3DSMC Final Report: KinectFusion (Team 9)

Xuyuan Han
Technical University of Munich
xuyuan.han@tum.de

Mohamed Kotb
Technical University of Munich
mohamed.kotb@tum.de

Feng Yang
Technical University of Munich
feng.yang@tum.de

Shengchao Zhang
Technical University of Munich
shengchao.zhang@tum.de

https://github.com/xuyuan-han/KinectFusion

Abstract

In this work, we present a comprehensive study and a reimplementation of KinectFusion, a notable system for real-time 3D reconstruction of indoor scenes using RGB-D data from Microsoft Kinect. We detail our efforts in reimplementing KinectFusion, exploring both multi-threaded CPU and CUDA-based GPU approaches to achieve efficient real-time performance. Our implementation extends the conventional KinectFusion pipeline by incorporating modern advancements such as high-resolution LiDAR data from iPhone sensors and integrating semantic information from 2D segmentation maps to enhance the reconstruction process. Through experiments conducted on the TUM RGB-D dataset and real-world data captured by an iPhone LiDAR sensor, we demonstrate the efficacy of our approach in producing detailed and accurate 3D reconstructions. Our results highlight the significant speedup achieved by the CUDA implementation, achieving real-time reconstruction at 30 fps, which is 14 times faster than its multi-threaded CPU counterpart. This work not only proves that consumer-grade sensors can effectively reconstruct 3D scenes but also highlights how integrating semantic segmentation can enhance reconstruction quality.

1. Introduction

Microsoft Kinect is an RGB-D camera with high-resolution depth information and at a lower cost in 3D sensing technology compared with other traditional 3D cameras, such as stereo cameras [5]. A notable application of the Kinect sensor is KinectFusion [7]. This RGB-D dense reconstruction framework permits real-time, high-detail mapping and tracking of complex indoor scenes using the Kinect sensor.

We reimplemented KinectFusion in our project based on theories drawn in Newcombe [9] in two versions, one multi-threaded CPU version (without CUDA), and one CUDA version to meet the real-time requirement. We use TUM RGB-D Dataset [1] as input data. Furthermore, given that the modern iPhone and iPad are released with a high-resolution LiDAR sensor for depth detection [8], we also applied the KinectFusion method to process data streamed from the iPhone. Moreover, this report discusses whether an appropriate deep-learning model for segmentation can enhance the reconstruction process.

2. Related Work

Before KinectFusion emerged, dense scene reconstruction had been extensively explored in various fields. In robotics, for instance, occupancy mapping gained popularity. This approach involves dividing a scene into a grid of cells, where each cell represents the probability of occupancy. These probabilities are updated iteratively using Bayesian updates with new depth frames [4]. The introduction of Signed Distance Field (SDF), as described in [2], revolutionized partial depth frame fusion while addressing issues associated with mesh-based reconstruction. Additionally, the research highlighted in [6] demonstrated that employing Bayesian probabilistic inference for optimal surface reconstruction, assuming a simple Gaussian noise model on depth measurements yields a straightforward algorithm of averaging weighted signed distance functions into a global frame.

In the context of implementing KinectFusion, numerous open-source implementations were available, including the CUDA implementation by Christian Diller [3]. Although our objective was to develop both CPU and CUDA versions and integrate various modifications such as incorporating segmentation maps into reconstruction and utilizing an iPhone LiDAR sensor for scanning (details of these modifications will be discussed later), Christian's implementa-

tion offered invaluable insights into specific implementation details and the data structures employed.

3. Method

There are four main modules in this framework of KinectFusion(Figure 1). The pipeline begins with surface measurement, processing raw depth data to generate dense vertex and normal maps. These measurements are then integrated into a volumetric scene model using a Truncated Signed Distance Function (TSDF) representation. Real-time alignment between the live depth frame and the scene model is achieved through raycasting in the surface prediction stage, followed by camera pose estimation using multi-scale ICP alignment for real-time tracking of the camera's position and orientation relative to the scene. In this section, we discuss the original KinectFusion paper's implementation for each module. The details of our CPU multi-threaded implementation and GPU implementation utilizing CUDA will be described in Appendix.

