Edge-SLAM: Edge-Assisted Visual Simultaneous Localization and Mapping

Ali J. Ben Ali
alijmabe@buffalo.edu
University at Buffalo
Buffalo, New York

Zakieh Sadat Hashemifar
zakiehsa@buffalo.edu
University at Buffalo
Buffalo, New York

Karthik Dantu
kdantu@buffalo.edu
University at Buffalo
Buffalo, New York

ABSTRACT

Localization in urban environments is becoming increasingly important and used in tools such as ARCore [11], ARKit [27] and others. One popular mechanism to achieve accurate indoor localization as well as a map of the space is using Visual Simultaneous Localization and Mapping (Visual-SLAM). However, Visual-SLAM is known to be resource-intensive in memory and processing time. Further, some of the operations grow in complexity over time, making it challenging to run on mobile devices continuously. Edge computing provides additional compute and memory resources to mobile devices to allow offloading of some tasks without the large latencies seen when offloading to the cloud. In this paper, we present Edge-SLAM, a system that uses edge computing resources to offload parts of Visual-SLAM. We use ORB-SLAM2 as a prototypical Visual-SLAM system and modify it to a split architecture between the edge and the mobile device. We keep the tracking computation on the mobile device and move the rest of the computation, i.e., local mapping and loop closure, to the edge. We describe the design choices in this effort and implement them in our prototype. Our results show that our split architecture can allow the functioning of the Visual-SLAM system long-term with limited resources without affecting the accuracy of operation. It also keeps the computation and memory cost on the mobile device constant which would allow for deployment of other end applications that use Visual-SLAM.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; **Client-server architectures**; • **Human-centered computing → Ubiquitous and mobile computing**; • **Computing methodologies → Vision for robotics**.

KEYWORDS

visual simultaneous localization and mapping, edge computing, split architecture, mobile systems, localization, mapping

ACM Reference Format:
Ali J. Ben Ali, Zakieh Sadat Hashemifar, and Karthik Dantu. 2020. Edge-SLAM: Edge-Assisted Visual Simultaneous Localization and Mapping. In *The 18th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '20), June 15–19, 2020, Toronto, ON, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3386901.3389033

MobiSys '20, June 15–19, 2020, Toronto, ON, Canada

ACM ISBN 978-1-4503-7954-0/20/06…$15.00
https://doi.org/10.1145/3386901.3389033

1 INTRODUCTION

Advances in sensing, computing, communication and actuation are bringing in a new set of mobile devices into our daily lives. Service robots operate in our homes cleaning our spaces and delivering condiments in hotels. Augmented reality apps on smart phones allow us to navigate in indoor environments, provide visualizations of spatial reconfigurations without actually doing it, or play games in the real world by augmenting it with virtual objects. Augmented reality glasses are used for collaboration across the globe. There are many more envisioned applications, including better seamlessness via mixed reality as well as telepresence using robots. Most of these applications rely on sensing spatial context, in particular spatial localization and place recognition indoors in GPS-denied scenarios.

Spatial sensing has been a research topic for several decades. Depending on the application, there are several modalities of spatial sensing. Examples include (i) place recognition which takes a sensor snapshot (an image from a camera, for example) of a location and matches it with known locations from prior measurements, (ii) tracking or estimation of the path followed by the mobile device from a starting point e.g. odometry, and (iii) localization which is the absolute positioning of a mobile device with respect to known landmarks. Each of these classes of sensing is useful for various applications and has tradeoffs in terms of computational complexity as well as utility. More recently, Simultaneous Localization and Mapping (SLAM) has evolved as a class of algorithms useful for accurate spatial context. It is the process of localizing a mobile device with respect to an absolute coordinate system as well as mapping the traversed space with respect to the same coordinate system. In particular, there has been much recent interest in using visual sensing (cameras, depth sensors, LiDARs) for SLAM leading to several Visual-SLAM algorithms.

Typical Visual-SLAM algorithms perform three main tasks. First, as the mobile device is moving, the algorithm performs a frame-frame alignment. This is the process of relating the pose of the mobile device that captured frame (image) *k* with the pose when capturing frame *k+1*. Usually, this is achieved by detecting features in each frame and finding feature correspondence between the two frames. The second step is to perform map adjustments locally. This step involves adjusting the frame-frame alignment performed in step 1 and identifying "keyframes" or frames of significance to be used in step 3. Finally, step 3 is loop closure, or the ability of the algorithm to identify when the mobile device is back at a location that it has previously visited. This step requires the algorithm to compare the new frame with all previous frames to identify matches.

A challenge with this task is the growing complexity of this task as the map grows. A typical solution to alleviate this problem is the use of keyframes identified in step 2 for these comparisons, and not compare the new frame with all previous frames. Other algorithms limit the number of comparisons to a subset of frames by various methods [15] such as the use of a short-term and long-term memory [33], or clustering using other sensing [22]. However, most of these solutions perform a tradeoff of accuracy to computational complexity that leads to mixed results.

Recently, there has been much excitement in edge computing architectures [37, 46, 47, 49]. Such an architecture advocates for the use of edge computing resources, typically relatively local to the mobile device and one hop over the local network away, to alleviate some of the computational tasks on mobile devices. In this work, we use edge computing resources to improve Visual-SLAM. To this end, we make the following contributions:

- We take ORB-SLAM2 [41], a popular Visual-SLAM system, and adapt it to the edge computing architecture. Our system is called Edge-SLAM.
- To this end, we de-couple the tracking and local mapping processes, thereby improving local mapping and loop closure efficiency without compromising the functionality of the tracking module.
- We evaluate Edge-SLAM on two different mobile devices using our datasets as well as benchmark datasets such as TUM [26].
- We open-source our Edge-SLAM implementation[1] allowing other practitioners to compare with our system.

Our results show that Edge-SLAM architecture is a good way to distribute Visual-SLAM computation between the edge and the mobile device. In addition to performance, there are several additional benefits in deploying Visual-SLAM in the edge computing paradigm such as control of map complexity, privacy, concurrency as well as reasoning with dynamics. We hope to study these ideas in future work.

2 RELATED WORK

The area of edge computing has been the topic of research for the last decade [37, 46, 47, 49]. It proposes a paradigm with sizeable computing and storage resources placed at the edges of the Internet closer to mobile and IoT devices that generate a lot of data. The idea is to utilize computing and storage closer to the sensors to improve processing latency while not burdening the resource-scarce devices.

There has been some work on offloading tasks from mobile devices to the edge/cloud previously. MAUI [10] and CloneCloud [8] perform cloud offloading of tasks at various granularities. MARVEL [7], VisualPrint [30] and [36] present application-specific techniques for offloading to the edge or the cloud. These papers work on decreasing offload latency or masking it from the end user in time-sensitive applications.

Simultaneous localization and mapping (SLAM) has been a topic of research in robotics and mobile systems for several decades [5, 13, 50]. Initial research focused on depth sensors such as sonar [17] and 2-D LiDAR [9, 12]. Other sensors such as Wi-Fi signal strength have been used for SLAM as well [18, 25, 39].

[1] http://droneslab.github.io/edgeslam

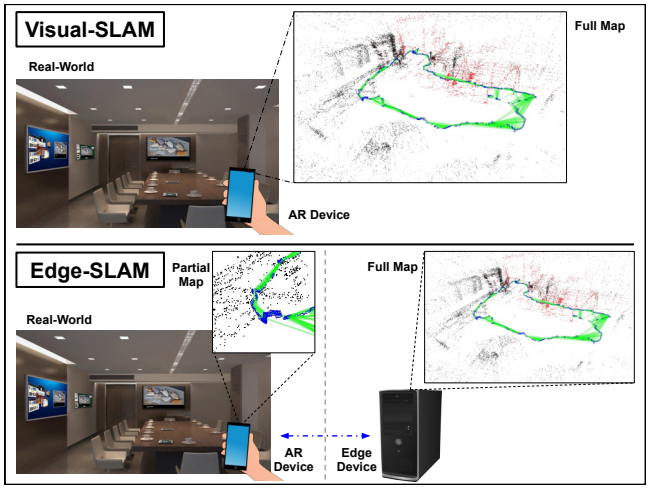

Figure 1: Visual-SLAM vs. Edge-SLAM. An augmented reality device running Visual-SLAM (top), and an augmented reality device running Edge-SLAM in collaboration with an edge device in the environment (bottom) [1–3, 26, 41]

Visual-SLAM has grown rapidly in the last decade [16, 24]. This includes the use of RGB cameras, RGB-D cameras, and LiDAR sensors. Systems such as PTAM [31], DTAM [43], LSD-SLAM [14] used monocular cameras for SLAM. A recent trend has been the use of color images with depth images. Some of more well-known Visual-SLAM examples include RGBD-SLAM [16], RTAB-Map [34], and ORB-SLAM [40, 41]. They build on initial work from systems such as Kinect Fusion [42], and Kintinuous [51] that first used RGB-D sensors for 3-D modeling of environments. Current trends also improve on basic Visual-SLAM by reasoning about semantics [21, 23] as well as object permanence in maps [20].

