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

ALI J. BEN ALI, Binghamton University, USA
MARZIYE KOUROSHLI, University at Buffalo, USA
SOFIYA SEMENOVA, University at Buffalo, USA
ZAKIEH SADAT HASHEMIFAR, Zoox, USA
STEVEN Y. KO, Simon Fraser University, Canada
KARTHIK DANTU, University at Buffalo, USA

Localization in urban environments is becoming increasingly important and used in tools such as ARCore [18], ARKit [34] and others. One popular mechanism to achieve accurate indoor localization and 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 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 [50] 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 closing, 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 the deployment of other end applications that use Visual-SLAM. We perform a detailed performance and resources use (CPU, memory, network, and power) analysis to fully understand the effect of our proposed split architecture.

CCS Concepts: • **Computer systems organization → Cloud computing**; **Client-server architectures**; **Embedded systems**; **Real-time system architecture**; • **Human-centered computing → Ubiquitous and mobile computing**; • **Computing methodologies → Multi-agent systems**; **Computer vision**.

Additional Key Words and Phrases: visual simultaneous localization and mapping, edge computing, split architecture, mobile systems, localization, mapping, concurrency

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.

Authors' addresses: Ali J. Ben Ali, abenali@binghamton.edu, Binghamton University, 4400 Vestal Parkway East, Binghamton, New York, USA, 13902; Marziye Kouroshli, marziye.kouroshli@gmail.com, University at Buffalo, 106 Davis Hall, Buffalo, New York, USA, 14260; Sofiya Semenova, sofiyase@buffalo.edu, University at Buffalo, 106 Davis Hall, Buffalo, New York, USA, 14260; Zakieh Sadat Hashemifar, zhashemifar@zoox.com, Zoox, 1149 Chess Drive, Foster City, California, USA, 94404; Steven Y. Ko, steveyko@sfu.ca, Simon Fraser University, 8888 University Drive, Burnaby, British Columbia, Canada, V5A1S6; Karthik Dantu, kdantu@buffalo.edu, University at Buffalo, 106 Davis Hall, Buffalo, New York, USA, 14260.

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 closing, 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 [22] such as the use of a short-term and long-term memory [39], or clustering using other sensing [28]. 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 [43, 57, 58, 60]. 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 adapt ORB-SLAM2 [50], a popular Visual-SLAM system, to the edge computing architecture. Our system is called Edge-SLAM.
- Our design accomplishes this adaptation by designing a novel data structure called *Local-Map*. This mapping structure allows us to decouple the Tracking and Local-Mapping tasks, thereby improving Local-Mapping and Loop-Closing efficiency without compromising the functionality of the Tracking module.
- We evaluate Edge-SLAM on a prototypical mobile device using two large custom datasets as well as two open-source datasets (TUM [33]).
- We improve the map update implementation on the mobile device to perform faster updates and smoother tracking. These improvements reduce the map update reconstruction latency on the mobile device by 19% compared to the original Edge-SLAM [7] implementation.
- Implement a map update controller interface on the mobile device to provide control over the updates to the user and/or motion planner algorithm. Such interface has significantly improved experimentations with Edge-SLAM by reducing the chances of losing track due to previously known scenarios such as sharp turns.

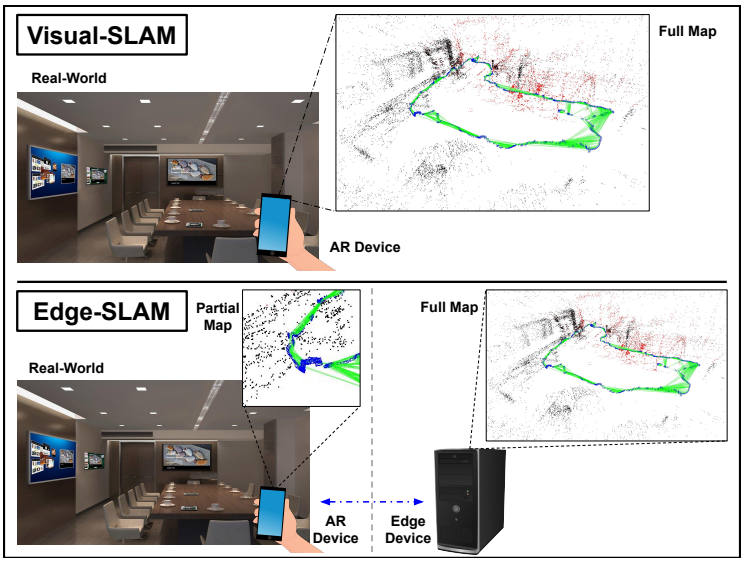

Fig. 1. Visual-SLAM vs. Edge-SLAM. An augmented reality device running Visual-SLAM (top), and an augmented reality device running Edge-SLAM in collaboration with an edge device in the environment (bottom) [1–3, 33, 50]

- We perform a complete resource usage analysis including CPU, memory, and network (latency and bandwidth) usage as well as power consumption.
- 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. Figure 1 shows the difference between a current Visual-SLAM system and Edge-SLAM when run on a mobile device. The mobile device in Edge-SLAM works with a partial map of the environment and is assisted by an edge 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

2.1 Edge/Cloud Offloading

The area of edge computing has been the topic of research for the last decade [43, 57, 58, 60]. 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 [17] and CloneCloud [14] perform cloud offloading of tasks at various granularities. MARVEL [11], VisualPrint [36] and [41] present application-specific techniques for offloading to the

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

edge or the cloud. These papers work on decreasing offload latency or masking it from the end user in time-sensitive applications.

In the cloud robotics area, several studies have looked into simplifying the integration between the robots and the cloud. Rapyuta [47] is a cloud platform to enable computation offloading for robots. [5] proposes a multi-query motion planning system that uses serverless functions on the cloud. [13] offers an offloading strategy using deep reinforcement learning to help a robot make an informed decision on when it is best to offload a computation. PoundCloud [44] is a communication framework to simplify the integration of robots and the cloud. DewROS [8] is a communication architecture to distribute the computation among the robot, the network device, and the cloud.

2.2 Simultaneous Localization and Mapping (SLAM)

Simultaneous localization and mapping (SLAM) has been a topic of research in robotics and mobile systems for several decades [6, 20, 61]. Initial research focused on depth sensors such as sonar [23] and 2-D LiDAR [16, 19]. Other sensors such as Wi-Fi signal strength have been used for SLAM as well [24, 31, 46].

Visual-SLAM has grown rapidly in the last decade [22, 30]. This includes the use of RGB cameras, RGB-D cameras, and LiDAR sensors. Systems such as PTAM [37], DTAM [52], LSD-SLAM [21] used monocular cameras for SLAM. A recent trend has been the use of color images with depth images, as well as using point and line features [53, 69]. Some more well-known Visual-SLAM examples include RGBD-SLAM [22], RTAB-Map [39], ORB-SLAM [10, 49, 50], and VINS-Mono [54]. They build on initial work from systems such as Kinect Fusion [51], and Kintinuous [64] that first used RGB-D sensors for 3-D modeling of environments. Current trends improve on basic Visual-SLAM by reasoning about semantics [27, 29, 67], reasoning about object permanence in maps [26], and introducing deep learning techniques to replace or augment various parts of the traditional Visual-SLAM pipeline [45, 62].

More recently, there is increased interest in the use of multiple sensors to perform SLAM. [63] combines event camera frames, traditional camera images, and IMU sensor data for accurate visual SLAM in high dynamic range and low light scenes. Several recent works combine Wi-Fi with visual sensing for improved SLAM. In [35], 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. [55] utilizes a training phase for Wi-Fi modeling and then applies particle filters for fusing different sensors. [28] 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.

