Onboard Real-time Dense Reconstruction of Large-scale Environments for UAV

Anurag Sai Vempati[1,2], Igor Gilitschenski[1], Juan Nieto[1], Paul Beardsley[2] and Roland Siegwart[1]

Abstract— **In this paper, we propose a GPU parallelized SLAM system capable of using photometric and inertial data together with depth data from an active RGB-D sensor to build accurate dense 3D maps of indoor environments. We describe several extensions to existing dense SLAM techniques that allow us to operate in real-time onboard memory constrained robotic platforms. Our primary contribution is a memory management algorithm that scales to large scenes without being limited by GPU memory resources. Moreover, by integrating a visual-inertial odometry system, we robustly track the camera pose even on an agile platform such as a quadrotor UAV. Our robust camera tracking framework can deal with fast camera motions and varying environments by relying on depth, color and inertial motion cues. Global consistency is achieved via regular checking for loop closures in conjunction with a pose graph, as a basis for corrective deformation of the 3D map. Our efficient SLAM system is capable of producing highly dense meshes up to 5mm resolution at rates close to 60Hz fully onboard a UAV. Experimental validations both in simulation and on a real-world platform, show that our approach is fast, more robust and more memory efficient than state-of-the-art techniques, while obtaining better or comparable accuracy.**

I. Introduction

Dense representations of the environment is a key for many robotics applications such as navigation, scene understanding, and scene manipulation. Research in dense methods for SLAM has recently grown with the advancements in computational power. This enables to implement real-time applications that use complete images, without the need for pre-processing such as feature detection and descriptor evaluation. Powerful Graphics Processing Units (GPU) are now widely available, allowing vision algorithms to harness the power of parallelization, thus enabling higher amounts of data to be processed in real-time. *KinectFusion* [1] is one such approach that is capable of building dense 3D maps in real-time using the commercially available Kinect[1] sensor.

Several challenges need to be addressed in order to make dense SLAM suitable for real-world applications. For instance, many existing dense SLAM systems do not scale to large scenes because they are constrained by GPU memory. In the context of this paper, we define large-scale scenes based on the surface area and/or volume of the scene scanned, rather than the length of the camera trajectory, since they are more relevant to the amount of memory used to store the resulting reconstruction. Another limitation is the inability to handle fast and abrupt camera motion, which is particularly important for agile drones or handheld devices. Finally, camera tracking may drift over time, resulting in inaccurate maps of the environment. Going beyond state-of-the-art approaches addressing some of these issues, we propose a framework that offers new levels of accuracy and robustness suitable for robotics applications.

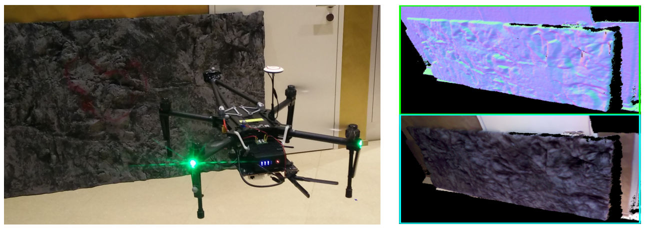


Fig. 1: UAV performing onboard real-time scan and the dense mesh being generated in the GUI. Our lightweight system is capable of generating sub-centimeter resolution meshes at rates upto 60Hz, fully onboard.

Overall, the contributions of our work can be summarized as follows:

- An efficient *Memory Management* pipeline that enables scalable highly-dense map building in real-time, while guaranteeing a fixed GPU memory footprint (Sec. IV-C).
- A new raycasting technique which generates maps of previously visited areas. This enables robust drift quantification during loop closures, without having to additionally store submaps or depth measurements from previous time instants (Sec. IV-D, IV-E).
- A method to robustly track the camera under fast camera motions or complex scene geometry by integrating visual-inertial odometry together with depth and photometric cues (Sec. IV-A, IV-B).

Our SLAM framework further involves a pose graph system capable of performing bundle adjustments on the dense map whenever loop closures are detected, thus resulting in globally optimal 3D maps (Sec. IV-F). The proposed system is lightweight and is capable of real-time performance on embedded platforms onboard commercially available quadrotor UAVs (Fig. 1). Our modular framework with a *Frontend* and *Backend* architecture is presented in Section III.

[1]Anurag Sai Vempati, Igor Gilitschenski, Juan Nieto and Roland Siegwart are with the Autonomous Systems Lab at ETH Zurich, Leonhardstrasse 21, 8092, Zurich, Switzerland. `avempati@ethz.ch`

[2]Anurag Sai Vempati and Paul Beardsley are with Disney Research Zurich, Stampfenbachstrasse 48, 8006, Zurich, Switzerland. `anurag.vempati@disneyresearch.com`

[1]`https://dev.windows.com/en-us/kinect`

II. Related Work

With advances in GPU architectures and GPGPU algorithms, vision and robotics communities have seen a rise in real-time dense mapping algorithms. DTAM [2] is one of the first SLAM systems that uses dense methods for scene reconstruction from an RGB camera in real-time using a parallelized framework based on a GPU. In contrast, we focus on the use of low-cost RGB-D sensors such as Microsoft Kinect and Intel RealSense[2] as they are widely available and readily provide accurate depth information.

