2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation

Kestrel: Video Analytics for Augmented Multi-Camera Vehicle Tracking

Hang Qiu[†], Xiaochen Liu[†], Swati Rallapalli[*], Archith J. Bency[‡], Kevin Chan[**]
Rahul Urgaonkar[♯], B. S. Manjunath[‡], Ramesh Govindan[†]
[†]University of Southern California, [‡]UCSB, [*]IBM Research, [**]ARL, [♯]Amazon
{hangqiu, liu851, ramesh}@usc.edu, {archith, manj}@ece.ucsb.edu
srallapalli@us.ibm.com, kevin.s.chan.civ@mail.mil, urgaonka@amazon.com

***Abstract*—In the future, the video-enabled camera will be the most pervasive type of sensor in the Internet of Things. Such cameras will enable continuous surveillance through *heterogeneous camera networks* consisting of fixed camera systems as well as cameras on mobile devices. The challenge in these networks is to enable *efficient video analytics*: the ability to process videos cheaply and quickly to enable searching for specific events or sequences of events. In this paper, we discuss the design and implementation of Kestrel, a video analytics system that tracks the path of vehicles across a heterogeneous camera network. In Kestrel, fixed camera feeds are processed on the cloud, and mobile devices are invoked *only* to resolve ambiguities in vehicle tracks. Kestrel's mobile device pipeline detects objects using a deep neural network, extracts attributes using cheap visual features, and resolves path ambiguities by careful association of vehicle visual descriptors, while using several optimizations to conserve energy and reduce latency. Our evaluations show that Kestrel can achieve precision and recall comparable to a fixed camera network of the same size and topology, while reducing energy usage on mobile devices by more than an order of magnitude.**

***Index Terms*—Cyber-physical Systems, Video Analytics, Vehicle Trajectory Inference, Heterogeneous Camera Network**

I. Introduction

Video cameras will soon be, if they are not already, the most pervasive type of sensor in the Internet of Things. They are ubiquitous in public places, available on mobile devices and drones, in cars as dashcams, and on security personnel as bodycams. Such cameras are valuable because they provide rich contextual information, but pose a challenge for the very same reason: it requires significant manual effort to extract meaningful semantic information from these videos. To address this, recent research [20] has started exploring the design of *video analytics* systems, which automatically process videos in order to extract semantic information.

Heterogeneous Camera Networks: Future video analytics systems will need to operate on *heterogeneous camera networks*. These networks consist of fixed camera surveillance systems which can be engineered such that camera feeds can be transmitted to a cloud-based processing system [20], but also more constrained camera devices (mobile cameras, dash cams and body cams) that may be wirelessly connected, have limited processing power, and, in some cases, may be battery powered.

The advantage of heterogeneous camera networks is that they can greatly increase accuracy and coverage. This is, by now, well established even in the public sphere. After the success in using mobile phone images and videos in the Boston marathon bombing a few years ago [30], mobile devices are now in widespread use in police and security work [39]. In particular, cities like Chicago [11], and New York [34] have already equipped police officers with body cameras, some of which are wireless-enabled [44], and police cruisers [45] also have dash cams for recording traffic stops.

The disadvantage of such networks is that they pose significant challenges for video analytics. Processing videos on the cloud is hard enough. Computer vision algorithms follow a resource-accuracy tradeoff, where more resources can enable higher accuracy, at higher cost (*cloud resources can be expensive*), so finding the right combination of algorithms to satisfy this tradeoff is an important systems challenge. This becomes harder with heterogeneous devices, like mobile cameras or body cams: because of bandwidth constraints, some processing has to be done locally on the device, but energy and CPU constraints limit this processing.

Video Analytics for Vehicle Tracking: As a first step to understanding how to design video analytics systems in heterogeneous camera networks, we take a specific problem: *to automatically detect a path taken by a vehicle through a heterogeneous network of non-overlapping cameras*. In our problem setting, each mobile or fixed camera either continuously or intermittently captures video, together with associated metadata (camera location, orientation, resolution, *etc.*). Conceptually, this creates a large corpus of videos over which users may pose several kinds of queries either in *near real-time*, or *after the fact*, the most general of which is: What path did a given car, seen at a given camera, take through the camera network?

Commercial surveillance systems do not support automated multi-camera vehicle tracking, nor a heterogeneous camera network. They either require a centralized collection of videos [3], or perform object detection on an individual camera, leaving it to a human operator to perform association of vehicles

Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-09-2-0053 (the ARL Network Science CTA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

across cameras by manual inspection [1].

Contributions: This paper describes the design of Kestrel[1] (§II), a video analytics system for vehicle tracking. Users of Kestrel provide an image of a vehicle (captured, for example, by the user inspecting video from a static camera), and the system returns the sequence of cameras (*i.e.*, the path through the camera network) at which this vehicle was seen. This system *carefully selects, and appropriately adapts, the right combination of vision algorithms to enable accurate end-to-end tracking, even when individual components can have less than perfect accuracy, while respecting mobile device constraints*. Kestrel addresses the challenge described above using one key architectural idea (Figure 1). Its *cloud pipeline* continuously processes videos streamed from fixed cameras to extract vehicles and their attributes, and when a query is posed, computes vehicle trajectories across these fixed cameras. *Only when there is ambiguity in these trajectories*, Kestrel queries one or more mobile devices (which runs a *mobile pipeline* optimized for video processing on mobile devices). Thus, mobile devices are invoked only when absolutely necessary, significantly reducing resource consumption on mobile devices.