2.3 Collaborative SLAM

Collaborative SLAM has been explored in recent works through combining edge/cloud computing with SLAM systems in different ways. [59] built a collaborative monocular SLAM on top of ORB-SLAM2 [50] 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. [25] 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. [56] describes C2TAM which is a collaborative SLAM framework that is built on top of PTAM [37]. 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 [40], the authors present CORB-SLAM, which is a multi-robot SLAM system built on top of ORB-SLAM2 [50]. 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. [48] presents a collaborative mapping system using robots. In this system, each robot maintains a local map and sends a copy of that map to the cloud. The cloud merges and optimizes the local maps received from the robots into a global map. The cloud would then push to each robot an optimized version of its local map. 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. [15] introduces a decentralized Visual-SLAM system through the integration of multiple decentralized SLAM components. However, unlike Edge-SLAM and the previously discussed work, their system does not depend on a singular, central edge or cloud device.

2.4 SLAM Offloading

Some studies have run the entire SLAM pipeline on the cloud. FogRos [12] and FogRos2 [32] are cloud platforms to enable deploying robot systems on the cloud. They were used to deploy ORB-SLAM2 on the cloud. For this, a robot was used to transfer camera frames to the cloud. On the cloud, the full ORB-SLAM2 pipeline was running to build and maintain a map of the robot environment. Few recent studies have considered splitting the SLAM pipeline between a mobile device and an edge or a cloud device. One such study is edgeSLAM [66], which 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 [49]. 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 usage (CPU, memory, network, and power) or overhead of synchronizing the edge and mobile devices. Further, incorporating semantic segmentation makes their design harder to generalize across other Visual-SLAM systems. Another study, CloudSLAM [65], proposes a split architecture for cloud-assisted Visual-SLAM system for autonomous driving. This study aims to minimize network usage of the system while preserving an acceptable level of consistency between the vehicle and the edge device. The authors claim that the level of consistency in the

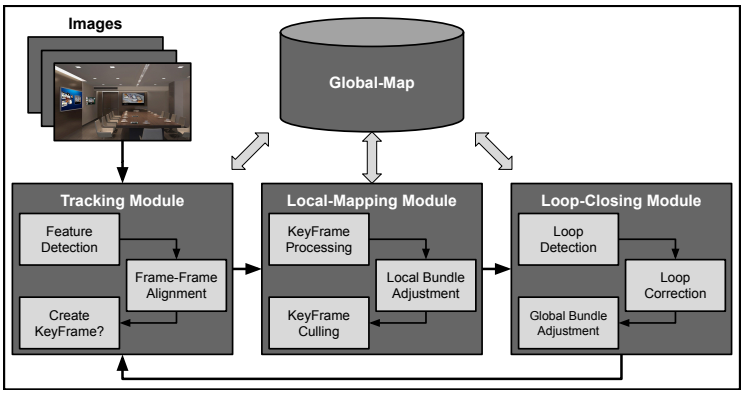

Fig. 2. Architecture of a typical Visual-SLAM system [2]

system is enough to calculate an accurate trajectory. In contrast, Edge-SLAM is built on top of ORB-SLAM2 [50] (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 (CPU, memory, network, and power) on the mobile device, and our work extensively evaluates these aspects of the implementation.

3 SYSTEM DESIGN

3.1 Overview of a Typical Visual-SLAM System

Shown in Figure 2 is an architecture diagram of a typical feature-based Visual-SLAM system. Several SLAM systems adhere roughly to this architecture including PTAM [37], LSD-SLAM [21], ORB-SLAM [49], ORB-SLAM2 [50], and ORB-SLAM3 [10]. 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 three main modules described below.

3.1.1 Tracking Module. 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.

3.1.2 Local-Mapping Module. If Tracking module 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.

3.1.3 Loop-Closing Module. 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 [38].

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. This has been the research topic for several Visual-SLAM systems such as RTAB-Map [39] which uses long-term and short-term memory to reduce this complexity. 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, 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 [50] 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 8, 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 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 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

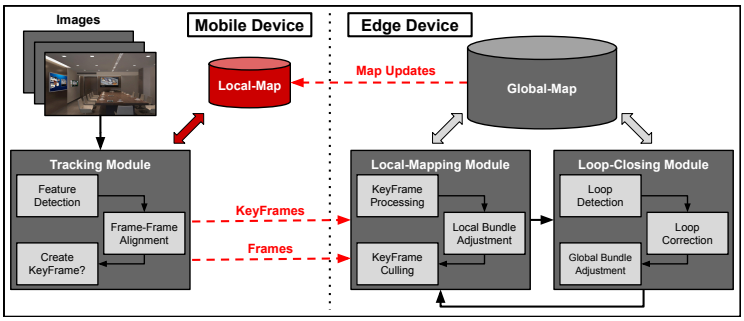

Fig. 3. Envisioned architecture of the Edge-SLAM system—our modifications are shown in red [2]

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, 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. 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. The second connection is used to pass the KeyFrame if Tracking module decided to create a new KeyFrame, and is labeled *KeyFrames* in Figure 3. The third connection is used to update the Local-Map. This is shown in the top portion of Figure 3 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.

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

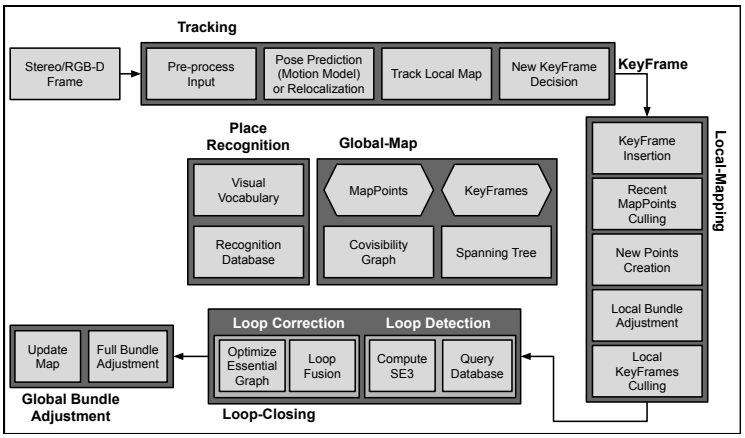

Fig. 4. ORB-SLAM2 system architecture [50]

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.

3.3.3 Adapting Edge-SLAM Architecture in Visual-SLAM Systems. To demonstrate the feasibility of our Edge-SLAM architecture, we prototyped our idea on ORB-SLAM2 [50], 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. For example, to apply Edge-SLAM architecture to VINS-Mono [54], we can map VINS's Measurement Preprocessing and Initialization module to Edge-SLAM's Tracking module on the mobile device. Then, we map VINS's Local Visual-Inertial Odometry with Relocalization module to Edge-SLAM's Local-Mapping module on the edge device. Finally, we map VINS's Global Pose Graph Optimization and Reuse module to Edge-SLAM's Loop-Closing module on the edge device.

Other non-Visual-SLAM systems such as LiDAR-SLAM might be less applicable to our Edge-SLAM architecture depending on the particular system pipeline. For example, LOAM [68] is a LiDAR-based mapping system where the pipeline components can not be directly mapped to Edge-SLAM components. For such systems, further exploration of the details is necessary.

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.

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 [50]

4.1 ORB-SLAM2

4.1.1 Overview. ORB-SLAM2 [50] 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.

ORB-SLAM2 consists of three threads, one per module: Tracking, Local-Mapping, and Loop-Closing as shown in Figure 4. 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 [49], and the fifth in the second paper [50]. 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 MapPoints 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-Closing 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 *MapPoints*. A KeyFrame, as described above, is an image frame that contains unique segment or viewpoint of the environment. A MapPoint stores the position, feature descriptor, and references to all KeyFrames that observe it. In ORB-SLAM2, the MapPoints are obtained by detecting the ORB-features in an image. Therefore, the feature descriptors are ORB-descriptors. Each KeyFrame points to several MapPoints. Also, multiple KeyFrames could point to the same MapPoints if the same features are seen from multiple KeyFrames. Together, the set of all KeyFrames currently constructed, and all their observed MapPoints form the current Global-Map. These three data structures depend on each other, and maintain several additional information—details can be found in [49, 50]. 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 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.

