CarMap: Fast 3D Feature Map Updates for Automobiles

Fawad Ahmad and Hang Qiu, *University of Southern California;* Ray Eells, *California State Polytechnic University, Pomona;* Fan Bai, *General Motors;* Ramesh Govindan, *University of Southern California*

https://www.usenix.org/conference/nsdi20/presentation/ahmad

This paper is included in the Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)

February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-13-7

Open access to the Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20) is sponsored by

CarMap: Fast 3D Feature Map Updates for Automobiles

Fawad Ahmad, *USC*
Hang Qiu, *USC*
Ray Eells, *Cal Poly, Pomona*
Fan Bai, *General Motors R&D*
Ramesh Govindan, *USC*

Abstract

Autonomous vehicles need an accurate, up-to-date, 3D map to localize themselves with respect to their surroundings. Today, map collection runs infrequently and uses a fleet of specialized vehicles. In this paper, we explore a different approach: near-real time *crowd-sourced* 3D map collection from vehicles with advanced sensors (LiDAR, stereo cameras). Our main technical challenge is to find a lean representation of a 3D map such that new map segments, or updates to existing maps, are compact enough to upload in near real-time over a cellular network. To this end, we develop CarMap,[1][2] which finds a parsimonious representation of a feature map, contains novel object filtering and position-based feature matching techniques to improve localization robustness, and incorporates a novel stitching algorithm to combine *map segments* from multiple vehicles for unmapped road segments and an efficient map-update operation for updating existing segments. Evaluations show that CarMap takes less than a second to update a map, reduces map sizes by 75× relative to competing strategies, has higher localization accuracy, and is able to localize in corner cases when other approaches fail.

1 Introduction

Autonomous vehicles use a three-dimensional (3D) map of the environment to position themselves accurately with respect to the environment. A 3D map contains *features* in the environment (§2), and their associated positions. As a vehicle drives, it perceives these features using advanced depth perception sensors (such as LiDAR and stereo cameras), then *matches these to features in the map*, and using the feature positions, triangulates its own position.

Maps need to be updated whenever there are significant changes to the environment. Changes to the environment can impact the set of features visible to a vehicle. For example, road or lane closures due to construction or accidents, parked delivery vans impeding traffic flow, parked vehicles on the road-side, or closures for sporting events can cause the set of features in the map to be different from the set of features visible to the vehicle. This impacts feature matching, and can reduce localization accuracy. Figure 1 quantifies this in a simple scenario. In the image on the left, a street has been closed due to an accident. With an outdated map, a car is unable to position itself; an updated map is necessary for accurate positioning.

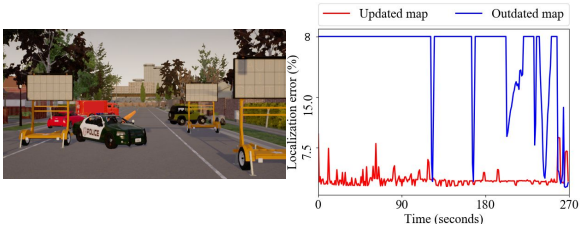

Figure 1: If short timescale events like traffic accidents (left) are not updated in maps, a vehicle cannot localize itself (blue line) because it cannot match the scene with the map. On the other hand, vehicles with updated maps (red line) can localize themselves accurately.

Keeping this map up to date can be tedious. Today, large companies (*e.g.,* Waymo [56], Uber [14], Lyft [12], Here [31], Apple [6], Baidu [7], Kuandeng [11], Mapper [5]) employ fleets of vehicles equipped with expensive sensors (LiDAR, Radar, stereo cameras) and GPS devices. For instance, Apple Map [1] uses vans equipped with a high-precision GPS device, 4 Lidar Arrays, and 8 cameras beside other equipment for capturing mapping data. These vehicles scan neighborhoods periodically with a frequency determined by cost considerations, which could be up to several thousand dollars per kilometer [4]. The scan frequency determines the timescale of environmental changes captured by the map [2]. To capture these changes, vehicle fleets have to continuously traverse the mapped area at very fine timescales [8], which can be prohibitively expensive.

In this paper, we take a first step towards answering the question: *What techniques and methods can ensure near real-time updates to 3D maps?* The most promising architectural approach to this question, which we explore in this paper, is *crowd-sourcing*.[3] In this approach, which leverages the increasing availability of depth perception sensors in vehicles, each vehicle, as it drives through a road segment, uploads *map updates* in near real-time over a cellular network to a cloud service. The cloud service, which acts as a rendezvous point, applies these updates to the map and makes these updates available to other vehicles.

Given today's cellular bandwidths, this architecture is most suitable for a class of 3D maps in which landmarks are *sparse*

[1] https://github.com/USC-NSL/CarMap
[2] Video demo

[3] Incentives for crowd-sourcing are beyond the scope of this paper. Waze has successfully employed crowd-sourcing from vehicles, by providing a navigation service and CarMap can use similar techniques.

features in the environment. Even so, today's feature-based 3D maps of the kind generated by Simultaneous Localization and Mapping (SLAM) algorithms require an order of magnitude higher bandwidth than cellular speeds (§2).

Contributions. Our first contribution (§3) is to identify the most parsimonious representation of feature maps. SLAM feature maps preserve a large number of features, even transient ones, and build indices to enable fast and effective *feature matching*. We show that it is possible to preserve fewer features, and reconstruct the indices, without impacting localization accuracy while reducing map size significantly.

Because our lean map representation throws away information, we have had to re-think feature matching. Our second contribution leverages the observation that, unlike robots, cars have approximate position information (*e.g.,* from GPS). Thus, instead of using statistics of features alone for matching, we also use position information to enable a more robust feature search, leading to improved localization accuracy.

Vehicles will use feature maps over longer time-scales than SLAM maps used by robots,[4] so we must avoid including features (*e.g.,* from parked cars, or pedestrians) that may disappear over those time-scales. We observe that *semantic segmentation* algorithms can identify such features. Our third contribution is a robust resource-aware algorithm that incorporates the semantics of objects in the scene to perform *dynamic object filtering*.

Updates to a map can be of two kinds: *map segments* representing a previously unseen road segment, and *map diffs* representing a transient in a previously-mapped road segment. Our last contribution is a collection of algorithms for map update: a fast and efficient map diff algorithm which generates compact diffs and can integrate these quickly into the map, and a robust map segment *stitching* algorithm that reliably identifies areas of overlap between the map segment and the existing map, and uses features within the overlapped region to transform the segment into the existing map's coordinate frame of reference.

We have embodied these contributions in a system called CarMap. Using experiments on an implementation (§4) of CarMap built upon the top-ranked visual open-source SLAM algorithm [41], and real traces as well as traces from a game-engine simulator [27] we show that (§5): CarMap requires 75× lower bandwidth than competing algorithms; it can generate a map update, disseminate it to a participating vehicle, and integrate the update into the vehicle's map *in less than a second*; its localization accuracy is better than state-of-the-art SLAM algorithms especially when a map is used in dramatically different conditions (*e.g.,* denser traffic) than when it was collected; it can localize a vehicle in some cases when other competitors cannot, such as when a map obtained from one lane is used in another lane in a multi-lane street; its computational overhead is comparable to, and sometimes better than competing strategies; and its feature labeling achieves upwards of 95% accuracy in distinguishing static from non-static objects even when the underlying segmentation algorithms have lower accuracy.

[4] In a robot, SLAM algorithms perform mapping and localization simultaneously. For vehicular use, a SLAM map is collected once, updated intermittently, and used often.

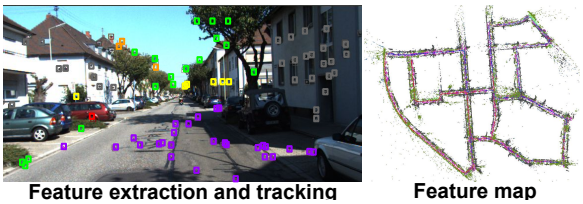

Figure 2: Localization using a feature based map. The picture on the left shows the features in an image, and the picture on the right shows the feature map generated for an area. Features are color-coded by the type of object those features belong to.

2 Background and Motivation

SLAM Principles. SLAM represents a map by a set of *landmarks* and their associated positions [19]. As a vehicle traverses the environment, its sensors (LiDAR, cameras) continuously generate *measurements* of the environment. SLAM continuously outputs (a) detected landmarks, and (b) the current *pose* (position and orientation) of the vehicle. It does this by using maximum *a posteriori* (MAP) estimation [42], finding the landmark position and vehicle pose that best explain the observed measurements.

Feature-based Maps: Terminology. SLAM maps can contain either feature-based landmarks (extracted from cameras [41] or LiDAR [57, 58]) or dense representations such as image frames [28] and occupancy grids [54]. In this paper, we explore crowd-sourcing *feature-based maps* (Figure 2), leaving denser representations for future work[5]. A *feature* is a lower-dimensional representation of some high-dimensional entity in the environment (*e.g.,* a leaf on a tree, or a part of a letter on a roadside sign), and is represented by a feature signature. Features are usually extracted from LiDAR or camera frames. For storage efficiency, SLAM implementations store features from approximately every k frames (so called *keyframes*), for small k. These implementations associate each feature in a keyframe with a relative 3D position with respect to that keyframe. They extract landmarks for the feature-based map from a subset of these features; we call these *map-features*. Maps have a single coordinate frame of reference, and map-features have 3D positions relative to the map's coordinate frame of reference.

SLAM Practice. Practical SLAM implementations are complex (Figure 3) because they have to deal with sensor and estimation errors. We briefly describe SLAM components

[5] Which map technology a vehicle uses is generally proprietary information, but we conjecture, based on anecdotal evidence that lower levels of autonomous driving [43] or vehicles that use stereo cameras will use feature-based maps [9] for cost reasons, while higher-end fully-autonomous vehicles with LiDAR will use denser maps.

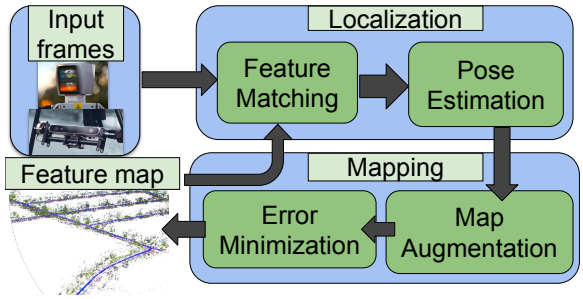

Figure 3: Components of feature-based map generators.

here, and introduce additional background in later sections.

Feature matching. Feature matching (or data association) is the process of matching features in the current frame with features seen in one or more keyframes in the map. SLAM implementations match features in a number of different ways: *e.g.,* image feature matching uses the similarity of image signatures (feature descriptors), and LiDAR 3-D features use feature geometry. Matching is a crucial building block for identifying map-features (as described below). SLAM implementations contain two data structures to speed up feature matching. A *map-feature index* associates map-features to keyframes they occur in. A *feature index* can search for the keyframe whose features most closely match the features in a given frame.

Pose estimation. This component contains algorithms that estimate the pose of the vehicle. As a vehicle traverses an environment, it first extracts features from each frame received from its sensor. Then, the vehicle matches the extracted features with those extracted in the last frame. At this point, the vehicle knows (a) the pose estimate in the previous frame, (b) the positions of the matched features in the previous frame and the current frame. It then uses MAP estimation [42] to estimate the current pose of the vehicle. If the feature matching step does not return enough features to estimate pose accurately, the vehicle uses the feature index to search the entire map for keyframes containing features matching those seen in this frame, a step called *relocalization*.