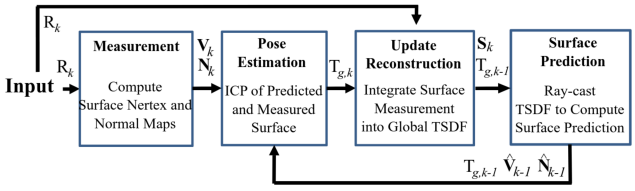


Figure 1. Overall system workflow [9]

3.1. Surface Measurement

As the first module, the surface measurement generates a vertex map and computes a corresponding normal map based on the input depth map provided by the virtual sensor. The output of this module is a vertex map pyramid and a normal map pyramid.

First, we apply a bilateral filter to the depth map. This filter can reduce the noise while preserving a fairly sharp edge [10]. Each vertex in the vertex map is obtained by multiplying the inverse of the camera intrinsic matrix, provided by the virtual sensor, by the vector $(u*d, v*d, 1*d)^T$, where u represents the row index of the pixel, v represents the column index of the pixel, and d represents the depth of the pixel. The normal of each vertex is the cross-product between the neighboring vertices, and then it should be normalized. The initial depth map constitutes the bottom level. Subsequently, the depth map pyramid is generated by downsampling the lower level to half resolution. Each level of the vertex map pyramid and the normal map pyramid is computed based on the corresponding depth map level.

3.2. Surface Reconstruction

The surface reconstruction module accumulates information from each depth map to maintain a global Truncated Signed Distance Function (TSDF) volume. It takes processed depth frames and predicted camera poses as inputs, updating the global volume accordingly.

The scene is represented by a Truncated Signed Distance Volumetric representation, storing distance to the nearest surface and whether the point is inside or outside the surface. The "truncated" nature optimizes computational resources by cutting off the function beyond a specific distance. The global frame is divided into a grid of voxels with predefined resolution.

For each depth frame, every observed voxel's TSDF and weight are calculated iteratively. The TSDF is the difference between a voxel's depth in the depth frame and its distance from the camera center in the global frame. When V is closer to the surface (less than the measured depth), the TSDF is positive, indicating its location in front of the surface. Conversely, if V is farther away, the TSDF is negative, placing it behind the surface. When V matches the measured depth, the TSDF is zero. To meaningfully accumulate information from multiple depth frames, we employ the weight concept from KinectFusion. Here, the weight serves as a counter representing how often a voxel has been observed. Utilizing this weight, we compute a weighted average between old and new TSDF values. Initially, depth frame updates heavily influence the voxel's TSDF, but with more observations, confidence in the stored TSDF increases, diminishing the impact of new readings.

3.3. Surface Prediction

The surface prediction module is implemented by ray casting the Truncated Signed Distance Function (TSDF) volume to deliver the predicted vertex and normal maps, which will be used in the subsequent camera pose estimation step. The surface prediction module takes the global TSDF volume and global color volume from the surface reconstruction module as inputs to perform the ray-casting algorithm.

To predict vertex and normal maps, a per-pixel ray is marched, starting from the minimum depth and stopping when a zero crossing is found, to indicate the surface interface at zero value. The ray marching is only valid if the crossing from a positive to a negative TSDF value occurs. After a zero-value crossing is found, the linear interpolation is used to predict the vertex. Also, it is assumed that the gradient of the TSDF for points at or near the interface is orthogonal to the zero-level set. Therefore, the surface normal is computed based on the difference in TSDF value across the voxels. In the voxel grid, trilinear interpolation is used to estimate the TSDF values of non-integer coordinate points. This involves solving a cubic polynomial.

3.4. Pose Estimation

When reconstructing the TSDF model of the scene, the pose estimation module is responsible for tracking the camera pose. In this module, we take the vertex and normal maps from the surface measurement and surface prediction as input and estimate the camera pose of the current frame. Using a high frame-rate input RGB and depth stream, we assume that the camera pose changes smoothly between frames, which allows us to use linearized Iterative Closest Point (ICP) to minimize the point-to-plane metric and to estimate the camera pose as shown in equations (16)-(26) in the original paper [9]. We filter point correspondences in the vertex and normal maps from the surface measurement and surface prediction modules by checking the distance between the possible corresponding vertices and the angle between the possible corresponding normals to obtain valid vertex-normal element correspondences. By calculating and accumulating the contribution of each vertex-normal element correspondence to the equation (25)(26) and performing an LU decomposition on the linear system, we can minimize the point-to-plane metric iteratively to estimate the camera pose incrementally. The whole process operates within a three-level coarse to fine framework, starting from the coarsest level and iteratively refining the camera pose estimation.