4.2 Edge-SLAM Implementation

Figure 5 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.

4.2.1 Global-Map. The Global-Map is created and stored on the edge. This contains the complete set of KeyFrames, set of MapPoints, the Co-Visibility Graph, and the Spanning Tree. The Co-Visibility Graph connects KeyFrames based on their shared MapPoints observations. The Spanning

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

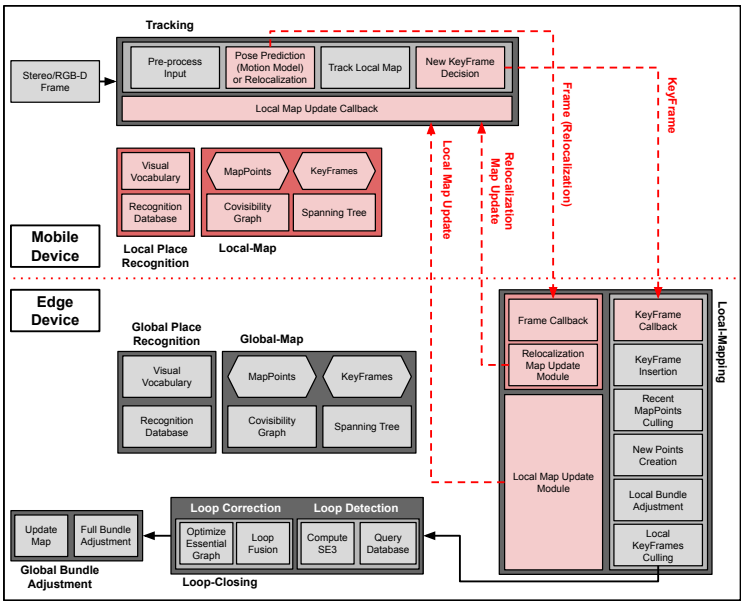

Fig. 5. Edge-SLAM system architecture—our modifications are shown in red

Tree is a subset of the Co-Visibility Graph connecting every KeyFrame with the KeyFrame that they share the most MapPoints.

4.2.2 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, MapPoints, 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.

4.2.3 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 MapPoints 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 Local-Map updates or relocalization map updates from the edge to the mobile device. Because each connection is running on a separate thread and in a nonblocking/simultaneous fashion with other working threads, we implemented three callback sub-modules as shown in Figure 5. *Local-Map Update Callback* in the Tracking thread on the mobile device. *Frame Callback* and *KeyFrame Callback* in the Local-Mapping thread on the edge. Upon successful transmission of a Frame/KeyFrame/Map-Update object, the corresponding callback sub-module will be triggered to process the incoming

data. 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), 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 MapPoints, 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, the Tracking thread 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:

$$TC = ITC/((KFN/LMU) + 1) \tag{1}$$

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. We also discuss some additional measures we take to address this challenge in Section 4.7.

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

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

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-Closing 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 *Local-Map 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 MapPoints. 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 MapPoints in the Local-Map. Such KeyFrames and MapPoints will also contribute to creating newer KeyFrames and MapPoints 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 MapPoint 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 ORB-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. In this subsection, we first discuss Edge-SLAM Parameters, then we discuss the Local-Map Update Callback sub-module in the Tracking thread.

Edge-SLAM Parameters. Edge-SLAM has 4 parameters that have been selected after careful and extensive testing. The parameters are:

- **Local-Map Update Size (discussed in Section 4.7.3)** this is set to six KeyFrames, which is the minimum number of KeyFrames required for the system to operate in the normal mode. Otherwise, the system would run in the initialization mode. This parameter is so sensitive as it directly affects the mobile device performance. Increasing the size of the Local-Map update will increase the time required to freeze the Tracking module to apply the map update and consequently increase the chances of the mobile device losing track. Therefore, we set this parameter to the minimum possible number of KeyFrames to keep the map update overhead at the lowest rate.
- **Local-Map Update Frequency (discussed in Section 4.5)** this is set to five seconds, such that small map updates are sent frequently in short time intervals from the edge device. This parameter indirectly affects the mapping accuracy of the system. Increasing the map update frequency will increase the network usage but may help reduce the mapping drifts on the mobile device. On the other hand, reducing the map update frequency will reduce the network usage but may increase the mapping drifts since the mobile device will work more with unoptimized mapping data. Therefore, this parameter may be adjusted as necessary depending on the conditions under which the system has to run.
- **Relocalization Frame Frequency (discussed in Section 4.7.5)** this is set to half a second, such that one frame is sent every half a second from the mobile device to the edge device to receive map assistance for relocalization. This time interval would prevent the mobile device from sending additional redundant frames to the edge device. Typically, cameras produce 30 frames per second. This parameter defines how frequently a frame is sent from the mobile device to the edge device when the system is in relocalization mode. Increasing the frequency will increase the network usage but may also increase the chances of successful relocalization if the scene changes rapidly; otherwise, the sent frames will be redundant. Reducing the frequency, on the other hand, will reduce the network usage but may also reduce the chances of successful relocalization. Therefore, this parameter can be adjusted as necessary depending on the conditions under which the system has to run.
- **Time Constraint to Accept a Local-Map Update** 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. When selecting an initial time constraint value, our objective was to control how big the Tracking Local-Map can get before it is updated, especially during a high KeyFrame creation rate. Also, we wanted to allow the mobile device to work independently for short periods 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. We found through experiments that working with 50 unoptimized KeyFrames in the Local-Map is a fair number before needing a map update. The "Local-Map Update Frequency" parameter

discussed above defines how frequently a map update is sent from the edge device. The "Time Constraint to Accept a Local-Map Update" parameter defines how frequently a map update is accepted on the mobile device. This parameter is sensitive to changes as it directly affects how the mobile device behaves in different scenarios. Increasing the time constraint will reduce the frequency of accepted map updates, leading to mapping drifts. However, this may still help in challenging mapping scenarios by preventing a map update from being applied when the scene is rapidly changing. Reducing the time constraint, however, will increase the frequency of accepted map updates which may increase the chances of losing track soon after the system gets into any challenging scenario. Therefore, careful consideration is necessary when setting this parameter.

Our experiments show that the above parameters work with most scenarios and datasets without adjusting them. Further, Edge-SLAM is sensitive to ORB-SLAM2 parameters. Thus, not setting such parameters correctly would affect the system's performance.

Local-Map Update Callback. The Local-Map Update Callback submodule in the Tracking thread is responsible for processing and reconstructing the Local-Map when an update is received from the edge. Before processing an incoming update, the submodule would first check if all the following conditions are true:

- The map update controller (discussed in Section 4.7.2) did not disable map updates.
- The system is not in initialization mode.
- The Local-Map has changed since the last applied map update.
- The current KeyFrame creation rate is not high.

If the above conditions are true, the submodule pauses the Tracking thread from processing incoming camera frames. It would then fully erase the Local-Map on the mobile device. Since a copy of every new KeyFrame is sent to the edge device on creation, no information will be lost when the Local-Map is erased. Next, the received map update is used to rebuild the Local-Map. Once the map is rebuilt and all connections have been established, the submodule will resume the Tracking thread to process the incoming frames. The Tracking thread processes the latest camera frame when the frame processing resumes. Since the latency of updating the map is small, the amount of change in the scene (i.e., when comparing the last processed frame before the update and the first processed frame after the update) is typically accepted by the system, which allows tracking to continue without issues. Occasionally, applying a map update would cause tracking to be lost, but that only happens due to unforeseen circumstances, as discussed in Section 4.7.2.