A prominent example using this sensing modality is KinectFusion [1], which performs accurate real-time mapping of complex indoor scenes, harnessing the parallel computing power of a GPU. A single global implicit surface model of the observed scene is generated and rendered in real-time resulting in a highly-accurate reconstruction with a great level of detail. A 6DOF pose of the camera is estimated by performing coarse-to-fine Iterative Closest Point (ICP) algorithm for aligning the current depth scan with the global scene. Variants of KinectFusion involve ElasticFusion [3], which builds dense globally consistent surfel-based maps online in an incremental fashion, without needing pose graph optimization or any post-processing steps. Local surface loop closure optimizations are applied to detect overlapping mapped regions and non-rigidly deform the whole scene model to maintain global consistency. Though this method achieves high quality consistent maps, it is known to not scale well for larger datasets.

Dense SLAM methods are in general constrained by the memory available onboard a GPU, thus limiting the size of the area that can be mapped. Several methods in the past have tried to optimize the GPU memory usage to achieve scalable mapping of larger areas by moving memory in and out of a GPU. Kintinuous [4] addresses this problem by only keeping the local environment in the GPU and perform dynamic memory updates as the camera moves. They propose a moving TSDF volume method which involves a technique to virtually translate the TSDF grid and relocate the camera to the origin of the grid as it starts to move outside of it. All the voxels that fall out of the grid as a result of this operation are transferred out of the GPU. In [5], the authors further extended this to include pose graph optimization for non-rigid deformation of the dense map, enabling loop closures on large datasets. However, both these works [4][5] represent all the space using voxels, despite it being occupied or not. Our method on the other hand, only allocates memory for occupied space using the advantages of voxel hashing technique proposed in [6]. Further, [4][5] dump all the TSDF information soon as it exits the moving volume, making it necessary to regenerate it when a particular area is revisited, resulting in issues with redundancy in the map representation. Patch Volumes [7] try to combine accuracy of GPU based volumetric methods with global consistency of RGB-D SLAM systems. There, a dynamic patch swapping algorithm is proposed for swapping whole patches in and out of GPU memory to map larger areas. But, this results in unknown time lags depending on patch-size, thus compromising its real-time performance. Similarly, [8] tries to address this problem by building submaps which are actively swapped in and out of the GPU. However, their submap-based graph optimization only accounts for transformations between pairs of submaps and hence cannot correct for drifts if they occur within a submap. These problems do not arise in our method as it does not require storing and managing any additional submaps/patches of previously visited areas in the CPU, thus maintaining a lower memory footprint. In addition, unlike all these methods, we have an upper limit on the number of memory transfers in each iteration, thus ensuring the transfer delay is always bounded.

The goal of the present paper is to make dense mapping suitable for robotics applications by robustifying the system. Specially, it was found from our experiments that the works presented in [1][3], do not deal well with fast motions and scenes with clutter. On the other hand, unlike works that use an additional odometry stage for achieving robustness in camera tracking, such as [5], our approach does not require uninterrupted availability of depth data. Therefore we are not limited by the requirement that at least some objects in the mapped environment need to remain within the range of the depth sensor, thus limiting the possible applications.

III. System Overview

We choose a modular approach where the system architecture is divided into *Frontend* and *Backend* processing blocks. The *Frontend* involves all the main processing performed every time a new image from the sensor is received. The *Backend* involves all the processes running in the background asynchronously that assist *Frontend* operations. Such an architecture allows for a distributed operation where one or more UAVs could be running the *Frontend* and a base station platform could be running the computationally expensive *Backend*. Fig. 2 depicts a flowchart illustration of all the individual modules and the data being transferred between them. A detailed description of modules with significant contributions are further elaborated in Section IV.

A. Frontend

The *Frontend* consists of the *Main Processing Pipeline* which is heavily GPU parallelized. Our volumetric representation is based on a voxel grid of truncated signed distance function (TSDF) entries and we use parts of the framework from InfiniTAM [9], which extends KinectFusion by implementing voxel based hashing with efficient data structure handling and several optimizations in data allocation, data transfers, and raycasting. The *Main Processing Pipeline* consists of basic components originally proposed in KinectFusion with some crucial changes. *Data Pre-processing* performs data acquisition from the depth sensor and performs pre-processing involving conversion of disparity readings to depth point clouds, hole filling in depth data, finding pixel colors for depth measurements using calibration data and building image pyramid. The *Camera*

[2]`software.intel.com/en-us/realsense/r200camera`

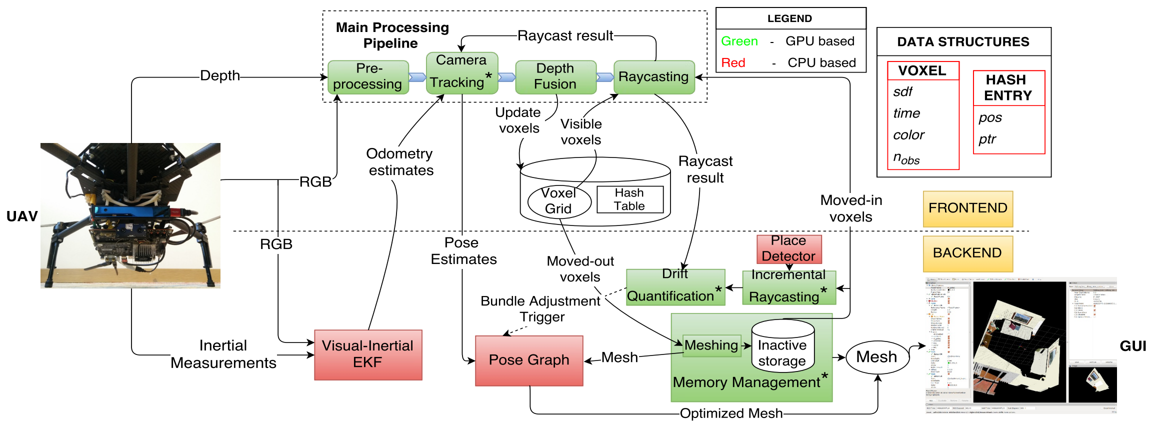


Fig. 2: Flowchart indicating individual modules and data transfers. Modules with major contributions are indicated with a $\star$.