Map augmentation. Pose estimation can estimate the 3D positions of features in each keyframe. It adds some of these features as map-features, but only after filtering transient features (those that do not occur across multiple frames [41, 58]) or dynamic features (*e.g.,* features that belong to moving vehicles) whose position is not stable across frames.

Error minimization. This component minimizes the error accumulated in the feature map. *Local error minimization* rectifies error accumulation in successive frames using, for example, extended Kalman filters for LiDARs and bundle adjustment [55] for cameras. When vehicles visit a previously-traversed part of the environment, a *loop closure* algorithm finds matches between features in the current frame and features already in the map, then reconciles their position estimates (while also correcting positions of features discovered within the loop), thereby reducing error.

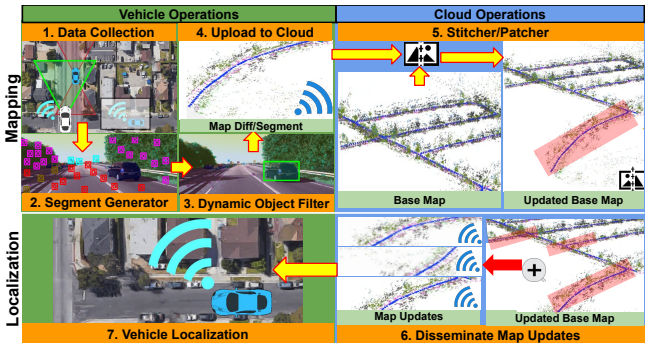

Figure 4: Architecture and workflow of CarMap

Challenges. CarMap faces four challenges.

Map size. CarMap could simply upload, over the cellular network, a SLAM map to the cloud, but these maps, which include map-features, keyframes, and the two indices, can be large. A 1 km stretch of our campus generates a 1.5 GB map. A car traveling at 30 kph would require a sustained bandwidth of 100 Mbps, well above achievable LTE speeds [3].[6] Our first challenge is to find a lean map representation that fits within wireless bandwidth constraints.

Environmental dynamics. CarMap maps are meant to be used over a longer timescale than SLAM maps used by robots, so they must be robust to environmental dynamics. For example, if a map includes features from a parked car that has since moved, localization error can increase.

Effective feature matching. As in SLAM, CarMap relies heavily on accurate feature matching for pose estimation, relocalization, and loop closure. However, because CarMap's lean map has less information than SLAM's, its feature matching accuracy can be lower, so CarMap must use a fundamentally different strategy.

Fast map-updates. CarMap must devise fast algorithms to (a) *stitch* additions to the map received from vehicles traversing a previously unmapped road segment (decentralized SLAM algorithms [29] have a similar capability but differ significantly in the details, §6), (b) generate and incorporate changes to the map from temporary obstructions.

3 Design of CarMap

Architecture and Workflow. As vehicles traverse streets (Step 1, Figure 4), they derive lean representations of feature maps using a *map segment generator* that runs on the vehicle (Step 2, §3.1). To this representation, CarMap applies a *dynamic object filter* to improve robustness to environmental dynamics (Step 3, §3.3). CarMap then determines whether this is a new map segment (not available in its own base map). If so, it uploads the entire map segment, else it uploads a

[6] With standard compression techniques (*e.g.,* gzip [26]) the sustained bandwidth is approximately 60 Mbps. Moreover, gzip compression adds latency: it takes approximately 25 seconds to compress a 500MB map collected over 4 minutes.

map diff (Step 4) to a cloud service. The cloud service runs a *stitcher* to add a new segment to the map, or a *patcher* to patch the diff into the existing map (Step 5, §3.4).

A vehicle receives, from the cloud service, segments or diffs contributed by other vehicles (Step 6), *reconstructs* the complete map, and uses it for localizing the vehicle (Step 7, §3.5). Diff generation, stitching, patching, and reconstruction use a *position-based feature index* for feature matching (§3.2), resulting in high feature matching accuracy.

The on-vehicle compute resources needed to run map generation, matching, diff generation, and reconstruction are comparable to those provided by commercial on-vehicle computing platforms like the NVIDIA Drive AGX [13]. CarMap uses (a) cloud storage as rendezvous for map updates from vehicles, and (b) cloud compute to integrate map updates. Extensions to this architecture to use road-side units for storage and processing are left to future work.

3.1 Map Segment Generator

The Problem. As a vehicle traverses a street, it produces map segments. The map segment generator must find the *leanest* representation of the map that respects cellular bandwidth constraints while permitting accurate localization.

As discussed in §2, a complete map contains four distinct components: (A) map-features, (B) features associated with every keyframe, (C) a map-feature index that associates map-features with keyframes used to generate the map-features (recall that a map-feature is one whose position is stable across several keyframes), and (D) a feature index that finds the most similar keyframe to the current frame. Uploading the complete map is well beyond cellular bandwidths (§2).

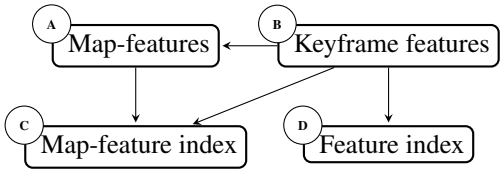

Figure 5: Dependencies between map components

CarMap's Approach. Consider Figure 5 in which an arrow from B to A indicates that B is needed to generate A. Thus, for example, map features are generated from keyframe features. Similarly, to generate the map-feature index, we need both map-features and keyframe features.

From this figure, it is clear that *all other components can be generated from keyframe features*. Thus, in theory, it would suffice for CarMap to upload only the keyframe features, thereby reducing the volume of data to be uploaded. Unfortunately, this does not provide significant bandwidth savings. For a 1 km stretch of a street on our campus (§2), the keyframe features require 400 MB. At 30 kmph, this would require an upload bandwidth of 26.67 Mbps, still above nominal LTE speeds. At higher speeds, CarMap would require proportionally greater bandwidth since the vehicle covers more of the environment (§A.3, Figure 19).

A Lean Map. CarMap uses a slightly non-intuitive choice of map representation: the *map-features* alone. Each map-feature contains the feature signature, the 3D position in the map's frame of reference, and the list of keyframes in which the map-feature appears. In §5, we show that this representation permits real-time map uploads.

Reconstruction. However, to understand why this is a reasonable representation, we describe how one can reconstruct a full SLAM map from these map-features. Map-features have, associated with them, a list of keyframes in which they appear. From these, we can generate keyframe features (a sequence of keyframes and features seen in those keyframes). From these keyframe features, it is possible to generate the feature index and the map-feature index, resulting in the complete SLAM map. §3.5 presents the details.

However, the CarMap map contains *only* map-features whereas a SLAM map contains *all* features seen in every keyframe. These fewer features can potentially impair feature matching accuracy. To address this, CarMap employs a better feature search strategy.

3.2 Robust and Scalable Feature Matching

Background. Feature matching is a crucial component in feature-based localization, and determines both the robustness of feature matching as well as scalability. Feature matching requires two operations: given a frame F, (a) find keyframes with the most similar features, and (b) given a feature f in F and a keyframe K, find those map-features m in K that are most similar to f. The first operation is used in relocalization and loop closure, and the second operation is used for these two tasks as well as fine-grained pose estimation (§2).

Similarity matching. Both of these operations use similarity matching techniques. For example, if a feature is represented by a vector, then, the most similar feature is one closest by Euclidean distance to this feature. Similarly, if a frame F can be represented by a signature in a multi-dimensional space, then the most similar keyframe K is one that is closest by some distance measure.

Scaling similarity matching. To derive scalable feature matching, many SLAM implementations arrange keyframe features in fast data structures. We have used the term feature index in §3.1 to describe these data structures. In practice, implementations construct multiple indices.

To ground the discussion, we take a concrete example from a popular visual SLAM [41] implementation. This implementation discretizes the space of features into hypercubes, and represents each hypercube by a *word*. For example, if a feature f is represented by a vector $< 1, 5 >$, and the hypercube has a side of 10 units, then, f falls into the hypercube defined at the origin. Suppose the hypercube is assigned the word "0". Then, any feature f' that is assigned "0" (*i.e.,* falls into the

same hypercube) is close in feature space to f.

Search indices. The two feature matching operations can be implemented in scalable fashion using this word-based discretization. The first operation uses an *inverted index* I that maps each word to all the keyframes that it appears in. To find a keyframe closest to a given frame F, we can use the following algorithm. (i) Map each feature f in F to a word w_f. (ii) For each w_f in F, find all keyframes K associated with w_f in I. (iii) Take the intersection of all keyframes across all words w_f, then find those keyframes whose word histogram is most similar to F. The second operation requires a *word search tree per keyframe* K that maps a given w_f to those features in K that are closest to w_f in feature space.

The Problem. While these data structures work well for indoor robot navigation in relatively static environments, they can fail in more dynamic environments for outdoor vehicles. For example, keyframe word histogram matching can fail when a map's keyframe K was collected from an unobstructed view, while frame F, *taken at the same position*, had a car in front of it which obscured many of the features in K. As another example, consider a map of a 2-lane street where the map was taken from the right lane, but the vehicle using the map is on the left lane; in this case, a feature's signature may change if perceived from a different 3D position and orientation and hence result in a mismatch if the matching is based on feature similarity. In these cases, feature matching can result in *false positives*: a keyframe K far from the vehicle's current position may better match the current frame F than the correct match K' because features at completely different locations in a frame may look visually similar (*e.g.,* features from trees of the same species).

CarMap's Approach. To address these problems, instead of searching all keyframes in the map, CarMap *searches for matches in the vicinity of the vehicle's current position*. CarMap relies on a vehicle's GPS position to scope the search. However, GPS is known to be erroneous, especially in highly obstructed environments [40], so CarMap searches over a large radius around the current GPS position (in our experiments, 50 m, larger than the maximum error reported in [40]).

Keyframe matching. Specifically, in addition to using the inverted index and word histogram similarity to find matching keyframes in the base map, CarMap maintains a global k-d tree [16] of keyframes and uses it to search for all keyframes in the map within a given radius. Then, to localize a vehicle with a frame F in a given map, CarMap uses the GPS coordinates of the vehicle to get all keyframes within a large radius around the GPS position. It then finds the subset of these keyframes that most closely resemble F based on histogram matching. If it cannot find any resembling keyframes, then CarMap uses the keyframes closest to the vehicle's GPS coordinates. For each keyframe K in this subset, CarMap tries to find, for each feature f in F, the closest matching feature in K. To do this, it first performs a coordinate transformation to find the position of f in the map, assuming that F is at K's position, and then performs feature matching.

Feature matching. Based on the position hints of the features, CarMap also maintains another global k-d tree of map features, which partitions 3-D space into different regions to find all features in the map that are closest (by position) to a given feature f. Then, for each feature f in frame F, CarMap finds all map-features that are spatial neighbors, and uses feature similarity to identify the matching features. Using these matching features, it can perform pose estimation. CarMap then attempts to *refine* this pose estimate by searching nearby (in position) map-features for additional feature matches.