4.7.2 Map Update Controller Interface. After testing ORB-SLAM2 and Edge-SLAM with various datasets, we have observed that the mapping pipeline is sensitive to the following scenarios:

- **Quick Rotation** is when the speed at which the mobile device changes direction is more than what the system can handle, i.e., the system cannot track the changes in the scene during the rotation.
- **Sharp Turn** is when instead of the mobile device turning gradually while moving forward, it makes the turn in place, resulting in a sharp angle turn. Such a turn would most likely result in a totally different scene from the one before the turn.
- **Shaking Scene** is when the mobile device shakes while moving at a degree that would result in a blurry scene that is not visible enough to extract features.
- **Featureless Scene** there are moments where the environment that the mobile device is navigating has very few features, which may result in difficulties for the system to continue tracking scene changes.

- **Dynamic Scene** is when the scene includes some non-static moving objects such as a walking person or a moving car. Such dynamicity can cause a rapid change in the scene, making it difficult to track.

When such movements happen, the system loses track and needs to perform relocalization or restart its map. This is because the system needs to compute the amount of change in the scene after processing every single frame. The scene change is calculated by extracting features from the current frame and matching those features to the features of the last created KeyFrame. For the tracking to persist, the number of matched features after processing each frame must meet a preset threshold in ORB-SLAM2 [49, 50]. If the threshold is not met, the tracking will be lost. Therefore, any challenging movement, such as the ones mentioned above, would lead to the same outcome, losing track if the minimum number of feature matches is not found. Because continuous tracking is time-consuming and computationally heavy, losing track chances significantly increases when an unforeseen movement happens.

Further, in Edge-SLAM, when the mobile device receives a map update, it needs to freeze processing the incoming camera frames while it reconstructs an updated map. Such updates could take hundreds of milliseconds and affect the system's trackability. The process of freezing the tracking and constructing the map is particularly exaggerated if the mobile device is experiencing any of the above scenarios, which can pull the number of matched features in the scene to zero, causing tracking to be lost.

To address this, we take the following measures in Edge-SLAM to reduce the chances of losing track due to the scenarios mentioned above:

- We reconsider the data structures used in ORB-SLAM2 to maintain the map's KeyFrames and MapPoints and then restructure the map using other data structures that would enable fast retrieval. One such data structure that we have decided to use is the `unordered_map` data structure from the C++ Standard Template Library (STL), which is based on hash tables and can, in most cases, retrieve an item in a constant time. This is particularly important because the map in ORB-SLAM2 is highly connected. Each KeyFrame object should have a reference to each of its MapPoints, and each MapPoint object, on the other hand, should reference each KeyFrame that observes it. In ORB-SLAM2, each KeyFrame could have hundreds of MapPoints, and each MapPoint could be observed by multiple KeyFrames. When a map update is received in Edge-SLAM, the Local-Map should be reconstructed from the update by looping through the KeyFrames and MapPoints to rebuild the map connections. Such operation would require at least quadratic time complexity when used with the map data structures in ORB-SLAM2 to reconstruct the map. To address this, we use `unordered_map`, a hash-based data structure to map each KeyFrame and each MapPoint to its corresponding unique identification value. This way, every KeyFrame, and MapPoint can be retrieved on average in a constant time, and the map rebuild operation will take a linear time complexity on average. Our objective in restructuring the map is to reduce the time it takes to rebuild the Local-Map from an update on the mobile device and consequently reduce the time needed to freeze the Tracking module.
- We implement a map update controller interface for the mobile device to enable or disable map updates at specific moments. Our rationale behind the controller is that several undesirable movements such as those mentioned above might be known to the user or other algorithms such as motion planning. The controller would enable a user or an algorithm to use such knowledge to prevent a map update from causing tracking loss. The controller has a graphical user interface (GUI) and a programming interface.

The mobile device's environment can be highly dynamic, which can introduce many navigation challenges beyond a single system's ability to understand and properly handle. Therefore, having a map update controller interface on the mobile device for the user and other algorithms can improve the mapping quality and assist in correctly addressing navigation challenges that can cause unnecessary tracking loss.

For example, the planner could use the controller on a robot to indicate an upcoming sharp turn by turning off the map updates. The map update controller should be toggled to disable map updates on the mobile device before receiving a new update from the edge device during an unwanted scenario. The controller can then re-enable map updates as soon as the undesirable scenario is over.

4.7.3 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.

4.7.4 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.

4.7.5 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 using the *Relocalization Map Update Module*, as shown in Figure 5, 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 5.3.2.

5 EVALUATION

To study Edge-SLAM, we conduct several experiments to demonstrate the performance as well as overheads from our design. We use a setup that mimics that of the modern mobile devices to

study the performance. We use two datasets collected at University at Buffalo along with two publicly available datasets. The collected datasets are much longer than the publicly available ones, allowing us to demonstrate the performance of Edge-SLAM for more extended periods. We study the mapping accuracy, performance, memory use, CPU use, network use, and power consumption. Finally, we look at the impact of variable network bandwidth on Edge-SLAM performance by using a network shaper tool and limiting the upload/download bandwidth available.

5.1 Experiment Setup

To evaluate Edge-SLAM, we run experiments using two 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, 8GB Memory, and Connects using 802.11ac WLAN—running Ubuntu 18.04LTS. We denote this device as **JTX2**. The second mobile device is a DELL Latitude laptop—Intel Core i7-7600U, Intel HD Graphics, 16GB Memory—running Ubuntu 18.04LTS. We denote this device as **LAPT**. The edge device is a DELL XPS desktop—Intel Core i7 9700K, NVIDIA GeForce GTX 1080, 32GB Memory—running Ubuntu 18.04LTS. We denote this device as **DESK**. In this section we evaluate our system Edge-SLAM and compare it to ORB-SLAM2. In ORB-SLAM2 experiments, only a mobile device is required, so we either use JTX2 or LAPT. In Edge-SLAM experiments, a mobile device and an edge device are needed, and for that, we use either JTX2-DESK or LAPT-DESK.

We use two pre-collected RGB-D datasets of our campus building floors as the input source for long-running experiments. Our datasets are collected using a robot equipped with a Kinect 360 RGB-D sensor and a Velodyne VLP-16 LiDAR for the ground truth. Our first and primary dataset consists of 52,427 frames, runs for a total of 1,774 seconds (≈30 minutes), and has a trajectory of ≈155 meters. We denote this dataset as **D1**. Our second dataset consists of 39,374 frames, runs for a total of 1,315 seconds (≈22 minutes), and has a trajectory of ≈140 meters. We denote this dataset as **D2**. We also use two popular public datasets from the Technical University of Munich called TUM [33] RGB-D datasets for short-running experiments. The first TUM dataset is called *freiburg2_pioneer_slam2*, which we denote as **D3**. The second TUM dataset is called *freiburg2_pioneer_slam3*, which we denote as **D4**. All the datasets frames are read from storage by the mobile device and published as ROS topics[4] for consumption by either ORB-SLAM2 or Edge-SLAM.

We repeat the experiments on each platform by replaying the dataset frames as if they were being collected. This allows us to provide an exact comparison of performance across different platforms and is a standard mechanism to compare performance across SLAM systems [9]. The speed at which the robot traversed D1 and D2 datasets are not constant. Due to the campus building corridor dynamics at the time of the dataset collection, the robot had to make several stops in different parts of the datasets. Therefore, the speed cannot be calculated purely from the distance and duration of the dataset. A typical frame rate of the Kinect is between 15-30fps. Due to the sensitivity of the Tracking module discussed in Section 4.7.1 and Section 4.7.2, we replay our datasets for the long-running experiments using 15fps and use the map update controller at turns as necessary to reduce the chances of losing track. Note that this is a shortcoming of ORB-SLAM2, and fixing this was beyond the scope of this work. We use the same configurations to evaluate ORB-SLAM2 and Edge-SLAM in all the long-running experiments results presented below. For the short-running experiments, we use the public datasets, which we replay using 30fps for all platforms.

The mobile devices are connected to a private campus Wi-Fi network (Download=84Mbps, Upload=92Mbps). The edge is connected to a private campus network through a wired connection (Download=92Mbps, Upload=93Mbps), emulating an actual deployment.

