GPU Accelerated Robust Scene Reconstruction

Wei Dong[1], Jaesik Park[2], Yi Yang[1], and Michael Kaess[1]

***Abstract*— We propose a fast and accurate 3D reconstruction system that takes a sequence of RGB-D frames and produces a globally consistent camera trajectory and a dense 3D geometry. We redesign core modules of a state-of-the-art offline reconstruction pipeline to maximally exploit the power of GPU. We introduce GPU accelerated core modules that include RGB-D odometry, geometric feature extraction and matching, point cloud registration, volumetric integration, and mesh extraction. Therefore, while being able to reproduce the results of the high-fidelity offline reconstruction system, our system runs more than 10 times faster on average. Nearly 10Hz can be achieved in medium size indoor scenes, making our offline system even comparable to online Simultaneous Localization and Mapping (SLAM) systems in terms of the speed. Experimental results show that our system produces more accurate results than several state-of-the-art online systems. The system is open source at https://github.com/theNded/Open3D.**

I. Introduction

Dense 3D reconstruction of scenes is a fundamental component in localization and navigation tasks for intelligent systems such as robots, drones, and surveillance systems. Accurate 3D models of real-world scenes is a key element for mixed and virtual reality applications, because it is directly related to realistic content creation or telepresence. In recent years, research in scene reconstruction using RGB-D frames has flourished with the presence of affordable, high-fidelity consumer-level RGB-D sensors, and a number of off-the-shelf reconstruction systems have been introduced so far.

Reconstruction systems can be generally classified into online and offline systems. Systems including VoxelHashing [1], InfiniTAM [2], and ElasticFusion [3] are based on dense SLAM algorithms. These algorithms suffer from *pose drift*, which usually causes corrupted models in large scenes. BundleFusion [4], as an online system, is more similar to offline structure-from-motion systems. It keeps track of all the RGB-D keyframes, hence is computationally expensive and requires two high-end graphics cards to run. Offline systems such as Open3D [5] and Robust Reconstruction Pipeline [6], on the other hand, include hierarchical global constraints such that the camera poses are globally consistent and accurate even in large scenes. However, the expected running times of such systems span from several hours to days.

In this paper, we present an accelerated offline RGB-D reconstruction system. The basic algorithm is based on offline systems such as Open3D [5], [6]. On top of this, the major contributions in this work include:

- GPU acceleration of core modules in an offline reconstruction system such as RGB-D odometry, Iterative Closest Point (ICP), global registration, volumetric integration, and Marching Cubes.
- Designing new GPU data containers to boost basic operations.
- 10x faster on average than baseline offline systems, comparable to online systems in terms of speed.
- While being fast as an offline system, the reconstruction accuracy is maintained on most datasets compared to the state-of-the-art offline system [6].

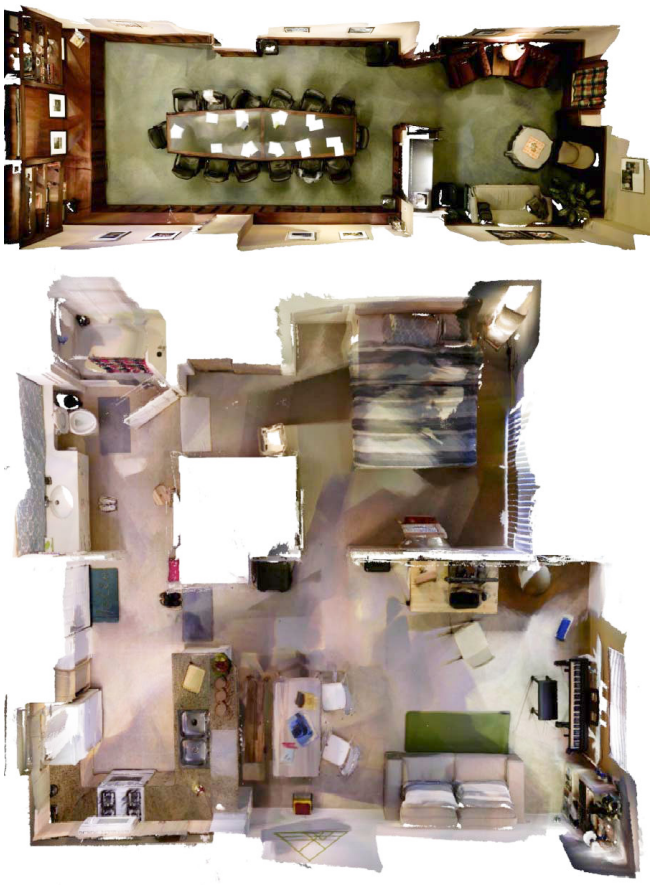


Fig. 1. Reconstructed large indoor scenes, *boardroom* and *apartment*. The largest scene *apartment* ($>$ 30K RGB-D frames) is reconstructed by our system within 1.5 hours, 6Hz on average. These scenes typically require more than 10 hours to run on the offline system [5], and will fail on online systems [3], [2]

[1]Wei Dong, Yi Yang, and Michael Kaess are with Robotics Institute, Carnegie Mellon University, PA, USA {weidong, yiy4, kaess}@andrew.cmu.edu

[2]Jaesik Park is with Compute Vision Lab, POSTECH, Pohang, South Korea jaesik.park@postech.ac.kr

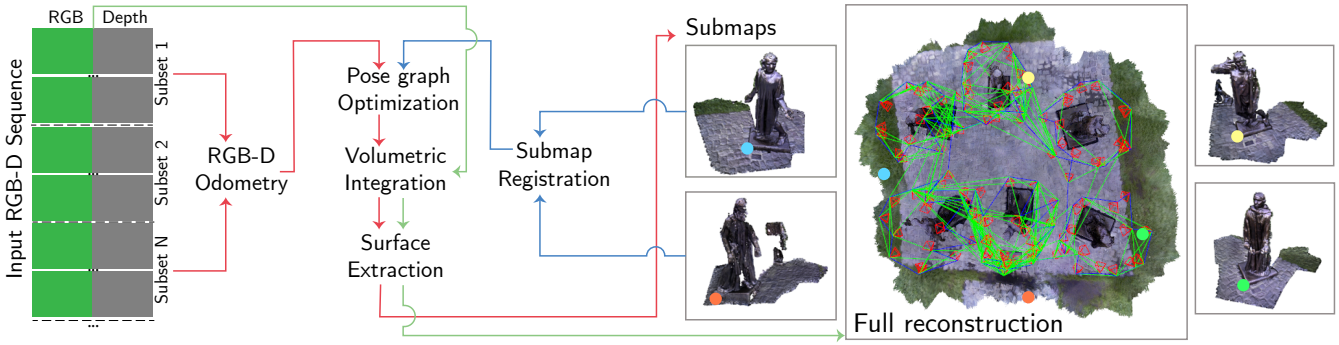


Fig. 2. System overview. Left shows workflows of the 3 major stages, displayed in red (*make submaps*), blue (*register submaps* and *refine registration*), and green (*integrate scene*) lines respectively. Right shows local and global reconstruction results, along with the pose graph of submaps. Best viewed in color.