Tracking module performs camera pose estimation using photometric, visual-inertial odometry estimates and depth data. *Depth Fusion* module performs data integration of new depth data into a locally stored voxel grid containing TSDF values. The *Raycasting* module shoots rays along the current camera viewing direction to evaluate zero crossing points in the TSDF voxel grid, which will be used to build surface maps for performing tracking in the next iteration.

In our framework, each voxel consists of an *sdf* value, *time* it was last updated at, its *color* and n_{obs} observations that are fused so far. Each hash table entry consists of a pointer *ptr* to a voxel grid entry and its position *pos* in global reference frame.

B. Backend

The *Backend* involves several crucial extensions to Kinect-Fusion. A *Visual-Inertial EKF* (Extended Kalman Filter) is used to obtain robust odometry estimates to assist the *Camera Tracking* module. The *Memory Management* module moves data between active memory (GPU) and inactive memory (CPU) and always maintains only a relevant section of the voxel grid within the active memory of GPU. It also involves a *Meshing* process that performs marching cubes [10] on moved-out voxel blocks before storing them in inactive memory. We propose a novel *Incremental Ray-casting* technique together with a *Place Detector* which incrementally builds a map from previous visit of a place. *Drift Quantification* is performed every few frames to detect smaller drifts in the pose estimates and a *Pose Graph* module involves the creation of a factor graph and graph optimization. Finally, a Graphical User Interface (*GUI*) is designed to visualize the camera trajectory and 3D map being built in real-time.

IV. Approach

In this section we describe, in detail, various stages of our pipeline that involve modules (please refer to Fig. 2) with our main contributions and how we differ in our design choices as compared with other earlier works.

A. Camera Tracking

We chose to implement a modular design for the *Camera Tracking* module in order to cope with data streams arriving at different frame rates and/or intermittent sensory data. Since our framework is ROS based, we use callbacks whenever sensory data is available, which allows us to incrementally update the camera pose without having to wait for other sensory data.

For each odometry measurement available from the EKF, we perform a *Odometry Update* which evaluates a differential pose change and updates camera pose as follows:

$$T_{g_c}^{c,t+1} = \left(\left(T_c^i \right)^{-1} T_{g_o}^{i,t+1} \left(T_{g_o}^{i,t} \right)^{-1} T_c^i \right) \times T_{g_c}^{c,t} \quad (1)$$

where, $T_{g_c}^{c,t}, T_{g_o}^{i,t}$ are camera and IMU pose at time t referenced with respect to their respective inertial frames g_c and g_o. The transformation T_c^i is the extrinsic calibration parameter describing misalignment between IMU and camera.

We further minimize two different metrics for better alignment of depth data with the scene, while also maintaining color consistency. These metrics are independently minimized as and when the depth or color data is available, in accordance with our modular approach. The first is a point-to-plane metric:

$$E_t^D(p_t) = \sum_{p_t} \| \{ (R_c^{g_c,t} p_t + t_c^{g_c,t}) - p_s \}^T N_s(p_s) \|_2 \quad (2)$$

where, p_t is a live depth measurement point at time t and p_s is a corresponding point on the surface with a normal value $N_s(p_s)$, obtained using projective data association during ray-casting stage. This metric is evaluated between current depth image and raycasted point map obtained from the TSDF voxel grid, and is minimized by employing an ICP based tracker. $T_c^{g_c,t} = [R_c^{g_c,t}, t_c^{g_c,t}]$ are initialized to the estimates from *Odometry Update* and an incremental transform is applied using a small angle assumption to get an updated transform. Linearizing around previous estimate allows for an iterative solution for minimizing Equation 2, a GPU parallelizable implementation of which is provided in [1]. We also employ a hierarchical approach based on image pyramid of varying resolutions, which allows for faster convergence.

To ensure color consistency between the current color image and the scene, we also minimize the following photometric error:

$$E_t^C(p_s) = \sum_{p_s} \|C_t\left(\pi\left(R_{g_c}^{c,t} p_s + t_{g_c}^{c,t}\right)\right) - C_s(p_s)\|_2 \quad (3)$$

which is evaluated for each scene point p_s obtained during ray-casting with color values $C_s(p_s)$ and projecting it into color image C_t at time t with projection matrix π. During our experiments, using the photometric error term was found to help in avoiding mis-alignments in scenes containing predominantly flat areas.

We further employ Mahalanobis distance based outlier rejection and the camera pose is reverted back to previous best estimate in case of an outlier.

B. Visual-Inertial EKF

We use the robust visual-inertial EKF system (ROVIO) described in [11], where inertial measurements from an IMU are used for filter propagation while multi-level feature patches in the image are tracked for performing filter updates. The odometry estimates from ROVIO are fed into the *Camera Tracking* module to perform odometry update step as mentioned in Section IV-A.