3.5. iPhone LiDAR Sensor Data Stream

Since the release of the iPhone 12 Pro, Apple has equipped its Pro series iPhones with a camera-LiDAR system. Although the resolution of this system is not as high as that of the Kinect v1, the widespread availability of the iPhone makes it a more accessible option, and we have one readily available for use. We intend to explore methods that allow room reconstruction with the lower-resolution LiDAR sensor on the iPhone without significantly compromising quality compared to using the Kinect v1.

As for the implementation details of using the iPhone LiDAR sensor data stream, we used the Record3D app to capture and transfer the RGB and depth image stream to a laptop with the help of the C++ library provided by the app. With the library, we can stream RGB images up to 1440x1920 and depth images up to 192x256 with the rear camera-lidar system on the iPhone at 30 fps, and reconstruct our apartment from the RGB-D data stream. A similar idea of using iPhone and Record3d to record depth data was also proposed in a recent paper [11].

3.6. Incorporating Semantic Information from 2D

Semantic segmentation in image space has advanced significantly with deep learning, leading to improved performance. Our project focuses on constructing a class volume from noisy per-frame 2D semantic segmentation maps, akin to surface reconstruction. We assign voxel classes based on TSDF proximity to the surface. To address inconsistencies inherent within the 2D segmentation maps, we leverage the weight assigned to each voxel as a counter, tracking how many times a voxel was segmented with its current class. When encountering a different class in the segmentation map, we probabilistically assign the class, considering the ratio of the old weight to the sum of the old and new weights. The benefits of incorporating semantic information into 3D include:

- Enhanced understanding compared to 2D maps, allowing for outlier removal using local neighborhood information.
- Improved correspondence finding in pose estimation.
- Extension to class-specific reconstruction, enabling selective inclusion/exclusion of classes for tasks like mapping static scenes with dynamic individuals in the scene.

4. Results

We visualize the results of our framework. We tested the pipeline on a laptop with 32 threads CPU (Intel Core i9-13900HX) and 9728 CUDA-core GPU (NVIDIA RTX 4090 Laptop) on Ubuntu 22.04.

4.1. TUM Dataset result on CUDA implementation

We reconstruct the scene freiburg1_xyz in the TUM dataset both on CPU implementation and CUDA implementation. The reconstruction results of both implementations are quite similar. Therefore, we only show the visualized results of CUDA implementation here. The reconstruction process can reach an average frame rate of 30 fps on CUDA implementation. We converted the color volume to a point cloud for better visualization. The final 3d reconstructed color volume of rgbd_dataset_freiburg1_xyz is shown in Figure 2. The intermediate results during the reconstruction process are shown in Figure 3.

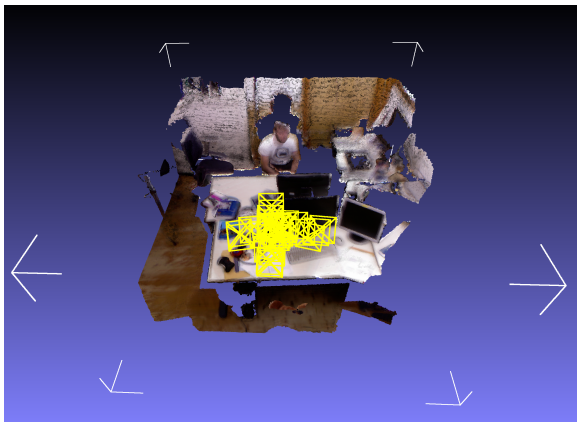
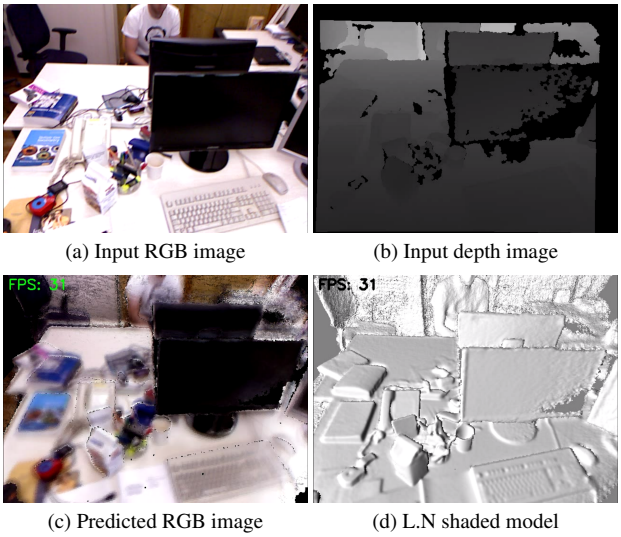