Kestrel uses novel techniques for fast execution of deep (those with 20 or more layers) Convolutional Neural Networks (CNNs) on mobile device embedded GPUs (§II-A). These deep CNNs are more accurate, but mobile GPUs cannot execute these deep CNNs because of insufficient memory. By quantifying the memory consumption of each layer, we have designed a novel approach that offloads the bottleneck layers to the mobile device's CPU and pipelines frame processing without impacting the accuracy. Kestrel leverages these optimizations as an essential building block to run a deep CNN (YOLO [21]) on mobile GPUs for *detecting* objects and drawing bounding boxes around them on a frame.

Kestrel employs an accurate and efficient object *tracker* for objects within a single video (§II-B), enabling an order of magnitude more efficiency than per-frame object detection. This tracker runs on the mobile device, and we use it to extract object attributes, like the direction of motion as well as object features for matching. Existing general purpose trackers (§IV) are computationally expensive, or can only track single objects. Kestrel only periodically detects objects (thereby reducing the energy cost of object detection), say every k video frames, so that the tracking algorithm has to only be accurate in between frames at which detection is applied.

Kestrel uses novel sensor fusion to achieve accurate vehicle *association* (§II-C). Given an object in a camera's video, association can determine which object in a second camera best matches the given object. Association is performed both on the cloud and in mobile devices, and uses three ways to winnow candidate matching objects: the travel time between cameras, the direction of motion of the object exiting the first camera and entering the second, and the color distribution of the vehicle. Association is used by a *path inference* algorithm that uses a dynamic programming approach to find the most likely *paths* through the camera network that a vehicle may have traversed.

[1]Kestrel video demo: 'https://youtu.be/vSO7mYUpEhs'

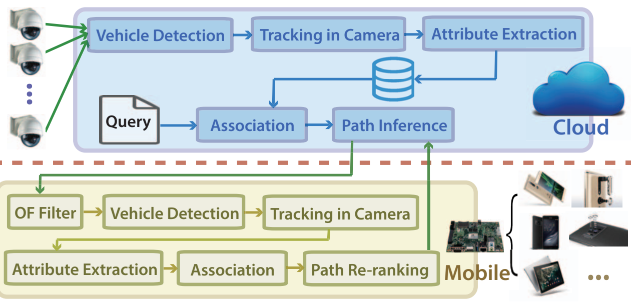

Fig. 1—Kestrel System Architecture

The evaluation (§III) of Kestrel uses a dataset of videos collected from a heterogeneous camera network. Specifically, our dataset consists of a total of 4 hours of video footage collected on a university campus comprising of 11 static and 6 mobile cameras. In this dataset, we have manually annotated a ground truth dataset of ~120 cars. Kestrel is able to achieve over 90% precision and recall for the path inference problem across multiple hops in the camera network, with minimal degradation as the number of hops increases, even though, with each hop, the number of potential candidates increases exponentially. Its overall performance on a heterogeneous network is comparable to an identical fixed camera network: the use of mobile devices only minimally impacts performance, while reducing energy usage by an order of magnitude. Finally, the association has nearly 97% precision and recall with each winnowing approach contributing to the association performance.

II. Kestrel Design

Figure 1 shows the system architecture of Kestrel, which consists of a *mobile device pipeline* and a *cloud pipeline*. Videos from fixed cameras are streamed and stored on a cloud. Mobile devices *store videos locally* and *only send metadata to the cloud* specifying when and where the video is taken. Given a vehicle to track through the camera network, the cloud detects cars in each frame (*object detection*), determines the direction of motion of each car (*tracking*) in each single camera view and extracts visual descriptors for each car (*attribute extraction*). Then, across nearby cameras, it tries to *match* cars (*cross-camera association*), and uses these to infer the path the car took (*path inference*) *only* on videos from the fixed cameras. Only when the cloud determines that it has low confidence in the result, it *uses metadata from the mobile devices* to query relevant mobile devices *to increase tracking accuracy*. Once the mobile device receives the query, it performs roughly the same steps, but only in a small segment of the captured video (and some of these steps are optimized to save energy). More specifically, the cloud sends to the mobile device its inferred candidate paths, and the mobile pipeline *re-ranks* these paths. The outcome is increased confidence in the estimated path. In the following sections, we describe Kestrel's components: some components run on the cloud alone, others run on both the cloud and the mobile device.

A. Object Detection: Deep CNNs

Kestrel uses GPUs on mobile devices to run Convolutional Neural Networks (CNNs) for vehicle detection. Many mobile devices incorporate GPUs (like nVidia's TK1 and TX1 or Qualcomm's Adreno), including mobile phones such as Lenovo Phab 2 Pro [25] and Asus ZenFone AR [2], as well as tablets like the Pixel C [16], the Google's Tango [15].

Unfortunately, the memory requirements of deep CNNs are outpacing the growth in mobile GPU memory. The trend today is towards deeper CNNs (*e.g.*, ResNet has 152 layers), with increasing memory footprint. Kestrel carefully optimizes CNN memory footprint to enable state-of-the-art object detection on mobile GPUs.

YOLO: Kestrel uses YOLO [21], a deep CNN for performing object detection. As shown in Figure 2, YOLO not only classifies the object but also draws a bounding box around it. YOLO is structured as a 27 layer CNN, with 24 convolutional layers, followed by 2 fully-connected (FC) layers and a detection layer. The convolutional layers extract the features that best suits the vision task, while the FC layers use these features to predict the output probabilities and the bounding box coordinates.

Running YOLO on mobile GPUs requires more memory than is available on these devices. For example, YOLO requires 2.8GB memory on TK1 (which has only 2GB). A normal GPU programming workflow involves loading the data to be operated upon (in our case the trained CNN weights) first in CPU RAM, then copying them to GPU memory and starting the kernel operation. If we had enough resources to hold the weights in memory, then multiple video frames can be dispatched for processing in sequence and the computation is very fast. However, when we ran YOLO on the TK1 using the workflow described above, it *failed to execute* due to insufficient memory resources. This motivated us to explore several methods to manage the memory constraint on mobile GPUs, based on the observation that almost 80% of the memory allocated to the network parameters is taken by the first fully-connected (FC) layer [23] of the neural network.