More recently, there is increased interest in the use of multiple sensors to perform SLAM. Several recent works combine Wi-Fi with visual sensing for improved SLAM. In [29], they model Wi-Fi signal strength using a Gaussian process and use it for finding an initial seed estimate of the robot's location which is then refined with RGB-D data. [44] utilizes a training phase for Wi-Fi modeling and then applies particle filters for fusing different sensors. [22] provided a general way to integrate wireless signal strength from Wi-Fi APs to Visual-SLAM algorithms. [4] uses a Wi-Fi map to merge multiple visual maps from multiple agents.

Collaborative SLAM has been explored in recent works through combining edge/cloud computing with SLAM systems in different ways. [48] built a collaborative monocular SLAM on top of ORB-SLAM2 [41] for Unmanned Aerial Vehicles (UAVs). The system runs a smaller version of ORB-SLAM2 (tracking thread and local mapping thread) on every UAV, to maintain and optimize a limited local map independently, and runs place recognition and map fusion on a centralized server, to merge and optimize the UAVs local maps into a global map. This study focuses on enabling collaborative SLAM on multiple UAVs. Whereas Edge-SLAM addresses the increasing resource usage (compute, storage, etc.) of Visual-SLAM on mobile devices by splitting the Visual-SLAM pipeline between a mobile device and an edge device. [19] presents a mapping framework for

Micro Arial Vehicles (MAVs). It consists of one-way communication between multiple MAVs and a server. Every MAV extracts features, estimates relative-motion, and then sends the information to the server to build a separate map and detect loops, as well as merge the MAVs maps. This framework does not maintain a local map on the MAVs and runs the SLAM pipeline on the server. In Edge-SLAM, the system maintains a local and global map by splitting Visual-SLAM pipeline between a mobile device and an edge device. [45] describes C2TAM which is a collaborative SLAM framework that is built on top of PTAM [31]. This system keeps PTAM's tracking thread on the client and moves PTAM's mapping thread to the cloud. The client sends new keyframes to the server, while the server sends the full map to the client after every optimization. Such a mechanism has the potential of causing the client to run out of memory and generate increasing network traffic as the map size gets bigger. Edge-SLAM keeps memory and network usage under control by only maintaining a local map on the client instead of a global map. In [35], the authors present CORB-SLAM, which is a multi-robot SLAM system built on top of ORB-SLAM2 [41]. CORB-SLAM runs on multiple robots and a centralized server. Every robot runs an instance of ORB-SLAM2 to build a map of its environment and sends it to the server. The server merges the maps received from robots and feed the complete merged map back to every robot. Sending back the full map to the client can cause the same memory and network issues mentioned for C2TAM. Collaborative SLAM studies might have some overlap with what we do; however, there are differences between the approaches as well as the goals. Most collaborative SLAM works focus on building an accurate joint global map from smaller maps built by individual robots. Their architecture has no computation offloading; in these systems, multiple agents run independently to map an area. Therefore, they will likely not be comparable in resource use to Edge-SLAM. Building a global map from smaller maps has other sources of error beyond what Edge-SLAM does. Thus, comparing Edge-SLAM with these systems would not be an apples-to-apples comparison of the functioning of the two pipelines.

A recent study [52] addressed offloading Visual-SLAM to the edge. In this study, the authors propose an edge-assisted monocular SLAM system built on top of ORB-SLAM [40]. At a high level, their goal is similar to our work. However, examining it closely reveals significant differences in design and implementation. The authors made offloading decisions by looking at the internal pieces of ORB-SLAM modules and not by looking at each module as one piece. Further, the authors incorporated a semantic segmentation algorithm into their system for improved accuracy. Unlike us, the focus of this study is not on resource constraints on mobile devices. Correspondingly, their design does not address relocalization, and their study does not measure resource (CPU, memory) usage or overhead of synchronizing the edge and mobile devices. Further, incorporating semantic segmentation makes their design harder to generalize across other Visual-SLAM systems. In contrast, Edge-SLAM is built on top of ORB-SLAM2 [41] (improved version of ORB-SLAM) and incorporates all aspects of Visual-SLAM including relocalization as well as ability to work with monocular, stereo and RGB-D cameras. Our design and implementation focus on resource usage on the mobile device, and our work extensively evaluates these aspects of the implementation.

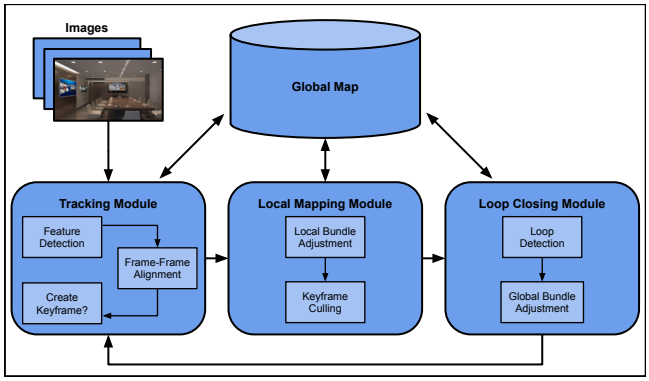

Figure 2: Architecture of a typical Visual-SLAM system [2]

3 SYSTEM DESIGN

3.1 Overview of a Typical Visual-SLAM System

Shown in Figure 2 on page 3 is an architecture diagram of a typical feature-based Visual-SLAM system. Several SLAM systems adhere roughly to this architecture including PTAM [31], LSD-SLAM [14], ORB-SLAM [40] and ORB-SLAM2 [41]. The input to a typical Visual-SLAM system are series of images (aka frames) captured from a camera. While we describe this generic system as one that accepts regular images (RGB), many SLAM systems are capable of accepting stereo images, depth images as well as color and depth images together. Most Visual-SLAM systems have the following three components:

Tracking: The tracking module detects features in the incoming image (frame). Typical features can be SIFT, SURF, ORB or corners. The tracking module then uses these features to find correspondences with a previous reference image (also called keyframe in many cases). Based on the correspondences in features between the two frames, it calculates the relative odometry (labeled frame-frame alignment) between the reference keyframe and the current frame. The tracking module then determines if this frame should be added as a keyframe to the map based on a set of criteria such as number of feature matches. If it decides to add a keyframe, it passes the current frame to the local mapping module.

Local Mapping: If tracking deemed the current frame to be a new keyframe, the local mapping module is invoked. This module creates correspondences between the new keyframe and other keyframes in the map. It then performs local bundle adjustment; a process of refining the relative coordinates of where the images were taken given the detected common features between keyframes. The bundle adjustment is local because it limits the reasoning to keyframes with common features.

Loop Closure: Every so often (frequency depends on the particular algorithm), the SLAM system runs the loop closure procedure. Conceptually, this might need to run every time a new keyframe is added. The new keyframe is compared to all the other keyframes in the map to check if the current location (place where the current image was taken) is the same as a previously visited location. If the current keyframe is similar to a previous one, this module will perform fusion of these keyframes and all related ones. It might also perform pose optimization, typically as a graph optimization [32].

3.2 Challenges in Deploying Visual-SLAM Systems

3.2.1 Computational Complexity. Typically, loop closure, or the process of identifying previously visited places is extremely time consuming. This is because the complexity of this task grows with the size of the map. Secondly, the process of merging map data structures from two distinct locations could be arbitrarily complex depending on the local structure at those places. Finally, the step of refining poses after the merge involves solving an optimization problem which might also be complex.

Historically, SLAM algorithms were designed to run on robots that had reasonably powerful computing onboard. As these technologies move to mobile/wearable devices, running a Visual-SLAM algorithm at reasonable rates is extremely challenging. Further, SLAM is typically a service to identify the location or recognize a place. This service is used by an application to perform additional tasks that could need additional computing power making it even more challenging to run a complete SLAM system on a mobile/wearable device.

3.2.2 Tight Coupling between Modules. An idea would be to run some modules in a SLAM algorithm on the mobile device while running others on the edge/cloud. However, this is challenging as all modules are tightly coupled. As shown in Figure 2 on page 3, all of them operate on the global map and require to compare, modify and trim the map. Latency in access to this shared data structure or between the modules would result in improper function of the overall system. Therefore, it is challenging to simply offload parts of the computation in a Visual-SLAM system. To better visualize the complexity of de-coupling Visual-SLAM modules, we traced the modules that access parts of the global map in ORB-SLAM2 [41] in Section 4.1.2. The number of locks, and the accessing of various data structures from multiple locations demonstrates the tight coupling of the modules.