Figure 2. Reconstructed color volume of scene freiburg1_xyz



(a) Input RGB image (b) Input depth image

(c) Predicted RGB image (d) L.N shaded model

Figure 3. TUM rgbd_dataset_freiburg1_xyz results on CUDA implementation. (a) Input RGB image captured by Kinect sensor. (b) Input depth image captured by Kinect sensor. (c) Predicted RGB image by surface prediction. (d) The L.N shaded model, which is rendered by computing the dot product of each vertex's normal with the direction of a static light source. This calculation determines the intensity of each pixel, producing the shading effect.

4.2. iPhone LiDAR Sensor Data Stream Result

We reconstructed our apartment using the LiDAR sensor data streamed from an iPhone 15 Pro. The reconstruction was performed in real-time by scanning the room with the iPhone and streaming the camera-LiDAR data to a laptop via a Thunderbolt 3 cable. The final 3D reconstructed volume, rendered in color, was saved as a point cloud and is presented in Figure 4. Both the motion of the camera and the structure of the room were captured accurately, and the objects within the room are clearly identifiable.

Figure 4. The reconstruction of our apartment using iPhone LiDAR sensor stream

4.3. Incorporating Semantic Information from 2D

To evaluate segmentation, we utilized a pre-trained deep learning 2D segmentation model called SegFormer [12]. To streamline the process, we pre-segmented all frames in our dataset using this model. However, it's worth noting that SegFormer is inherently efficient and has the potential to operate in real-time. Refer to Figure 5 for an example of a fused segmented volume.

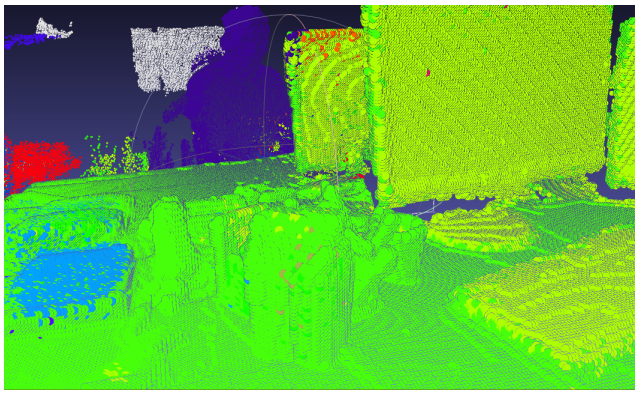


Figure 5. Sample of the segmented volume on the TUM Freiburg1 xyz dataset

5. Analysis

We conducted an experiment to compare the reconstruction time between CPU and CUDA implementation.

5.1. Processing time comparison

We reconstruct the scene freiburg1_xyz in the TUM dataset both on CPU implementation and CUDA implementation with a volume size of $(700, 400, 800)$ and volume scale (dimension of each voxel) of 5 mm. The dataset has 798 frames in total. The total time comparison between CPU and CUDA implementation and the average time per frame comparison is shown in Table 1. The results show that the CUDA implementation can achieve real-time performance in 3D scene reconstruction at 30 fps and is 14 times faster than the multi-threaded CPU implementation. We also compare the time of processing one frame and the time that each module takes for both CPU and CUDA implementation in Table 2. It is worth noting that there is an approximate 25 ms difference in the per-frame processing time between Table 1 (475.9 ms for CPU and 33.1 ms for CUDA) and Table 2 (447.0 ms for CPU and 7.4 ms for CUDA). The difference is attributed to the time-consuming nature of displaying intermediate images and saving output videos, especially in the CUDA implementation.