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

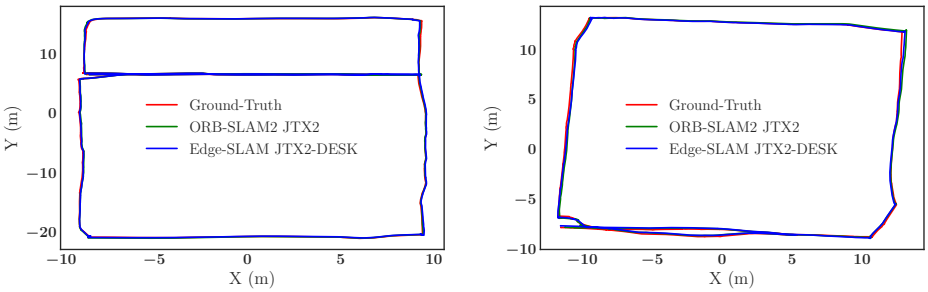

Fig. 6. ORB-SLAM2 and Edge-SLAM trajectories mapping D1 (left) and D2 (right) datasets compared to the ground-truth. ORB-SLAM2 is running on JTX2, and Edge-SLAM is running on JTX2 and DESK

Accuracy Measure \ Visual-SLAM	ORB-SLAM2 JTX2	Edge-SLAM JTX2-DESK
Mean Localization Error for D1 Dataset (cm)	19.08 ±9.4	18.69 ±8.55
Mean Localization Error for D2 Dataset (cm)	21.36 ±10.46	18.94 ±7.53

Table 2. Mean Localization Error of ORB-SLAM2 and Edge-SLAM mapping D1 and D2 datasets

5.2 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. This subsection will compare the localization and mapping accuracy of ORB-SLAM2 and Edge-SLAM with a ground-truth trajectory. For this, we perform four experiments. The first two experiments map D1 and D2 datasets using ORB-SLAM2 running on a mobile device (JTX2). The second two experiments map D1 and D2 datasets using Edge-SLAM running on a mobile device (JTX2) and an edge device (DESK).

The 2-D trajectories of the path traced by the mobile device (JTX2) as constructed by ORB-SLAM2 and Edge-SLAM are shown in Figure 6. 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 for both datasets in Table 2. Edge-SLAM performs slightly better on both datasets with a difference of ≈1cm (0.01m) on average on a trajectory of ≈150m. For most applications, this is quite acceptable for the feasibility of deploying accurate localization and mapping long-term on mobile devices.

Also, ORB-SLAM2 mapping is highly accurate. From the results in Table 2, we see that it only drifts for ≈20cm (0.2m) after traveling ≈150m, which is small. Inherent latency, splitting the architecture, and working with a smaller map (Local-Map) tend to increase Edge-SLAM's potential drift/error. However, our results show that the drift of running Edge-SLAM is similar to ORB-SLAM2, i.e., ≈19cm (0.19m), after traveling the same trajectory length. This is because, in ORB-SLAM2 and Edge-SLAM, full bundle adjustment is performed after every loop closure. This process optimizes the Global-Map to reduce the drift. In our experiments, the D1 dataset has two loops, and the D2 dataset has one loop, where each system got to run full bundle adjustment at least once in each dataset. However, in Edge-SLAM, the full bundle adjustment gets to run on the edge device with

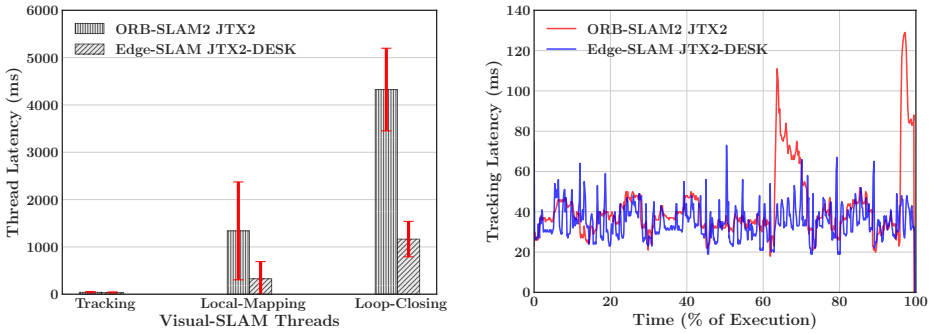

Fig. 7. Overall latency of ORB-SLAM2 and Edge-SLAM while mapping D1 dataset on JTX2. The average latency per-module (left) shows that Edge-SLAM offloads the two CPU-intensive tasks. Tracking module latency on the mobile device (JTX2) over time (right) better shows the latency for that module in each system

more computing power and lower latency, which would help achieve either the same or better accuracy than ORB-SLAM2.

5.3 Performance

In this subsection, we measure the overall performance of ORB-SLAM2 and Edge-SLAM. We look at the latency of the modules as well as the latency of the relocalization operation. Our goal is to determine how the split architecture has affected the working of the Visual-SLAM pipeline.

5.3.1 Module Latency. In this subsection, we present the results of two experiments. In the first experiment, we map our D1 dataset using ORB-SLAM2 running on JTX2. In the second experiment, we map our D1 dataset using Edge-SLAM running on JTX2 as the mobile device and DESK as the edge. If not specified, all the results are averaged over our dataset. As a reminder, the two goals of Edge-SLAM are 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 entirely on the mobile device (ORB-SLAM2) and running Visual-SLAM with edge offloading (Edge-SLAM).

Shown in Figure 7 (left) are the average times taken (in ms) to run the individual modules (Tracking, Local-Mapping, and Loop-Closing) in each of the two configurations. As seen from Figure 7 (right), the Tracking thread takes less than 60ms 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, except for the Tracking module in Edge-SLAM interacting with the Local-Map, which does not have any significant performance impact. However, when a loop is detected and closed, at ≈65% and ≈95% execution time, we observe that the Tracking latency on ORB-SLAM2 goes up to ≈120ms. In contrast, the Tracking latency on Edge-SLAM continues working at the same rate given the loop closure operation is performed on the edge side. Further, we see a large difference between the execution time for Local-Mapping as well as Loop-Closing modules.

We would like to make two observations regarding 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 the Loop-Closing 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

Relocalization \ Visual-SLAM	ORB-SLAM2 JTX2	Edge-SLAM JTX2-DESK
D3: # of Successful Relocalization	8	2
D3: Relocalization Latency (ms)	32.13 ±2.91	30.5 ±1.5
D4: # of Successful Relocalization	3	8
D4: Relocalization Latency (ms)	24 ±1.15	37 ±5.26

Table 3. Relocalization statistics for ORB-SLAM2 and Edge-SLAM on JTX2 mapping D3 and D4 public datasets

first place. *This accomplishes our first objective of reducing the overall computational load on the mobile device.*

5.3.2 Relocalization Latency. Relocalization is an essential feature of Visual-SLAM systems. It enables the system to resume mapping after losing track due to sudden movement. This subsection shows that the split architecture in Edge-SLAM has minimal effect on the relocalization performance. We map two challenging TUM [33] RGB-D public datasets, D3 and D4, at 30fps using ORB-SLAM2 and Edge-SLAM. We use JTX2 as our mobile device and DESK as our edge device. The camera in these datasets heavily shakes multiple times while in motion causing both systems to lose track and try to relocalize.

Table 3 shows that both systems successfully relocalize multiple times in each of the two datasets. Successful relocalization mainly depends on how well the system is performing with respect to feature extraction and matching. This is why each system might relocalize better or worse than the other system in each dataset. However, since the Tracking module in Edge-SLAM uses a Local-Map, it reasons with fewer KeyFrames and features than ORB-SLAM2; therefore, it has a slightly higher chance of losing track compared to ORB-SLAM2.