3.3 Edge-SLAM Design

3.3.1 Edge-SLAM Design Goals. Our primary goal in designing Edge-SLAM is to reduce the computational and memory overhead on the mobile/wearable device without affecting the accuracy of the execution of the Visual-SLAM system. As described previously, this is challenging given the tightly coupled nature of the modules. A second goal is to keep the overall resource usage (CPU, memory) constant to allow smooth working of applications on the mobile device. As seen in Figure 7 on page 10, running a Visual-SLAM pipeline could potentially require a large, and an increasing amount of resources over time. *Our objective is to keep that constant for long-term operation.*

3.3.2 Edge-SLAM Architecture. Shown in Figure 3 on page 4 is the Edge-SLAM architecture. Our goal is to offload some of the computing to a "nearby" edge device. However, this is non-trivial as described previously. To make it possible, we make two major changes. First, we propose to run the tracking module on the mobile device and move the local mapping as well as the loop closing to the edge device along with the global map. However, the tracking module needs the map for its tasks. To address this, we introduce a *local map*—a partial map that resides on the mobile device. This is

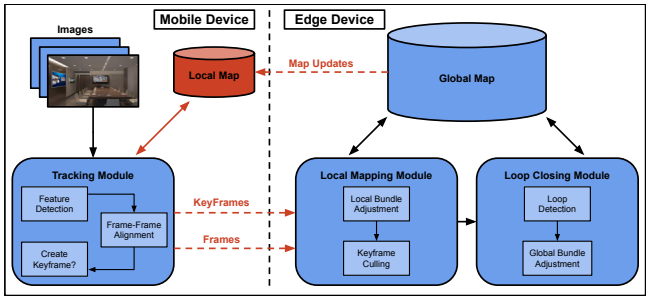

Figure 3: Envisioned architecture of the Edge-SLAM system—our modifications are shown in red [2]

our second modification. We designed the local map structure to meet one of our primary motivations, which is to keep the resource overhead (CPU, memory) on the mobile device constant. From the local/global map structure, the split followed. The tracking module could work completely using the local map and is, therefore, on the mobile device. The local mapping and the loop closing modules need the global map for some of their computation. Therefore, they were moved to the edge. We then provision communication mechanisms between the tracking module and the local mapping module. Since local mapping and loop closing modules frequently update the global map, we also need a mechanism to update the local map when the global map changes. This is discussed further below.

Network Design: A key design challenge was the synchronization between the tracking module on the mobile device and the local mapping/loop closing modules on the edge device. First, we assume that there is a reasonable connection (such as a reliable wireless connection with speeds similar to a local wireless network) between the two sides. Without this, it is challenging to sustain the amount of synchronization required. As shown in Figure 3 on page 4, we designed *three separate network connections* between the mobile device and the edge device. Each of these network connections operates independently of the other so that there is no sequencing of the communication and corresponding delays.

The first two connections are used to pass the output of the tracking module to the edge device—they are shown in red in the lower portion of Figure 3 on page 4. One connection is used by the tracking module during relocalization to communicate the processed frame including features and local geometry, and is labeled *Frames* in Figure 3 on page 4. The second connection is used to pass the keyframe if tracking decided to create a new keyframe, and is labeled *KeyFrames* in Figure 3 on page 4. The third connection is used to update the local map. This is shown in the top portion of Figure 3 on page 4 and labeled *Map Updates*. This communication is typically from the edge device to the mobile device. This connection keeps the local map updated with the global map. The global map keeps track of the differences between its state and the current local map. If the difference is deemed to be large, it creates an update and sends the update to the mobile device. Depending on the current status of tracking, the mobile device can decide if it wants to accept the update or not.

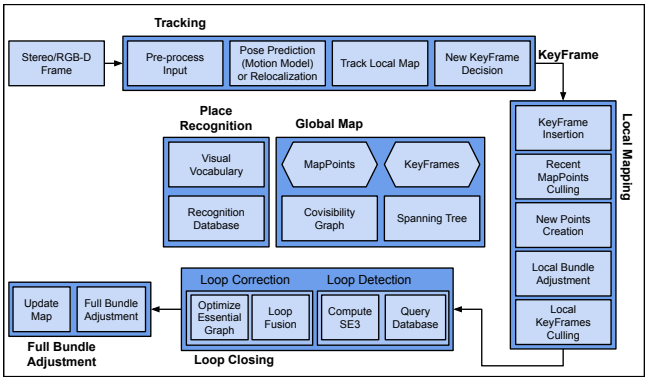

Figure 4: ORB-SLAM2 system architecture [41]

Updating the Local Map: Both the local mapping and loop closing modules update and optimize the global map. They look for local and global relationships between keyframes and constantly work on optimizing the overall map for consistency. However, since Edge-SLAM creates a new map structure for the mobile device, it is important to keep it synchronized with the global map for correct execution of the tracking module.

Each time the tracking module creates a new keyframe, it passes it on to the local mapping module on the edge. Correspondingly, the edge keeps track of the local map on the mobile device. As the global map changes, it computes map updates and sends them to the mobile device for update. However, map updating on the mobile device is a time consuming process as will be shown later. Therefore, we modify the tracking module to have the capability to accept or reject the map updates. Given this tradeoff, we empirically determine a timeout mechanism to decide when the local map is stale and requires updating. The exact mechanism is discussed in the implementation of our prototype.

While conceptually, these changes seem straightforward, engineering everything to work together with realistic network latencies is challenging. To demonstrate the feasibility of this architecture, we prototyped our idea on ORB-SLAM2 [41], a well-known Visual-SLAM system. Our work provides a conceptual design to offload Visual-SLAM. In order to apply it to other Visual-SLAM systems, one would need to map the components of our design to the implementation of the particular Visual-SLAM system to determine the exact implementation of the offloading.

4 Edge-SLAM IMPLEMENTATION IN ORB-SLAM2

Section 3 described our overall design to offloading some tasks in Visual-SLAM. We prototyped our idea using ORB-SLAM2 since it is open-source. We will now describe ORB-SLAM2, and our Edge-SLAM prototype implementing the split architecture using ORB-SLAM2. Please note that we present some of the details including several magic numbers that make the system work for clarity.

4.1 ORB-SLAM2

4.1.1 Overview. ORB-SLAM2 [41] is the recent state-of-the-art graph-based Visual-SLAM algorithm that can use a monocular (RGB) camera, stereo cameras, or RGB-D camera to build sparse 3-D maps. The map is a graph where vertices correspond to image frames, and edges correspond to 3-D visual transformations between them.

Data Structure	Lock	No. of Operations Acquiring the Lock per Module
`MapPoint`	`mGlobalMutex` (static)	5—Tracking 1—Local Mapping 1—Loop Closing 1—Full Bundle Adjustment
`MapPoint`	`mMutexPos`	12—Tracking 6—Local Mapping 4—Loop Closing 1—Full Bundle Adjustment
`MapPoint`	`mMutexFeatures`	12—Tracking 6—Local Mapping 4—Loop Closing 1—Full Bundle Adjustment
`KeyFrame`	`mMutexPose`	2—Tracking 4—Local Mapping 4—Loop Closing 1—Full Bundle Adjustment
`KeyFrame`	`mMutexConnections`	4—Tracking 7—Local Mapping 4—Loop Closing 1—Full Bundle Adjustment
`KeyFrame`	`mMutexFeatures`	7—Tracking 8—Local Mapping 6—Loop Closing
`KeyFrameDatabase`	`mMutex`	1—Tracking 1—Local Mapping 1—Loop Closing
`Map`	`mMutexMapUpdate`	1—Tracking 1—Local Mapping 1—Loop Closing 1—Full Bundle Adjustment
`Map`	`mMutexPointCreation`	4—Tracking 1—Local Mapping
`Map`	`mMutexMap`	6—Tracking 6—Local Mapping 1—Loop Closing 1—Full Bundle Adjustment

Table 1: Global map locks in ORB-SLAM2 [41]

ORB-SLAM2 consists of three threads, one per module: tracking, local mapping, and loop closure as shown in Figure 4 on page 5. The tracking thread loops through incoming image frames for their initial pose estimation and decides which frame to accept as a keyframe, based on five conditions where the first four were introduced in the first ORB-SLAM paper [40], and the fifth in the second paper [41]. They are:

(1) If relocalization occurred, then 20 frames should pass to insert a new keyframe.
(2) Either 20 frames have passed after the last inserted keyframe, or local mapping thread is not busy.
(3) The current frame is tracking at least 50 features.

(4) The current frame is tracking fewer than 90% points compared to the frame's reference keyframe (i.e., the keyframe most similar to the current frame).
(5) If the current frame tracks less than 100 close points, and can create more than 70 new close points.

As described later, this detail is important to Edge-SLAM because our split requires us to reason about some of these conditions on the mobile side and others on the edge side.

Local mapping adds accepted keyframes to a global map and performs map-points and keyframes optimization as well as bundle adjustment on the map local to the accepted keyframe. Next, local mapping passes the accepted keyframe to the loop closure thread, which checks for loops, which checks if the current keyframe is similar to any previously stored keyframe. If they are similar, it performs loop correction where it corrects keyframes poses and optimizes the global map.

There are two central data structures to the operation of ORB-SLAM2—*KeyFrames* and *Map-Points*. A keyframe, as described above, is an image frame that contains unique segment or viewpoint of the environment. A map-point stores the position, feature descriptor, and references to all keyframes that observe it. In ORB-SLAM2, the map-points are obtained by detecting the ORB-features in an image. Therefore, the feature descriptors are ORB-descriptors. Each keyframe points to several map-points. Also, multiple keyframes could point to the same map-points if the same features are seen from multiple keyframes. Together, the set of all keyframes currently constructed, and all their observed map-points form the current global map. These three data structures depend on each other, and maintain several additional information—details can be found in [40, 41]. Part of the map is a visual vocabulary. ORB-SLAM2 stores this so it is easy to compare frames. Each incoming frame gets tagged with the list of observed words from this dictionary. This can be used later for quick lookup of similar frames—for loop closure for example.