The CPU offload Optimization: Prior work has considered offloading computation to overcome processor limitations [8, 18]. We explore offloading only the FC layer computation to the CPU, because, since the CPU supports virtual memory it can more effectively manage memory over-subscription required for the FC layer. With CPU offloading, we observe that when the CPU is running the FC layer on the first video frame, the GPU cores are idle. So we adopt a *pipelining* strategy to start running the second video frame on the GPU cores rather than letting them idle. To achieve this kind of pipelining, we run the FC layers on the CPU on a separate thread, as GPU computation is managed by the main thread.

Other memory optimizations: In our evaluation (§III-E), we compare offloading with pipelining with other memory optimizations that have been proposed in the literature. We explore *reducing the size* of the FC layer ([12, 40] has explored similar techniques in different settings), which can potentially reduce detection accuracy. To reduce the size of the FC layer, we reduce the number of filters and the number of outputs in the network configuration and re-train. To reduce the memory footprint of the FC layer, we can also *split the computation* in some layers [29]. Specifically, in the FC layer, the input vector of previous layer is multiplied by the weight matrix to obtain the output vector. Splitting this matrix multiplication and reading the weight matrix in parts allows us to reuse memory. However, re-using memory and overwriting the used chunks incurs additional file access overheads.

B. Attribute Extraction

After detecting objects, Kestrel extracts two *attributes* of the object from a video, including its (i) direction of motion, and (ii) a low-complexity visual descriptor. These attributes are used to associate objects across cameras (§II-C). To estimate the direction of motion, it is important to *track* the object across successive video frames.

Light-weight Multi-Object Tracking: To be able to track objects[2] in a video captured at a single camera, Kestrel needs to take objects detected in successive frames and associate them across multiple frames of the video. Our tracking algorithm has two functions: (i) it can reduce the number of times YOLO is invoked, which in turn reduces the latency and conserves energy, and (ii) it can be used to estimate direction of motion. The state-of-the-art object tracking techniques [48, 13] take as input the pre-computed bounding box of the object, then extract sophisticated features like SIFT [9], SURF [17], *etc.* from the bounding box, and then iteratively infer the position of these key-points in subsequent frames using feature matching algorithms like RANSAC [28].

However, designing the most robust tracker for a single object is different from effectively tracking all the objects that appear within a given time window in a video. In dynamic mobile camera scenarios, objects (especially fast moving vehicles) can enter and exit the scene frequently. This means that Kestrel needs to run expensive object detection algorithms like YOLO frequently in order not to miss out on tracking new objects. To avoid this, Kestrel uses two optimizations. First, it performs optical flow [3] based scene change detection and whenever a scene change is detected, it runs YOLO to detect new objects. Second, to avoid running YOLO on every subsequent frame, we run YOLO every k frames (for small k) and stitch the detected objects together using a tracker that tracks light-weight features (Good Features To Track [22]). In §III, we compare our tracker against other state-of-the-art trackers to quantify the latency vs. accuracy tradeoffs between the two approaches.

Specifically, given a video stream, Kestrel's tracker detects *keypoints* of each frame and uses a keypoint tracker, Kanade-Lucas-Tomasi (KLT [5]), to track keypoints across frames. For every k frames (called YOLO frames), Kestrel runs YOLO to detect the objects of interest in the frame. Each detected

[2]While our paper is about tracking vehicles, many of the components we use are generic and can be used to detect and track objects in general (*e.g.*, people). Where a component is specific to vehicles, we will indicate as such.

[3]An optical flow in a video is the pattern of motion of points, edges, surfaces, objects in the scene.

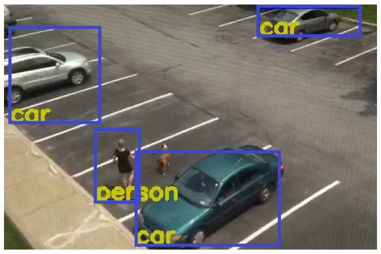

Fig. 2—Object Detection

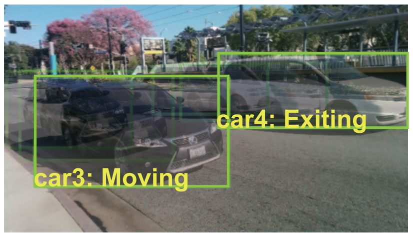

Fig. 3—Motion Analysis via Multi-Object Tracking

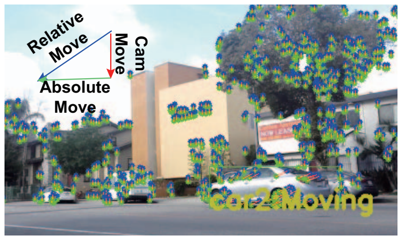

Fig. 4—Camera Movement Subtraction

object is stored in Kestrel with a series of attributes: object ID (OID), the bounding box coordinates, the coordinates and features description of the keypoints in the bounding box, *etc*. (Kestrel computes other attributes during this process, as discussed below). It tries to associate each detected object with its previous appearance in the last YOLO frame by tracking the object over k frames. In the new YOLO frame, Kestrel finds the match by comparing the keypoints of tracked objects with those in the bounding box of the newly detected one. The new objects that are successfully associated with a tracked object will inherit the existing object ID, otherwise a new unique ID is assigned. Also, for each matching existing box, Kestrel resets the tracking attributes, *i.e.*, the box coordinates are updated with the new box coordinates, the feature keypoints are updated with those from the YOLO frame inside the new box. Between two YOLO frames, Kestrel tracks the object by updating the box attributes frame by frame. Specifically, it calculates the average displacement of all the keypoints associated with one object, and updates the box coordinates accordingly. By integrating the optimized YOLO with KLT, we have built a fast and effective mobile video processing toolkit that achieves real-time robust object detection, localization and tracking on mobile devices. Figure 3 shows Kestrel tracking 2 vehicles.