ROVIO has several parameters to be defined, making it difficult to be used out of the box for a new type of sensor. It was found that its performance is quite sensitive to the IMU used, since the prediction routine of the EKF is performed solely based on IMU readings. As we use two different visual-inertial sensors (refer to Section V) for our experiments, we had to tune the prediction parameters of EKF to fit the particular IMU. More specifically, we had to tune for covariance parameters related to the prediction of position, velocity, attitude along with the gyroscope and accelerometer biases. First, we estimate the IMU intrinsics using extended version of Kalibr [12]. The accelerometer and gyroscope noise parameters obtained through Kalibr are then used to obtain decent initial estimates for EKF covariance parameters. To further improve the parameters, we collected various datasets with different motion patterns within the motion capture area. A simple non-linear, local optimization script is then executed to find best fitting parameters for the EKF such that the RMS error on the camera trajectory is minimized. We used a derivative-free, bound-constrained BOBYQA algorithm [13] as implemented in the NLopt non-linear optimization package [14].

Having a high-frequency odometry system together with our modular *Camera Tracking* module allows continuous tracking despite sensory drop outs in RGB-D cameras.

C. Memory Management

In this section, we describe the voxel operations involved in GPU-CPU memory transfers. Fig. 3 shows an instant of the submap inside the GPU which is stored in the form of a voxel grid and a corresponding hash table. As the

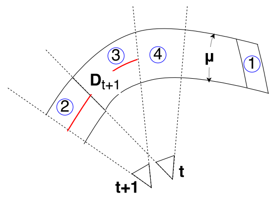


Fig. 3: Submap corresponding to the active memory and the regions (1-4) affected during each memory management iteration.

camera spans across the scene ($t \rightarrow t+1$), following actions are performed: (a) Voxels which have not been updated for more than t_{old} sec (Region 1) are moved out to the inactive memory. (b) Voxel locations belonging within the truncation bandwidth μ around the new depth measurement D_{t+1} (Region 2+3) are updated as follows: In the absence of a voxel at a required location, its memory is allocated in the active memory, else, the new depth measurement is integrated and that voxel's entry is updated in the *Depth Fusion* stage. (c) Inactive memory is searched for presence of previously stored voxels belonging to Region 2+3. In case of a hit, the required voxel entries are requested by the active memory. (d) Finally, the voxels moved in as a result of previous request (Region 3+4) are integrated into the active memory. We use transfer buffers as described in [9] for implementing voxel transfers between GPU and CPU.

D. Incremental Raycasting

In the previous Section IV-C, we mentioned how voxels that are earlier moved out to inactive memory are requested by the active memory when a place is revisited. In this section, we describe a novel raycasting technique that incrementally estimates the zero crossing of these moved-in voxels, thus making it possible to detect drift by comparing it against the current raycast output. Further details on drift detection and quantification are explained in Section IV-E.

Among the voxels being moved in, the ones with smaller SDF values (sdf) and the ones with higher number of fused observations in evaluating the SDF value (n_{obs}) are considered more reliable. So, we define voxel reliability measure $\mathcal{R}(v) \in [0, 1]$ for a voxel v as follows:

$$\mathcal{R}(v) = \alpha\left(\frac{n_{obs}}{n_{max}}\right) + \beta\left(1 - \frac{|sdf|}{\mu}\right) \quad (4)$$

where, n_{max} is the maximum number of observations after which no more observations are fused for evaluating the SDF value, and μ is the truncation bandwidth for the SDF values. α and β are weights for the individual reliability terms.

Each individual voxel v and its corresponding hash entry h can then be used to estimate a point $p_{0,v}$ on zero crossing surface using its SDF value as follows:

$$\vec{r} = \frac{h.pos - p_c}{\|h.pos - p_c\|}$$
$$p_{0,v} = h.pos + \{\vec{r} \times sdf\} \quad (5)$$

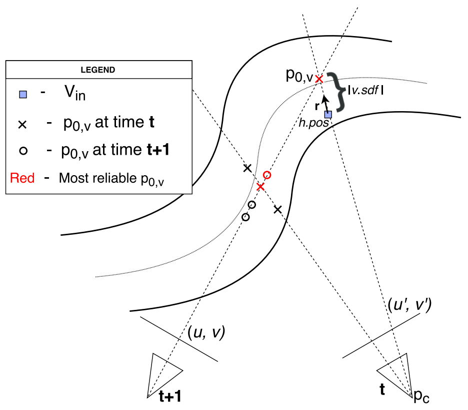


Fig. 4: Incremental raycast technique. Zero crossing of the TSDF voxel grid is incrementally estimated as the voxels are moved-in from the inactive memory.