4.1.2 Complexity. ORB-SLAM2[2] is an open-source system with large code-base consisting of 20 classes and ≈18,000 lines of code. The system depends on three main threads running simultaneously, where each thread runs one of the modules, i.e., tracking, local mapping, and loop closing. Further, the system initiates a fourth thread on-demand after every loop closure to perform full bundle adjustment. ORB-SLAM2 complexity lies in all these threads working on a shared global map structure. To demonstrate the level of coupling between these threads, Table 1 on page 5 lists the set of locks used in the ORB-SLAM2 code, the data structures they control access to, and the number of times they are called in various modules. For example, we observe in the table that the `Map` data structure lock `mMutexMap` is acquired to perform six operations in the tracking module such as `Tracking::UpdateLocalMap()`, to perform six operations in the local mapping module such as `LocalMapping::MapPointCulling()`, to perform one operation `LoopClosing::CorrectLoop()` in the loop closing module, and to perform full bundle adjustment.

In ORB-SLAM2, all three threads assume to execute on the same computing device and have access to a global map maintained by the system. The operation of each of the threads is dependent on each other because of their reliance on the shared data structures.

[2]https://github.com/raulmur/ORB_SLAM2

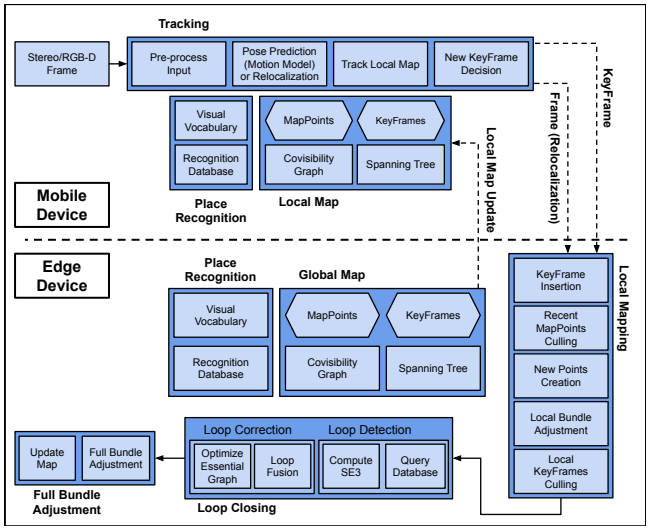

Figure 5: Edge-SLAM system architecture

4.2 Edge-SLAM Implementation

Figure 5 on page 6 shows a breakdown of Edge-SLAM into components that run on the mobile device and the edge. As described earlier, in Edge-SLAM split architecture, the Tracking thread runs on the mobile device while the Local Mapping and the Loop Closing threads run on the edge.

Global Map: The global map is created and stored on the edge. This contains the complete set of keyframes, set of map-points, the Co-Visibility Graph, and the Spanning Tree. The Co-Visibility Graph connects keyframes based on their shared map-points observations. The Spanning Tree is a subset of the Co-Visibility Graph connecting every keyframe with the keyframe that they share the most map-points.

Local Map: We create our new map structure called the Local Map in the tracking thread on the mobile device. It includes a subset of the latest created keyframes, map-points, Co-Visibility Graph, and Spanning Tree from the global map. It uses the same visual vocabulary and recognition database as the one on the edge.

Map Synchronization: The edge periodically sends local map updates with the latest optimized changes to the mobile device. The mobile device, on the other hand, instantly sends newly created keyframes along with its map-points to the edge. On receiving an update, the mobile device can choose if it wants to update its local map or not. The edge always accepts new keyframes from the mobile device.

4.3 Mobile-Edge Network Setup

As described in Section 3, the latency between modules could greatly affect the working of the Visual-SLAM system. To this end, we have three separate connections between the mobile device and the edge—one each to transmit frames from the mobile device to the edge upon relocalization event, to transmit keyframes from

the mobile device to the edge, and the third to send map updates from the edge to the mobile device. With system performance a priority, we use a fast blocking concurrent queue implementation[3] for inter-thread communication. Our results show that such setup has low overhead on the overall working of the Edge-SLAM system.

We will now describe Edge-SLAM operation on the mobile as well as the edge.

4.4 Mobile Device Operation

In our design (Figure 5 on page 6), the mobile device runs the tracking thread. In order to decouple the mobile device operation, we allow the tracking thread to maintain a local map, create new keyframes, create new map-points, and accept map updates from the edge. As in ORB-SLAM2, the tracking thread in Edge-SLAM continuously processes input feed/frames from the camera, tracks the local map, estimates the initial position of the current frame, and decides which frame to be a keyframe. However, in Edge-SLAM, the local mapping on the edge does not accept all keyframes created by the tracking on the mobile device. This is because we split the keyframe creation conditions, described in Section 4.1, between the mobile device and the edge depending on where each condition can be validated. We moved the conditions 1 and 2 to the local mapping on the edge and kept the conditions 3, 4, and 5 on the mobile device. If a new keyframe is to be created, it creates the new keyframe in its local map and sends a copy to the edge to be considered for addition to the global map as well.

When the mobile device receives a local map update, it first checks if it is not redundant. A map update is considered redundant if no new keyframe created since the last applied map update. Second, the mobile device checks if applying the local map update would significantly increase the chance of losing track due to its latency. We apply a time constraint starting at 300 milliseconds, which decreases as the number of keyframes in the local map increases. Such a time constraint will limit the size of the local map on the mobile device. It will also prevent a local map update from being applied when the keyframe creation rate is high. Typically, more keyframes are created when there are large changes in the scene indicating that the device is moving fast. In such cases, interruptions for map updates will lead to losing track, and is undesirable. Thus, a local map update is accepted only if more than the time constraint has passed since the last created keyframe. We use the following formula to compute the time constraint:

$$TC = ITC/((KFN/LMU) + 1) \quad (1)$$

Where TC is Time Constraint, ITC is Initial Time Constraint (set to 300ms), KFN is current number of keyframes in the local map, and LMU is the Local Map Update size (set to six keyframes). We discuss why we choose such numbers in Section 4.7.

When a local map update is accepted, the tracking thread would temporarily stop operation and not process any new frame. The mobile device updates the local map by fully clearing the current one, and then constructs a new map using the received update. Because keyframes are sent to the edge upon creation, no information is lost when the current local map is cleared. This is a time-consuming process and we show the latencies involved in Section 5.

[3]https://github.com/cameron314/concurrentqueue

4.5 Edge Operation

As described earlier, the edge runs two threads: local mapping and loop closing. Local mapping thread receives keyframes from the mobile device as they get created and sends periodic local map updates back to the mobile device. On the other hand, loop closing thread interacts with local mapping thread to receive new keyframes, after they get processed and added to the global map, and then it continues processing as in ORB-SLAM2. When the local mapping thread on the edge device receives a new keyframe, it checks the remaining keyframe insertion conditions, i.e., conditions 1 and 2 described in Section 4.1, before accepting the keyframe to be inserted into the global map.

In Edge-SLAM, the mobile device maintains a local map to keep the system going. This map is not intended to be used for a long-term run. Because the local mapping and loop closure run on the edge, the tracking local map (on mobile device) does not get optimized and might drift and affect the system accuracy if it does not receive an update regularly. Thus, our objective is to maximize the number of updates to minimize such drift in the tracking thread. However, maximizing the number of updates would also mean more network usage as well as adding map reconstruction overhead to the tracking thread. In Edge-SLAM, we implemented a timer-based update module in local mapping thread to regularly send a local map update with the minimum number of keyframes possible at short time intervals. Such an update would correct any drifts and inconsistencies in the mobile device's local map. In our update module, a local map update is sent every five seconds and consists of the six most recent keyframes inserted into the global map along with all of their map-points. By sending small map updates at short time intervals, we are achieving our objectives to minimize the drift, minimize the map reconstruction overhead, and limit the network usage. We will quantify all these in Section 5. In Section 4.7, we discuss why a local map update consists of six keyframes in more detail.

4.6 Implementation Tradeoffs

4.6.1 Local Map Update Strategies. There are two typical methods to update the local map on the mobile device. The first method is to apply edge changes to the mobile device's current local map. The second method is to clear the mobile device's current local map and replace it with the new received local map update from the edge. After running several experiments, we identified the following issues with the first method, which makes it not efficient and unsafe:

- If we continue reusing the current local map on the mobile device by applying changes to it, then we will accumulate lots of unprocessed and unoptimized keyframes and map-points in the local map. Such keyframes and map-points will also contribute to creating newer keyframes and map-points that are inaccurate and increase the chances of drift and lower accuracy in the global map on the edge.
- Applying changes to the current local map require an expensive search for every single keyframe and map-point in the update to find all their references in the local map structure. This would significantly increase the time complexity of applying an update and reduce the mobile device performance.

- Due to the complex structure of OBR-SLAM2, there exist lots of cyclic references in the data structures. Thus, applying changes to the current local map increases the chances of memory issues such as memory leaks.
- The local map structure on the mobile device is shared between the various threads. This requires mechanisms to avoid concurrent operations on the map for correct functioning. Therefore, synchronization mechanisms such as locks are used by individual threads to streamline their access. Isolating such data structure to update it would be very time consuming and might result in the erroneous operation of the whole system.

Thus, whether we receive updates immediately as it happens or regularly over time, the process of applying the updates would rapidly increase the chance of losing track as well as decrease the mobile device performance. The second method, by contrast, is safer, more efficient, and has fewer side effects on the mobile device. In this method, a local map update fully replaces the existing mobile device local map with the minimum possible overhead on the mobile device performance due to the small size of the update.