Object State Extraction: An object seen in a single camera can go through different states: it can *enter* the scene, *leave* the scene, *move* or be *stationary*. Determining these states is important to filter out uninteresting objects like permanently parked cars, etc. Kestrel differentiates moving objects from stationary ones by detecting the difference of the optical flow in the bounding boxes and in the background. For moving objects, the state of the object changes from entering, moving, to exit, as it moves through the camera scene. A moving object can become stationary in the scene, a stationary object can also start moving at any time. For example, a car may come to a stop sign or a traffic light at an intersection, or park temporarily for loading passengers. If the object is detected as *exited* from the scene, then the object can be evicted. Kestrel uses *fast eviction*: after an object exits the scene, all the feature points and information for tracking are evicted from memory. This, not only helps save memory, but also improves accuracy because evicting the keypoints from previous objects can reduce the search space for matching candidates (§II-C).

Extracting the Direction of Motion: Kestrel estimates the direction of motion of a vehicle to on a mobile device to eliminate ambiguity. Suppose a vehicle is moving towards camera A as shown in Figure 5, but one of the cloud-supplied

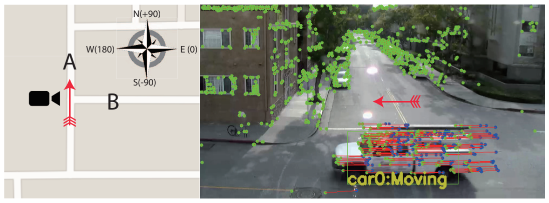

Fig. 5—Extracting the Direction of Motion and Coordinate Transformation

paths shows that it went through a different camera, say B located away from the vehicle's direction of motion. Since it could not possibly have gone through B, the mobile pipeline can lower that path's rank (§II-D). Extracting the direction poses two challenges: how to deal with the *movement of the mobile device*, and how to *transform the direction of motion* to a global coordinate frame.

Kestrel addresses the first challenge by *compensating for camera movement*. In general, to extract the direction of motion in the frame coordinates, we can use the difference between the bounding box positions from consecutive frames. In our experience, however, using the optical flow in the bounding box gives us more fine-grained direction estimation that is robust to the errors in YOLO's bounding boxes. To compensate for camera movement, we subtract the optical flow of the background from the optical flow of the object bounding box. The result is the direction of the object (Figure 4).

Kestrel needs to transform the direction of motion to a global coordinate frame in order to reason about, and re-rank, other vehicle paths provided by the cloud. For an arbitrary camera orientation, the exact moving direction in global coordinates can be calculated using a homography transformation [49]. However, in most practical scenarios, cameras have a small pitch and roll (*i.e.*, no tilt towards ground or sky and no rotation, videos are often recorded in either portrait or landscape) especially across a small number of frames. So, we only use the azimuth of the camera, and infer only the horizontal direction of moving objects (Figure 5). As per the Android convention (that we use to obtain our dataset), right (east) is $0°$ and counterclockwise is positive in camera (global) coordinates. The transformation is simply $\theta_g = \gamma + (\theta_c - 90°)$ where θ_c (θ_g) is the direction of motion in the camera (global) coordinates, and γ is the azimuth of the camera orientation.

To obtain the azimuth for mobile cameras we use the orientation sensor on off-the-shelf mobile devices. We have a custom Android application that records meta-data of the video (GPS, orientation sensor, accelerometer data, *etc*.) along with the video. However, to use this information directly is not

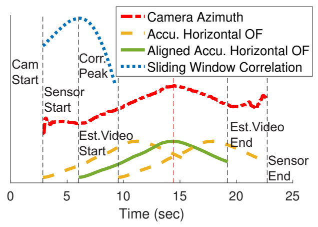

Fig. 6—Time Alignment of Motion Sensor and Optical Flow

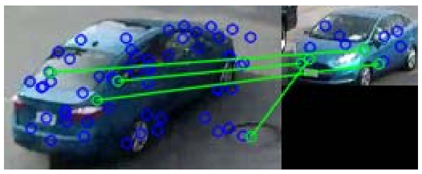

Fig. 7—MisMatch of SURF Features on The Same Car Captured at Two Cameras from Different Angle

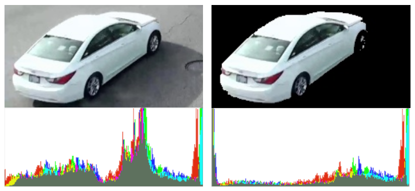

Fig. 8—Object Thumbnail and Corresponding Color Histogram Before (Left) and After (Right) Background Removal.

feasible as we cannot align the orientation sensor readings to a particular frame of the video. The timestamp of the video being taken on a mobile device may or may not be available. To find the correct time alignment between frames and mobile sensors, Kestrel takes the first few seconds optical flow pattern as a sliding window, and calculates its correlation with orientation sensors, as it slides through the beginning sequence of the sensor readings. By selecting the correlation peak, Kestrel can find the time shift and calibrate the timestamp of each video frame. Figure 6 demonstrates this for an experiment in which the camera is horizontally panned towards the left, and then towards the right.