II. Related Work

State-of-the-art online dense 3D reconstruction systems [3], [1], [2], [4] track the pose of the current frame using visual odometry (VO), fuse data into the dense map, and search for loop closures to reduce drift. While these methods are fast enough to track input sequences, inevitable drift can cause incorrect dense maps.

ElasticFusion [3] proposes to correct drift in a surfel-based [7] map by introducing graph deformation. However, the deformation is sensitive to user parameters and may easily fail on various benchmarks. In addition, as surfels accumulate quickly, the system may not scale to larger scenes. InfiniTAM [2] and VoxelHashing [1] apply frame-to-model tracking [8] to overcome the drift issue. RGB-D data is fused into a Truncated Signed Distance Field (TSDF) [9] with estimated poses, and the system extracts the model from the TSDF for tracking. However, it may break down when incorrect camera poses corrupt the dense map, even only locally. InfiniTAM [2] introduces a relocalizer to recover the system from tracking failures, but there is no strategy to handle the corruptions in the TSDF map.

BundleFusion [4] performs exhaustive feature-based bundle adjustment on the keyframes in the entire sequence to correct camera poses, and uses brute-force re-integration [10] to correct the TSDF map. For this reason, the system requires two high-end graphics cards[1] to ensure real-time, which is beyond consumer level. Moreover, it keeps track of all the keyframes and performs global optimization frequently, hence it is only applicable to scenes with $\leq$ 25000 RGB-D frames. It suggests that the computational cost may be reduced by splitting large scenes and processing hierarchical optimization.

On the other hand, offline reconstruction systems [6], [5] address the drift issues in the online system by adopting a hierarchical framework, as mentioned in [4]. The system splits input sequences of varying lengths into subsets and generates submaps using methods similar to online SLAM systems. Afterwards, these submaps are registered pairwise to form a pose graph [11], which is optimized to provide accurate poses for submaps. Finally, all RGB-D frames with accurate poses are integrated into a global TSDF volume to extract a complete 3D model. This approach can easily modularize the overall procedure into small tasks such as VO, dense mapping, and surface registration. Each decoupled component can benefit from available state-of-the-art algorithms.

The offline systems use direct RGB-D odometry to estimate the pose between subsequent frames. They can fully utilize available depth and color information by *directly* minimizing joint photometric and geometric error [12], [13], [14], as well as overcome challenging textureless scenes where feature-based method [15] may fail. Assuming that pose drift is small over the course of each submap, the systems fuse raw RGB-D data into a TSDF volume [8], [1] and extract the mesh or point cloud as a submap.

At the level of submaps, point cloud registration is required to estimate relative poses between corresponding point sets. With proper initialization, classical ICP [16] iteratively computes data association and registration. The state-of-the-art Colored ICP [14] achieves high accuracy between dense colored point clouds. In cases where the pose initialization is not reliable, globally optimal methods are preferred. [17] searches the $SE(3)$ space with bound-and-branch strategy for potential initializations and then performs ICP. A faster yet accurate algorithm Fast Global Registration (FGR) [18] relies on Fast Point Feature Histograms (FPFH) [19] to generate possible matches, and filters incorrect matches with fast and robust line processes [20] defined in [18].

The obvious limitation of aforementioned offline systems is that they take a very long time to reconstruct large scale scenes. In this paper, we propose a GPU-accelerated offline system to reach the level of online system performance while not sacrificing accuracy.

III. Reconstruction System

The basic principle of the proposed system is similar to state-of-the-art offline reconstruction systems [5], [6]. The overall procedure of the system consists of four major stages, as shown in Fig. 2:

[1] An NVIDIA TITAN X and a TITAN black card were used for the real-time demonstration.

1) *Make submaps*. Frame-to-frame camera pose is estimated for evenly divided subsets of the input sequence. Afterwards, TSDF volume integration is performed for each subset of the sequence, and a *submap* is extracted in the form of a triangle mesh.
2) *Register submaps*. Pairwise submap registration is applied. The submap pairs that are temporally adjacent are registered using *Colored ICP* [14] with reliable initial relative poses from RGB-D odometry. Temporally distant submaps are registered using geometric feature-based *FGR* [18] to detect loop closures. A pose graph is constructed and optimized to determine the poses of submaps in the *global* coordinate system.
3) *Refine registrations*. The relative poses between each registered submap pair, including both the adjacent and the distant submap pairs, are further refined using multi-scale Colored ICP. A subsequent pose graph optimization determines the final global poses for every submap.
4) *Integrate scene*. Given the optimized poses of the submaps and the poses of every frame, TSDF integration fuses the entire RGB-D sequence, and produces the final 3D reconstruction.

A. Basic Data Containers for GPU

As the basic infrastructure, we designed several GPU data structures for general usage in our system.

- *1D arrays*. We implement atomic *push_back* operation to support multi-thread writing, mimicking a vector on CPU.
- *2D arrays*. They are stored in *row* major order. In a typical parallel execution on a large 2D array, each thread is designed to iterate over one *column*. Since the threads are approximately accessing the same *row* simultaneously, they are more likely to share the cache that loads the same section of global memory, which increases accessing efficiency.
- *Linked list and hash table*. A generic templated (key, value, and hashing function) hash table is implemented for GPU. Each hashed key corresponds to a fixed size bucket array plus a dynamic size bucket linked list to address conflicts. The hash table is essential for spatial hashing used in 3D volume storage.