Table 3 also shows that the split architecture of Edge-SLAM has minimal effect on the relocalization latency. When our system loses track, it tries to relocalize using the existing Local-Map. In the meantime, it requests relocalization assistance from the edge. From the table, we observe that, on average, the relocalization latency on both systems is less than ≈40ms. ORB-SLAM2 takes on average 28ms, and Edge-SLAM takes on average 34ms. The additional overhead of Edge-SLAM comes from serializing a frame every half a second to send it to the edge to get assistance.

5.4 CPU and Memory Usage

Figure 8 shows the CPU and memory usage on the mobile device (JTX2) while running ORB-SLAM2 and Edge-SLAM to map the D1 dataset. This is our next significant result that we demonstrate with our system.

Figure 8 (left) shows the instantaneous CPU usage through the execution on the mobile device (JTX2). On average, the CPU usage for ORB-SLAM2 is at ≈30% while using the JTX2. In comparison, the CPU usage is at ≈15% when using the JTX2 for Edge-SLAM. Overall, there is ≈50% reduction in CPU use while using Edge-SLAM due to offloading.

Figure 8 (right) shows the memory usage on the mobile device (JTX2) 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 ≈65% and ≈95% execution time). This

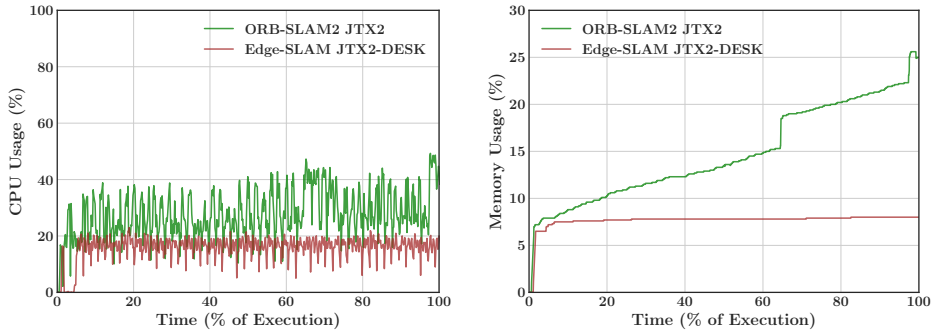

Fig. 8. Resource usage of ORB-SLAM2 and Edge-SLAM on the mobile device (JTX2) while mapping D1 dataset—CPU (left) and Memory (right). The jumps in memory use at 65% time and 95% time (right) are due to loop closures in ORB-SLAM2

Measurement \ Visual-SLAM	Edge-SLAM JTX2-DESK without MUCI	Edge-SLAM JTX2-DESK with MUCI
Keyframe Transmit Latency from Mobile Device to Edge (ms)	162.43 ±2.90	97.03 ±3.58
Construct Map Update Latency on Edge (ms)	57.09 ±0.69	61.82 ±0.64
Map Update Publish Frequency on Edge (s)	8.24 ±0.48	8.76 ±0.57
Map Update Acceptance Frequency on Mobile Device (s)	9.16 ±0.55	9.62 ±0.65
Re-Construct Map Update Latency on Mobile Device (ms)	411.43 ±4.84	335.72 ±1.79
Mean Localization Error for D1 Dataset (cm)	19.23 ±11.32	18.69 ±8.55

Table 4. Map update latencies and frequencies on the mobile device (JTX2) and on the edge (DESK) while mapping the D1 dataset using Edge-SLAM. We provide measurements for Edge-SLAM with and without Map Update Controller Interface (MUCI). The measures for the "without MUCI" are based on [7]

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, which is performed on the edge. *This accomplishes our second objective of keeping the resource use on the mobile device constant.*

5.5 Network Usage

5.5.1 Latency. As described in Section 5.1, we use a regular on-campus wireless network to connect the mobile device with the edge device. We also performed our experiments during normal working hours when the access points are used by other users. We did so to understand the network latency imposed in a typical urban setup and its effect on the Edge-SLAM system. In Figure 3, we show three links between the mobile and the edge device. We characterize the delay on these links for Edge-SLAM running on the mobile device (JTX2) and the edge (DESK) while mapping the D1 dataset.

The biggest source of delay is in the map update from the Global-Map on the edge to the Local-Map on the mobile device. This latency is shown in Table 4 (with MUCI). Totally, there are three parts to this latency. First is the latency to construct the map update on the edge. Second to transmit the map update across the network. Third, once the update is received, the mobile device needs to reconstruct its Local-Map using the received map update. The table shows that preparing the map

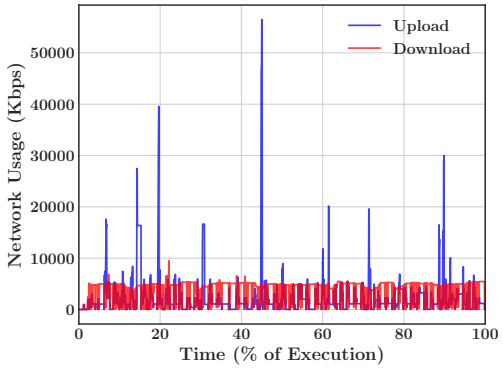

Fig. 9. Network usage over time for Edge-SLAM running on LAPT mapping D1 dataset without limiting the bandwidth

Measure \ Bandwidth	High (U=51Mbps, D=67Mbps	Medium (U=20Mbps, D=20Mbps)	Low (U=15Mbps, D=10Mbps)
# of Dropped KeyFrames on Mobile Device	11	236	392
Mean Localization Error (cm)	23.3 ±10.13	19.59 ±8.65	28.50 ±12.49

Table 5. Comparing the overhead of limiting the mobile device's (LAPT) network bandwidth on the performance of Edge-SLAM to map D1 dataset

update on the edge takes ≈62ms. Reconstructing the map update on the mobile device, however, takes ≈336ms. We should note that the mobile device cannot execute the Tracking module while reconstructing the map update. The reconstruction latency might seem long on the mobile device, but with the current optimizations, it is short enough for the Tracking module to continue mapping smoothly.

Each map update consists of the latest six KeyFrames added to the edge Global-Map. Shown in Table 4 (with MUCI) is the end-to-end latency in transmitting each KeyFrame across the wireless network. This latency includes the time from when a KeyFrame is assigned for transmission on one side to when it is received and deserialized on the other side. We see that each KeyFrame transmission takes on average ≈97ms, which is small. We should note that all network connections are non-blocking, and each runs on a separate thread simultaneously with other mapping operations. Therefore, Edge-SLAM can tolerate reasonable network latencies and delays.

Finally, Table 4 (with MUCI) shows the frequency of the map update in our experiment. Note that the mobile device might choose not to accept map updates transmitted by the edge. We offer the average frequency of map updates sent from the edge and the frequency of map updates accepted on the mobile device. Our results show that the mobile device rejects map updates rarely, at least on the D1 dataset. Map updates are published every ≈9s from the edge and accepted every ≈10s on the mobile device.

5.5.2 Bandwidth. In order to measure the effect of available network bandwidth on the performance of Edge-SLAM, we use the wondershaper [42] tool to control the bandwidth on the mobile device. We also identify the minimum required bandwidth for Edge-SLAM to work correctly. Because wondershaper is not compatible with JTX2, we use LAPT as our mobile device for the bandwidth

experiments. In these experiments, we map our primary dataset, D1, using Edge-SLAM. We run three experiments, first with a high bandwidth (no limits), second with medium bandwidth, and third with low bandwidth (minimums).

Figure 9 shows network usage for Edge-SLAM on the mobile device LAPT while mapping the D1 dataset without limiting the bandwidth. The mobile device in Edge-SLAM uses the network to download map updates from the edge every 5 seconds and upload KeyFrames to the edge immediately after creation. We observe that the download rate is steady at ≈5Mbps. This is because the Local-Map updates are small and consist of only six KeyFrames per update. On the other hand, the upload rate is ≈2Mbps most of the time, but it spikes when the KeyFrame creation rate is high. The figure shows around eight upload spikes that go as high as ≈56Mbps. These spikes correspond to eight turns that exist in the D1 dataset. When the robot turns, the scene changes rapidly, causing the system to create a higher number of KeyFrames to keep up with the changes.