Extracting Object Descriptors: Kestrel's pipeline requires a method to *re-identify* cars seen in cloud-provided vehicle paths with vehicles detected by its object detector. For this, Kestrel extracts *descriptors* of objects. A good descriptor has the property that is *low-complexity* (can be computed easily and requires minimal network bandwidth to transmit), yet can effectively distinguish different objects. The simplest descriptor, a thumbnail image, can distinguish objects but can consume significant bandwidth (§III).

Kestrel extracts *visual features* for use as descriptors. Visual features fall into two groups, local descriptive keypoints (SIFT, SURF, *etc*.), and global features (color histogram (CH), color layout descriptor (CLD), *etc*.). In multi-camera scenarios, local features do not work well since objects can be captured in different angles with varying degrees of occlusion, illumination, *etc*. Using SURF features and RANSAC [28] for keypoint matching (Figure 7) demonstrates how slightly different views can lead to mismatches: the four matching keypoints (connected by green lines) are all false positives. In §III, we quantify the performance tradeoffs of these approaches.

For these reasons, Kestrel extracts the color histogram. Before doing so, it preprocesses the bounding box to exclude background pixels. GrabCut [6] efficiently removes the background in a bounding box, leaving only the pixels of the object of interest. As we see in Figure 8 (left), with so many gray background pixels the color histogram can look very different from the color histogram of the car seen in Figure 8 (right) after removing the background pixels. Background removal helps other visual features as well, since keypoints from the background can confound matching during association.

The choice of color histogram does have drawbacks: vehicles with similar colors can have high correlation scores. Figure 9 illustrates, in our dataset, both an easy case, where Kestrel searches for a red sedan among other vehicles of different colors, and one of the most challenging cases, where three white SUVs have very similar color layout that would require careful human inspection. *But Kestrel does not rely on color alone*: it uses other filters (direction, travel time estimation) to further narrow the search space and achieve high object association accuracy. Indeed, in our experiments, Kestrel was able to correctly distinguish between the three SUVs.

C. Pair-Wise Instance Association

An *instance* corresponds to a specific object captured at a specific camera, together with the associated attributes (§II-B). *Instance association* determines if object instances at two cameras represent the same object or not. A camera network can be modeled as an arbitrary topology shown in Figure 10. Assume that each intersection has a camera ($Cam\ c$), and each camera extracts several instances. We define the nth object instance of Cam c as $o_{c,n}$. Kestrel infers the association between any pair of object instances $(o_{x,i}, o_{y,j})$, using three key techniques: visual features, travel time estimation and the direction of motion.

Preprocessing: Before associating instances, Kestrel preprocesses each instance to narrow the search space. Since YOLO can detect several types of objects, the preprocessing stage filters all object types other than vehicles. Also, it filters all stationary objects, such as parked cars on the side of the streets. Even so, there could still be hundreds of instances to search from multiple neighboring cameras. Kestrel estimates the travel time $ET(x, y)$ between a pair of camera x, y, and sets a much smaller but coarse time window (which is refined in the next step) to limit the number of candidate instances at this preprocessing stage.

Spatio-temporal Association: To associate between two object instances $(o_{x,i}, o_{y,i})$, Kestrel uses the timestamp of the object's first scene entry TI and its last appearance exiting the scene TO. Kestrel calculates the total travel time from $Cam\ x$ to $Cam\ y$ as $\Delta T(o_{x,i}, o_{y,j}) = TI_{y,j} - TO_{x,i}$, and compares this to the estimated time $ET(x, y)$ needed to travel from $Cam\ x$ to $Cam\ y$. Taking the exit timestamp from the first camera and the entry timestamp in the second camera filters out any variance due to a temporary stop (*e.g.*, at a stop sign) while in the camera view.

One way to estimate travel time between two cameras is to use the Google Directions API [14]. For dense camera deployments typical of campuses, travel times between cameras are small (~30 secs) and can be inflated by noise. To avoid this,

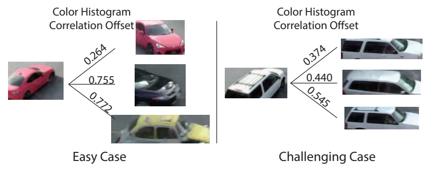
Color Histogram Correlation Offset
0.264
0.755
0.772
Easy Case
Color Histogram Correlation Offset
0.374
0.440
0.545
Challenging Case

Fig. 9—Object Descriptor Correlation

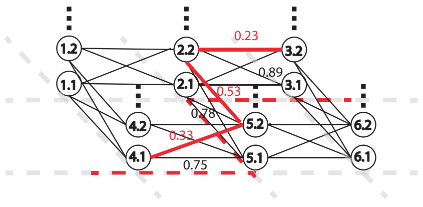

Fig. 10—Cross-Camera Association Using Correlation As Link Weight In A Network of Camera Captured Object Instances

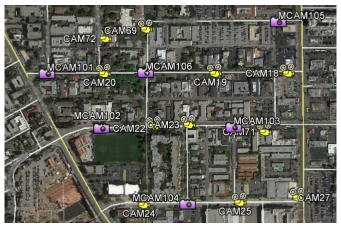

Fig. 11—Evaluation Setup: A Heterogeneous Camera Network

we estimate the travel time between each pair of cameras using a small annotated ground truth dataset. Further, since cars have different travel speeds and may stop in between, there is some variance in the travel time between cameras. We also estimate this variance from our training dataset. To filter out unlikely associations (*e.g.*, a car cannot appear at two cameras that are one block away from each other within 1 sec), we apply temporal constraints derived from the dataset: specifically, if the estimated travel time is x, over 95% of the vehicles take a travel time of $x \pm 0.6x$.

