scene volume contains 6200K triangle faces and 3400K vertices. As shown in Figure 4, though the size of the scene is large, the reconstructed model still possesses abundant details.



OF9	D=7	D=8	D=9	Aux	Total
PreAlloc	8.0	6.4	51.2	16.0	81.6
Used	8.0	4.0	27.6	16.0	55.6

OF10	D=7	D=8	D=9	D=10	Aux	Total
PreAlloc	8.0	6.4	51.2	245.8	51.2	362.6
Used	8.0	3.0	19.8	145.4	51.2	227.4

Fig. 2. Memory comparison of a static scene "chairs"

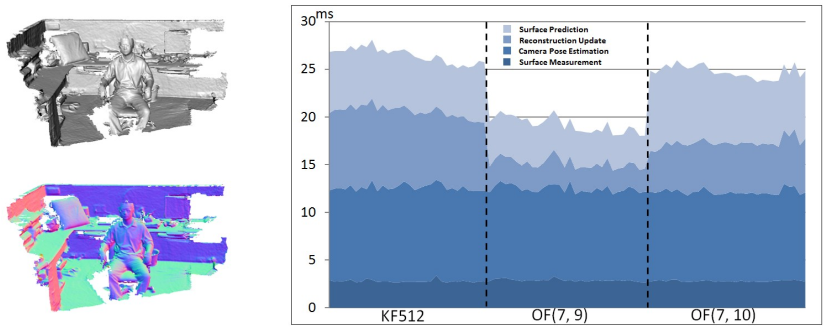


Fig. 3. Processing time for a test scene. Left are the Phong-shaded rendering and the normal map of the test scene, respectively. Right is the processing time on each stage of different methods.

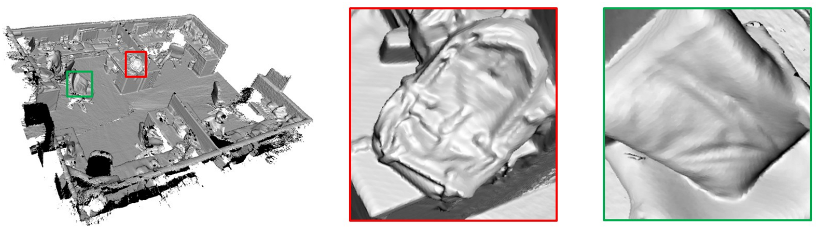


Fig. 4. Reconstruction result of a large scene and two zoom-in parts

8 Conclusion and Future Work

We propose an octree based KinectFusion. Our method represents the scene data in an octree structure, and maintains this octree by “add node” operations according to changes of the scene. We also modify KinectFusion to adapt the octree representation to highly utilize parallelism computation ability of GPU. Experiments show that our method costs only about 10% memory of original KinectFusion and runs about 2$\times$ faster than original KinectFusion on the improvement parts. Our method can reconstruct 3D scenes 8 times larger than that of KinectFusion.

The system can be extended in several ways. One is to design a more efficient memory management solution to replace current pre-allocation method. Another possible improvement may come from the combination of our method and Kintinuous [15]. A straightforward way is to use our method as a building block in Kintinuous, which may also involve some modifications both on data structures and the volume shifting algorithm of Kintinuous.

Acknowledgments. We would like to thank the anonymous reviewers for their valuable comments, and thank Bo Jiang, Zizhao Wu and Xuan Cheng for proof reading and video editing. This work was partially supported by NSFC (No. 60970074), China 973 Program (No. 2009CB320801), Fok Ying Tung Education Foundation and the Fundamental Research Funds for the Central Universities.

References

1. Chen, Y., Medioni, G.: Object modeling by registration of multiple range images. Image and Vision Computing (IVC) 10(3), 145–155 (1992)
2. Fitzgibbon, A.W., Zisserman, A.: Automatic Camera Recovery for Closed or Open Image Sequences. In: Burkhardt, H.-J., Neumann, B. (eds.) ECCV 1998. LNCS, vol. 1406, pp. 311–326. Springer, Heidelberg (1998)
3. Frisken, S.F., Perry, R.N., Rockwood, A.P., Jones, T.R.: Adaptively sampled distance fields: a general representation of shape for computer graphics. In: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2000, pp. 249–254. ACM Press/Addison-Wesley Publishing Co., New York (2000)
4. Harris, M., Owens, J.D., Sengupta, S., Zhang, Y., Davidson, A.: Cudpp homepage (2007), `http://gpgpu.org/developer/cudpp`
5. Harris, M., Sengupta, S., Owens, J.D.: Parallel prefix sum (scan) with CUDA, ch. 39. Addison Wesley (August 2007)
6. Izadi, S., Kim, D., Hilliges, O., Molyneaux, D., Newcombe, R., Kohli, P., Shotton, J., Hodges, S., Freeman, D., Davison, A., Fitzgibbon, A.: Kinectfusion: real-time 3d reconstruction and interaction using a moving depth camera. In: Proceedings of the 24th Annual ACM Symposium on user Interface Software and Technology, UIST 2011, pp. 559–568. ACM, New York (2011)
7. Michael, K., Matthew, B., Hugues, H.: Poisson surface reconstruction. In: Proceedings of the Fourth Eurographics Symposium on Geometry Processing, SGP 2006, pp. 61–70. Eurographics Association, Aire-la-Ville (2006)
8. Microsoft. Microsoft kinect project (2010), `http://www.xbox.com/kinect`
9. Newcombe, R.A., Izadi, S., Hilliges, O., Molyneaux, D., Kim, D., Davison, A.J., Kohli, P., Shotton, J., Hodges, S., Fitzgibbon, A.: Kinectfusion: Real-time dense surface mapping and tracking. In: Procedings of IEEE/ACM International Symposium on Mixed and Augmented Reality, pp. 127–136 (2011)
10. Newcombe, R.A., Lovegrove, S., Davison, A.J.: Dtam: Dense tracking and mapping in real-time. In: International Conference on Computer Vision, pp. 2320–2327 (2011)
11. Pollefeys, M., Gool, L.V., Vergauwen, M., Verbiest, F., Cornelis, K., Tops, J., Koch, R.: Visual modeling with a hand-held camera. International Journal of Computer Vision 59(3), 207–232 (2004)
12. Richard, A.J.D., Newcombe, A.: Live dense reconstruction with a single moving camera. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR), pp. 1498–1505 (June 2010)
13. Stühmer, J., Gumhold, S., Cremers, D.: Real-Time Dense Geometry from a Handheld Camera. In: Goesele, M., Roth, S., Kuijper, A., Schiele, B., Schindler, K. (eds.) DAGM 2010. LNCS, vol. 6376, pp. 11–20. Springer, Heidelberg (2010)
14. Sun, X., Zhou, K., Stollnitz, E., Shi, J., Guo, B.: Interactive relighting of dynamic refractive objects. ACM Trans. Graph. 27(3), 35:1–35:9 (2008)
15. Whelan, T., McDonald, J., Kaess, M., Fallon, M., Johannsson, H., Leonard, J.J.: Kintinuous: Spatially extended kinectfusion. In: RSS Workshop on RGB-D: Advanced Reasoning with Depth Cameras (July 2012)
16. Zhou, K., Gong, M., Huang, X., Guo, B.: Data-parallel octrees for surface reconstruction. IEEE Transactions on Visualization and Computer Graphics (TVCG) 17(5), 669–681 (2011)