All the data structures support memory transfer between GPU and CPU. Since global memory allocation is expensive on GPU, we pre-allocate enough memory, and rely on a dynamic memory manager to manually allocate and free memory on GPU on demand. Reference counting is implemented for efficient GPU memory sharing.

B. RGB-D Odometry

In RGB-D odometry, we seek to optimize an error function of relative pose between two consecutive RGB-D frames $\langle \mathcal{F}_s, \boldsymbol{\xi}_s \rangle$ and $\langle \mathcal{F}_t, \boldsymbol{\xi}_t \rangle$.[2] The error function is constructed as in [14], including both the photometric error r_I and the difference of depth r_D:

$$E(\boldsymbol{\xi}_s^t) = \sum_{\mathbf{p}} (1-\sigma)\ r_I^2(\boldsymbol{\xi}_s^t, \mathbf{p}) + \sigma\ r_D^2(\boldsymbol{\xi}_s^t, \mathbf{p}), \tag{1}$$

$$r_I = \mathcal{I}_s[\mathbf{p}] - \mathcal{I}_t[\mathcal{W}(\boldsymbol{\xi}_s^t, \mathbf{p}, \mathcal{D}_s[\mathbf{p}])], \tag{2}$$

$$r_D = \mathcal{T}(\boldsymbol{\xi}_s^t, \mathbf{p}, D_s[\mathbf{p}]).z - \mathcal{D}_t[\mathcal{W}(\boldsymbol{\xi}_s^t, \mathbf{p}, \mathcal{D}_s[\mathbf{p}])], \tag{3}$$

where $\boldsymbol{\xi}_s^t \in SE(3)$ is the relative pose from $\mathcal{F}_s$ to $\mathcal{F}_t$, $\mathbf{p} \in \mathbb{R}^2$ is the pixel in the RGB-D image $\mathcal{F}_s$, and $\sigma \in [0, 1]$ is a hyper parameter to balance the appearance and geometry terms. We use $\mathcal{T}(\boldsymbol{\xi}, \mathbf{p}, d)$ and $\mathcal{W}(\boldsymbol{\xi}, \mathbf{p}, d)$ to denote rigid transformation and warping (transformation + projection) of a 3D point with its pixel coordinate $\mathbf{p}$ and depth value d respectively. For simplicity, the intrinsic matrix is not displayed in these functions.

To minimize the non-linear error function, Gauss-Newton optimization is applied. By computing the first-order linearization, we have

$$r_I(\boldsymbol{\xi}_s^t + \Delta\boldsymbol{\xi}, \mathbf{p}) \approx r_I(\boldsymbol{\xi}_s^t) + J_I(\boldsymbol{\xi}_s^t, \mathbf{p})\Delta\boldsymbol{\xi}, \tag{4}$$

$$r_D(\boldsymbol{\xi}_s^t + \Delta\boldsymbol{\xi}, \mathbf{p}) \approx r_D(\boldsymbol{\xi}_s^t) + J_D(\boldsymbol{\xi}_s^t, \mathbf{p})\Delta\boldsymbol{\xi}, \tag{5}$$

where J_I and J_D are Jacobian matrices for each term per pixel. Finally, the problem reduces to solving $\Delta\boldsymbol{\xi}$ in the least squares system and updating $\boldsymbol{\xi}_s^t$ iteratively:

$$\sum_p A(\mathbf{p})\Delta\boldsymbol{\xi} = -\sum_{\mathbf{p}} b(\mathbf{p}), \tag{6}$$

$$\boldsymbol{\xi}_s^t = \Delta\boldsymbol{\xi} \oplus \boldsymbol{\xi}_s^t, \tag{7}$$

where the system is built up with

$$A(\mathbf{p}) = (1-\sigma)\ J_I^T J_I(\boldsymbol{\xi}_s^t, \mathbf{p}) + \sigma\ J_D^T J_D(\boldsymbol{\xi}_s^t, \mathbf{p}), \tag{8}$$

$$b(\mathbf{p}) = (1-\sigma)\ J_I^T r_I(\boldsymbol{\xi}_s^t, \mathbf{p}) + \sigma\ J_D^T r_D(\boldsymbol{\xi}_s^t, \mathbf{p}). \tag{9}$$

To ensure faster convergence, we implement coarse-to-fine RGB-D odometry using a image pyramid.

It is straightforward to parallelize building the Jacobian matrix, because each pixel contributes to the Jacobian independently. However, proper handling of thread conflicts in the summation operation is critical here, because the number of GPU threads is often larger than a few thousand, and thread conflicts degrade performance. While an `Atomic-Add` operation is supported on most GPUs to avoid conflicts, the more advanced technique `Reduction` [21] can accelerate the process further by fully utilizing *shared* memory instead of *global* memory. There are two variations of Reduction, the original one [21] and `Warp Shuffle` used in [3]. Based on our experiments, the original version runs two times faster than built-in atomic-add, and outperforms warp shuffle.

As $A(\mathbf{p})$ is a symmetric matrix, only the upper-right 21 elements are summed, while the other elements are duplicated on CPU. Plus 6 elements in $b(\mathbf{p})$, we need to sum 27 elements separately. To accelerate, we allocate 3 arrays of shared memory and sum 3 elements in the linear system at one time. With reasonable shared memory consumption, the cost of frequent thread synchronization is reduced: reduction

[2] We define the terms as follows. $\langle \mathcal{F}, \boldsymbol{\xi} \rangle$ denotes an RGB-D frame. Here, $\mathcal{F} = \langle \mathcal{I}, \mathcal{D} \rangle$ consists of color and depth images, and $\boldsymbol{\xi} \in SE(3)$ represents the 6-DOF pose vector in the local submap's coordinate system.

takes around 2∼3 ms to sum elements in the linear system for a 640×480 image. This general reduction module is also used in Sec. III-E, where similar linear systems are defined.

C. Integration

With the known pose $\boldsymbol{\xi}$ of each frame $\mathcal{F}$, we can use TSDF integration to fuse raw RGB-D data into the volumetric scalar field. To improve the storage efficiency, we use spatial hashing as described in [2], [1], with the modified hash table structure described in Sec. III-A. The space is coarsely divided into voxel blocks for spatial hashing; each block contains an array of $8 \times 8 \times 8$ voxels for fast and direct parallel memory access by GPU threads.