Visual Similarity: The visual similarity of object instances is also used in object association. Given two color histograms H_1, H_2 from two instances, Kestrel measures their similarity using a *correlation offset*, which is 1 minus the correlation between the two histograms. The lower the offset is, the more likely the two instances are the same. We find that this metric can discriminate well between similar and dissimilar objects. From a sample of 150 images (figure omitted for space reasons), we find that over 95% of identical objects have an offset of less than 0.4, while over 90% of different objects have an offset larger than 0.4. In a more general setting, this threshold can be learned from data.

Direction Filter: We also use the direction of motion (§II-B) to prune the search space for association. For example, in Figure 10, assume that the moving direction of instance $o_{1,1}$ from Cam 1 is from east to west, then Kestrel ignores Cam 4, and searches over all object instances of Cam 2 within the temporal window around the expected travel time between Cam 1 and Cam 2, and considers only those instances exiting Cam 2 towards Cam 1.

Instance Association in the Mobile Pipeline: The mobile pipeline also contains some elements of instance association necessary for re-ranking paths. Specifically, the mobile pipeline must eliminate stationary objects, but does not need spatio-temporal filtering and temporal filtering in the preprocessing stage (because Kestrel selects the mobile device at the relevant location, and supplies the device a time window over which to search for associations). The mobile pipeline computes visual similarity and applies the direction filter.

D. Path Inference

Using its object detection, attributes and association components, Kestrel can infer, *in the cloud*, the path of a target object through a multi-camera network. Our pair-wise association simplifies the object instance network and retains only the valid paths *i.e.*, filters out objects moving in incorrect directions, those outside the temporal window and those whose visual correlation is too low. In this *pruned network*, the *weight* $w(o_x, o_y)$ of a *link* $l(o_x, o_y)$ is defined as the color histogram correlation offset (lower the offset, better is the correlation). A path from a source instance o_s to destination o_d in an instance network is defined as an *instance path* $p(o_s, o_d) = o_s, o_1, o_2, ..., o_d$. A physical route in the real world, termed a *camera path*, is defined by the sequence of cameras traversed by a vehicle: multiple instance paths can traverse the same camera path.

Consider one object of interest $o_{x,i}$, captured by either a mobile user or a camera operator from $Cam\ x$. Kestrel seeks to infer the instance path it takes and answers this query in near-real time, generating the path and instances at each camera as a feedback to the user.

One straightforward approach is to assign the correlation offset as a weight to the links in the pruned instance network, and use maximum likelihood estimation (MLE) to find the minimum weighted path from any given instance. However, this approach fails to capture the correlation between non-neighboring instances, as it only relies on the link weight.

Instead, Kestrel takes a hop-by-hop iterative approximation approach to find the best matching last hop instance each time. At every hop with multiple candidate paths joining at one instance, Kestrel evaluates the weight of each path from the candidate set and uses the Viterbi decoding algorithm [4] to eliminate paths with heavy weight to keep the algorithm at polynomial complexity. Given an instance in the instance network (Figure 10), we define a valid neighbor as an instance that has a direct link and passes all the filters defined in §II-C. Every time a new instance is added to a path, the path weight is multiplied by an amount equal to the average correlation offset of this instance from every other instance in the path. The worst case complexity is $O(kn^2)$, assuming every camera is right next to each other, and every instance has a valid link to any instances. In practice, each camera has an average of 2.2 neighbors, and vehicles usually traverse each node in the network only once except for looping scenarios.

Re-ranking Paths in the Mobile Pipeline: The output of path inference can be ambiguous, where multiple instance paths have high path weights. In this case, the cloud pipeline attempts to find (from previously uploaded metadata) a set of mobile devices which might have videos that could help resolve this ambiguity. The cloud pipeline sends to the mobile device its set of ranked instance paths. Given this set, the mobile camera does *pair-wise association* with the instances

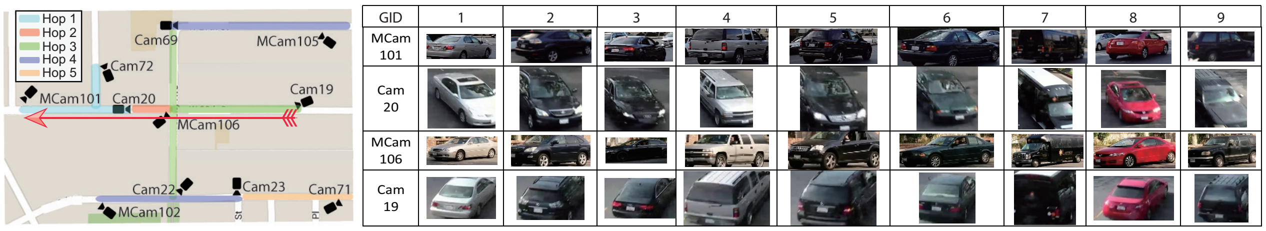

Fig. 12—Example Ground Truth Instance Paths Across The Heterogeneous Camera Network

from both the last hop and the next hop static cameras of each instance paths. If the locally extracted instance matches the path's direction, a new instance path is generated whose instance path weight is augmented with a *pair-wise association* score. Note that a single instance path could generate multiple new instance paths with different intermediate mobile instances and path weights. The mobile pipeline can re-rank the paths and transmit these to the cloud. For each new mobile instance, it only needs to transmit the color histogram and the instance's direction of motion in case the cloud needs to query another mobile device.

III. Evaluation

In this section, we evaluate the end-to-end performance of Kestrel, and the performance and overhead of each submodule.