4.7 Engineering Edge-SLAM Modules For Efficient Operation

4.7.1 Tracking Thread. Edge-SLAM has 4 parameters—local map update size (discussed in Section 4.7.2), local map update frequency (discussed in Section 4.5), relocalization frame frequency (discussed in Section 4.7.4), and time constraint to accept a local map update (discussed in this section). Further, Edge-SLAM is sensitive to ORB-SLAM2 parameters. Thus, not setting such parameters correctly would affect the system performance.

As we discussed in Section 4.4, the tracking thread computes a time constraint value which is used to decide whether to accept a local map update or not. We initially set this to 300ms. Our objective when selecting an initial time constraint value was to control how big the tracking local map can get before it is updated, especially during high keyframe creation rate. Also, we wanted to allow the mobile device to work independently for small periods and regardless of connectivity to the edge. After several experiments, we found that during high keyframe creation periods, an initial time constraint value of 300ms would most likely lead the tracking thread to accept a local map update before the size of the local map gets higher than 50 keyframes.

4.7.2 Local Mapping Thread. As described earlier in Section 4.5, when the local mapping thread prepares a local map update to send to the mobile device, it sets the size of the update to six keyframes. The main objectives we had when setting the size of the local map update was to reduce network usage and to reduce map reconstruction overhead on the mobile device. Thus, after looking into the tracking thread initialization process, we found that if the system loses track and there are less than six keyframes in the map, the tracking thread would reset the whole system. It would assume the system lost track right after initialization. Based on this condition, the minimum number of keyframes the local map can have to continue working without resetting the system is six, which is what we choose as our local map update size.

4.7.3 Reset Function. The reset function can be called by the system as well as the user. The system calls reset function after an unsuccessful initialization. Whenever the reset function is called, the system will clear all data structures and restart the mapping process. In Edge-SLAM, we did not add any new data structure to perform a full (mobile-edge) system reset. Instead, the tracking thread uses the same keyframes connection to resend the most recent keyframe after setting the keyframe's reset flag to true. This way, both sides would reset instantly upon receiving a request either from the system or the user.

4.7.4 Relocalization Function. Relocalization function is called when the tracking thread loses track where it tries to re-compute the camera pose using the global map. Relocalization is particularly useful when the tracking thread loses track at a location that has been previously visited and mapped. In Edge-SLAM, we want relocalization to be as robust as ORB-SLAM2. To this end, when our system loses track, it not only tries to relocalize using the current local map but also sends a relocalization request to the edge for assistance. This is why we dedicated one of the connections between the mobile device and the edge to the transmission of frames that assist in relocalization. When the tracking thread on the mobile device loses track, it transmits a frame to the edge every half a second. We chose to send a frame every half a second to allow some change to happen in the scene, so we do not send redundant frames. The local mapping on the edge uses the received frames to detect candidate keyframes from the global map for relocalization. It then sends a relocalization map update to the mobile device so the mobile device can try estimating the camera pose from the map. A relocalization map update consists of candidate keyframes in addition to the keyframes connected to each one of them. When Edge-SLAM is trying to relocalize, it lifts all time and size limits imposed on local map updates. This is because successful relocalization is a priority over performance during such time. We compare Edge-SLAM and ORB-SLAM2 relocalization statistics in Section 5.5.

5 EVALUATION

5.1 Experiment Setup

To evaluate Edge-SLAM system, we run experiments using two distinct mobile devices and an edge device. The first mobile device is an NVIDIA JETSON TX2—64-bit NVIDIA Denver and ARM Cortex-A57 CPUs, NVIDIA Pascal GPU with 256 CUDA-cores, 8 GB Memory, Connects to 802.11ac WLAN—running Ubuntu 18.04LTS. This is a prototypical computing platform comparable to the processing capabilities on the Magic Leap One AR glasses [28]. The second mobile device is a Dell Latitude laptop—Intel Core i5-520M (2.4GHz, 3M cache) (Dual-Core), Intel HD Graphics with dynamic frequency, 8 GB Memory—running Ubuntu 18.04LTS. This computing platform is loosely comparable to the resources on a Microsoft HoloLens [38]. The edge device is a Dell XPS desktop—Intel Core i7 9700K (8-Core/8-Thread, 12MB Cache, Overclocked up to 4.6GHz on all cores), NVIDIA GeForce GTX 1080, 32 GB Memory—running Ubuntu 18.04LTS.

We use a pre-collected RGB-D dataset of one of our campus building floors as the input source for long-running experiments. Our dataset is collected using a robot equipped with a Kinect 360

RGB-D sensor as well as a Velodyne VLP-16 LiDAR for ground truth. The dataset consists of 52,427 frames and runs for a total of 1,774 seconds (≈30 minutes). We repeat our experiments on each platform by replaying the frames as if they are being collected at that time. This allows us to provide exact comparison of performance across different platforms and is a standard mechanism to compare performance across SLAM systems [6]. Typical frame rate of the Kinect is 30fps which is what we replay our dataset at. However, at eight locations, when the robot was making a turn, we observe that the ORB-SLAM2 system is unable to find correspondences. After repeated experimentation, we found that slowing the frame rate down to 15fps at the ends of the corridors when the robot is turning allows the system to keep the correspondences and continue to build the map. Note that this is a shortcoming of ORB-SLAM2, and fixing this was beyond the scope of this work. Therefore, for each experiment, we reduce the frame rate to 15fps close to the turns and replay it at 30fps at other locations. This replay pattern is the same for the evaluation of ORB-SLAM2 as well as Edge-SLAM in all the results presented below.

We also use a popular public dataset from Technical University of Munich called TUM [26] RGB-D dataset for short-running experiments and run them at 30fps. The frames are read from storage and published as ROS topics[4] for consumption by either ORB-SLAM2 or Edge-SLAM. The mobile devices are connected to different campus Wi-Fi networks, and the edge is connected to the campus network through a wired connection emulating a realistic deployment. Please note that we performed experiments at all times of day and our results inherently capture network dynamics due to multiple users using the same access points.

Using this setup, we compare results from four configurations:

- Running ORB-SLAM2 on the JETSON TX2. We refer to this experiment by **ORB-SLAM2 JTX2** in the results.
- Running ORB-SLAM2 on the Dell Latitude laptop. We denote this experiment by **ORB-SLAM2 L** in the results.
- Running Edge-SLAM on the JETSON TX2, and the edge Dell XPS desktop. We denote this experiment by **Edge-SLAM JTX2-D** in the results.
- Running Edge-SLAM on the Dell Latitude laptop, and the edge Dell XPS desktop. We denote this experiment by **Edge-SLAM L-D** in the results.

5.2 Edge-SLAM Performance

As a reminder, the two goals of Edge-SLAM is to reduce computational load on the mobile device and keep the load constant. Our first set of results show the computational complexity of running Visual-SLAM completely on the mobile device (ORB-SLAM2) and running Visual-SLAM with offloading (Edge-SLAM). If not specified, all of them are results averaged over our dataset.

Shown in Figure 6 on page 10 (left) are the average times taken (in ms) to run the individual modules (tracking, local mapping and loop closing) in each of the four configurations. As seen from Figure 6 on page 10 (right), the tracking thread takes less than 80ms on average. There is also not much difference in performance between latency in execution of the original ORB-SLAM2 tracking module and the tracking module in Edge-SLAM. This is expected given the two modules are similar with the exception of the tracking module in Edge-SLAM interacting with the local map, which does not have any significant performance impact. However, there is a large difference between the execution time for local mapping as well as loop closure modules. We would like to make a couple of observations related to these results:

- These modules run on the JETSON-TX2/Laptop in the case of ORB-SLAM2 while they run on the edge (Desktop) in the case of Edge-SLAM which has a more powerful CPU and can correspondingly execute the modules faster. However, this comparison is reasonable as this is the main reason offloading is appealing.
- While the tracking module is continuously executed to process the incoming frames, local mapping and loop closure modules are not. The local mapping module is executed only when the tracking module deems that a frame needs to be registered as a keyframe. Loop closure module runs partially for every created keyframe to check for a loop, and runs entirely if a loop is detected. Our dataset has a total of ≈52,400 frames from which our system gets to process ≈39,500 frames. It creates ≈1,200 keyframes. Therefore, local mapping and loop closure (partial run) modules get called ≈1,200 times.
- Finally, there were two loop closure occurrences in our dataset, which happen toward the middle and the end. It can be observed in Figure 7 on page 10 (both sub-figures) at ≈60% and ≈95% execution time of ORB-SLAM2 running on mobile devices. While the loop closure occurs only twice in our dataset, one can envision more occurrences in other scenarios based on the trajectory followed by the user.

We would like to make two observations with regards to the results of average latency in processing for each of the modules of Visual-SLAM. First, Edge-SLAM reduces the latency in the local mapping and loop closing modules by offloading them to the edge. It reduces the latency of loop closure module dramatically, allowing for faster map updates. Our second observation is that by offloading the intensive tasks, we reduce the variability of performance on the mobile device and allow the mobile device to run end user applications which are the main reason to run Visual-SLAM in the first place. *This accomplishes our first objective of reducing overall computational load on the mobile device.*

The next significant performance result is shown in Figure 7 on page 10. Figure 7 on page 10 (left) shows the instantaneous CPU usage through the execution on the mobile device. On average, the CPU usage for ORB-SLAM2 is at ≈30% while using the JETSON TX2 and ≈40% while using the laptop. In comparison, the CPU usage is ≈15% when using the JETSON TX2 for Edge-SLAM and ≈25% when using the laptop. Overall, there is ≈35-50% reduction in CPU use while using Edge-SLAM.