Table 5 shows how lower network bandwidths could affect the performance of Edge-SLAM. Our main objectives in performing the network bandwidth measurements are:

- Show that Edge-SLAM can work reliably in different network conditions.
- Identify the minimum network bandwidth required for Edge-SLAM to map properly.

After performing several experiments, we have identified the upload rate of 15Mbps and the download rate of 10Mbps as the minimum required bandwidths for Edge-SLAM to work accurately without mapping issues. We have used these minimum rates for our low bandwidth experiment. From the low bandwidth measurements, we observe some overhead with respect to the number of dropped KeyFrames and the mean localization error. This overhead is still within an acceptable range, given that the system could fully map the dataset without any mapping issues. As for the medium bandwidth experiment, we see no overhead in the mean localization error, which is even better than the high bandwidth's mean localization error. However, we do see more drops of created KeyFrames, but without affecting the system mapping accuracy. We would like to note here that ORB-SLAM2 and consequently Edge-SLAM depend on multiple factors when run; therefore, their outcomes are slightly different after each run. These factors include but are not limited to the device's computational load, resource availability, network condition, and optimization quality. Thus, as long as the network bandwidths are above the minimum, it is normal to observe slightly mixed measurements for medium and high bandwidths. i.e., higher KeyFrame drops, lower localization error for medium bandwidth, lower KeyFrame drops, and higher localization error for high bandwidth. However, the final mapping outcome for both bandwidths should still be consistent and within an acceptable accuracy range. Therefore, Edge-SLAM can smoothly map with a wide range of today's bandwidths and variable network conditions.

5.6 Map Update Controller Interface Analysis

As described in Section 4.7.2, we implemented a map update controller interface along with other map update improvements for the mobile device to prevent unnecessary tracking loss during challenging mapping scenarios. In this subsection, we analyze these improvements by comparing different aspects of the system with the previous version. For this, we use measurements for Edge-SLAM running on the mobile device (JTX2) and the edge (DESK) while mapping the D1 dataset. The measurements for Edge-SLAM without the map update controller interface are based on [7].

Table 4 compares the measurements of the previous version of Edge-SLAM (without MUCI) to the current improved version of Edge-SLAM (with MUCI). For the edge operations (i.e., map update construction latency and publish frequency), these stayed almost the same as they are not affected by the map update controller and the map restructuring improvements. The improvements are designed to mainly enhance mobile device operations. For instance, when the controller prevents

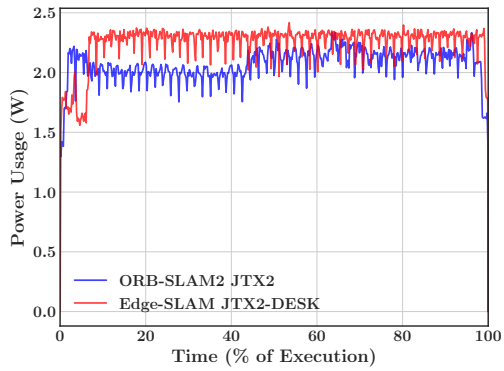

Fig. 10. Power consumption over time for ORB-SLAM2 and Edge-SLAM on JTX2 mapping D1 dataset

applying map updates during challenging scenarios, it enables the mobile device to better utilize its resources for other mapping and non-mapping operations. For mapping operations, we observe that the map update controller has slightly improved the mean localization error (i.e., accuracy). We also observe that the controller has almost no impact on the frequency at which the mobile device accepts map updates. For non-mapping operations, we look at the latency to prepare (serialize) and transmit newly created KeyFrames from the mobile device to the edge. Since all network connections are non-blocking, and each runs on a separate thread simultaneously with other mapping operations, we observe a %40 reduction in the KeyFrame transmission latency from the mobile device to the edge when map updates are only applied at the proper times.

Restructuring the map to support fast retrieval operation has improved the time it takes for the mobile device to rebuild the Local-Map from an update and establish the map connections. This can be seen in Table 4, showing a %19 reduction in the map update reconstruction latency.

Finally, we want to highlight another main benefit of the map update controller interface: reducing the number of experiments that must be performed to successfully map the long-running datasets with challenging scenarios. For instance, in the D1 dataset shown in Figure 6, there are eight sharp turns the robot needs to make without losing track to successfully map the dataset, and that is challenging. Thus, without the map update controller, we had to map the dataset multiple times from the beginning until getting a run where the system made it through all turns. *However, after implementing the map update controller, we successfully mapped the entire dataset in a single run.*

5.7 Power Consumption

We also measure power consumption on the mobile device for ORB-SLAM2 and Edge-SLAM. For this, we first map our main dataset, D1, using ORB-SLAM2 running on the JTX2. Second, we map the D1 dataset using Edge-SLAM running on the JTX2 as the mobile device and DESK as the edge device. The power measurements are collected on the JTX2.

To measure power consumption, we collect voltage readings throughout the run of each system. Then, we calculate the current using the obtained voltage using Equation 2. Finally, we calculate the power using Equation 3.

$$Current = (Voltage - 2.5)/0.1 \tag{2}$$

$$Power = Voltage * Current \tag{3}$$

Figure 10 shows no significant difference in power consumption between ORB-SLAM2 and Edge-SLAM on the JTX2. In ORB-SLAM2, the power consumption varies between 2 and 2.25 Watts. We believe this variation in power consumption is due to the loop closure and global optimization operations on the mobile device happening at ≈65% and ≈95% execution time. These operations are computationally heavy and run when a loop is detected. In Edge-SLAM, the power consumption is steady at ≈2.25 Watts. We believe this steadiness is due to moving the occasional computation-heavy operations such as loop closure and global optimization to the edge device. On the other hand, due to the continuous Wi-Fi network usage throughout the experiment, Edge-SLAM power consumption is equal to ORB-SLAM2's maximum power value of 2.25 Watts. ORB-SLAM2 does not use the network while mapping. Therefore, the increase of 0.25 watts on average can be attributed to the consistent use of the Wi-Fi. Even with this addition, Edge-SLAM power consumption is comparable to that of ORB-SLAM2.

6 CONCLUSION

Many mobile applications require spatial localization, including augmented reality apps and libraries such as ARCore, ARKit, and HoloLens API. 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 proposed idea by prototyping the Edge-SLAM architecture using ORB-SLAM2, a popular Visual-SLAM system. In particular, we kept the Tracking module on the mobile device and moved the Local-Mapping and 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 mainly overcome two challenges of ORB-SLAM2. First, we limit the growth in memory usage due to increasing map size and keep the mobile device memory usage constant. Second, we move the bursty computational tasks (Local-Mapping and Loop-Closing) to the edge device, allowing the mobile device to function more efficiently and run other apps. Overall, we achieved this with minimal loss of accuracy in the final map and the trajectory taken. We demonstrated this using our own datasets as well as publicly available datasets. We have internally tested our system on multiple other datasets, and the results are similar. We evaluated all aspects of resources used by our system, including CPU, memory, network, and power consumption. We open-source[5] our Edge-SLAM implementation and make it available to other researchers to evaluate their solutions.