3.3 Dynamic Object Filter

Background. As a vehicle traverses an environment, it encounters three types of objects: a) static, b) semi-dynamic, and c) dynamic objects. Static objects are those that are at rest when perceived by the vehicle and are likely to stay in the same position for a long time *e.g.,* roads, buildings, traffic lights, and traffic signs. Dynamic objects are those that are in motion when perceived by the vehicle *e.g.,* moving vehicles and pedestrians. Semi-dynamic objects are those that have the ability to move but might not be in motion when perceived by the vehicle *e.g.,* parked vehicles, construction trucks.

The Problem. SLAM algorithms contain techniques to estimate whether a feature belongs to a dynamic object or not; if it does, that feature is not used in the map (§2). However, for a system designed for vehicles like CarMap, this is insufficient. These techniques work only if the majority of the scene is static and fail in highly dynamic environment (as we show in §5). Similarly, unlike SLAM, CarMap maps are intended to be re-used over longer time scales, during which the environment might change significantly. If a map contains a feature f, say, belonging to a semi-dynamic object such as a parked car which has moved away by the time a vehicle uses that map (before another vehicle has contributed a map diff), keyframe matching and feature matching might fail.

Figure 6: Semantic segmentation of an image while driving.

CarMap's Approach. To counter this, CarMap uses *semantic segmentation* to classify the whole scene into static and (semi-)dynamic objects. Semantic segmentation can be performed on camera data as well as LiDAR data, and refers to the task of assigning every pixel/voxel in a frame a semantic label (Figure 6), such as "car", "building" *etc.* In addition to motion analysis (§2), CarMap leverages these semantic labels to determine whether to add features to the map.

Specifically, CarMap extracts features (Figure 4) and uses

semantic segmentation to label each point/pixel in the frame. It then associates each feature with the corresponding semantic label of the particular pixel(s) that the feature covers. As a result, when a feature is generated, besides its feature signature and 3D position, CarMap also appends a semantic label to it. If the semantic label belongs to a dynamic or semi-dynamic[7] object (*e.g.,* car, truck, pedestrian, bike *etc.*), CarMap does not add it to the map.

To detect moving objects we could have used *background subtraction*, but CarMap needs the ability to also detect semi-dynamic objects (*e.g.,* parked cars). Object detectors can generate loose bounding boxes for semi-dynamic objects, which can result in incorrect matches between features and their corresponding objects.

Challenges. Semantic segmentation poses two challenges in practice. First, it is prone to errors, especially at the boundaries of different objects. For example, a state-of-the-art segmentation tool, DeepLabv3+ [20], has an iIoU[8] score of 62.4% on a semantic segmentation benchmark (CityScapes [23]). Second, it uses deep convolutional neural networks that are computationally very expensive (*e.g.,* DeepLabv3+ runs at only 1.1 FPS on a relatively powerful desktop equipped with an NVIDIA GeForce RTX 2080 GPU).

Robust labeling. To tackle the first challenge, CarMap tracks feature labels across multiple frames and uses a majority voting scheme for deriving robust labels. Consider a feature f that is detected and tracked in multiple keyframes (only these features are likely to be added as map-features). In each keyframe, we determine the semantic label associated with the f. Instead of labeling each feature with its semantic label, we perform a coarser classification, determining whether that label belongs to a static (road surface, traffic signals, buildings, and vegetation *etc.*) or a non-static (cars, trucks, pedestrians) *etc.* This coarser classification overcomes boundary errors in segmentation: even if the segmentation algorithm identifies a pixel as belonging to a building when it actually belongs to a tree in front of the building, because both of these are static objects, the pixel would be correctly classified as static. CarMap then does a majority voting across these coarser labels to determine whether f is static or non-static. In §5, we show that this approach results in high classification accuracy.

Resource usage. Semantic segmentation CNNs can run at low frame rates. However, CarMap only needs to determine the label of a feature when creating map-features. These are assessed at keyframes, so, segmentation needs only be applied at keyframes. Depending on the vehicle's speed, SLAM algorithms [41] can generate keyframes at 1-10 frames per second. In §5, we explore a resource/accuracy trade-off: running slightly less accurate, but lower resource intensive CNNs still gives acceptable performance in our setting. When segmentation cannot run on every keyframe, we mark the missed keyframe's features as *unlabeled*.

[7] For brevity, we use the term *dynamic object filter* for this capability, but it can detect semi-dynamic objects as well.

[8] The IoU (intersection over union) metric is biased towards classes covering a large image area. Hence, for autonomous driving, the iIoU metric is preferred which is fairer towards all classes.

Figure 7: When adding a new region to the base map, the vehicle uploads the whole map segment (above). For updating a existing map segment, CarMap generates a map diff containing new map features (below, new map features marked in blue).

3.4 Map Updater

Map Diffs. When a vehicle traverses a segment that exists in its own map, CarMap generates a compact map diff to report newly discovered map features.

The Problem. CarMap may discover new features for two reasons. In Figure 7, if the feature map were constructed from the top image, a vehicle traveling through the same region at a later time (bottom image) might see new features previously occluded by the bus. Moreover, sparse SLAM algorithms are designed to capture only a small portion of all the features in the environment to ensure real-time operation, so a new traversal may discover additional features (Figure 7).

CarMap's Approach. A *map diff* compactly represents the newly discovered features. To explain how CarMap generates a map diff, consider a vehicle V, traversing a road segment R_A at time t_1, having an on-board map segment M_A of the same area from an earlier time t_0. CarMap loads the on-board map segment M_A into memory and marks all map elements (map-points and keyframes) as pre-loaded elements in the map. As the vehicle V traverses R_A, it localizes itself in the map segment M_A. At the same time, for every feature f_{road} the vehicle perceives, it uses CarMap's robust feature matching (§3.2) to query and match it against features f_A present in the map segment M_A in the same spatial vicinity. If the match is successful, that means the feature is already present in the map. If not, it is a new feature. This yields a set of features f_{diff} and keyframes K_{diff} that have been introduced in the time interval $\delta t = t_1 - t_0$. The vehicle uploads this *diff* map to the cloud service. The cloud service's patcher receives this and patches these map elements (f_{diff} and K_{diff}) into the base map. It also sends out the patch to all vehicles so that they can update their base maps.

Removing features no longer visible is tricky because those

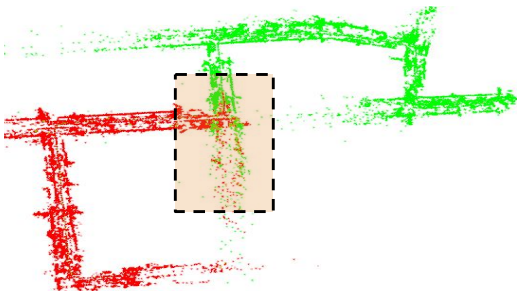

Figure 8: CarMap stitching together two feature maps. The highlighted regions represent overlapped sub-segments.

features could be, for example, occluded by a parked vehicle. It is, however, important to do this in practice (*e.g.,* features from objects present during a transient road closure). We are currently working on a robust algorithm for this.

Map Segment Stitching. When it traverses a previously unseen road segment, CarMap uploads the map segment to a *map stitcher* in the cloud.

The Problem. CarMap's stitcher adds map segments (§3.1) received from vehicles into its *base map* (Figure 8). CarMap must address three challenges while stitching a map segment into a base map. It must efficiently find potential regions of overlap between two map segments. The stitcher only has access to map-features at keyframes whereas SLAM algorithms preserve all features in each keyframe; feature matching can potentially be more difficult in CarMap. To scale well, CarMap must incrementally add new map segments to the base map without recomputing the whole map.

CarMap's Approach. Algorithm A.1 depicts the stitching algorithm. Suppose we have two map segments, the new incoming map segment M_s and the base map M_b. To stitch M_s with the base map M_b, CarMap first reconstructs (lines 4-5) (§3.5) the two map segments (R_b, R_s). Then, it uses (line 6) fast feature search (§3.2) to find the sub-segments (sequences of keyframes) that overlap (O_b, O_s). It then applies (line 7) feature matching between these sub-segments and uses these matches to compute the *coordinate transformation matrix* between M_s and M_b. It uses this matrix to transform M_s into M_b's coordinate frame of reference (lines 8-9). Finally, it removes duplicate features observed in both segments. §A.1 describes some of the details of this algorithm.

3.5 Reconstruction

Map Segment Download. Before a vehicle enters a street, it retrieves a map segment from the cloud service. This segment uses the lean representation described in §3.1.

Reconstruction Details. CarMap places map-features into keyframes, and adds them to the k-d tree structures. It then generates word histograms and per-keyframe word search trees as in SLAM. To do this, it must compute the 2D and 3D positions of each map-feature in the associated keyframe (recall that a map-feature's position in a map segment is with respect to the map's frame of reference). To reconstruct the position of a given map-feature f in a keyframe k, $P_f(k)$, CarMap uses the global 3D position of the map-feature $P_f(O)$, the respective keyframe's position $P_K(O)$ and rotation matrix $R_K(O)$ to perform an inverse transformation:

$$P_f(k) = \left[R_K^{-1}(O) * \left\{ P_f(0) - P_K(0) \right\} \right] \quad (1)$$

4 Implementation of CarMap

Software we use. We have implemented CarMap by modifying a visual SLAM algorithm, ORB-SLAM2 [41], the top-ranked open-source visual odometry algorithm for mono, stereo, and RGB-D cameras in the KITTI vision-based benchmarks [30] for self-driving cars. At least one other visual SLAM implementation [45] has a very similar implementation structure, so CarMap can be ported to it. It should also be possible to incorporate CarMap ideas into LiDAR SLAM implementations, but we have left this to future work.

For semantic segmentation, we use MobileNetV2 [51], a light-weight version of DeepLabv3+ [20] designed for mobile devices. We use OpenCV [18] for image transformations, the Point Cloud Library (PCL) [50] for point cloud operations, and the C++ Boost library [52] for serializing and transferring the map files over the network.

Our Additions. On top of these, we have added a number of software modules necessary for the six components described in §3. CarMap reuses the feature extraction, index generation, and similarity-based feature matching modules in ORB-SLAM2 (ORB-SLAM2 is 9620 lines of C++ code), but even so, it requires approximately 15,000 additional lines of C++ code. §A.2 discusses these additions in detail.

5 CarMap Evaluation

In this section, we evaluate (a) real-time end-to-end latency of map update using experiments and (b) the localization accuracy of CarMap using trace-driven simulation. We then report on microbenchmarks for its lean map representation, feature map stitching, segmentation, and spatio-temporal robustness in localization, using both synthetic and real-world traces.

5.1 Methodology

Traces. For our end-to-end accuracy evaluations, we use 15 km of stereo camera traces that we curated using CarLA [27], the leading simulation platform for autonomous driving supported both by car manufacturers and major players in the computing industry. CarLA can simulate multiple vehicles driving through realistic environments — the simulator has built-in 3D models of several environments including freeways, suburban areas, and downtown streets. Each vehicle can be equipped with stereo cameras or LiDAR sensors, and the simulator produces a *trace* of the sensor outputs as the cars drive through. When curating our CarLA traces, we model a stereo camera with the same properties (stereo baseline, focal length *etc.*) used in the KITTI dataset. When evaluating

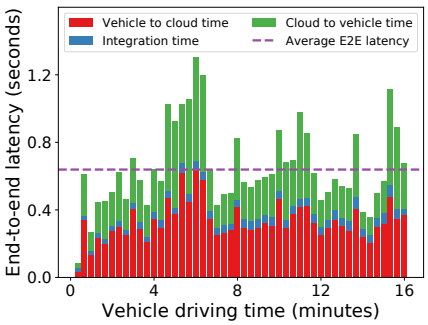

Figure 9: End-to-end latency results for CarMap's map update operation enables real-time map updates (average end-to-end latency is approximately 0.6 seconds.)