The integration consists of three passes. In the first pass, a point cloud is generated from frame $\mathcal{F}$ and transformed using its pose $\boldsymbol{\xi}$. The blocks that can cover the generated points will be created if they do not exist yet. In the second pass, all generated blocks in the $\mathcal{F}$'s viewing frustum will be collected into a buffer. Finally, parallel integration will be performed on collected blocks in the buffer. Each voxel in the blocks will be projected onto $\mathcal{F}$ to find the projective closest pixel. After that, the voxels update their stored TSDF value using a weighted average:

$$d = \phi(\mathcal{D}[\mathcal{W}(\boldsymbol{\xi}, \mathbf{x})] - \mathbf{x}.z), \tag{10}$$

$$\mathrm{TSDF}[\mathbf{x}] = \frac{\mathrm{TSDF}[\mathbf{x}] + d}{\mathrm{Weight}[\mathbf{x}] + 1}, \tag{11}$$

$$\mathrm{Weight}[\mathbf{x}] = \mathrm{Weight}[\mathbf{x}] + 1, \tag{12}$$

where ϕ is the truncation function in [8], $\mathbf{x} \in \mathbb{R}^3$ is the voxel coordinate, and $\mathcal{W}(\boldsymbol{\xi}, \mathbf{x})$ projects $\mathbf{x}$ to the frame after transforming $\mathbf{x}$ with $\boldsymbol{\xi}$. Weighted average generally also applies to colors.

D. Mesh Extraction

After the TSDF has been generated, Marching Cubes (MC) [22] is applied to extract surfaces from TSDF volumes. We improve the mesh extraction framework in [23] both in terms of memory and speed.

As pointed out in [23], the vertex on the shared edge between two voxels can be computed only once. By setting up a one-to-one correspondence between edges and voxels, we can compute every unique vertex once. Directly maintaining the edge-vertex correspondences in voxels along with the TSDF creates a large overhead in memory. In the improved version, we detach the structure, so that it can be allocated and attached only to the active parts of the TSDF volume that require meshing.

One of the most computationally expensive operations in MC is normal estimation from the TSDF. Given a vertex $\mathbf{x}_v = (x_v, y_v, z_v)$, the normal is computed by normalized gradient $\nabla\mathrm{TSDF}(\mathbf{x}_v)/||\nabla\mathrm{TSDF}(\mathbf{x}_v)||$, where

$$\begin{aligned} &\nabla\mathrm{TSDF}(\mathbf{x}_v) \\ &= \begin{bmatrix} \mathrm{TSDF}(x_v+1, y_v, z_v) - \mathrm{TSDF}(x_v-1, y_v, z_v) \\ \mathrm{TSDF}(x_v, y_v+1, z_v) - \mathrm{TSDF}(x_v, y_v-1, z_v) \\ \mathrm{TSDF}(x_v, y_v, z_v+1) - \mathrm{TSDF}(x_v, y_v, z_v-1) \end{bmatrix}. \end{aligned} \tag{13}$$

Since the 6 query points are not grid points, 8 spatial queries are required to get a tri-linear interpolated TSDF value per query point. 48 spatial queries and 48 interpolations for one single point is expensive. We found that the final computed normal is *identical* to a simple interpolation of normals at the two adjacent grid points, while the normal computation at a grid point only requires 6 queries, as shown in Fig. 3. Therefore, we need only 12 spatial queries and 1 interpolation.

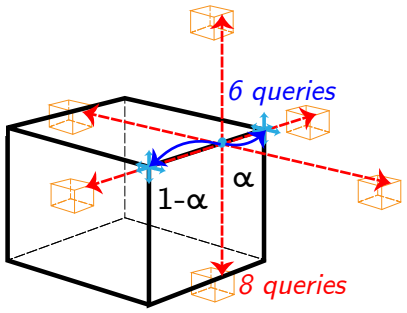


Fig. 3. Illustration of normal extraction. Red lines show the redundant queries (tri-linear interpolation for non-grid points) to extract normal. Blue lines show the optimal interpolation of normals at adjacent grid points. α is the interpolation ratio of normals, which can be directly computed in MC.

To further speed up MC in spatial hashed voxel blocks, we specifically implement MC for

1) Voxels inside a voxel block. Modified MC in [23] is directly performed on such voxels.
2) Voxels at the boundary of a voxel block, where frequent hash table queries are required. Despite the fact that hash tables are *supposed* to be $O(1)$ look up, the overhead is considerable. To speed up, for each block, we first cache pointers of all 26 neighbor blocks. Then each voxel can access neighbor voxels from the cached block instead of looking them up in the hash table.

E. Registration

Given extracted surfaces as submaps $\mathcal{S}$, we perform a multiway registration defined in [6] to get their poses $\boldsymbol{\zeta}$.

1) Colored ICP: Point cloud pairs with good initial pose guesses are registered using Colored ICP [14]. Typically, we can obtain reasonable initialization between two consecutive submaps by aligning the poses of the last RGB-D frame in the first submap and the first RGB-D frame in the second submap.

Colored ICP builds up a linear system similar to Eq. 2. While RGB-D images are dense in 2D, point clouds are sparse in 3D. Therefore, for point cloud registration, projective data association has to be replaced by nearest neighbor search. Since there are no satisfying alternatives on GPU, we use Fast Library for Approximate Nearest Neighbors (FLANN) [24] on CPU to find 3D nearest neighbors, as a legacy of [5]. In addition, the gradient operation that is natural on 2D images is not well-defined on point clouds, causing problems in computing Jacobians. It is easy to replace gradient of the depth image with each point's normal;