For each voxel v moved in at time t, a raycast estimate $R_t^\star$ can be generated by forward projecting the corresponding $p_{0,v}$ onto the camera's image plane. In the case where multiple $p_{0,v}$s project into the same pixel location, one with highest reliability score is stored in $R^\star$ along with the corresponding score. As the time passes, more voxels are moved in and the raycast estimate can be further refined incrementally. As depicted in Fig. 4, by picking the most reliable zero-crossing entry among the newly moved-in voxels at time $t+1$ and remapped raycast estimates $R_{t\rightarrow t+1}^\star$ from time t, the new raycast estimate $R_{t+1}^\star$ can be obtained as follows:

$$R_{t\rightarrow t+1}^\star(u,v) = \max_{\mathcal{R}(.)} \left\{ R_t^\star(\acute{u},\acute{v}) \ni \mathcal{F}_{t+1}\left(R_t^\star(\acute{u},\acute{v})\right) = (u,v) \right\}$$
$$R_{t+1}^\star(u,v) = \max_{\mathcal{R}(.)} \left\{ R_{t\rightarrow t+1}^\star(u,v) \cup \{p_{0,v} \ni v\epsilon V_{in}\} \right\} \tag{6}$$

where, $\mathcal{R}(.)$ is the reliability measure associated with a zero-crossing/raycast estimate and $\mathcal{F}_{t+1}$ is the forward projection operation for time $t+1$.

It was observed in our experiments that the remapping stage results in sparsification due to multiple raycast entries falling into same pixel location during forward projection operation. By initializing a random pixel entry neighboring to the collision pixel location with the most reliable raycast estimate among the colliding candidates, one can quickly obtain dense raycast maps. The reliability score associated with the newly initialized entry is same as the score of most reliable candidate, but scaled by a fixed factor.

E. Drift Quantification

Despite having an accurate *Camera Tracking* pipeline, over longer sequences the system will drift in the pose. This can cause problems in *Depth Fusion* stage, specially when a particular section of the scene is re-visited after some time. A slight drift in pose can result in wrongly fusing new depth values within the voxel grid resulting in a deformed map. This section explains how a raycast of the scene from previous time visit (Section IV-D) can be used to accurately quantify drift in the camera pose.

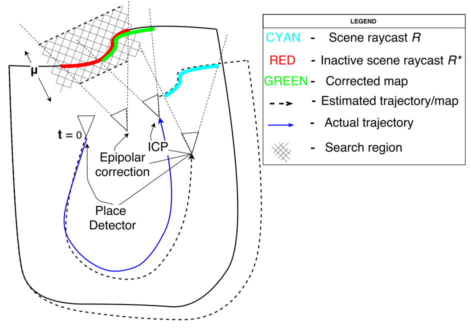


Fig. 5: Drift quantification using a *Place Detector*

Incremental Raycast module assumes that the previously visited part of the voxel grid is within a truncation bandwidth (μ) search region around the current depth measurement. Though this assumption is valid in cases of smaller drifts, it does not hold true for larger drifts such as in Fig. 5. Increasing the search region could solve this problem but comes with a heavy computational price. To address this problem, we use a *Place Detector* based on a binary bag-of-words approach [15] to provide a candidate image from previous time instants that resembles closest to the scene being observed. We first evaluate the essential matrix between the current image and this candidate image, using the matched feature pairs. Then, by performing Singular Value Decomposition (SVD) of this essential matrix, relative camera pose between these two instances can be obtained [16]. This relative pose is obtained to the true scale by ensuring that the matched features are triangulated to their true depth, which is known from the depth image. An "Epipolar Correction" is now performed on current camera pose such that it is consistent with this relative pose. The corrected pose is then used to generate the raycast map $R^\star$ (Section IV-D).

The drift is accurately quantified by performing an ICP routine between R and $R^\star$. If considerable drift is detected, a bundle adjustment is triggered within the *Pose Graph* stage (explained in Section IV-F). As can be seen in the Fig. 5, the resulting $R^\star$ (Red) overlaps very well with the current scene's raycast R (Cyan) after the correction (Green).

F. Global Consistency

We use GTSAM [17] to perform global optimization on the 3D maps, after a bundle adjustment trigger is set off by *Drift Quantification* stage (Section IV-E). Pose estimates from the *Camera Tracking* module are used to link consecutive nodes of the graph, while measurements are fed in the form of a triangular mesh from *Meshing* stage. We use unary factors to impose priors on the graph and a bundle adjustment is performed using Levenberg-Marquardt optimization. To summarize, Algorithm 1 describes how globally consistent mapping is achieved, over large scale, using the routines mentioned in previous sections.

Algorithm 1 Large Scale Mapping Algorithm

Input: I_t is the current image. V_{in} and V_{out} are list of voxels moved-in and moved-out respectively. H_{in} are hash entries and $R^\star$ ray-casting output corresponding to V_{in} generated using viewpoint $T_{R^\star}$. H_{out} are hash entries and $\mathcal{M}$ is mesh output corresponding to V_{out}.
Output: Corrected trajectory $T^\star$ and corrected 3D map $\mathcal{M}^\star$

```
procedure LargeScaleMapping
    $\mathcal{M} \leftarrow$ toMesh($V_{out}, H_{out}$)                          ▷ Sec. IV-C
    if isPlaceRevisited($I_t$) then                          ▷ Sec. IV-E
        $T_{R^\star} \leftarrow$ queryBoWDataBase($I_t$)
    else
        $T_{R^\star} \leftarrow T_{g_c}^{c,t}$
    end if
    $R_t^\star \leftarrow$ raycast($T_{R^\star}, R_{t-1}^\star, V_{in}, H_{in}$)               ▷ Sec. IV-D
    $T_{drift} \leftarrow$ quantifyDrift($R_t^\star, R_t$)                    ▷ Sec. IV-E
    if $T_{drift} \neq I_{4\times4}$ then                                   ▷ Sec. IV-F
        $T^\star \leftarrow$ performBundleAdjustment()
        $\mathcal{M}^\star \leftarrow$ deformMesh($\mathcal{M}, T^\star$)
    end if
end procedure
```

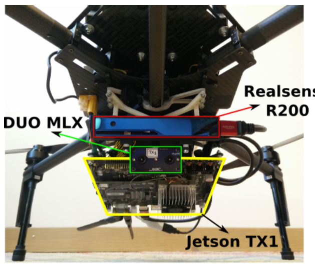


Fig. 6: DJI M100 UAV used for real-world experiments.