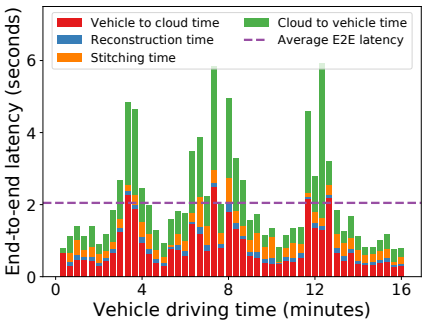

Figure 10: End-to-end latency for CarMap's map stitch operation. The stitch operation, on average, takes approximately 2.0 seconds for unmapped regions.

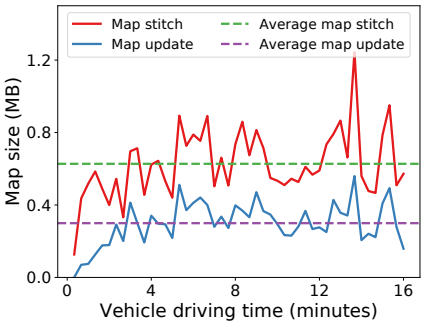

Figure 11: Vehicle map uploads for map stitch and update operations. Map updates reduce required bandwidth by 2x as compared to stitching map segments.

CarMap, we only extract the left and right images from the modeled camera after which we process the frames like we would for a real-world camera. We do not extract depth or segmentation labels from CarLA but instead generate them using ORB-SLAM2's stereo matching and a segmentation CNN respectively.

For some of our microbenchmarks, we also used 22 km of real-world traces from the KITTI odometry benchmark [30]. The KITTI benchmark traces only have a single run for each route, but for our end-to-end evaluations, we need one run to build the map, and another to use the map for localization. This is why we use traces from CarLA.

Finally, to validate real-time map updates (§5.2), we used 8 km of stereo camera data from our campus.

Metrics. For most evaluations, we are interested in end-to-end latency, localization accuracy, and map size. To calculate localization accuracy, we build a map for a region and then localize another vehicle that drives in the same region using that map. In this case, the localization error is the average translational/localization error (used in KITTI odometry benchmark [30]) between the ground truth position of the vehicle and its estimated position, averaged over the whole trace. In some experiments, we also measure compute times for various operations. These measurements were taken on an Alienware laptop equipped with an Intel i7 CPU running at 4.4 GHz with 16 GB DDR4 RAM and an NVIDIA 1080p GPU with 2560 CUDA cores.

Scenarios. For the end-to-end accuracy experiments, we generate CarLA traces to mimic three different kinds of driving conditions: a) suburban streets (light traffic and some parked vehicles), b) freeway roads (dense traffic), and c) downtown roads (dense traffic, with parked vehicles on both sides). For each of these, we generate traces for a static scene (no traffic), and for a dynamic scene (with traffic). This allows us to evaluate maps built for one kind of scene (*e.g.,* static), but used in another (*e.g.,* dynamic).

Comparison. In all these evaluations, we compare the performance of: a) maps generated by ORB-SLAM2, b) a stitched map generated by QuickSketch [15], and c) a stitched CarMap map. QuickSketch is a competing approach to map crowdsourcing that does not attempt near real-time map updates. In QuickSketch, map segments are raw stereo camera traces, and the stitching algorithm feeds new map elements from the camera trace into an existing base map generated by ORB-SLAM2. QuickSketch uses ORB-SLAM2's relocalization and feature matching components. We repeat each experiment three times and report the average values.

5.2 Near Real-Time Map Updates

Methodology. To measure end-to-end latency of map updates, we drove a vehicle for 16 minutes (8 km) equipped with an Alienware laptop tethered to a phone with an LTE connection. The laptop sends map updates to a remote server which runs CarMap's diff integration and stitching operations, then sends the map updates back to the vehicle. The end-to-end latency includes: update generation and transmission on the sender, update processing on the cloud, and update transmission and integration on the receiver. We conducted two experiments.

Map Diffs. In the first experiment, we measure end-to-end latency when all updates are in the form of map diffs (*i.e.,* the vehicle drives through a previously mapped area). CarMap generates map diffs every 10 s. As Figure 9 shows, the *average end-to-end latency for CarMap's map update operation is 0.6 s*[9]. Update transmission times dominate the cost, since diff integration is fast (§3.4).

Map Segment Stitching. In the second real-time experiment, we measure the end-to-end latency when all updates are in the form of map segments (*i.e.,* the vehicle drives through a previously un-mapped area). As before, CarMap generates map segments every 10 s; in this case, however, the cloud service needs to perform an expensive stitch operation (§3.4).

The overall end-to-end latency for map segment updates in CarMap (Figure 10), although about 3.2× more than map-

[9] As an aside, vehicles rely on these maps only to localize themselves, not for safety-critical operations (for which they use their sensors).

updates, is *still only 2.1 seconds on average*. Two factors contribute to a higher end-to-end latency. First, map segments are about 2-4× larger than map diffs (Figure 11). Second, they require about 10× more computation than map updates. Map update integration is only 50 ms whereas the partial reconstruction (§A.1) and stitching take nearly 500 ms. Even so, transmission and reception times dominate.

In summary, in CarMap, map updates can be made available to other vehicles in under a second. Even in the rare event that a vehicle traverses an un-mapped road segment, map updates can be made available in about 2 s.

5.3 End-to-End Localization Accuracy

We now demonstrate that CarMap has comparable or better localization accuracy than ORB-SLAM2 and QuickSketch [15] for three different scenarios: static scene, dynamic scene, and multi-lane localization.

Static Scene Maps. In this scenario, we build a map from a static scene with no dynamic or semi-dynamic objects (a *static-map*). We then use this map to localize a vehicle that drives in: a) the same static scene (resulting in a *static-trace*), and b) the same scene with parked and moving vehicles (resulting in a *dynamic-trace*). Figure 12 shows the average error and map sizes for each scheme and scenario. (We show the error distributions in Figure A.9 and Figure A.11).

In all three environments (suburbia, downtown, and freeway), the localization error for the static-trace in the static-map shows that CarMap is able to *localize as accurately as ORB-SLAM2* even though the *map sizes are 23-26*× smaller. Similar results hold for CarMap when compared against recent map crowdsourcing work, QuickSketch. This is because CarMap preserves all map-features that contribute most towards accurate localization.

However, for the dynamic-trace on the static-map, CarMap has nearly *28*× *better localization accuracy* than ORB-SLAM2 and Quicksketch. These differences arise from two features in our scenarios: traffic, and the presence of parked cars, which impact localization accuracy in different ways.

To understand why, consider a dynamic-trace on a suburban street. If the location or number of parked cars in the dynamic-trace are different from those in the static-map, the signature of the observed frame (its word histogram) is different in the trace than in the map. Because ORB-SLAM2 relies on word-histogram matching for re-localization, it fails to find the right keyframe candidates to localize. In contrast, because CarMap filters features belonging to parked cars, the vehicle in the suburban street sees similar features as in the map, and can re-localize more accurately.

Now consider a dynamic-trace on the freeway, in which a vehicle's view can be obscured by other vehicles, so it is unable to observe many of the features in the map. This causes ORB-SLAM2's word histogram matching to fail. CarMap uses all keyframes within a 50 m radius of its current position, so it always has keyframe candidates to search from. Even when histogram matching succeeds, ORB-SLAM2 uses per-keyframe word search trees that can result in false-positive feature matches. CarMap uses feature position based search to avoid this. In this scenario, moreover, ORB-SLAM2 believes features belonging to vehicles moving in the same direction to be stable (since their relative speed is near zero), makes them map-features and uses them to track its own motion. CarMap's dynamic object filter avoids this pitfall.

Dynamic Scene Maps. In this scenario, we build a map from a dynamic scene (a *dynamic-map*) and then use the map to localize in a dynamic- or static-trace. Figure 13 summarizes the results from this experiment (Figure A.10 plots the distribution of mapping errors).

The results for the dynamic-map are more dramatic than those for the static-map. CarMap's map is 15-36× smaller than ORB-SLAM2's or QuickSketch's map. Despite this, these two approaches *fail to localize* (denoted by ∞) on static-traces in downtown and suburban streets. In the static-trace, very few of the perceived features appear in the dynamic-map, and relocalization fails completely. CarMap does well here because it filters out all cars (parked or moving). For the dynamic-trace, its accuracy is nearly 50× better than ORB-SLAM2 and QuickSketch. CarMap's accuracy is lowest for the downtown dynamic-trace (with a 5% translational error) in which parked and moving cars obscure a lot of features in the map, resulting in fewer matches.

Multi-Lane Localization. In this set of experiments, we consider a somewhat more challenging case, for each of our scenarios: building a map by traversing one lane of a multi-lane street (4 freeway lanes, or 2 lanes in the suburban and downtown streets), and then trying to localize the vehicle in each of the remaining lanes. As before, we build both static-maps (Figure 14) and dynamic-maps (Figure 15).

For the freeway static-map, ORB-SLAM2 cannot localize beyond the second lane, while CarMap can localize across all four lanes. For the dynamic-map, a more challenging case, CarMap can localize one lane over, but ORB-SLAM2 and QuickSketch cannot localize at all (denoted by ∞). In all these cases, ORB-SLAM2's search strategy fails because its keyframe search relies on the vehicle's perspective being the same as the map's perspective: in these experiments, that assumption does not hold. CarMap, by contrast, matches features by position not perspective, so is much more robust.

Similar results hold for suburban and downtown streets: ORB-SLAM2 and QuickSketch are unable to localize, but CarMap is able to localize in all cases, with low error.

In §A.4, we show that CarMap's *mapping accuracy*, which measures the inherent error introduced by mapping, is comparable to ORB-SLAM2.

5.4 Other Performance Measures

Map Sizes in Real-World Traces. §5.3 shows that CarMap's maps are lean relative to competing strategies, but these are

Mapping scheme	Freeway			Suburbia			Downtown		
	Static error (%)	*Dynamic error (%)*	*Map size (MB)*	*Static error (%)*	*Dynamic error (%)*	*Map size (MB)*	*Static error (%)*	*Dynamic error (%)*	*Map size (MB)*
ORB-SLAM2	1.06	24.3	157.6	0.60	36.7	101.2	1.06	26.2	320.7
QuickSketch	0.95	24.4	157.6	0.60	37.0	99.1	0.90	22.1	303.0
CarMap	1.09	**2.06**	**5.94**	0.68	**1.01**	**3.94**	1.15	**3.87**	**14.3**

Figure 12: Mapping error and map sizes for a static-map used with static- and dynamic-traces, for each scenario. ∞ indicates that the scheme was not able to localize at all.