We also tested our pipline on the scene freiburg2_xyz (3666 frames) in the TUM dataset with a much smaller volume size of $(250, 160, 180)$ and volume scale (dimension

	CPU	CUDA
Total time	380.0 s	26.4 s
Time per frame	475.9 ms / 2 fps	33.1 ms / 30 fps

Table 1. Total computation time for processing the TUM rgbd_dataset_freiburg1_xyz of CPU and CUDA implementation (including the time for image display and video saving)

	Time (CPU)	Time (CUDA)
Surface measurement	14.2 ms	0.6 ms
Pose estimation	30.9 ms	3.5 ms
Surface reconstruction	333.3 ms	1.3 ms
Surface prediction	68.6 ms	2.0 ms
Total time of the frame	447.0 ms	7.4 ms

Table 2. Computation time comparison of each module between CPU and CUDA implementation of one example frame (excluding the time for image display and video saving)

	CPU	CUDA
Total time	548.8 s	123.1 s
Time per frame	149.7 ms / 6.7 fps	33.6 ms / 29.8 fps

Table 3. Total computation time for processing the TUM rgbd_dataset_freiburg2_xyz of CPU and CUDA implementation (including the time for image display and video saving)

of each voxel) of 10 mm. The computation time is shown in Table 3. Compared with freiburg1_xyz, the computation time of CPU significantly decreased, while the computation time of GPU remains almost the same. This is because the main bottleneck of CPU version is the computation of the algorithm, but the main bottleneck of GPU version is the transfer of data between the GPU memory and CPU memory. Smaller volume size results in less computational load on the CPU.

6. Conclusion

In this paper, we have detailed the development and optimization of KinectFusion through both CPU and GPU-based frameworks, achieving a substantial performance improvement with our CUDA version, capable of real-time 3D scene reconstruction at 30 frames per second — a 14-fold speed increase over the CPU-based approach. Our exploration extended to leveraging the iPhone's LiDAR capabilities for data acquisition and incorporating 2D semantic segmentation to refine the reconstruction quality. The high efficiency of CUDA implementation demonstrates the significant advantages of GPU parallel computation in 3D reconstruction tasks. We validated our approach's performance through practical experimentation, including the detailed reconstruction of environments using iPhone LiDAR data. These advancements not only showcase the capabilities of modern hardware and software in 3D reconstruction but also suggest new directions for our future work.

References

[1] Computer Vision Group - Dataset Download. https://cvg.cit.tum.de/data/datasets/rgbd-dataset/download. 1

[2] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, page 303–312, New York, NY, USA, 1996. Association for Computing Machinery. 1

[3] Christian Diller. Kinectfusionapp. https://github.com/chrdiller/KinectFusionApp, 2018. 1

[4] Alberto Elfes and Larry Matthies. Sensor integration for robot navigation: Combining sonar and stereo range data in a grid-based representataion. In *26th IEEE Conference on Decision and Control*, volume 26, pages 1802–1807, 1987. 1

[5] Jungong Han, Ling Shao, Dong Xu, and Jamie Shotton. Enhanced computer vision with microsoft kinect sensor: A review. *IEEE transactions on cybernetics*, 43(5):1318–1334, 2013. 1

[6] Carlos Hernández, George Vogiatzis, and Roberto Cipolla. Probabilistic visibility for multi-view stereo. *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2007. 1

[7] Shahram Izadi, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, Dustin Freeman, Andrew Davison, et al. 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*, pages 559–568, 2011. 1

[8] Gregor Luetzenburg, Aart Kroon, and Anders A Bjørk. Evaluation of the apple iphone 12 pro lidar for an application in geosciences. *Scientific reports*, 11(1):22221, 2021. 1

[9] Richard A Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE international symposium on mixed and augmented reality*, pages 127–136. Ieee, 2011. 1, 2, 3

[10] Sylvain Paris, Pierre Kornprobst, Jack Tumblin, Frédo Durand, et al. Bilateral filtering: Theory and applications. *Foundations and Trends® in Computer Graphics and Vision*, 4(1):1–73, 2009. 2

[11] Hongchi Xia, Yang Fu, Sifei Liu, and Xiaolong Wang. RGBD Objects in the Wild: Scaling Real-World 3D Object Learning from RGB-D Videos, Jan. 2024. 3

[12] Enze Xie, Wenhai Wang, Zhiding Yu, Anima Anandkumar, Jose M. Alvarez, and Ping Luo. Segformer: Simple and efficient design for semantic segmentation with transformers, 2021. 4

Appendix

1. Surface Measurement

1.1. CPU Implementation

Following the algorithm described above, we implement a compute_map() function to compute the normal and vertex maps. We use the bilateral filter and down-sampling from the OpenCV library. A multi-threading version has been implemented to reduce processing time by parallelizing depth map processing across different threads, optimizing efficiency in handling depth data.