Figure 7 on page 10 (right) shows the memory usage on the mobile device for both ORB-SLAM2 as well as Edge-SLAM. As described in Section 3, as the size of the map increases, the overall memory required to store it also goes up. If the global map is stored on the mobile device (as in ORB-SLAM2), this will result in growing memory use which is highly undesirable. Also note that the memory use goes up when loop closure is performed (at ≈60% and ≈95%

[4] https://wiki.ros.org

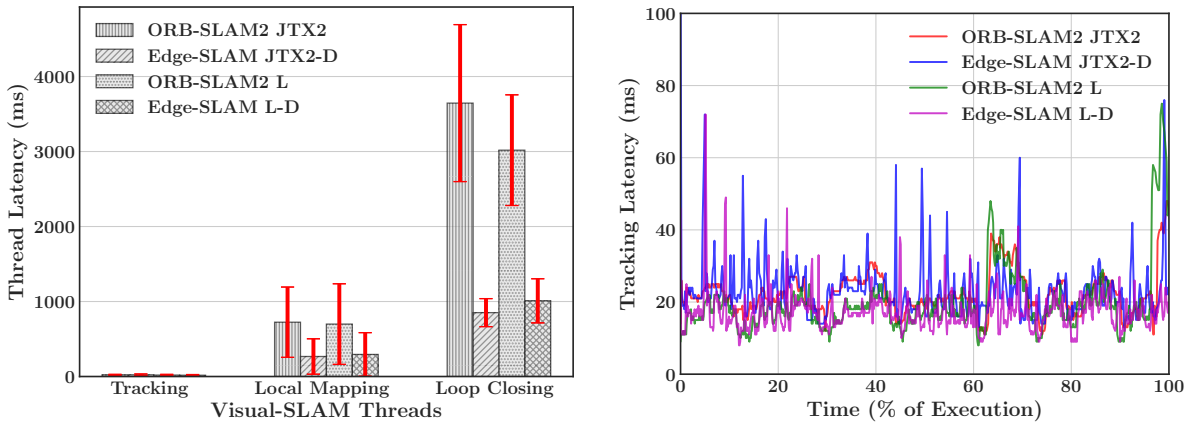

Figure 6: Overall latency of ORB-SLAM2 and Edge-SLAM. The average latency per-module (left) shows that Edge-SLAM offloads the two CPU-intensive tasks. Tracking module latency on the mobile device over time (right) better shows the latency for that module in each configuration

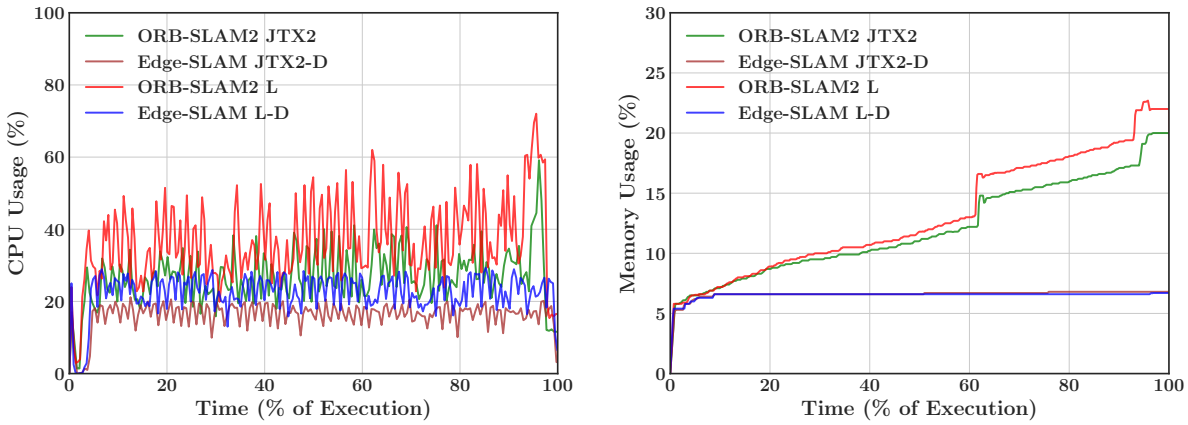

Figure 7: Resource usage of ORB-SLAM2 and Edge-SLAM on the mobile device—CPU (left) and Memory (right). The jumps in memory use at 60% time and 95% time (right) are due to loop closures in ORB-SLAM2

execution time). This is also undesirable. Because Edge-SLAM stores a fixed size local map on the mobile device, the memory usage of Edge-SLAM is constant. It remains constant even during loop closure. *This accomplishes our second objective of keeping the resource use on the mobile device constant.*

5.3 Network Performance

As described in Section 5.1, we use a regular on-campus wireless network to connect the mobile device with the edge. We also performed our experiments in regular working hours when the access points are used by many users. We did so to understand the network latency imposed in a regular urban setup, and its effect on the Edge-SLAM system. In Figure 3 on page 4, we show three links between the mobile and the edge device. We characterize the delay on each of these links for both configurations of Edge-SLAM we experimented with. The biggest source of delay is in the map update

Edge-SLAM / Map Update	Edge-SLAM JTX2-D (ms)	Edge-SLAM L-D (ms)
Construct Map Update on Edge	57.09 ±0.69	58.30 ±0.66
Re-Construct Map Update on Mobile Device	411.43 ±4.84	285.68 ±3.18

Table 2: Local map update latency on mobile device and edge

from the global map on the edge to the local map on the mobile device. This latency is shown in Table 2 on page 10. Totally, there are three parts to this latency. First is the latency to construct the map update on the edge. Second to transmit the update to the mobile device. Finally, once the update is received, the mobile device needs to reconstruct the map. Table 2 on page 10 shows the first

Map Update \ Edge-SLAM	Edge-SLAM JTX2-D (s)	Edge-SLAM L-D (s)
Map Update Publish Frequency on Edge	8.24 ±0.48	7.92 ±0.43
Map Update Acceptance Frequency on Mobile Device	9.16 ±0.55	8.74 ±0.48

Table 3: Local map update frequency on mobile device and edge

Keyframe Update \ Edge-SLAM	Edge-SLAM JTX2-D (ms)	Edge-SLAM L-D (ms)
Transmit Keyframe from Mobile Device to Edge	162.43 ±2.90	142.53 ±6.38

Table 4: Keyframe transmission latency from mobile device to edge

Accuracy Measure \ Visual-SLAM	ORB-SLAM2 JTX2	Edge-SLAM JTX2-D	ORB-SLAM2 L	Edge-SLAM L-D
Mean Localization Error (cm)	20.59 ±10.92	19.23 ±11.32	20.90 ±12.77	21.39 ±9.16

Table 5: Mean Localization Error of ORB-SLAM2 and Edge-SLAM

and third parts of this latency. Constructing the update is done on the edge and takes ≈58ms. Reconstruction, however, takes between 285ms and 411ms which is long. We should also note that when the local map is being reconstructed on the mobile device, it cannot execute the tracking module. Each map update consists of the latest six created keyframes. Shown in Table 4 on page 11 are the network latencies in transmitting keyframes across a wireless network—both dedicated as well as public. Given the on-campus public network has higher bandwidth than the private network; the keyframe transmission is faster on the public network compared to the private network—142ms vs. 162ms on average. However, like we show in the next subsection, the Edge-SLAM system works in both scenarios with little added error in mapping.

Finally, Table 3 on page 11 shows the frequency of the map update in our experiments. Note that the mobile device might choose not to accept map updates transmitted by the edge. We show both the average frequency of map updates transmitted as well as the frequency of map updates accepted. Our results show that the mobile device rejects map updates rarely, at least on our dataset. Map updates are accepted every ≈9s.

5.4 Mapping Accuracy

Our primary objective was to improve the execution performance of Visual-SLAM while running on mobile devices. Implicit in this objective is to retain the accuracy of the localization and mapping achieved by the redesigned Visual-SLAM system. In this subsection, we will compare the performance of Edge-SLAM with ORB-SLAM2. The 2-D trajectories of the path traced by the mobile device as constructed by ORB-SLAM2 and Edge-SLAM are shown in Figure 8 on page 12. There is minimal difference between the mapped trajectories and the ground truth trajectory demonstrating that the Edge-SLAM system is comparable in accuracy to ORB-SLAM2. For a more detailed examination, we show the mean localization error in centimeters of ORB-SLAM2 and Edge-SLAM on both platforms in Table 5 on page 11. Each system performs better on one platform, and in each case, the difference is ≈1cm on average on a trajectory of ≈150m length. For most applications, this is quite acceptable for the feasibility of deploying accurate localization/mapping long-term on mobile devices.

Also, ORB-SLAM2 mapping accuracy is very good. From the results in Table 5 on page 11, we see that it only drifts for ≈20cm after traveling ≈150m, which is relatively small. Inherent latency, as well as working with a smaller map (local map), tend to increase the potential drift/error in Edge-SLAM. However, our results show that the drift of running Edge-SLAM is similar to ORB-SLAM2, i.e., ≈20cm, after traveling ≈150m. This is because ORB-SLAM2 performs full bundle adjustment after every loop closure. This process optimizes the global map to reduce the drift. In our experiments, our dataset has two loops, where ORB-SLAM2 has to run full bundle adjustment. However, Edge-SLAM got to run full bundle adjustment three times because the loop closing module runs on the edge with more computing power and lower latency, and this enabled the detection and running of an additional full bundle adjustment for one of the loops.