Mapping scheme	Freeway			Suburbia			Downtown		
	Static error (%)	*Dynamic error (%)*	*Map size (MB)*	*Static error (%)*	*Dynamic error (%)*	*Map size (MB)*	*Static error (%)*	*Dynamic error (%)*	*Map size (MB)*
ORB-SLAM2	6.33	62.3	110.6	∞	45.4	105.6	∞	25.9	291.1
QuickSketch	19.6	64.7	113.3	∞	44.7	108.4	∞	22.4	267.3
CarMap	**1.39**	**1.5**	**5.25**	**1.22**	**0.86**	**7.2**	**1.39**	**5.05**	**7.91**

Figure 13: Mapping error and map sizes for a dynamic-map used with static- and dynamic-traces, for each scenario. ∞ indicates that the scheme was not able to localize at all.

Mapping scheme	Freeway			Suburbia	Downtown
	2nd Lane	*3rd Lane*	*4th Lane*	*Parallel lane*	*Parallel lane*
ORB-SLAM2	3.79	∞	∞	∞	∞
QuickSketch	4.29	∞	∞	∞	∞
CarMap	**2.26**	**3.52**	**4.85**	**1.96**	**4.03**

Figure 14: Mapping error (%) for multi-lane localization in static environments using maps collected from one lane in other parallel lanes.

Mapping scheme	Freeway	Suburbia	Downtown
	Parallel lane	*Parallel lane*	*Parallel lane*
ORB-SLAM2	∞	∞	∞
QuickSketch	∞	∞	∞
CarMap	**4.80**	**5.24**	**9.81**

Figure 15: Mapping error (%) for multi-lane localization in dynamic environments using a map collected from one lane in other parallel lanes. CarMap is robust to spatio-temporal changes.

Mapping environment	Mapping scheme	Map: dynamic Trace: static	Map: static Trace: dynamic
Suburbia	QuickSketch	∞	∞
	CarMap	0.8	1.6
Downtown	QuickSketch	∞	∞
	CarMap	0.93	4.91

Figure 16: Mapping errors (m) for stitching map segments from different traffic conditions. CarMap is robust to temporal changes because: a) removes dynamics, and b) robust feature search.

DCNN	FPS	Label segmentation accuracy	Object segmentation accuracy
DeepLabV3+	1.1	76.7	96.6
Tuned DeepLabV3+	2.2	71.6	96.4
MobileNetV2	5	72.8	97.6

Figure 17: Semantic segmentation accuracy for different DCNNs. By classifying labels into static and dynamic objects, the segmentation accuracy for all DCNNs is above 96%.

for synthetically generated traces. Figure 18 shows the map sizes for the 11 real-world KITTI sequences. Across all sequences, CarMap *reduces map size 20×*. About 20% of this savings comes from removing the reconstructible indices (the *No Index* column), and another 60% from removing keyframe-features after generating the indices (the *No Keyframe-features* column).

As a vehicle travels faster, feature maps capture more data from the environment and generate data at a higher rate. We validate this in Figure 19 by calculating the bandwidth requirements of all 11 KITTI traces. Maps generated by ORB-SLAM2 and the No-Index approach are impractical at all speeds for LTE wireless upload and impractical for LTE download at speeds over 40 kph. The No Keyframe-features alternative is impractical for LTE upload at speeds over 60 kph. CarMap requires less than 3 Mbps up to 80 kph (the highest speed in the KITTI traces). Similar results hold for CarLA-generated traces (§A.3).

Other factors determine map size, including visual richness of the environment, lighting, weather *etc.* CarMap's map size should still be an order of magnitude smaller than competing approaches; future work can validate this.

Localization Time. CarMap's accuracy comes at the cost of a slightly higher per-frame localization time. During localization, CarMap's feature search adds overhead. To quantify this, we built a map from a very large trace with 4541 frames and then tried to localize in the same trace. ORB-SLAM2 has a per-frame localization cost of 0.023 s, while CarMap's is only marginally higher (0.033 s).

Map Load Time. When it receives a map segment, CarMap needs to read the segment from disk, reconstruct the keyframe features, and the indices. Figure 20 quantifies the total cost of these operations (called the map *load time*) for each of the 11 KITTI sequences. The load times for other alternatives are normalized by those for CarMap.

Interestingly, except for sequences 00, 01 and 06, load times for CarMap *are less than ORB-SLAM2* (on average, 0.95×). For most sequences, CarMap's load time is lower than ORB-SLAM2 because the latter's map is large enough that the time to load it from disk exceeds CarMap's reconstruction overhead. Other alternatives (*No Index* and *No Keyframe-features*) have large maps and high reconstruction overhead. When CarMap's reconstruction cost is (marginally) higher than ORB-SLAM2, it is because the corresponding scenes have a dense map-feature index, leading to a slightly higher reconstruction cost. (See §A.5 for details). Denser map-feature indices are found in environments with keyframes that have a large number of common map-features (*e.g.,* freeways). We have verified both these observations (equivalent map-load times and slightly higher load times for dense map-feature indices) for CarLA sequences.

Loop Closure. Loop closure is an important component of SLAM systems. For the KITTI dataset, we have verified that, even though its maps contain only map-features, CarMap can perform all loop closures that ORB-SLAM2 can.

5.5 Robustness

Robust Feature Matching. We compare CarMap's feature matching performance to that of ORB-SLAM2's native feature matching approach (we use ORB-SLAM2's default parameters for matching). For this, we build a map segment for a static trace and then use that trace to localize: a) the same static trace, b) a static trace from a parallel lane, c) a dynamic trace from the same lane, and d) a dynamic trace from a parallel lane. We collect the trace using CarLA on a freeway, and use two metrics: a) feature matching ratio (the percentage of map-features matched in the current trace), and b) localization error (m).

Figure 21 shows that for all scenarios, robust feature matching *is able to find more matches and hence results in lower*

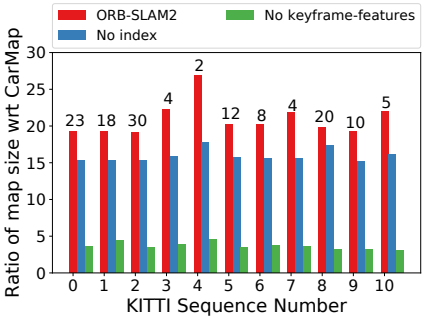

Figure 18: Map sizes on KITTI traces: for each alternative, the map size is normalized by CarMap's map size. The number on top of each group of bars shows the size in MB of CarMap's map for the corresponding KITTI trace. CarMap reduces map size by 20x for unmapped regions.

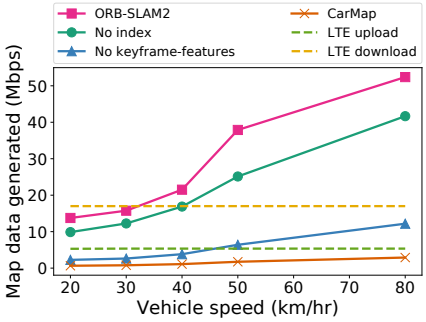

Figure 19: Bandwidth requirements for the four mapping schemes averaged over diverse environments in all 11 KITTI sequences at different speeds. CarMap can support near-real time uploads over LTE at speeds up to 80 kph whereas other schemes fail even at low speeds.

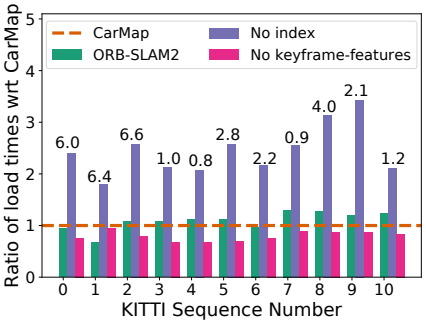

Figure 20: Load times on KITTI traces: for each alternative, the load times are normalized by CarMap's load time (whose absolute value is on top of each group of bars). CarMap loads faster than ORB-SLAM2 (*i.e.,* ORB-SLAM2's load time ratio > 1), except for 3 KITTI sequences.

Scheme	Feature matching (FM) ratio (%)	Localization error (m)
Map: Lane 1 (static) Trace: Lane 1 (static)		
Normal FM	70	0.83
Robust FM	**96**	**0.51**
Map: Lane 1 (static) Trace: Lane 1 (dynamic)		
Normal FM	49	1.4
Robust FM	**66**	**0.57**
Map: Lane 1 (static) Trace: Lane 2 (static)		
Normal FM	52	1.2
Robust FM	**74**	**0.66**
Map: Lane 1 (static) Trace: Lane 2 (dynamic)		
Normal FM	32	1.5
Robust FM	**40**	**0.6**

Figure 21: CarMap's robust feature matching finds more features in different conditions and thus localize better than ORB-SLAM2.

Base map lane	Mapping scheme	Map segment lane			
		1st	2nd	3rd	4th
1st	QuickSketch	0.67	0.90	∞	∞
	CarMap	0.70	**0.88**	∞	∞
2nd	QuickSketch	0.34	0.68	0.90	∞
	CarMap	1.1	**0.62**	0.97	**1.15**
3rd	QuickSketch	∞	5.8	0.68	3.2
	CarMap	**1.13**	**0.96**	**0.63**	**1.19**
4th	QuickSketch	∞	∞	10.83	0.67
	CarMap	∞	**1.0**	**1.0**	**0.66**

Figure 22: Mapping error (m) for multi-lane stitching. CarMap's stitching algorithm uses a more robust feature search based on position hints to stitch map segments two lanes apart where competing strategies fail (∞ shows an unsuccessful stitch operation.)

localization error as compared to ORB-SLAM2's feature matching. The base case (static-map used by a static trace) shows that normal feature matching fails to detect 30% of the features even though the same trace is used for mapping and localization. The introduction of dynamic objects reduces the feature matching ratio because features are occluded by vehicles and hence cannot be detected even with robust matching.

Making Semantic Segmentation Robust. CarMap makes segmentation robust by voting across multiple keyframes, and using a coarser static vs. non-static classification. Figure 17 shows CarMap's overall accuracy, for three different versions of DeepLabv3+. These DNNs are the DeepLabv3+ trained on the CityScape dataset pre-trained, a fined tuned DeepLabv3+ trained on the KITTI dataset and a light-weight version of DeepLabv3+ (MobileNetv2) for mobile devices. The third column shows that CarMap achieves upwards of 96% accuracy if we apply segmentation to every keyframe. Semantic segmentation, by itself, achieves only 70% accuracy in label assignment (second column).

The first column shows the frame rate these DNNs run at. The frame rate needs to be fast enough to process every keyframe, or at worst, every other keyframe (at which segmentation accuracy drops to about 85%, and below which it drops to unacceptable levels, §A.7). In the KITTI dataset, the average across the 11 sequences is 3.17 keyframes per second, well within the rate of the MobileNetv2 version. One of these sequences runs at 10 keyframes per second, so for this sequence MobileNetv2 would process every other keyframe. For more dynamic scenes, it might be necessary to devise faster semantic segmentation techniques, and we expect the vision community will make advances in this direction.

Multi-Lane Stitching. CarMap can stitch map segments collected from different lanes. For this experiment, we collect traces from four parallel lanes on a freeway in CarLA. Using each of these four traces as base maps, we try to stitch map segments from other lanes into it, then evaluate the mapping error for the new maps. Figure 22 shows the absolute mapping errors (in meters) for these stitched map segments. The first column shows the lane used to collect the base map and the last four columns show the absolute mapping error of a stitched map with each of these lanes. The ∞ sign represents a failure to stitch segments from the two lanes.