V. Experimental Setup

We perform thorough evaluations both in simulation and also in real-world. Our real-world experiments include indoor datasets collected both with a hand-held sensor-rig and a modified DJI Matrice M100 UAV[3] (Fig. 6). The hand-held unit's sensor-rig comprises of a Visual-Inertial sensor and a Kinect for depth sensing. Our DJI UAV is equipped with a sensor-rig comprising of a DUO-MLX visual-inertial sensor[4] and a Intel Realsense R200 camera for depth sensing. The UAV further has a NVIDIA Jetson TX1 embedded GPU platform[5] onboard for real-time computation. Simulator datasets are collected with a UAV in RotorS gazebo simulation platform [18]. Ground truth for camera pose is available from OptiTrack motion capture system[6] providing sub-millimeter accuracy. To measure the accuracy of reconstructed meshes, we compare our optimized meshes against groundtruth scans obtained from Leica Multistation MS50[7] laser scanner providing sub-millimeter precise scene structure.

VI. Results

In this section, we illustrate several experiments with results from our pipeline in terms of scalability to large scenes, computational efficiency, camera tracking accuracy and scene reconstruction quality. We provide quantitative results for five different datasets. We had to use custom datasets since there were not any open source RGB-D datasets with IMU measurements and groundtruth scene for real-world scenario, with the kind of variation in camera motion and scene size needed for our experiments. Though, note that we use the synthetic living room scene from ICL-NUIM RGB-D dataset [19] for our simulator environment, but scaled by a factor of 2 in size. Our dataset collection includes: (1) A *Loopy* dataset with a very loopy camera motion, thus resulting in a lot of small loop closures happening. (2) A *Fast* dataset with high translational and rotational velocities of camera. (3) A *Long* dataset with camera motion spanning very long distances resulting in meshes with tens of millions of vertices. (4) A *RotorS* dataset collected with a UAV simulated in RotorS [18] simulation environment. (5) A *UAV* dataset collected with our custom UAV in an indoor space. Some relevant statistics for each dataset are presented in Table I.

To benchmark our results against earlier works, we evaluated the performance of InfiniTAM [9] and ElasticFusion [3] on our datasets. We don't use IMU-aided version of InfiniTAM in our evaluations, since the software only supports pose inputs in the form of quaternions obtained from an external estimator, available for example in a tablet. Whereas, our system uses raw inertial measurements, i.e, the linear accelerations and angular velocities from the IMU. Our framework is shown to perform comparable or better for all the above datasets as compared with the benchmarks.

A. Scalability

Our efficient *Memory Management* module enables scalable mapping over longer sequences. To monitor the GPU memory footprint, we collected a minute long dataset and plotted the memory consumption. Fig. 7 shows the memory usage on the GPU (in MB) as the camera explores new areas. As can be seen, in the implementation without the memory management, more and more voxels had to be allocated as new area is explored. As a result very soon the memory usage hits the maximum limit of 1200 MB. Whereas, in the implementation with proposed memory management algorithm, the memory footprint has always stayed within 300-600 MB and only used half of the allocated memory despite more new area being covered. This proves that our technique can scale very well to longer sequences covering large areas, while maintaining a bounded memory footprint on the GPU.

Consequently, our system is capable of generating highly dense meshes of about 20m $\times$ 10m $\times$ 3m large indoor spaces at resolution of 1cm resulting in a mesh containing several tens of millions of vertices, as shown in Fig. 9b.

B. Timing

Timing results for our SLAM system are shown in Fig. 8. From our experiments, run time performance is found to be affected by sensors used, computational platform involved and also most importantly on the voxel size used. Results for **Simulator** and **Hand-held rig** are computed on a Desktop

[3]www.dji.com/matrice100
[4]duo3d.com/product/duo-minilx-lv1
[5]www.nvidia.com/object/jetson-tx1-dev-kit.html
[6]www.optitrack.com/
[7]leica-geosystems.com

Dataset	Source	Duration	# frames	Distance covered (total / end-to-end)	Average speed (translational / rotational)	Volume covered
Loopy	Hand-held rig	185 s	2784	42.379 m / 0.808 m	0.1650 m/s / 14.209 $°/s$	149.649 m^3
Fast	Hand-held rig	29.3 s	427	16.734 m / 0.893 m	0.4392 m/s / 35.484 $°/s$	77.469 m^3
Long†	Hand-held rig	84 s	1246	50.4 m / 6.2 m	0.4054 m/s / 20.895 $°/s$	549.828 m^3
RotorS	Simulator	58.8 s	1762	20.731 m / 0.512 m	0.3002 m/s / 14.235 $°/s$	1214.933 m^3
UAV	DJI M100	55.4 s	3308	9.46 m / 0.84 m	0.1732 m/s / 12.131 $°/s$	56.549 m^3

TABLE I: Dataset statistics. Note the variation in platform used, camera speeds, sequence length and volume covered.