Relocalization \ Visual-SLAM	ORB-SLAM2 JTX2	Edge-SLAM JTX2-D	ORB-SLAM2 L	Edge-SLAM L-D
# of Successful Relocalization	4	6	4	10
Relocalization Latency (ms)	18.25 ±2.29	38.50 ±4.61	12.00 ±0.82	22.90 ±3.92

Table 6: Relocalization statistics for ORB-SLAM2 and Edge-SLAM

5.5 Relocalization Latency

In this subsection, we show that the split architecture has minimal effect on the relocalization performance. We perform the four experiments using a public dataset—TUM [26] RGB-D dataset running at 30fps. While running this dataset, both systems, i.e., ORB-SLAM2 and Edge-SLAM, lose track and successfully relocalize multiple times. This is because the camera in this dataset shakes multiple times while in motion. Table 6 on page 11 shows that our system successfully relocalizes multiple times comparable to ORB-SLAM2. It even relocalizes a couple of times more. This is because the tracking module in Edge-SLAM uses a local-map, and thus it reasons with fewer features in comparison to ORB-SLAM2 since it has fewer keyframes in comparison to the global map. This is the cause of additional relocalizations.

Table 6 on page 11 also shows that the split architecture of Edge-SLAM has minimal effect on the relocalization latency. When our system loses track, it not only sends a relocalization request to the edge but also tries to relocalize using the existing local map in the meantime. Thus, on average, the relocalization latency on both systems is less than ≈40ms. ORB-SLAM2 takes on average between 12ms and 18ms and Edge-SLAM takes on average between 22ms

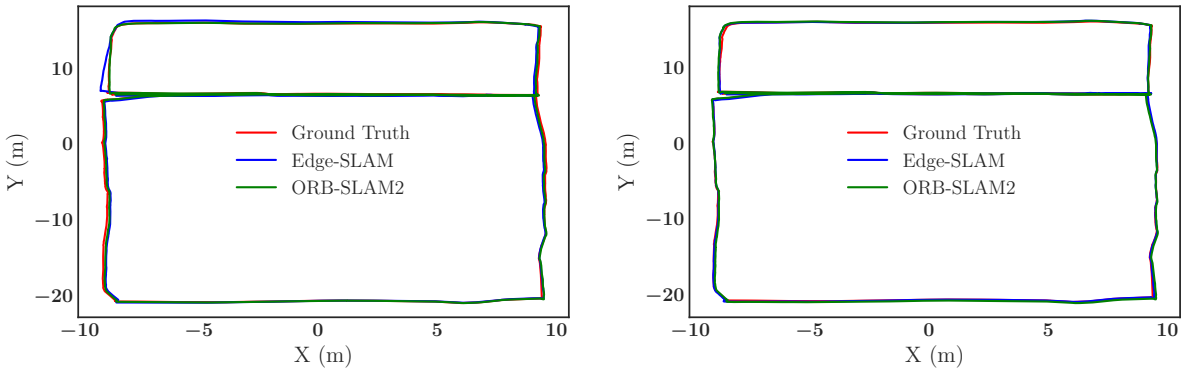

Figure 8: ORB-SLAM2 and Edge-SLAM trajectories running on the JETSON TX2 (left) and the laptop (right) compared to the ground truth. The Edge-SLAM is connected to on-campus private network on the JETSON TX2 and on-campus public network on the laptop

and 38ms. The additional overhead of Edge-SLAM comes from serializing a frame every half a second to send it to the edge. Otherwise, Edge-SLAM and ORB-SLAM2 have almost similar latencies.

6 CONCLUSION

Many mobile applications including augmented reality apps (and libraries such as ARCore, ARKit and HoloLens API) require spatial localization. One popular mechanism to achieve this is using Visual-SLAM. However, most Visual-SLAM systems are computationally intensive. In this work, we adapt Visual-SLAM to a split architecture called Edge-SLAM distributing the compute load between a mobile device and an edge device. We demonstrate our proposal by prototyping the Edge-SLAM architecture using ORB-SLAM2, a popular Visual-SLAM system. In particular, we moved the tracking module to the mobile device and the local mapping as well as the loop closing modules to the edge. We achieved this split by creating a new map structure called the local map on the mobile device for use by the tracking module. This local map only contains a local view of the global map, and gets periodically updated by the edge when needed.

In Edge-SLAM, we overcome two challenges of ORB-SLAM2. We limit the growth in memory usage due to increasing map size, and keep the mobile device memory usage constant. We also move the bursty computational tasks (local mapping and loop closing) to the edge device allowing the mobile device to function more efficiently and allow running other apps. Overall, we achieved this with minimal loss of accuracy in the final map as well as the trajectory taken. We demonstrated this using our own dataset as well as a publicly available dataset. We have internally tested our system on multiple other datasets and the results are similar. We open-source[5] our Edge-SLAM implementation and make it available to other researchers to evaluate their solutions using Edge-SLAM.

In the future, we would like to deploy Edge-SLAM in a long-term setting and observe the challenges of executing SLAM across days. We are interested in crowd-sourcing maps of large urban spaces as well as reasoning about localization across devices.

[5]http://droneslab.github.io/edgeslam

ACKNOWLEDGMENTS

We want to thank Yash Narendra Saraf for his assistance with the network implementation. We thank Steve Ko and Sofiya Semenova for early discussions about this work. The authors were supported through NSF#1846320 and NSF#1514395. We would like to thank our shepherd Nicholas Lane and the anonymous reviewers for their feedback in improving this work.

REFERENCES

[1] 2020. Computer Cpu Desktop - Free vector graphic on Pixabay. https://pixabay.com/images/id-156768/.
[2] 2020. Interior Design Tv Multi-Screen - Free image on Pixabay. https://pixabay.com/images/id-828545/.
[3] 2020. Smartphone Android Technology - Free vector graphic on Pixabay. https://pixabay.com/images/id-3358735/.
[4] Charuvahan Adhivarahan and Karthik Dantu. 2019. WISDOM: WIreless Sensing-assisted Distributed Online Mapping. In *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 8026–8033.
[5] T. Bailey and H. Durrant-Whyte. 2006. Simultaneous localization and mapping (SLAM): part II. *IEEE Robotics Automation Magazine* 13, 3 (Sep. 2006), 108–117. https://doi.org/10.1109/MRA.2006.1678144
[6] M. Bujanca, P. Gafton, S. Saeedi, A. Nisbet, B. Bodin, M. F. P. O'Boyle, A. J. Davison, P. H. J. Kelly, G. Riley, B. Lennox, M. LujÃąn, and S. Furber. 2019. SLAMBench 3.0: Systematic Automated Reproducible Evaluation of SLAM Systems for Robot Vision Challenges and Scene Understanding. In *2019 International Conference on Robotics and Automation (ICRA)*. 6351–6358. https://doi.org/10.1109/ICRA.2019.8794369
[7] Kaifei Chen, Tong Li, Hyung-Sin Kim, David E. Culler, and Randy H. Katz. 2018. MARVEL: Enabling Mobile Augmented Reality with Low Energy and Low Latency. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems* (Shenzhen, China) *(SenSys '18)*. ACM, New York, NY, USA, 292–304. https://doi.org/10.1145/3274783.3274834
[8] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. CloneCloud: Elastic Execution Between Mobile Device and Cloud. In *Proceedings of the Sixth Conference on Computer Systems* (Salzburg, Austria) *(EuroSys '11)*. ACM, New York, NY, USA, 301–314. https://doi.org/10.1145/1966445.1966473
[9] D. M. Cole and P. M. Newman. 2006. Using laser range data for 3D SLAM in outdoor environments. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.* 1556–1563. https://doi.org/10.1109/ROBOT.2006.1641929
[10] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services* (San Francisco, California, USA) *(MobiSys '10)*. ACM, New York, NY, USA, 49–62. https://doi.org/10.1145/1814433.1814441

[11] Google Developers. 2020. Build new augmented reality experiences that seamlessly blend the digital and physical worlds. https://developers.google.com/ar.

[12] M. W. M. G. Dissanayake, P. Newman, S. Clark, H. F. Durrant-Whyte, and M. Csorba. 2001. A solution to the simultaneous localization and map building (SLAM) problem. *IEEE Transactions on Robotics and Automation* 17, 3 (June 2001), 229–241. https://doi.org/10.1109/70.938381

[13] H. Durrant-Whyte and T. Bailey. 2006. Simultaneous localization and mapping: part I. *IEEE Robotics Automation Magazine* 13, 2 (June 2006), 99–110. https://doi.org/10.1109/MRA.2006.1638022

[14] Jakob Engel, Thomas Schöps, and Daniel Cremers. 2014. LSD-SLAM: Large-Scale Direct Monocular SLAM. In *Computer Vision – ECCV 2014*, David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars (Eds.). Springer International Publishing, Cham, 834–849.

[15] Nikolas Engelhard, Felix Endres, Jürgen Hess, Jürgen Sturm, and Wolfram Burgard. 2011. Real-time 3D visual SLAM with a hand-held RGB-D camera. In *Proc. of the RGB-D Workshop on 3D Perception in Robotics at the European Robotics Forum, Vasteras, Sweden*, Vol. 180. 1–15.