1.2. CUDA Implementation

In CUDA, both the normal map and the vertex map are computed concurrently using multiple threads. The OpenCV CUDA module provides functions for bilateral filtering and sub-sampling. The GPU implementation closely resembles the CPU counterpart. The image is divided into several blocks, ensuring that a dedicated GPU thread handles each pixel. The number of blocks is determined based on the image size, with each block containing 32×32 threads (totaling 1024 threads). To cover the entire image, even when its size is not a perfect multiple of the block size, the number of blocks required in the x and y dimensions is calculated by rounding up.

2. Surface Reconstruction

2.1. CPU Implementation

Once the algorithm is defined and understood, the implementation becomes straightforward. We represent the volume as an OpenCV matrix with two channels: one for TSDF Value and one for weight. The volume matrix has a shape of $(Volumesize.y \times Volumesize.z, Volumesize.x)$. This data structure facilitates the transition to CUDA. Additionally, we employ a color volume of the same shape with three channels for RGB values for each voxel.

To enhance efficiency, we leverage multi-threading by dividing the rows of our volume matrix among different threads, with the number of rows for each thread determined by the available threads for the CPU.

The reconstruction implementation is also uncomplicated. For a voxel V and its position P in the global frame, we obtain its pixel coordinates by projecting it using camera intrinsics. Subsequently, we use the pixel coordinates to index the depth map. If the calculated TSDF is within the allowed truncation threshold, we update the TSDF and weight, as described in the previous section.

2.2. CUDA Implementation

The CUDA implementation closely resembles the multi-threaded approach. The key adjustment involves transitioning from a regular OpenCV Mat data structure to a GPU Mat, with memory allocation carried out using CUDA. The task distribution among threads and blocks in the GPU is analogous to the multi-threading approach. Specifically, a grid of threads is allocated, with each block containing 32×32 threads. The number of blocks is dynamically calculated based on the volume size.

3. Surface Prediction

3.1. CPU Implementation

The implementation on the CPU follows the algorithm. The main part is composed of a ray-casting function and a trilinear interpolation function. To accelerate the prediction speed and effectively utilize the hardware resources, we use multi-threading to parallel process image data. Each image is divided into several parts, with each thread being responsible for processing one part of the image. We calculate the number of rows that each thread should process. Each thread invokes the ray-casting function to process a segment of the image. Finally, we use the join method to ensure that the main thread waits for all child threads to finish their work before continuing.

3.2. CUDA Implementation

The GPU implementation is quite similar to the CPU implementation. The main difference is that we divide the image by blocks instead of dividing the image by the number of rows. we define the number of blocks based on the size of the image being processed, ensuring that each pixel is handled by one GPU thread. We define that each block contains 32×32 threads, which equals 1024 threads. The number of blocks required in the x and y dimensions of the image is calculated rounded up to ensure coverage of the entire image, even when its size is not an exact multiple of the block size.

4. Pose Estimation

4.1. CPU Implementation

To speed up the pose estimation process, we utilize the multi-thread feature provided by the C++ standard library. We divide the rows of vertex and normal maps into several parts and assign each part to a thread to calculate the contribution of vertex-normal element correspondences in parallel. The number of parts is determined by the number of available threads in the CPU. After obtaining the result of each part, we accumulate the contribution of each part to the linear system and use the Eigen library to perform

the LU decomposition. Lastly, we update the camera pose based on the result of the LU decomposition.

4.2. CUDA Implementation

The CUDA implementation of the pose estimation module is similar to the CPU implementation. We divide the pixels of vertex and normal maps into blocks of size 32×32 to form a grid of blocks. Each pixel in the block is assigned to a GPU thread to calculate the contribution of vertex-normal element correspondences in parallel and to save the result of each pixel to the shared memory. Parallel reduction is performed on the data in the shared memory, to sum up the contribution of each pixel in the block. After obtaining the result of each block, we accumulate the contribution of each block by another parallel reduction and download the results from GPU to CPU to use the Eigen library to perform the LU decomposition. Finally, we update the camera pose based on the result of the LU decomposition.