A. Methodology

Dataset: We collected a video dataset from 17 distributed *non-overlapping* cameras (6 mobile cameras and 11 surveillance cameras) on our campus, deployed in a whole block of a residential area (Figure 11) adjacent to the campus. The surveillance cameras (in yellow in the figure) are commercial Axis Q-6035 and Q-6045 cameras, with a resolution of 1280×720 at 10 frames per second, and mounted on light poles. The mobile cameras (in pink in the figure) are from off-the-shelf smartphones held by human users at street level, and these capture 1920×1080 video at 30fps using a customized app which also captures inertial sensor readings from the phone. Users collect video by sweeping the visible area for interesting events: thus, unlike fixed cameras, the orientation (yaw and pitch) of the cameras are continuously changing.

Ground Truth: To evaluate the association, we *manually annotate* a ground truth list of instances that match across cameras (Figure 12). Specifically, each instance is labeled with a camera ID and an object ID, *i.e.,* the first car that appears in camera 20 would be labeled as $20(1)$. We visually track each vehicle from camera to camera, and label their corresponding instances in each camera with the same global ID. For example, if a vehicle travels through cameras 101, 20, 106, 19, appearing as the 8th, 12th, 15th, 4th object of each camera respectively, will form an *instance path* {101(8), 20(12), 106(15), 19(4)}, and can be assigned a global ID 2. We also evaluate Kestrel's accuracy in detecting the *camera path*, the sequence of camera traversed by the car, and represented by the corresponding set of camera IDs.

From the dataset of 17 cameras, we have annotated, over a total of 235 minutes of footage, 120 global objects / vehicles. In most cases, a vehicle going through the camera network can be seen at three to four cameras. To form as many valid ground truth paths as possible, we issue a query with each instance of the path, except the last instance before it disappears from the camera network. Kestrel runs those queries to infer backwards as many hops as possible. For example, if one car goes through 3 cameras, a, b, c, Kestrel can infer from the instance at c to get b and a, or infer from b to get a. That gives both a two-hop path and a one-hop path for the evaluation. Collectively, we are able to establish 311 one-hop paths, 197 two-hop paths, 125 three-hop paths, 55 four-hop paths, and 23 five-hop paths. There are a small number of longer paths, which we ignore because their number is too small to draw valid conclusions.

Metrics: The primary performance metric for Kestrel is accuracy. We use *recall* and *precision* to evaluate the accuracy of Kestrel's association and path inference. Among all the given paths, recall ($\frac{TP}{TP+FN}$) measures the fraction of the paths for which Kestrel can make a successful inference, where TP is the number of true positives and FN the number of false negatives. Given one instance of an object, precision ($\frac{TP}{TP+FP}$) measures how often Kestrel correctly identifies all the instances on the path within its top k choices, where FP measures the number of false positives. Ideally, Kestrel should exhibit high recall and precision for as small a k as possible.

We explore two aspects of Kestrel's accuracy. Its *camera path* accuracy measures how accurately Kestrel can identify the sequence of cameras traversed by a vehicle. We measure this by majority voting across the top five instance paths. We also measure the accuracy of determining the *instance path*: such a path identifies the correct instances at each camera matching the queried vehicle. For this, we present the top-k accuracy: how often its top k choices contain the ground truth path. Specifically, if the annotated ground truth is among those top k choices, Kestrel is said to generate a true positive (TP), otherwise a false positive (FP). Meanwhile, all the other unselected choices are considered as negatives (N). The negatives are true (TN) when they belong to different objects, otherwise false negatives (FN).

We evaluate several other metrics as well. We measure mobile pipeline energy consumption in Joules per frame, by using a current clamp attached to the positive wire of the TK1's power supply. By inducing a magnetic field on the conductor and making use of a Hall effect sensor, we can compute the current passing through. We use a DataQ DI-149 Data Acquisition Kit [7] that samples at 80 Hz and allows us to save the readings in a file for post-processing. We measure

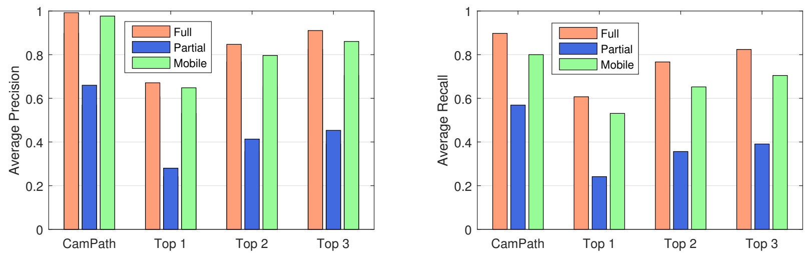

Fig. 13—Augmenting Sparse Camera Network with Mobile Cameras to Achieve Full Performance

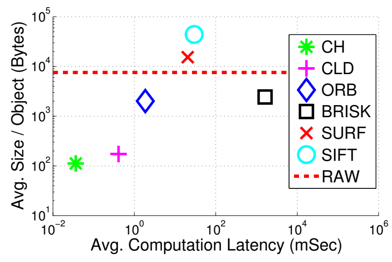

Fig. 14—Feature Size vs Computation Latency

detection latency of the object detector by CPU elapsed time, and detection accuracy by mean average precision (mAP [10]), which captures false positives as well as the false negatives.

B. *Camera Path and Instance Path Accuracy*

In this section, we evaluate Kestrel's accuracy. Kestrel supports a heterogeneous camera network, where videos captured on mobile devices can augment an existing fixed camera deployment. We would like to understand (a) how much the mobile cameras improve accuracy over the existing fixed camera deployment, and (b) how well a heterogeneous camera network performs relative to a fixed camera deployment *of the same size*. To this end, we evaluate accuracy over our 17-camera dataset by forming three topologies: a *Mobile* topology where the mobile cameras run the Kestrel mobile pipeline, a *Full* topology where all cameras are treated as fixed and run the cloud pipeline, and a *Partial* topology, which consists only of the 11 fixed cameras.