Although QuickSketch's base map has 20× more features than CarMap and it localizes a stereo camera trace in that base map instead of another map segment (CarMap), it cannot stitch two lanes away. On the other hand, CarMap's stitching algorithm uses robust feature matching (§3.2) and can stitch map segments collected two lanes away (*e.g.,* map segments

from lane 1 and lane 3). CarMap's robustness comes purely from using position hints to find the set of key-frames to match, and to find matching map features, while QuickSketch uses ORB-SLAM2's built-in matching methods (in this experiment, we do not compare against ORB-SLAM2 because it does not contain a map stitch operation).

Stitching in Different Traffic Conditions. Besides being robust to spatial changes, crowdsourced map collection and update requires robustness to temporal changes as well (*e.g.,* changes in traffic during different times of day). To evaluate this, we collect stereo camera traces from CarLA in suburban and downtown areas in the same environment during different traffic conditions (no traffic and heavy traffic). Using these traces, we evaluate the ability of the mapping schemes to stitch these map segments by comparing their mapping error.

Figure 16 shows that QuickSketch is unable to stitch because it fails to relocalize a trace in different traffic conditions (§5.3). This, again, is because its stitching is solely based on appearance-based matching whereas CarMap uses position hints as well to make its stitching more robust. By contrast, CarMap is able to stitch map segments collected across different traffic conditions. We evaluate the sensitivity of stitching accuracy to the degree of map segment overlap in §A.6.

6 Related Work

Decentralized SLAM. Decentralized SLAM systems [24] leverage multiple agents to run SLAM in unknown environments. CarMap can be considered an instance of decentralized SLAM [22] with some differences. In decentralized SLAM, the agents (robots) have limited compute-power and only run visual odometry [29]. This leads to inaccurate localization whereas vehicles in CarMap localize more accurately because they run both mapping and localization. Decentralized SLAM sends all keyframe features to a central collector which performs all mapping operations [53] whereas CarMap only sends map-features to a cloud service to ensure real-time map exchanges. Similarly, in decentralized SLAM, the collector finds overlap between maps of different agents using the histogram word approach, does not remove environmental dynamics and hence is not robust like CarMap. Decentralized SLAM [47] uses features from a single keyframe overlap to compute the transformation matrix whereas CarMap is more robust and uses features from multiple keyframes.

Visual SLAM. Although we have implemented CarMap on top of ORB-SLAM2 [41], our study of other SLAM systems shows that it can be easily ported to other keyframe-based visual SLAM algorithms like S-PTAM [45]. In future work, we can extend CarMap to group features into higher-dimensional planes [32] to further improve localization accuracy. As wireless speeds increase, it might be possible to design over-the-air map updates for dense mapping systems like [38] using techniques similar to ours. We have left this to future work.

Long Term Mapping. Our implementation uses traditional computer vision-based features (ORB [49]) to build the map, but these can be replaced with better, more stable CNN-based features [25]. After running a feature extractor, CarMap uses motion tracking and semantic segmentation to select stable features to build the map. Mask-SLAM [33] proposes a similar dynamic object filter to CarMap but CarMap uses majority voting and robust labeling to account for limited on-board computational resources and boundary segmentation errors. Other approaches [17, 34] remove dynamic features from multiple maps collected along the same trace using background subtraction. Even the most static features are not persistent for larger timescales. Future work for longer timescale mapping can integrate CarMap with a persistence filter presented in [48] that estimates the life period of a feature based on an environmental evolution model. CarMap benefits from map-element culling techniques [35] that scale maps sizes by the scale of the environment rather than the number of miles driven. Mobileye [10] crowdsources collecting 3D maps for vehicles using monocular cameras whereas CarMap is designed for 3D sensors like LiDARs, and stereo cameras.

Vehicle Sensing and Communication. LiveMap [21] uses GPS and monocular cameras to automate road abnormality detection (*e.g.,* pothole detection). With its depth perception capabilities, CarMap can more accurately position roadside hazards. AVR [46] extends vehicular vision using feature maps and would benefit from CarMap. Although the bandwidth requirements for CarMap are within the LTE speeds today, it can benefit from systems [36] that schedule redundant transmissions over multiple networks. Recent work in object detection on mobile devices [39] introduces a fast object tracking method that can be used in CarMap to enable faster segmentation. For stitching map segments from rural, unmapped regions, CarMap can benefit from [44] which enables autonomous navigation in such areas.

7 Conclusion

CarMap enables near real-time crowd-sourced updates, over cellular networks, of feature-based 3D maps of the environment. It finds a lean representation of a feature map that fits within wireless capacity constraints, incorporates robust position-based feature search, removes dynamic and semi-dynamic features to enable better localization, and contains novel map update algorithms. CarMap has better localization accuracy than competing approaches, and can localize even when other approaches fail completely. Future work can explore LiDAR sensors, mapping over timescales in which even relatively static features can disappear, dense map representations, infrastructure-based sensing for map updates in low vehicle density areas, and automated update of semantic map overlays (accidents, available parking spots).

Acknowledgements. Our shepherd Kyle Jamieson and the anonymous reviewers provided valuable feedback. The work was supported by grants from the US National Science Foundation (Grant No. CNS-1330118) and General Motors.

References

[1] Apple Is Rebuilding Maps From the Ground Up. https://techcrunch.com/2018/06/29/apple-is-rebuilding-maps-from-the-ground-up/, 2018.

[2] Here Self-Healing Maps. https://go.engage.here.com/self-healing.html, 2018.

[3] State of Mobile Networks: USA - OpenSignal. https://opensignal.com/reports/2018/07/usa/state-of-the-mobile-network, 2018.

[4] The Golden Age of HD Mapping for Autonomous Driving. https://medium.com/syncedreview/the-golden-age-of-hd-mapping-for-autonomous-driving-b2a2ec4c11d, 2018.

[5] There's No Google Maps for Self-Driving So This Startup Is Building It. https://www.technologyreview.com/s/612202/theres-no-google-maps-for-self-driving-cars-so-this-startup-is-building-it/, 2018.

[6] Apple Maps Image Collection. https://maps.apple.com/imagecollection/, 2019.

[7] Baidu. https://www.baidu.com/, 2019.

[8] Carmera. https://www.carmera.com/fleets/, 2019.

[9] GM's Hands-free Driving Feature to Work on 70,000 Additional Miles of Highways This Year. https://www.theverge.com/2019/6/5/18653628/gms-super-cruise-hands-free-driving-feature-highway-milage, 2019.

[10] HERE and Mobileye: Crowdsourced HD Mapping for Autonomous Cars. https://360.here.com/2016/12/30/here-and-mobileye-crowd-sourced-hd-mapping-for-autonomous-cars/, 2019.

[11] Kuandeng. http://www.kuandeng.com/html/1/index.html, 2019.

[12] Lyft Level 5. https://level5.lyft.com/, 2019.

[13] NVIDIA Drive AGX. https://www.nvidia.com/en-us/self-driving-cars/drive-platform/hardware/, 2019.

[14] Upgrading Uber's 3D Fleet. https://medium.com/uber-design/upgrading-ubers-3d-fleet-4662c3e1081, 2019.

[15] Fawad Ahmad, Hang Qiu, Xiaochen Liu, Fan Bai, and Ramesh Govindan. QuickSketch: Building 3D Representations in Unknown Environments using Crowdsourcing. In *2018 21st International Conference on Information Fusion (Fusion)*, pages 2314–2321. IEEE, 2018.

[16] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, September 1975.

[17] Julie Stephany Berrio, James Ward, Stewart Worrall, and Eduardo Nebot. Identifying Robust Landmarks in Feature-based Maps. *arXiv preprint arXiv:1809.09774*, 2018.

[18] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[19] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, Jose Neira, Ian Reid, and John J. Leonard. Past, Present, and Future of Simultaneous Localization and Mapping: Toward the Robust-perception Age. *Trans. Rob.*, 32(6):1309–1332, December 2016.

[20] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-decoder with Atrous Separable Convolution for Semantic Image Segmentation. In *ECCV*, 2018.

[21] Kevin Christensen, Christoph Mertz, Padmanabhan Pillai, Martial Hebert, and Mahadev Satyanarayanan. Towards a Distraction-free Waze. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pages 15–20. ACM, 2019.

[22] Titus Cieslewski, Siddharth Choudhary, and Davide Scaramuzza. Data-efficient Decentralized Visual Slam. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2466–2473. IEEE, 2018.

[23] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The Cityscapes Dataset for Semantic Urban Scene Understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3213–3223, 2016.

[24] Alexander Cunningham, Manohar Paluri, and Frank Dellaert. Ddf-sam: Fully Distributed Slam using Constrained Factor Graphs. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3025–3030. IEEE, 2010.

[25] Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. Superpoint: Self-supervised Interest Point Detection and Description. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 224–236, 2018.

[26] P. Deutsch. RFC1952: GZIP File Format Specification Version 4.3, 1996.

[27] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An Open Urban Driving Simulator. *arXiv preprint arXiv:1711.03938*, 2017.

[28] Jakob Engel, Thomas Schöps, and Daniel Cremers. LSD-SLAM: Large-scale Direct Monocular Slam. In *European conference on computer vision*, pages 834–849. Springer, 2014.

[29] Christian Forster, Simon Lynen, Laurent Kneip, and Davide Scaramuzza. Collaborative Monocular Slam with Multiple Micro Aerial Vehicles. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3962–3970. IEEE, 2013.

[30] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are We Ready for Autonomous Driving? the Kitti Vision Benchmark Suite. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3354–3361. IEEE, 2012.

[31] Here. The Self-healing Map From Here. https://go.engage.here.com/self-healing.html, 2019.

[32] Mehdi Hosseinzadeh, Yasir Latif, and Ian Reid. Sparse Point-plane Slam. In *Australasian Conference on Robotics and Automation 2017 (ACRA 2017)*.

[33] Masaya Kaneko, Kazuya Iwami, Toru Ogawa, Toshihiko Yamasaki, and Kiyoharu Aizawa. Mask-SLAM: Robust Feature-based Monocular SLAM by Masking using Semantic Segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 258–266, 2018.

[34] B Ravi Kiran, Luis Roldao, Beñat Irastorza, Renzo Verastegui, Sebastian Süss, Senthil Yogamani, Victor Talpaert, Alexandre Lepoutre, and Guillaume Trehard. Real-time Dynamic Object Detection for Autonomous Driving using Prior 3d-maps. In *European Conference on Computer Vision*, pages 567–582. Springer, 2018.

[35] Henrik Kretzschmar, Giorgio Grisetti, and Cyrill Stachniss. Lifelong Map Learning for Graph-based SLAM in Static Environments. *KI*, 24:199–206, 09 2010.

[36] HyunJong Lee, Jason Flinn, and Basavaraj Tonshal. Raven: Improving Interactive Latency for the Connected Car. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 557–572. ACM, 2018.

[37] Shiqi Li, Chi Xu, and Ming Xie. A Robust O (n) Solution to the Perspective-n-point Problem. *IEEE transactions on pattern analysis and machine intelligence*, 34(7):1444–1450, 2012.