ACKNOWLEDGMENTS

The authors were supported through NSF#1846320. We would like to thank our 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] C. Adhivarahan and K. Dantu. 2019. WISDOM: WIreless Sensing-assisted Distributed Online Mapping. In *2019 International Conference on Robotics and Automation (ICRA)*. 8026–8033. https://doi.org/10.1109/ICRA.2019.8793932

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

[5] Raghav Anand, Jeffrey Ichnowski, Chenggang Wu, Joseph M. Hellerstein, Joseph E. Gonzalez, and Ken Goldberg. 2021. Serverless Multi-Query Motion Planning for Fog Robotics. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*. 7457–7463. https://doi.org/10.1109/ICRA48506.2021.9561571

[6] 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

[7] Ali J. Ben Ali, Zakieh Sadat Hashemifar, and Karthik Dantu. 2020. Edge-SLAM: Edge-Assisted Visual Simultaneous Localization and Mapping. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services* (Toronto, Ontario, Canada) *(MobiSys '20)*. Association for Computing Machinery, New York, NY, USA, 325–337. https://doi.org/10.1145/3386901.3389033

[8] Alessio Botta, Jonathan Cacace, Riccardo De Vivo, Bruno Siciliano, and Giorgio Ventre. 2021. Networking for Cloud Robotics: The DewROS Platform and Its Application. *Journal of Sensor and Actuator Networks* 10, 2 (2021). https://doi.org/10.3390/jsan10020034

[9] 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. Luján, 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

[10] Carlos Campos, Richard Elvira, Juan J Gómez Rodríguez, José MM Montiel, and Juan D Tardós. 2020. ORB-SLAM3: An accurate open-source library for visual, visual-inertial and multi-map SLAM. *arXiv preprint arXiv:2007.11898* (2020).

[11] 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

[12] Kaiyuan Eric Chen, Yafei Liang, Nikhil Jha, Jeffrey Ichnowski, Michael Danielczuk, Joseph Gonzalez, John Kubiatowicz, and Ken Goldberg. 2021. FogROS: An Adaptive Framework for Automating Fog Robotics Deployment. In *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*. 2035–2042. https://doi.org/10.1109/CASE49439.2021.9551628

[13] Sandeep Chinchali, Apoorva Sharma, James Harrison, Amine Elhafsi, Daniel Kang, Evgenya Pergament, Eyal Cidon, Sachin Katti, and Marco Pavone. 2021. Network offloading policies for cloud robotics: a learning-based approach. *Autonomous Robots* 45, 7 (2021), 997–1012.

[14] 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

[15] Titus Cieslewski, Siddharth Choudhary, and Davide Scaramuzza. 2018. Data-efficient decentralized visual SLAM. In *2018 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2466–2473.

[16] 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

[17] 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

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

[19] 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

[20] 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

[21] 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.

[22] 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.

[23] 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

[24] 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

[25] 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

[26] Z. Hashemifar and K. Dantu. 2020. Practical Persistence Reasoning in Visual SLAM. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 7307–7313. https://doi.org/10.1109/ICRA40945.2020.9196913

[27] 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* (Munich, Germany) *(IoPARTS'18)*. Association for Computing Machinery, New York, NY, USA, 19–24. https://doi.org/10.1145/3215525.3215531

[28] 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.

[29] Z. S. Hashemifar, K. W. Lee, N. Napp, and K. Dantu. 2017. Consistent Cuboid Detection for Semantic Mapping. In *2017 IEEE 11th International Conference on Semantic Computing (ICSC)*. 526–531. https://doi.org/10.1109/ICSC.2017.78

[30] 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

[31] 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

[32] Jeffrey Ichnowski, Kaiyuan Chen, Karthik Dharmarajan, Simeon Adebola, Michael Danielczuk, Vıctor Mayoral-Vilches, Hugo Zhan, Derek Xu, Ramtin Ghassemi, John Kubiatowicz, Ion Stoica, Joseph Gonzalez, and Ken Goldberg. 2022. FogROS 2: An Adaptive and Extensible Platform for Cloud and Fog Robotics Using ROS 2. https://doi.org/10.48550/ARXIV.2205.09778

[33] 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.

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

[35] 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

[36] 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

[37] 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

[38] R. Kümmerle, 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

[39] M. Labbé 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

[40] 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.

[41] 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

[42] Canonical Ltd. 2021. Ubuntu Manpage: wondershaper - simple traffic shaping script. http://manpages.ubuntu.com/manpages/trusty/man8/wondershaper.8.html.

[43] 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

[44] Ricardo C. Mello, Sergio D. Sierra M., Wandercleyson M. Scheidegger, Marcela C. Múnera, Carlos A. Cifuentes, Moises R.N. Ribeiro, and Anselmo Frizera-Neto. 2022. The PoundCloud framework for ROS-based cloud robotics: Case studies on autonomous navigation and human–robot interaction. *Robotics and Autonomous Systems* 150 (2022), 103981. https://doi.org/10.1016/j.robot.2021.103981

[45] Stefan Milz, Georg Arbeiter, Christian Witt, Bassam Abdallah, and Senthil Yogamani. 2018. Visual slam for automated driving: Exploring the applications of deep learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 247–257.

[46] 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

[47] Gajamohan Mohanarajah, Dominique Hunziker, Raffaello D'Andrea, and Markus Waibel. 2015. Rapyuta: A Cloud Robotics Platform. *IEEE Transactions on Automation Science and Engineering* 12, 2 (2015), 481–493. https://doi.org/10.1109/TASE.2014.2329556

[48] Gajamohan Mohanarajah, Vladyslav Usenko, Mayank Singh, Raffaello D'Andrea, and Markus Waibel. 2015. Cloud-Based Collaborative 3D Mapping in Real-Time With Low-Cost Robots. *IEEE Transactions on Automation Science and Engineering* 12, 2 (2015), 423–431. https://doi.org/10.1109/TASE.2015.2408456

[49] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardós. 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

[50] R. Mur-Artal and J. D. Tardós. 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

[51] 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

[52] 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

[53] Albert Pumarola, Alexander Vakhitov, Antonio Agudo, Alberto Sanfeliu, and Francese Moreno-Noguer. 2017. PL-SLAM: Real-time monocular visual SLAM with points and lines. In *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE, 4503–4508.

[54] Tong Qin, Peiliang Li, and Shaojie Shen. 2018. VINS-Mono: A Robust and Versatile Monocular Visual-Inertial State Estimator. *IEEE Transactions on Robotics* 34, 4 (2018), 1004–1020.

[55] 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

[56] 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

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

[58] 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

[59] 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

[60] 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

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

[62] Lokender Tiwari, Pan Ji, Quoc-Huy Tran, Bingbing Zhuang, Saket Anand, and Manmohan Chandraker. 2020. Pseudo rgb-d for self-improving monocular slam and depth prediction. In *European Conference on Computer Vision*. Springer, 437–455.

[63] Antoni Rosinol Vidal, Henri Rebecq, Timo Horstschaefer, and Davide Scaramuzza. 2018. Ultimate SLAM? Combining events, images, and IMU for robust visual SLAM in HDR and high-speed scenarios. *IEEE Robotics and Automation Letters* 3, 2 (2018), 994–1001.

[64] 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

[65] Kwame-Lante Wright, Ashiwan Sivakumar, Peter Steenkiste, Bo Yu, and Fan Bai. 2020. CloudSLAM: Edge Offloading of Stateful Vehicular Applications. In *Proceedings of the Fifth ACM/IEEE Symposium on Edge Computing (SEC '20)*.

[66] J. Xu, H. Cao, D. Li, K. Huang, C. Qian, L. Shangguan, and Z. Yang. 2020. Edge Assisted Mobile Semantic Visual SLAM. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*. 1828–1837. https://doi.org/10.1109/INFOCOM41043.2020.9155438

[67] Chao Yu, Zuxin Liu, Xin-Jun Liu, Fugui Xie, Yi Yang, Qi Wei, and Qiao Fei. 2018. DS-SLAM: A semantic visual SLAM towards dynamic environments. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

IEEE, 1168–1174.

[68] Ji Zhang and Sanjiv Singh. 2014. LOAM: Lidar Odometry and Mapping in Real-time.. In *Robotics: Science and Systems*, Vol. 2. Berkeley, CA, 1–9.

[69] Xingxing Zuo, Xiaojia Xie, Yong Liu, and Guoquan Huang. 2017. Robust visual SLAM with point and line features. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 1775–1782.