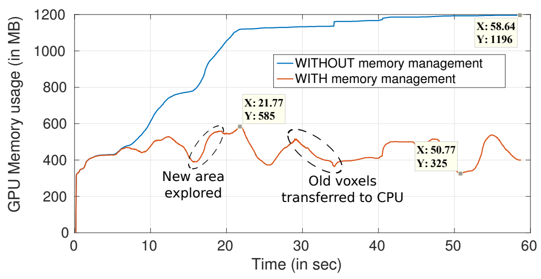


Fig. 7: GPU memory usage with and without *Memory Management*. Note that, with memory management, less than half the allocated GPU memory is used at all times.

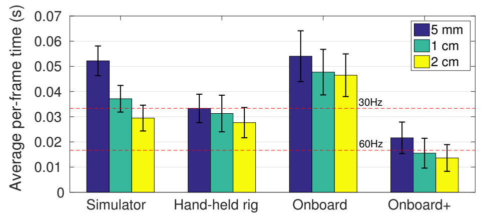


Fig. 8: Average per-frame timing results on different platforms with varying voxel sizes.

PC running Ubuntu 14.04 with an Intel i7 PCU running at 3.4GHz, 39GB RAM and an NVIDIA GeForce GTX 980 graphics card with 4GB memory. The difference in timing between these two setups was due to the fact that **Simulator** has a wide-angle depth camera which resulted in more voxels being viewed in each frame, thus resulting in higher computation times. Results for **Onboard** and **Onboard+** are computed on a Jetson TX1 embedded platform running Ubuntu 14.04, with RGB-D data available at 60Hz. For the **Onboard** setup, running the original system as is doesn't achieve desired run-time efficiency. So, to improve computational speeds, in **Onboard+** setup, we suppress the heavy visualization routines used in the GUI, since raycasting is one of the most expensive processes in the pipeline. Only the visible part of scene is raycasted for performing camera to model tracking. We further speed-up the performance by skipping ICP routine every few frames and only employ odometry and color based tracking for every frame. In this setup we only raycast the scene when ICP routine is performed otherwise just forward render the raycast from previous iteration. Having a modular tracking pipeline allows us to easily implement such changes. By saving time on raycasting and also the ICP routine, we achieve real-time performance at frame rates close to 60Hz. Moreover, no significant deterioration - both quantitatively and qualitatively - of tracking and mesh accuracy are observed due to these changes.

C. Camera Tracking

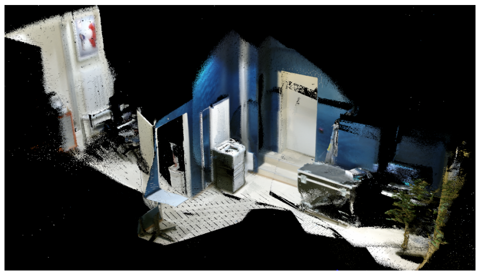
To evaluate the performance of camera tracking pipeline, we compared the estimated camera poses against the ground truth trajectories. Calibration between sensors is performed using Kalibr [20]. Calibration between visual and motion capture marker frame along with alignment of their respective inertial reference frames is done offline using batch estimation as described in [21]. Tracking results for individual datasets are provided in Table II. As it can be noted, the tracking accuracy only barely degrades despite translational and rotational velocities in *Fast* dataset being more than two times higher as compared with *Loopy* dataset.

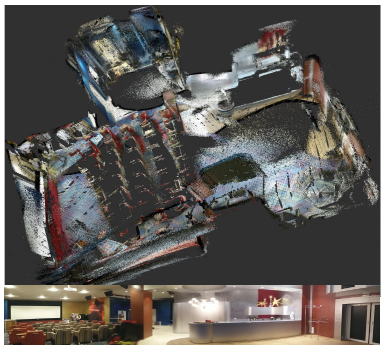
D. Meshing

To measure the accuracy of reconstructed meshes, we compared our optimized 3D meshes against groundtruth obtained from a high precision laser scanner. CloudCompare [22] is used to register the point clouds and evaluate alignment errors. Some sample meshes produced from various datasets can be seen in Fig. 9. Mesh accuracy results for individual datasets are provided in Table II, showing our superior reconstruction quality compared to benchmark methods.

VII. Conclusion

In this work, we described a framework for GPU accelerated dense SLAM using RGB-D+Inertial sensors with robust camera tracking and globally optimal map generation, that can scale well to large areas. We particularly showed satisfactory performance for different datasets involving variations in camera motions and size of the mapped area, both in simulation and real-world scenarios. Our results have shown that a robust camera tracking pipeline is crucial for using dense SLAM methods on agile robotic platforms such as a quadrotor UAV. Moreover, we have proven that its possible to achieve highly accurate mapping at very high frame rates even on computationally constrained platforms. In future work, we plan to test our framework for much longer sequences spanning even larger areas.


(a) *Loopy* dataset


(b) *Long* dataset

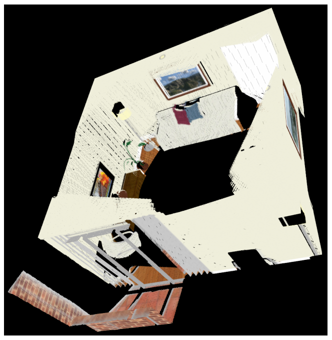

(c) *RotorS* dataset

Fig. 9: Meshing results from various datasets.