[38] Yonggen Ling and Shaojie Shen. Building Maps for Autonomous Navigation using Sparse Visual Slam Features. In *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*, pages 1374–1381. IEEE, 2017.

[39] Luyang Liu, Hongyu Li, and Marco Gruteser. Edge Assisted Real-time Object Detection for Mobile Augmented Reality. In *Proceedings of the 25th Annual International Conference on Mobile Computing and Networking*. ACM, 2019.

[40] Xiaochen Liu, Suman Nath, and Ramesh Govindan. Gnome: A Practical Approach to NLOS Mitigation for GPS Positioning in Smartphones. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, page 163–177, New York, NY, USA, 2018. Association for Computing Machinery.

[41] Raul Mur-Artal and Juan D Tardós. ORB-SLAM2: An Open-source Slam System for Monocular, Stereo, and Rgb-d Cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, 2017.

[42] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

[43] Society of Automotive Engineers International. Automated Driving Levels of Driving Automation Are Defined in New SAE International Standard J3016. (2014), 2014.

[44] Teddy Ort, Liam Paull, and Daniela Rus. Autonomous Vehicle Navigation in Rural Environments Without Detailed Prior Maps. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2040–2047. IEEE, 2018.

[45] Taihú Pire, Thomas Fischer, Gastón Castro, Pablo De Cristóforis, Javier Civera, and Julio Jacobo Berlles. S-ptam: Stereo Parallel Tracking and Mapping. *Robotics and Autonomous Systems*, 93:27–42, 2017.

[46] Hang Qiu, Fawad Ahmad, Fan Bai, Marco Gruteser, and Ramesh Govindan. AVR: Augmented Vehicular Reality. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services (Mobisys)*, MobiSys '18, pages 81–95, Munich, Germany, 2018. ACM.

[47] Luis Riazuelo, Javier Civera, and JM Martínez Montiel. C2tam: A Cloud Framework for Cooperative Tracking and Mapping. *Robotics and Autonomous Systems*, 62(4):401–413, 2014.

[48] David M Rosen, Julian Mason, and John J Leonard. Towards Lifelong Feature-based Mapping in Semi-static Environments. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1063–1070. IEEE, 2016.

[49] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An Efficient Alternative to Sift or Surf. 2011.

[50] Radu B Rusu and S Cousins. Point Cloud Library (pcl). In *2011 IEEE International Conference on Robotics and Automation*, pages 1–4, 2011.

[51] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted Residuals and Linear Bottlenecks. In *CVPR*, 2018.

[52] Boris Schling. *The Boost C++ Libraries*. XML Press, 2011.

[53] Patrik Schmuck and Margarita Chli. Multi-uav Collaborative Monocular Slam. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3863–3870. IEEE, 2017.

[54] Lu Sun, Junqiao Zhao, Xudong He, and Chen Ye. DLO: Direct Lidar Odometry for 2.5d Outdoor Environment. *2018 IEEE Intelligent Vehicles Symposium (IV)*, Jun 2018.

[55] Bill Triggs, Philip F McLauchlan, Richard I Hartley, and Andrew W Fitzgibbon. Bundle Adjustment—a Modern Synthesis. In *International workshop on vision algorithms*, pages 298–372. Springer, 1999.

[56] Waymo. Building Maps for a Self-driving Car. https://medium.com/waymo/building-maps-for-a-self-driving-car-723b4d9cd3f4, 2016.

[57] Ji Zhang and Sanjiv Singh. LOAM: Lidar Odometry and Mapping in Real-time. In *Robotics: Science and Systems*, volume 2, page 9, 2014.

[58] Ji Zhang and Sanjiv Singh. Visual-lidar Odometry and Mapping: Low-drift, Robust, and Fast. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2174–2181. IEEE, 2015.

A Appendix

A.1 Map Stitching Details

Algorithm A.1 describes the details of the stitching algorithm. The following two paragraphs discuss two key aspects of stitching.

Finding Overlap. To find potential regions of overlap, CarMap uses two strategies. When the cloud service receives the new map segment M_s, it uses the GPS positions and word-histograms associated with M_s to coarsely find potentially overlapping keyframes in the base map M_b. For this, CarMap reconstructs all the data structures in M_b and only word-histograms and keyframe-features of M_s using the methods described in §3.5.

Then, CarMap finds a finer-grained overlap O_b and O_s (granularity level of map-points) between M_s and M_b. For this, CarMap uses the reconstructed keyframe features of M_s. For each keyframe k_s in O_s, it uses the k-D tree to find *all* features (§3.2) in O_b that match features in k_s, instead of only matching features belonging to the two overlapping keyframes k_s and k_b. At the end of this process, there is a pairwise matching of features between O_b and O_s.

Input : Base map M_b and new map segment M_s
Output: Stitched base map M_b'

if M_b *is empty* **then**
  $M_b' \leftarrow M_s$;
else
  $R_b \leftarrow$ `Reconstruct(`M_b`)`;
  $R_s \leftarrow$ `PartialReconstruct(`M_s`)`;
  $O_b, O_s \leftarrow$ `FindOverlap(`R_b,R_s`)`;
  $T_{bs} \leftarrow$ `FindTransform(`O_b, O_s`)`;
  $M_s^\star \leftarrow T_{bs} * M_s$;
  $M_b' \leftarrow$ `Merge(`M_b, $M_s^\star$`)`;
end

Algorithm A.1: Stitching Algorithm

Computing the transformation matrix. In the next step, CarMap computes the transform (translation and rotation) to re-orient and position M_s in M_b. To do this, it finds the keyframe k_s from the new map segment with the maximum number of matched features from the previous step. Then it uses a perspective n-point (PnP [37]) solver to derive the coordinate transformation matrix, then transforms each map feature in M_s to M_b's frame of reference. After the transformation, CarMap removes all the duplicate map-features in the overlapping region O_b of the resulting base map M_b' that originated as a result of the transformation.

A.2 Implementation Details

The following paragraphs describe how we have implemented CarMap components on top of ORB-SLAM2.

Map segment generator. This component takes the output

of ORB-SLAM2 (which includes map-features, keyframe features, and the two indices), and simply strips all other components other than the map-features. We have also added the ability to periodically transmit complete *map segments*. On the receiver, we added a module to reconstruct (§3.5) the keyframe features from the received lean map, and re-generates the indices.

Fast feature search. For this, we added the k-D tree data structure, and associated code for manipulating the tree and searching in the tree, and re-used ORB-SLAM2 code for re-positioning a feature in a keyframe.

Stitcher. Stitching functionality does not exist in ORB-SLAM2. For stitching maps, we wrote our own modules for ORB-SLAM2. We also added support for finding overlapped keyframes and computing the transformation matrix.

Map updater. We wrote our own module for map updates. At the vehicle, our map update module uses a fast feature search for finding differences in the two feature sets (environment and base map). At the cloud, the module integrates these differences into the base map.

Dynamic object filter. We added a dynamic object filter to the mapping component of ORB-SLAM2 which invokes semantic segmentation and applies majority voting to decide the label associated with each map feature.

Map exchange. We added another module to allow the exchange of *map segments*, map updates, and the base map between the vehicles and the cloud service.

A.3 Bandwidth Requirements

Map Size with Change in Speed. As a vehicle's speed increases, it sees more features and hence generates larger maps. As such, we generated CarLA traces in which we increased the speed of the vehicle while keeping time constant. The goal of this experiment is to see if CarMap's maps can stay within the wireless bandwidth limits at different speeds. Figure A.1 shows that CarMap's maps are well below the wireless bandwidth limits today by a large margin and this is not true for competing strategies. ORB-SLAM2 and the No index approach's maps cannot be uploaded over current wireless networks at all speeds and cannot be downloaded for speeds greater than 10 kmph. The No keyframe-features approach is also infeasible for LTE upload for speeds over 15 kmph. We also validated this in Figure 19 for real-world traces from the KITTI dataset.

Bandwidth Savings with Map Updates. In this section, we evaluate the ability of CarMap's update operation to reduce the amount of bandwidth required to update the base map. For these experiments, we collected traces from the same area in CarLA in three different traffic conditions *i.e.,* static with no parked vehicles, semi-dynamic with only parked vehicles and dynamic with both parked and moving vehicles. We build a map for each traffic condition and then measure the amount of bandwidth required to update the existing map with features

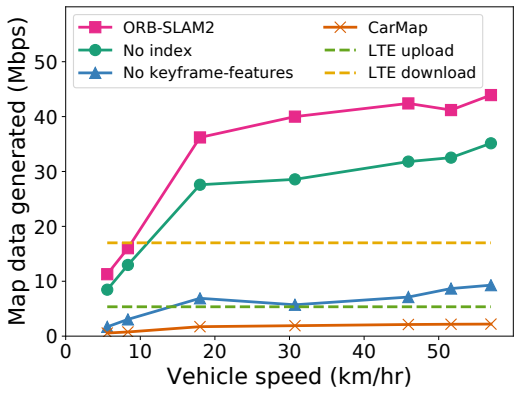

Figure A.1: Bandwidth requirements for mapping schemes at different speeds in CarLA. The bandwidth required to upload CarMap maps are well below the LTE upload limits.

Pre-loaded map	Trace	Map stitch size (MB)	Map update size (MB)
Static	Static	0.81	0.21
Static	Semi-dynamic	0.80	0.35
Static	Dynamic	0.82	0.29
Semi-dynamic	Static	0.81	0.27
Semi-dynamic	Semi-dynamic	0.80	0.08
Semi-dynamic	Dynamic	0.82	0.27
Dynamic	Static	0.81	0.24
Dynamic	Semi-dynamic	0.80	0.25
Dynamic	Dynamic	0.82	0.12

Figure A.2: Bandwidth requirements for map updates in CarMap under different traffic conditions

from a different set of conditions. The baseline we compare is with the map stitch case in which we would upload the whole map segment to the cloud service and the cloud service would only add the new map elements to the map.

The results from the experiment (Figure A.2) show that, given a base map of the area, map updates can reduce the amount of bandwidth required to integrate new features in the base map by 4-10× compared to sending the whole map segment (75× savings as compared to QuickSketch and ORB-SLAM2). This happens because the map update only sends new features whereas the map stitch sends the whole perceived map segment.

A.4 Mapping Accuracy

In this section, we evaluate how CarMap's reduced map sizes affect localization accuracy. For this experiment, we use all 11 real-world traces from the KITTI dataset. We generate maps for each of these traces, use them as base maps and localize the same trace in these maps. We compare the generated trajectory with the ground truth positions. Figure A.3 shows the average localization error divided by the length of the whole sequence for all the KITTI sequences. Even though CarMap reduces map sizes by a factor of 20, it is able to localize as accurately as ORB-SLAM2 in almost all KITTI sequences because: a) it preserves the most important map elements (map-features), and b) robust feature matching.

KITTI sequence number	Average translational error over whole trace (%)			
	ORB-SLAM2	No index	No keyframe-features	CarMap
00	0.16	0.17	0.17	0.16
01	0.69	0.69	0.68	0.74
02	0.18	0.17	0.17	0.18
03	0.31	0.32	0.32	0.36
04	0.33	0.31	0.39	0.40
05	0.07	0.07	0.07	0.08
06	0.12	0.21	0.20	0.20
07	0.14	0.15	0.14	0.15
08	0.33	0.30	0.26	0.30
09	0.32	0.32	0.30	0.29
10	0.46	0.50	0.50	0.48

Figure A.3: Localization error for CarMap over all KITTI sequences. Even though CarMap uses 20X fewer features in its map, its localization error is almost the same as ORB-SLAM2.