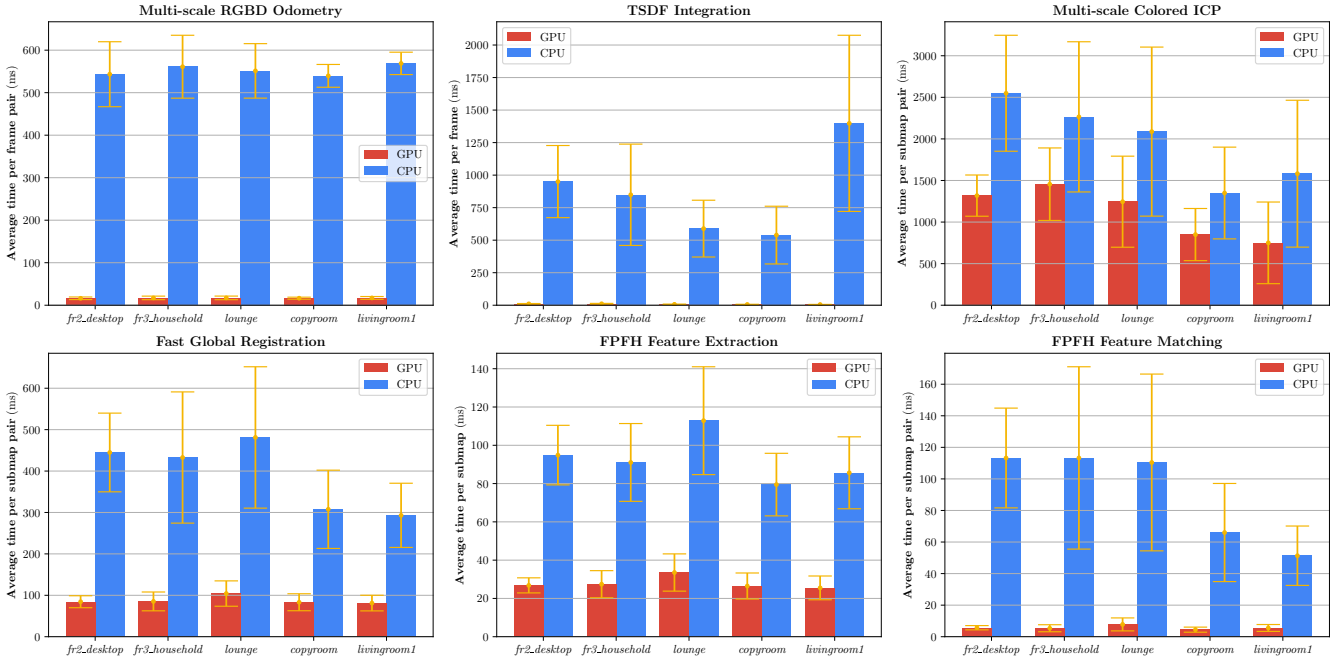


Fig. 4. Average runtime comparison of our system (GPU) and Open3D [5] (CPU).

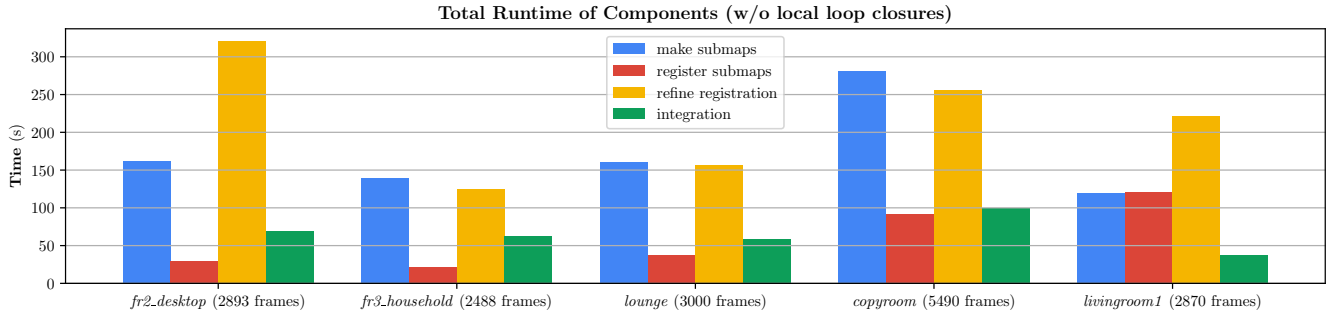


Fig. 5. Total system runtime on datasets. $\sim$ 8Hz on average is achieved on most datasets.

with the correspondences from FLANN, we can compute approximated color gradient per point, as defined in [14]:

$$\left(\sum_{\mathbf{p}'\in\mathcal{N}(\mathbf{p})} A(\mathbf{p}')^T A(\mathbf{p}') + \mathbf{n}(\mathbf{p})\mathbf{n}(\mathbf{p})^T\right)\ \mathbf{d}(\mathbf{p}) = \sum_{\mathbf{p}'\in\mathcal{N}(\mathbf{p})} A(\mathbf{p}')^T b(\mathbf{p}'). \tag{14}$$

Here $\mathbf{p} \in \mathbb{R}^3$ denotes the point we are processing, $\mathbf{n}(\mathbf{p})$ is its normal, $\mathbf{p}' \in \mathcal{N}(\mathbf{p})$ represents its neighbors, and $\mathbf{d}(\mathbf{p})$ is the target color gradient at $\mathbf{p}$. The linear system depends on the projective distances and color differences between $\mathbf{p}$ and its neighbors:

$$A(\mathbf{p}') = \mathbf{p}' - \mathbf{n}(\mathbf{p})^T(\mathbf{p}' - \mathbf{p})\ \mathbf{n}(\mathbf{p}), \tag{15}$$

$$b(\mathbf{p}') = \mathcal{C}(\mathbf{p}') - \mathcal{C}(\mathbf{p}), \tag{16}$$

where $\mathcal{C}(\mathbf{p})$ is the intensity of a point. To solve these small linear systems in parallel, we implement a complete module for LDL decomposition and forward substitution on GPU.

After preprocessing, our implementation iterates between finding data correspondences on CPU and building the linear system on GPU until convergence. Apart from Colored ICP, classical point-to-point and point-to-plane ICP are also supported within the same architecture.

2) Fast Global Registration: We compute pairwise registrations between non-adjacent submaps to detect potential loop closures. As no initialization is provided, we have to rely on 3D features to get correspondences. FPFH [19] is used in our case. FPFH only relies on local neighbors, therefore feature extraction is possible to run in parallel, provided nearest neighbors are found by FLANN.