Dataset	Method	Tracking accuracy	Mesh accuracy (mean ± std)	Avg. time per frame (On PC)
Loopy	Ours (1cm voxel)	**11.93 cm**	**24.06 ± 14.97 mm**	28.18 ms (with GUI)
	ElasticFusion	12.16 cm	25.79 ± 32.45 mm	28.33 (Core) + 11.03 (GUI) ms
	InfiniTAM	29.17 cm*	41.16 ± 25.83 mm	**23.61 ms** (with GUI)
Fast	Ours (1cm voxel)	**12.14 cm**	**24.98 ± 15.46 mm**	33.54 ms (with GUI)
	ElasticFusion	23.88 cm*	50.42 ± 64.70 mm	24.92 (Core) + 9.44 (GUI) ms
	InfiniTAM	84.41 cm*	highly distorted	**28.10 ms** (with GUI)
RotorS	Ours (1cm voxel)	**3.08 cm**	**5.25 ± 4.99 mm**†	36.01 ms (with GUI)
	ElasticFusion	3.37 cm	**6.41 ± 4.80 mm**	28.42 (Core) + 13.32 (GUI) ms
	InfiniTAM	3.21 cm*	5.34 ± 5.23 mm	**30.46 ms** (with GUI)

TABLE II: Tracking, reconstruction and timing results for some datasets. Compared to benchmark results, our results show significant improvement in accuracy at comparable frame rates. (*- Tracking fails halfway through the dataset. † - While our system achieved better mean, ElasticFusion achieved better standard deviation in RMS error.)

REFERENCES

[1] 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 *Mixed and augmented reality (ISMAR), 2011 10th IEEE international symposium on*. IEEE, 2011, pp. 127–136.

[2] R. A. Newcombe, S. J. Lovegrove, and A. J. Davison, "Dtam: Dense tracking and mapping in real-time," in *2011 international conference on computer vision*. IEEE, 2011, pp. 2320–2327.

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

[4] T. Whelan, M. Kaess, M. Fallon, H. Johannsson, J. Leonard, and J. McDonald, "Kintinuous: Spatially extended kinectfusion," 2012.

[5] T. Whelan, M. Kaess, J. J. Leonard, and J. McDonald, "Deformation-based loop closure for large scale dense rgb-d slam," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2013, pp. 548–555.

[6] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger, "Real-time 3d reconstruction at scale using voxel hashing," *ACM Transactions on Graphics (TOG)*, 2013.

[7] P. Henry, D. Fox, A. Bhowmik, and R. Mongia, "Patch volumes: Segmentation-based consistent mapping with rgb-d cameras," in *3DTV-Conference, 2013 International Conference on*. IEEE, 2013, pp. 398–405.

[8] O. Kähler, V. A. Prisacariu, and D. W. Murray, "Real-time large-scale dense 3d reconstruction with loop closure," in *European Conference on Computer Vision*. Springer, 2016, pp. 500–516.

[9] O. Kahler, P. V. Adrian, R. C. Yuheng, X. Sun, P. Torr, and D. Murray, "Very high frame rate volumetric integration of depth images on mobile devices." *IEEE transactions on visualization and computer graphics*, vol. 21, no. 11, pp. 1241–1250, 2015.

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

[11] M. Bloesch, S. Omari, M. Hutter, and R. Siegwart, "Robust visual inertial odometry using a direct ekf-based approach," in *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*. IEEE, 2015, pp. 298–304.

[12] J. Rehder, J. Nikolic, T. Schneider, T. Hinzmann, and R. Siegwart, "Extending kalibr: Calibrating the extrinsics of multiple imus and of individual axes," in *Robotics and Automation (ICRA), 2016 IEEE International Conference on*. IEEE, 2016, pp. 4304–4311.

[13] M. J. Powell, "The bobyqa algorithm for bound constrained optimization without derivatives," *Cambridge NA Report NA2009/06, University of Cambridge, Cambridge*, 2009.

[14] S. G. Johnson, "The NLopt nonlinear-optimization package," http://ab-initio.mit.edu/nlopt, 2014, [Online; accessed 20-Feb-2017].

[15] D. Gálvez-López and J. D. Tardós, "Bags of binary words for fast place recognition in image sequences," *IEEE Transactions on Robotics*, vol. 28, no. 5, pp. 1188–1197, October 2012.

[16] R. Hartley and A. Zisserman, "Multiple view geometry in computer vision," 2000.

[17] F. Dellaert, "Factor graphs and gtsam: A hands-on introduction," 2012.

[18] F. Furrer, M. Burri, M. Achtelik, and R. Siegwart, *Robot Operating System (ROS): The Complete Reference (Volume 1)*. Cham: Springer International Publishing, 2016, ch. RotorS—A Modular Gazebo MAV Simulator Framework, pp. 595–625. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-26054-9_23

[19] 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*, Hong Kong, China, May 2014.

[20] P. Furgale, J. Rehder, and R. Siegwart, "Unified temporal and spatial calibration for multi-sensor systems," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2013, pp. 1280–1286.

[21] M. Burri, J. Nikolic, P. Gohl, T. Schneider, J. Rehder, S. Omari, M. W. Achtelik, and R. Siegwart, "The euroc micro aerial vehicle datasets," *The International Journal of Robotics Research*, p. 0278364915620033, 2016.

[22] D. Girardeau-Montaut, "Cloud Compare," http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/, 2015, [Online; accessed 10-July-2016].