Total distance covered (m)	Average translational error (%)			
	ORB-SLAM2	No index	No keyframe-features	CarMap
50	3.62	3.61	3.75	2.70
100	2.25	2.15	2.22	1.85
150	1.55	1.54	1.60	1.50
200	0.88	1.22	1.27	1.30
300	0.73	0.91	0.94	0.99
400	0.66	0.75	0.78	0.80
600	0.56	0.60	0.62	0.58
800	0.53	0.55	0.53	0.49
1000	0.30	0.30	0.31	0.31
2000	0.32	0.30	0.30	0.32
3000	0.24	0.23	0.22	0.23
4000	0.18	0.17	0.17	0.18
5000	0.18	0.17	0.17	0.18

Figure A.4: Mapping accuracy of mapping schemes with varying distance, averaged over all KITTI sequences. The overall localization error decreases over longer distances and CarMap's localization error is almost the same as ORB-SLAM2's

Figure A.8 shows the error distribution of CarMap is similar to ORB-SLAM2, and QuickSketch for a map built from, and used in the first KITTI sequence, despite reducing map sizes by a factor of 20.

An important property of a map is to able to localize accurately over long distances. To study how CarMap's localization accuracy changes with the mapped area, we calculate the average translational error at different distances (*i.e.,* 50m to 5km) for all 11 KITTI sequences. We average these errors on all KITTI sequences and report the numbers in Figure A.4. As distance increases, the average translational error decreases and CarMap does as well as ORB-SLAM2 in almost all cases. The reason for this, as mentioned in §3, is that although CarMap removes keyframe-features, the robust feature matching (§3.2) makes up for the 20x fewer features with better matching.

A.5 Map Reconstruction

CarMap reduces map size by trading off compute for storage. The map load time for CarMap consists of the time to load the map from disk and the reconstruction time. After loading the map into memory, CarMap reconstructs two indices and infers the 2D and 3D position of map-features in keyframes (§3.5). Even so, as shown in Figure 20, except for sequence 00, 01 and 06, the load times for CarMap *are less than the ORB-SLAM2 baseline* (on average, 0.95×).

Figure A.5 shows the breakdown of the various map elements that contribute to map reconstruction time for all 11 KITTI sequences. In all sequences, reconstructing the feature-index takes around 40% of the overall reconstruction time. This, however, is still 2-4x less than the reconstruction time for keyframes that contain keyframe-features (in other mapping schemes) instead of just map-features. Calculating the 2D and 3D positions of map-features also takes an average 35% of the overall reconstruction time. The main reason for higher load times (Figure 20), as compared to ORB-SLAM2, in some cases (sequence 00, 01, and 06) is because of the variability map-feature index (orange bar) reconstruction times. The map-feature index is a graph that relates map-points to keyframes they were detected in. Hence, for environments like highways where the scene stays relatively constant, this graph is denser and so the reconstruction costs for the map-feature index are relatively greater. On the other hand, for environments where features change quickly *e.g.,* narrow streets, the map-feature index reconstruction times are lower because these graphs are not as dense. For instance, the feature-index reconstruction for sequence 00 (captured in narrow-streets) is approximately 3x greater than sequence 01 (captured on the highway).

A.6 Map Stitching Evaluation

In this section, we evaluate the ability of CarMap to accurately stitch map segments collected from different spatial and temporal conditions. We compare CarMap against two other map stitching schemes: progressive relocalization and QuickSketch. In progressive relocalization, as opposed to CarMap (one-shot stitching), we relocalize *every* keyframe from the incoming map segment instead of using the global transformation matrix. QuickSketch can only stitch a stereo camera trace with a QuickSketch generated map segment. So, for stitching, QuickSketch loads the QuickSketch map as a base map and then stitches by localizing the stereo camera trace in it.

We evaluate two metrics for stitching: mapping error, and stitching time. After stitching two map segments, we localize a trace in the stitched map and calculate the absolute translational error (m) for each frame. Mapping error is the mean of the translational errors over the whole trace. The stitching time is the amount of time required to do the whole stitch operation of two map segments.

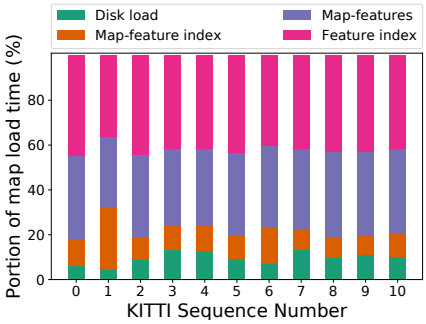

Figure A.5: Breakdown of reconstruction time for CarMap across all KITTI sequences

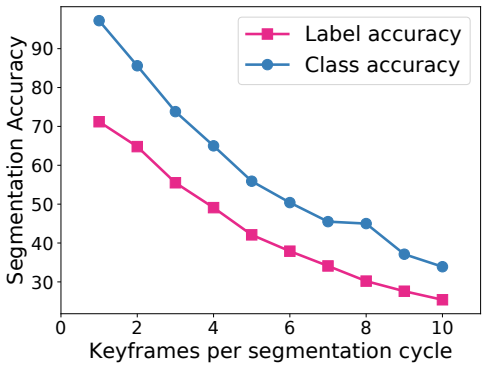

Figure A.6: Semantic segmentation accuracy at different frame rates. If CarMap segments every other keyframe, classification accuracy is 85%.

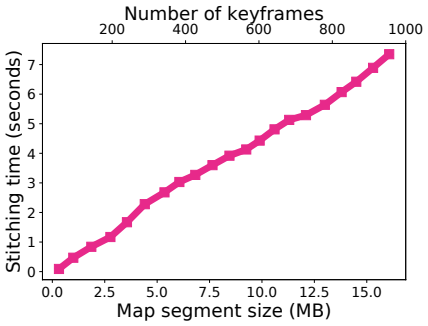

Figure A.7: Computational overhead of stitching. Even map segments as large as 1000 keyframes can be stitched in under 7 seconds.

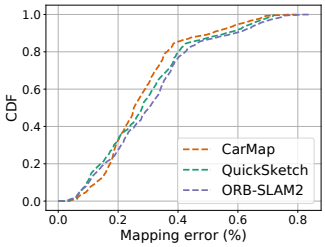

Figure A.8: For a map built, and used from a real-world trace (KITTI Trace 00) 80% of CarMap's mapping errors are less than 0.4% with respect to the length of the trace.

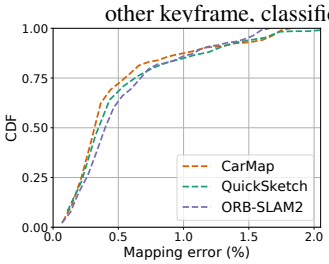

Figure A.9: For a map built, and used in a static trace collected from CarLA, 75% of the mapping errors for CarMap are less than 0.2% with respect to the length of the trace.

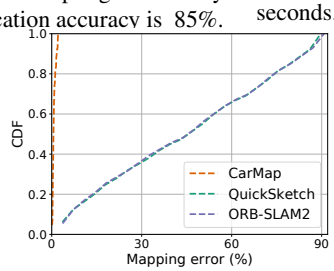

Figure A.10: For maps built, and used in CarLA's dynamic environments, CarMap has a maximum error of 2%. ORB-SLAM2 and QuickSketch have maximum errors of 90%.

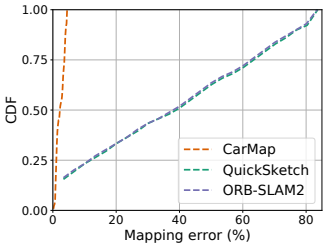

Figure A.11: For CarLA maps built from static, and used in dynamic environments, CarMap has a max error of 4%. ORB-SLAM2 and QuickSketch have maximum errors of 90%.

Overlapping frames	Absolute translational error (m)			Stitching time (seconds)		
	One shot stitching (CarMap)	Progressive relocalization	QuickSketch	One shot stitching (CarMap)	Progressive relocalization	QuickSketch
5	0.64	0.64	∞	1.50	46.90	∞
10	0.63	0.64	∞	1.62	44.29	∞
15	0.62	0.61	0.62	1.72	50.44	8.72
20	0.63	0.63	0.61	1.60	45.59	9.17
25	0.65	0.66	0.62	1.60	46.10	9.44
30	0.66	0.67	0.62	1.65	44.70	9.80
35	0.66	0.67	0.63	1.73	45.36	10.25
40	0.64	0.64	0.63	1.66	48.32	10.74
45	0.67	0.66	0.63	1.58	47.90	11.06
50	0.66	0.67	0.63	1.50	45.73	11.50

Figure A.12: Mapping error (m) with different overlapping regions. CarMap can stitch with fewer overlapping frames than QuickSketch and 30x faster than progressive relocalization.

Stitching Overlap. In the first experiment, we evaluate the mapping error and stitching time of the three mapping schemes as a function of the overlap between the two map segments. For this, we take a single stereo camera trace and split it into two traces with different overlaps. Figure A.12 shows that QuickSketch fails to stitch when the number of overlapping frames between the two map segments is less than 10 frames (1 second). This is because it is not able to find enough feature matches between the two map segments. On the other hand, CarMap can find enough feature matches even though it uses 20x fewer features due to its robust feature matching (§3.2). The mapping accuracy remains relatively constant irrespective of the amount of overlap because CarMap only needs to localize a single keyframe in the base map for a successful stitch operation. Although the mapping error of progressive relocalization is identical to CarMap, it takes approximately 30x more time to stitch the same area. In the stitch operation, localizing a keyframe in the base map is the most expensive operation. CarMap intelligently localizes a single keyframe in the base map and then uses a transformation matrix to shift the remaining map elements. On the other hand, progressive relocalization localizes all keyframes in the base map and hence takes a much longer time. So, as the size of the incoming map segment increases, the stitching time for progressive relocalization will increase significantly.

Stitching Overhead. To study the overhead of stitching, we take a KITTI trace and split it into two map segments (with a few overlapping frames). In doing so, we mark one as the base map and the other as the incoming map segment. We keep the size of the base map constant and vary the size of the incoming map segment. Figure A.7 shows that the stitching time increases with the size of the incoming map segment. It also shows that for map segments containing as many as 1,000 keyframes (15 MB), stitching takes only 7 seconds.

A.7 Semantic Segmentation

In this experiment, we evaluate the object label and class (static, and dynamic) estimation accuracy of CarMap against the frame rate of semantic segmentation. For this experiment, we generate stereo camera traces from CarLA. We segment these images with MobileNetV2. For ground truth, we use CarLA's own semantic segmented images.

Figure A.6 plots the accuracy of segmentation in CarMap using majority voting at different frame rates. We start by running segmentation every keyframe and evaluate till running segmentation every 10 keyframes. In the KITTI dataset, the average keyframes inserted per second is 3.17 and the worst case is 10 keyframes per second. The worse case corresponds to running segmentation every 2 keyframes *i.e.,* a class accuracy of 86% with CarMap using MobileNetv2 in a majority voting scheme.