As a high dimensional (33 dim) feature, FPFH is less efficient to match using FLANN. We turn to brute-force parallel NN matching by simplifying KNN-CUDA [25] to the 1-NN case. Brute force NN consists of 2 passes. In the first pass, a dense distance matrix is calculated in parallel after the matrix is divided into small submatrices. Here, the cache friendly column-wise iterations we designed in

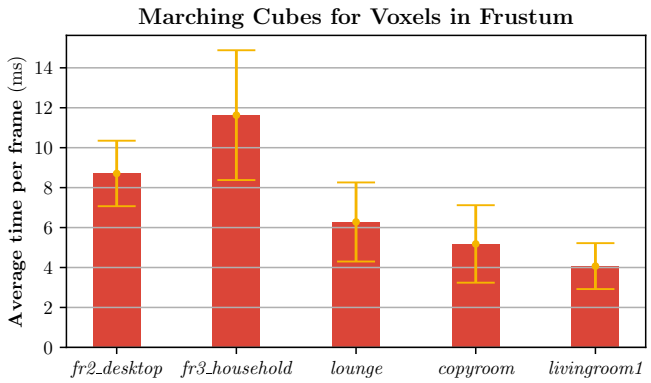


Fig. 6. Marching Cubes runtime. Note in this experiment, ground truth trajectories are used to ensure reasonable viewing frustums.

Sec. III-A help to accelerate the step by a factor of 2. In the second pass, each query point will search for their nearest neighbor by `Reduction` with the `min` operator, which resembles the `Reduction` with `add` we used. The drawback of brute-force matching is the memory cost. The dense distance matrix will consume all the GPU memory when the sizes of point clouds are large (> 20K). Under such circumstances, downsampling is required.

With known matches, we reuse our framework to build a linear system for the FGR algorithm [18]. As there are point-wise line processes involved, the error function is much different from Eq. 2. We refer readers to the original paper for more details. Before applying Gauss-Newton, feature matching tests are performed in parallel, where each thread handles a random test set. All the data stay on GPU after feature extraction, including parallel random number generation. No expensive CPU operation is required after FLANN has finished its job, hence FGR can run very fast and stable.

After registration, estimated poses ζ are inserted in a pose graph as nodes, with edges between adjacent submaps and potential loop closures. The information matrices on the edges are computed similar to the building process of linear systems with minor changes in residuals and Jacobians. We then perform robust multiway registration [6] to eliminate false loop closures and obtain optimized poses.

IV. Experimental Results

The proposed system can reproduce the results of the state-of-the-art offline reconstruction system implemented in Open3D [6], [5] with significantly reduced time budget. In this section, we compare runtime with the baseline offline reconstruction system Open3D [5], and show qualitative and quantitative results with state-of-the-art online reconstruction systems. The tested datasets include the TUM RGB-D dataset [26], the Stanford and Redwood simulated dataset [6], and the large Indoor LIDAR-RGBD Scan dataset [14]. The voxel size is set to 6mm in real-world scenes and 8mm in Redwood simulated scenes.

The GPU part is implemented in CUDA, where CPU utility functions are built upon Open3D [5]. The following experiments were run on a laptop with an Intel i7-6700 CPU and a NVIDIA 1070 graphics card with 8G GPU memory.

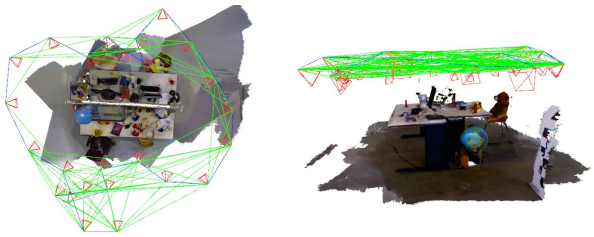


Fig. 7. Pose graph of submaps on *TUM household* and *desk* datasets. Valid loop closures are detected between submaps.

A. Runtime Results

In Fig. 4, we first show the acceleration of the components in our system.

- Multi-scale RGB-D Odometry runs with $\{20, 10, 5\}$ iterations from coarse to fine scale. It is accelerated to around $\sim$16ms per frame, around 40 times faster than the baseline. As it already achieves real-time, RGB-D SLAM systems may include this module.
- TSDF integration is accelerated to $\sim$10ms per frame on a volume with high resolution, approximately $50\sim120$ times faster. This component can also serve as a part of a real-time dense SLAM / mapping system.
- Multi-scale Colored ICP runs with $\{50, 30, 14\}$ iterations from coarse to fine scale. It is accelerated only with a factor of 1.5 to 2. The major reason is that we still rely on FLANN for data correspondences, which takes a large amount of time. Note the runtime excludes the slow point cloud downsampling, which takes place in both CPU and GPU methods.
- FGR is accelerated to around 10Hz, $4\sim5$ times faster than CPU version. GPU based feature extraction and matching significantly reduces the runtime. These modules may also be separately used in other tasks such as naive 3D object recognition and matching.

We also separately demonstrate the performance of parallel MC in Fig. 6. As it runs very fast for visualizing surfaces in the viewing frustum, it can serve as a visualizer for dense SLAM systems. A screenshot is shown in Fig. 11.

Fig. 5 provides a general runtime overview of the system on real-world datasets, where around 8Hz is achieved on most datasets. Note in the *make submaps* stage we only account for pure odometry. Optional feature-based loop closure detection may increase the local trajectory accuracy, with the cost of $2\sim3$ times runtime. A major time consuming operation in *refine registration* is point cloud downsampling on CPU, which can be accelerated on GPU.