[16] Nikolas Engelhard, Felix Endres, Jürgen Hess, Jürgen Sturm, and Wolfram Burgard. 2011. Real-time 3D visual SLAM with a hand-held RGB-D camera. In *Proc. of the RGB-D Workshop on 3D Perception in Robotics at the European Robotics Forum, Vasteras, Sweden*, Vol. 180. 1–15.

[17] M. F. Fallon, J. Folkesson, H. McClelland, and J. J. Leonard. 2013. Relocating Underwater Features Autonomously Using Sonar-Based SLAM. *IEEE Journal of Oceanic Engineering* 38, 3 (July 2013), 500–513. https://doi.org/10.1109/JOE.2012.2235664

[18] Brian Ferris, Dieter Fox, and Neil Lawrence. 2007. WiFi-SLAM Using Gaussian Process Latent Variable Models. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence* (Hyderabad, India) *(IJCAI'07)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2480–2485. http://dl.acm.org/citation.cfm?id=1625275.1625675

[19] C. Forster, S. Lynen, L. Kneip, and D. Scaramuzza. 2013. Collaborative monocular SLAM with multiple Micro Aerial Vehicles. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 3962–3970. https://doi.org/10.1109/IROS.2013.6696923

[20] Zakieh Hashemifar and Karthik Dantu. 2020. Practical Persistence Reasoning in Visual SLAM. In *accepted to appear in Proceedings of the International Conference on Robotics and Automation (ICRA '20)*. IEEE, Paris, France.

[21] Zakieh Hashemifar, Kyung Won Lee, Nils Napp, and Karthik Dantu. 2018. Geometric Mapping for Sustained Indoor Autonomy. In *Proceedings of the 1st International Workshop on Internet of People, Assistive Robots and Things*. 19–24.

[22] Zakieh S Hashemifar, Charuvahan Adhivarahan, Anand Balakrishnan, and Karthik Dantu. 2019. Augmenting visual SLAM with Wi-Fi sensing for indoor applications. *Autonomous Robots* 43, 8 (2019), 2245–2260.

[23] Zakieh S Hashemifar, Kyung Won Lee, Nils Napp, and Karthik Dantu. 2017. Consistent cuboid detection for semantic mapping. In *2017 IEEE 11th International Conference on Semantic Computing (ICSC)*. IEEE, 526–531.

[24] W. Hess, D. Kohler, H. Rapp, and D. Andor. 2016. Real-time loop closure in 2D LIDAR SLAM. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 1271–1278. https://doi.org/10.1109/ICRA.2016.7487258

[25] J. Huang, D. Millman, M. Quigley, D. Stavens, S. Thrun, and A. Aggarwal. 2011. Efficient, generalized indoor WiFi GraphSLAM. In *2011 IEEE International Conference on Robotics and Automation*. 1038–1043. https://doi.org/10.1109/ICRA.2011.5979643

[26] Computer Vision Group in Department of Informatics at Technical University of Munich. 2020. Computer Vision Group - Dataset Download. https://vision.in.tum.de/data/datasets/rgbd-dataset/download.

[27] Apple Inc. 2020. Augmented Reality - Apple Developer. https://developer.apple.com/augmented-reality/.

[28] Magic Leap Inc. 2020. Magic Leap 1 | Magic Leap. https://www.magicleap.com/en-us/magic-leap-1.

[29] S. Ito, F. Endres, M. Kuderer, G. Diego Tipaldi, C. Stachniss, and W. Burgard. 2014. W-RGB-D: Floor-plan-based indoor global localization using a depth camera and WiFi. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*. 417–422. https://doi.org/10.1109/ICRA.2014.6906890

[30] Puneet Jain, Justin Manweiler, and Romit Roy Choudhury. 2016. Low Bandwidth Offload for Mobile AR. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies* (Irvine, California, USA) *(CoNEXT '16)*. ACM, New York, NY, USA, 237–251. https://doi.org/10.1145/2999572.2999587

[31] G. Klein and D. Murray. 2007. Parallel Tracking and Mapping for Small AR Workspaces. In *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*. 225–234. https://doi.org/10.1109/ISMAR.2007.4538852

[32] R. KÃijmmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard. 2011. G2o: A general framework for graph optimization. In *2011 IEEE International Conference on Robotics and Automation*. 3607–3613. https://doi.org/10.1109/ICRA.2011.5979949

[33] Mathieu Labbe and Francois Michaud. 2013. Appearance-based loop closure detection for online large-scale and long-term operation. *IEEE Transactions on Robotics* 29, 3 (2013), 734–745.

[34] M. LabbÃĪ and F. Michaud. 2013. Appearance-Based Loop Closure Detection for Online Large-Scale and Long-Term Operation. *IEEE Transactions on Robotics* 29, 3 (June 2013), 734–745. https://doi.org/10.1109/TRO.2013.2242375

[35] Fu Li, Shaowu Yang, Xiaodong Yi, and Xuejun Yang. 2018. CORB-SLAM: A Collaborative Visual SLAM System for Multiple Robots. In *Collaborative Computing: Networking, Applications and Worksharing*, Imed Romdhani, Lei Shu, Hara Takahiro, Zhangbing Zhou, Timothy Gordon, and Deze Zeng (Eds.). Springer International Publishing, Cham, 480–490.

[36] Luyang Liu, Hongyu Li, and Marco Gruteser. 2019. Edge Assisted Real-time Object Detection for Mobile Augmented Reality. In *The 25th Annual International Conference on Mobile Computing and Networking* (Los Cabos, Mexico) *(MobiCom '19)*. ACM, New York, NY, USA, Article 25, 16 pages. https://doi.org/10.1145/3300061.3300116

[37] P. Mach and Z. Becvar. 2017. Mobile Edge Computing: A Survey on Architecture and Computation Offloading. *IEEE Communications Surveys Tutorials* 19, 3 (thirdquarter 2017), 1628–1656. https://doi.org/10.1109/COMST.2017.2682318

[38] Microsoft. 2020. HoloLens (1st gen) hardware | Microsoft Docs. https://docs.microsoft.com/en-us/hololens/hololens1-hardware.

[39] P. Mirowski, T. K. Ho, Saehoon Yi, and M. MacDonald. 2013. SignalSLAM: Simultaneous localization and mapping with mixed WiFi, Bluetooth, LTE and magnetic signals. In *International Conference on Indoor Positioning and Indoor Navigation*. 1–10. https://doi.org/10.1109/IPIN.2013.6817853

[40] R. Mur-Artal, J. M. M. Montiel, and J. D. TardÃşs. 2015. ORB-SLAM: A Versatile and Accurate Monocular SLAM System. *IEEE Transactions on Robotics* 31, 5 (Oct 2015), 1147–1163. https://doi.org/10.1109/TRO.2015.2463671

[41] R. Mur-Artal and J. D. TardÃşs. 2017. ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras. *IEEE Transactions on Robotics* 33, 5 (Oct 2017), 1255–1262. https://doi.org/10.1109/TRO.2017.2705103

[42] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon. 2011. KinectFusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality*. 127–136. https://doi.org/10.1109/ISMAR.2011.6092378

[43] R. A. Newcombe, S. J. Lovegrove, and A. J. Davison. 2011. DTAM: Dense tracking and mapping in real-time. In *2011 International Conference on Computer Vision*. 2320–2327. https://doi.org/10.1109/ICCV.2011.6126513

[44] M. Quigley, D. Stavens, A. Coates, and S. Thrun. 2010. Sub-meter indoor localization in unmodified environments with inexpensive sensors. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2039–2046. https://doi.org/10.1109/IROS.2010.5651783

[45] L. Riazuelo, Javier Civera, and J.M.M. Montiel. 2014. C2TAM: A Cloud framework for cooperative tracking and mapping. *Robotics and Autonomous Systems* 62, 4 (2014), 401 – 413. https://doi.org/10.1016/j.robot.2013.11.007

[46] M. Satyanarayanan. 2017. The Emergence of Edge Computing. *Computer* 50, 1 (Jan 2017), 30–39. https://doi.org/10.1109/MC.2017.9

[47] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. 2009. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing* 8, 4 (Oct 2009), 14–23. https://doi.org/10.1109/MPRV.2009.82

[48] P. Schmuck and M. Chli. 2017. Multi-UAV collaborative monocular SLAM. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*. 3863–3870. https://doi.org/10.1109/ICRA.2017.7989445

[49] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 3, 5 (Oct 2016), 637–646. https://doi.org/10.1109/JIOT.2016.2579198

[50] Sebastian Thrun et al. 2002. Robotic mapping: A survey. *Exploring artificial intelligence in the new millennium* 1, 1-35 (2002), 1.

[51] T. Whelan, H. Johannsson, M. Kaess, J. J. Leonard, and J. McDonald. 2013. Robust real-time visual odometry for dense RGB-D mapping. In *2013 IEEE International Conference on Robotics and Automation*. 5724–5731. https://doi.org/10.1109/ICRA.2013.6631400

[52] Jingao Xu, Hao Cao, Danyang Li, Kehong Huang, Chen Qian, Longfei Shangguan, and Zheng Yang. 2020. Edge Assisted Mobile Semantic Visual SLAM. (July 2020).