B. Reconstruction Results

We now show the qualitative reconstruction results on the TUM dataset in Fig. 7. The pose graph of the submaps is also illustrated, from which we can observe correct loop closures are found with pairwise FGR. Additional experimental results on *lounge, copyroom*, and *stonewall* can be viewed in Fig. 9. We can easily observe that the details

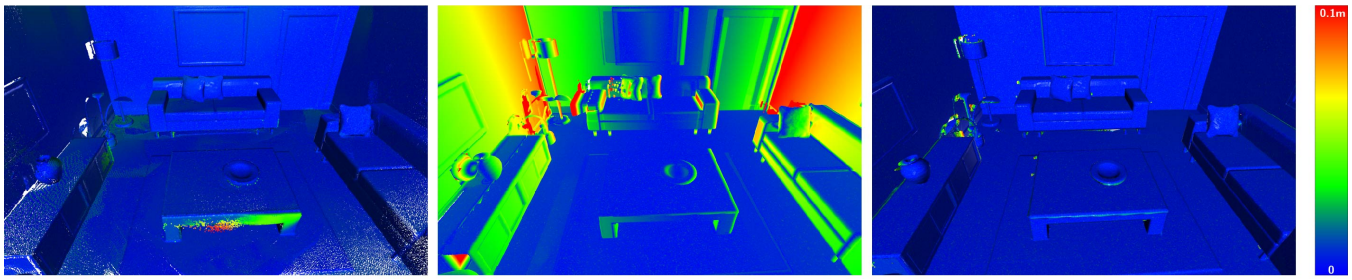


Fig. 8. Distance heatmap from reconstructed model to ground truth on *livingroom1*, generated from CloudCompare. From left to right, *ElasticFusion*, *InfiniTAM* (hybrid color and depth tracker fails, shift occurs in ICP tracker), ours.

are well preserved in the scenes without drift or apparent artifacts. Fig. 1 shows reconstruction results on more challenging scenes *boardroom* and *apartment* with more than 20K frames. Our system can still successfully close the loops and output accurate models, while other online SLAM systems fail.

Due to difficulties in configuration, we do not run BundleFusion [4] on our machine. Instead, we compare against their results on the BundleFusion dataset qualitatively. In Fig. 10 we can see that while producing similar results, our pipeline can better align surfaces.

As for quantitative results, we show the heatmap of cloud-to-cloud distances between estimated models and ground truth on the *livingroom* from the Redwood simulated dataset. The ground truth model is adapted from [27]. The heatmaps are computed by *CloudCompare* software. In Fig. 8, we can see there are only minor discrepancies between our output model and the ground truth, while a shift is observable in the online systems. More results on the Redwood simulated and Indoor LIDAR-RGBD datasets are listed in Table I. Our system has consistently higher reconstruction accuracy, and works on large scenes where other online systems fail to recover the whole scene.

C. Additional Result using Mobile Device

Our system not only works on laptops and PCs, but also on mobile devices supporting CUDA. Given real-world data collected from an Intel RealSense, our system is able to reconstruct the road in minutes on a NVIDIA Jetson TX2, see Fig. 12.

V. Conclusions and Future Work

We implemented a GPU-accelerated dense RGB-D offline reconstruction system which runs fast on real-world datasets while maintaining state-of-the-art reconstruction results. The system is open source, and the major components can be used separately in various applications, such as real-time SLAM and object recognition.

In the future, we intend to introduce advanced relocalizers to replace pairwise submap matching for higher speed and robustness in loop closure detection. Equipped with an efficient relocalizer, we may bring up an online system that accepts streaming input frames. Another research direction is a fast 3D nearest neighbor search. Advance techniques such as parallel cuckoo hashing in [28] may be considered.

Acknowledgement

We thank Ben Guidarelli for the outdoor road reconstruction experiments on NVIDIA Jetson TX2.

References

[1] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger, "Real-time 3D reconstruction at scale using voxel hashing," *ACM Trans. on Graphics*, vol. 32, no. 6, p. 169, 2013.

[2] O. Kahler, V. A. Prisacariu, C. Y. Ren, X. Sun, P. H. S. Torr, and D. W. Murray, "Very high frame rate volumetric integration of depth images on mobile device," *IEEE Transactions on Visualization and Computer Graphics*, vol. 22, no. 11, 2015.

[3] T. Whelan, S. Leutenegger, R. F. Salas-Moreno, B. Glocker, and A. J. Davison, "ElasticFusion: Dense SLAM without a pose graph," in *Robotics: Science and Systems (RSS)*, 2015.

[4] A. Dai, M. Nießner, M. Zollöfer, S. Izadi, and C. Theobalt, "Bundlefusion: Real-time globally consistent 3d reconstruction using on-the-fly surface re-integration," *ACM Trans. on Graphics*, 2017.

[5] Q.-Y. Zhou, J. Park, and V. Koltun, "Open3D: A modern library for 3D data processing," *arXiv:1801.09847*, 2018.

[6] S. Choi, Q.-Y. Zhou, and V. Koltun, "Robust reconstruction of indoor scenes," in *Proc. IEEE Int. Conf. Computer Vision and Pattern Recognition*, 2015.

[7] M. Keller, D. Lefloch, M. Lambers, S. Izadi, T. Weyrich, and A. Kolb, "Real-time 3D reconstruction in dynamic scenes using point-based fusion," in *Proceedings of 3DV*. IEEE, 2013.

[8] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon, "KinectFusion: Real-time dense surface mapping and tracking," in *IEEE and ACM Intl. Sym. on Mixed and Augmented Reality (ISMAR)*. IEEE, 2011.

[9] B. Curless and M. Levoy, "A volumetric method for building complex models from range images," in *SIGGRAPH*. ACM, 1996, pp. 303–312.

[10] N. Fioraio, J. Taylor, A. Fitzgibbon, L. Di Stefano, and S. Izadi, "Large-scale and drift-free surface reconstruction using online subvolume registration," in *Proc. IEEE Int. Conf. Computer Vision and Pattern Recognition*, 2015.

[11] G. Grisetti, R. Kummerle, C. Stachniss, and W. Burgard, "A tutorial on graph-based SLAM," *IEEE Intelligent Transportation Systems Magazine*, vol. 2, no. 4, pp. 31–43, 2010.

[12] F. Steinbrücker, J. Sturm, and D. Cremers, "Real-time visual odometry from dense RGB-D images," in *Proceedings of ICCV Workshops*. IEEE, 2011.

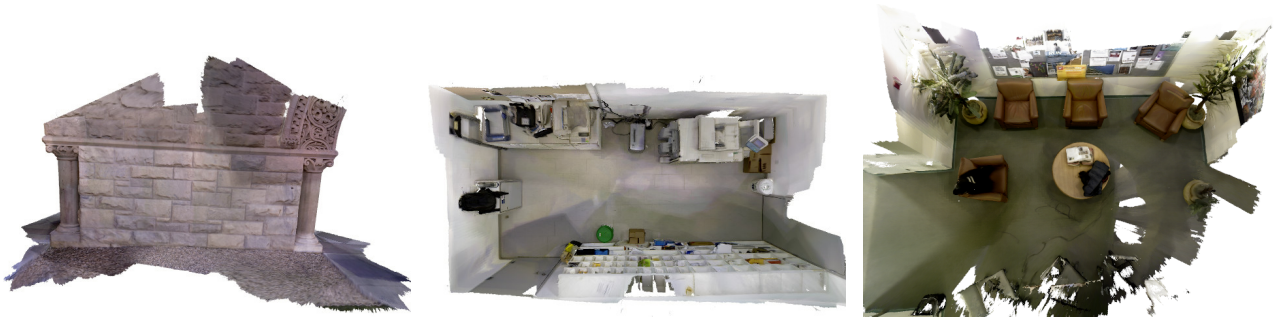
[13] C. Kerl, J. Sturm, and D. Cremers, "Robust odometry estimation for RGB-D cameras," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*. IEEE, 2013.

[14] J. Park, Q.-Y. Zhou, and V. Koltun, "Colored point cloud registration revisited," in *Intl. Conf. on Computer Vision (ICCV)*, 2017.

[15] R. Mur-Artal and J. D. Tardós, "ORB-SLAM2: An open-source slam system for monocular, stereo, and RGB-D cameras," *IEEE Trans. Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017.

[16] P. J. Besl and N. D. McKay, "A method for registration of 3-D shapes," in *Sensor Fusion IV: Control Paradigms and Data Structures*, vol. 1611. Intl. Society for Optics and Photonics, 1992, pp. 586–607.

[17] J. Yang, H. Li, D. Campbell, and Y. Jia, "Go-ICP: A globally optimal solution to 3D ICP point-set registration," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 38, no. 11, pp. 2241–2254, 2016.



(a) stonewall (b) copyroom (c) lounge

Fig. 9. Qualitative results on *stonewall, lounge*, and *copyroom*.

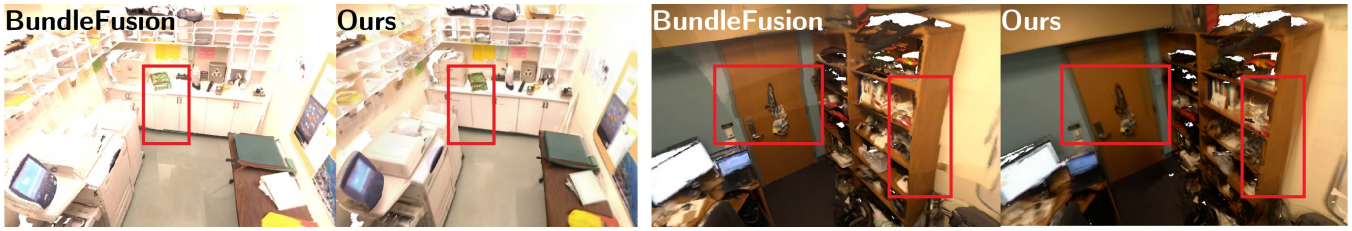


Fig. 10. Qualitative comparison to BundleFusion on *copyroom* and *office3* in *bundlefusion* dataset. Surfaces are better aligned in our reconstructed models.

TABLE I

RECONSTRUCTION ACCURACY (METER) ON SIMULATED AND REAL-WORLD DATASETS WITH GROUND TRUTH MODELS.

SYSTEM	*livingroom1*		*livingroom2*		*office1*		*office2*		*apartment*		*boardroom*	
	MEAN	STD	MEAN	STD	MEAN	STD	MEAN	STD	MEAN	STD	MEAN	STD
InfiniTAM [2]	0.0391	0.0451	**0.0045**	0.0018	0.0661	0.0504	**0.0052**	0.0028	fail	-	fail	-
ElasticFusion [3]	0.0060	0.0036	0.0100	0.0051	0.0258	0.0306	0.0074	0.0038	fail	-	fail	-
Ours	**0.0049**	0.0043	0.0068	0.0049	**0.0051**	0.0040	0.0054	0.0043	**0.0533**	0.0521	**0.0607**	0.0567

Fig. 11. A screenshot of real-time Marching Cubes for voxels in frustum.

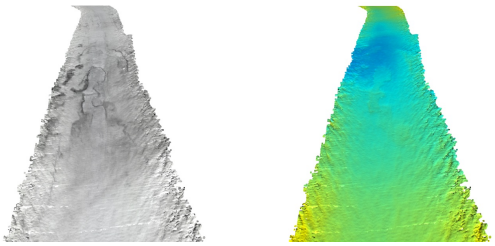


Fig. 12. Real-world road reconstrction and elevation visualization. Results are from Ben Guidarelli.

[18] Q.-Y. Zhou, J. Park, and V. Koltun, "Fast global registration," in *Eur. Conf. on Computer Vision (ECCV)*, 2016.

[19] R. B. Rusu, N. Blodow, and M. Beetz, "Fast point feature histograms (FPFH) for 3D registration," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, 2009.

[20] D. Geman and G. Reynolds, "Constrained restoration and the recovery of discontinuities," *IEEE Trans. Pattern Anal. Machine Intell.*, no. 3, pp. 367–383, 1992.

[21] M. Harris, "Optimizing parallel reduction in CUDA," 2007.

[22] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," in *SIGGRAPH*, vol. 21, no. 4. ACM, 1987, pp. 163–169.

[23] W. Dong, J. Shi, W. Tang, X. Wang, and H. Zha, "An efficient volumetric mesh representation for real-time scene reconstruction using spatial hashing," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, 2018.

[24] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 36, 2014.

[25] V. Garcia, E. Debreuve, and M. Barlaud, "Fast K nearest neighbor search using GPU," *IEEE Conf. on Computer Vision and Pattern Recognition Workshops*, 2008.

[26] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers, "A benchmark for the evaluation of RGB-D SLAM systems," in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2012.

[27] A. Handa, T. Whelan, J. McDonald, and A. Davison, "A benchmark for RGB-D visual odometry, 3D reconstruction and SLAM," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, 2014.

[28] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Real-time parallel hashing on the GPU," *ACM Trans. on Graphics*, vol. 28, no. 5, p. 154, 2009.