Figure 13 shows the average camera path inference precision and recall on these three different networks. To start with, with a Full topology where every intersection is monitored, Kestrel achieves a camera path precision of 99.2%, while Partial has a precision of 65%. Interestingly, the presence of a mobile camera enables Kestrel to effectively disambiguate path choices, *achieving a precision of 97.7%*. Recall results are qualitatively similar, but average recall numbers are slightly lower (explained in the next paragraph), with about 80% recall for Mobile vs. 90% for Full. Recall for partial is significantly lower. These results suggest that a heterogeneous camera network can approach the accuracy of an equivalently sized fixed network, and the addition of mobile devices to an existing fixed camera network can significantly improve performance.

For instance paths, Kestrel achieves good top-3 precision and recall for the Mobile topology, whose performance is close to the Full topology, and significantly better than Partial. In general, the recall performance for Mobile is lower than the Full topology (both for camera path and instance path accuracy) because of Kestrel's energy optimization to avoid invoking the object detector sometimes leads to missed detections. We explore the trade-off between energy and recall below.

Figure 15 shows Kestrel accuracy as a function of path length. Its top 3 choices ($k = 3$) almost always (~90% precision and recall) find the correct paths for up to three camera hops. Increasing k to 4 or 5 provides marginal improvements, while reducing it to 1 gives a precision and recall of slightly less than 80% for up to 3 hops. Mobile has comparable performance to Full with less than 10% degradation. In subsequent sections, we discuss how much of this performance can be attributed to object detection, attribute extraction, and association.

This accuracy is significant, considering that, statistically, for each hop, Kestrel has to find the correct instance among an average of 20.38 candidates from 2.2 cameras. More important, the distribution of the number of candidate paths increases exponentially as the number of hops increases; at 5 hops, the number of candidate instance paths in our dataset ranges from 10,000 to over a million. In this regime, a human operator can not manually identify the correct path just by visually inspecting pictures, but Kestrel can achieve nearly 70% recall and 80% precision at 5 hops, despite only using the color histogram descriptor, as a result of our techniques.

At 5 hops, the precision and recall might appear low, but Kestrel could be practicable even in this range, with a little operator input. Suppose that a user issues a path inference query and does not see the right 5-hop answer. If there is a correct 3-hop instance, she can re-issue a path inference query using as the starting hop instance the last correct instance, and stitch together the returned results.

C. *Energy and Latency*

Kestrel significantly reduces the energy consumption on the mobile device by only invoking the mobile device when presented with a query (unlike Kestrel, which processes every frame), and only when Kestrel has several high-ranked path candidates which a mobile device can help disambiguate. In our dataset, 77.0% of the path queries invoke only one mobile camera, 7.6% invoke two or more mobile cameras. If each mobile device were to continuously process the videos, instead of processing them on demand, they would consume 4806 J/min. A modern smartphones with a ~7 Wh/25200 J battery can only sustain such queries for 30 minutes. At 1 query/min, Kestrel only requires 82.2 J/m, an over 60× reduction.

When invoked, one particular optimization in the mobile pipeline is that Kestrel also reduces energy usage by careful design. The bottleneck of the mobile pipeline is running the neural net for object detection: on TK1, YOLO alone consumes 2.25 J/Frame whereas attribute extraction only drains 0.42 J/Frame. In terms of latency, YOLO takes an average of 0.259 Sec/Frame, while tracking only takes 0.081 seconds. Therefore, the fact that the optical flow motion filter can avoid running YOLO on every frame significantly conserves energy (81.3% less) and reduces the processing latency on mobile devices.

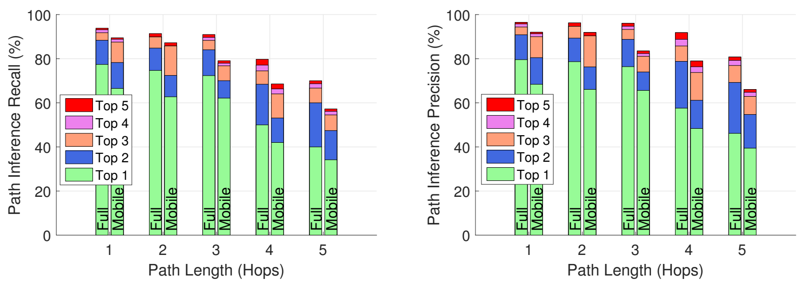

Fig. 15—Recall, Precision of Path Inference

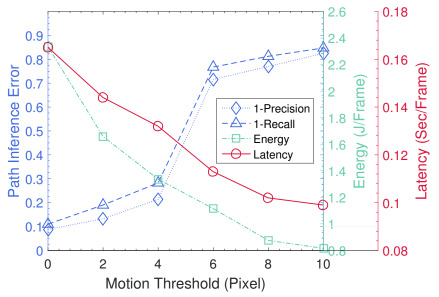

Fig. 16—Energy / Latency and Performance Trade-off

Energy and latency can also be traded-off for higher accuracy by adjusting the threshold for the optical flow filter. Figure 16 quantifies this tradeoff between energy / latency and the inference error. Intuitively, the higher the motion filter threshold, the fewer the YOLO invocations, which result in less energy and lower latency. Compared to YOLO running on every frame (motion threshold being 0), a small motion filter of 4 pixels can almost cut the energy usage in half, which can double mobile battery life, without significantly sacrificing path inference performance. On the contrary, if the motion filter is too insensitive (larger than 6 pixels), Kestrel will miss a lot of vehicles, which results in a much lower recall.

D. Association Performance

Next, we evaluate the performance of one-hop association. Given one instance at any camera, this component infers the associated instance among all the instances from all the neighboring cameras. When conducting this evaluation, we also evaluate the efficacy of each of the components of the association algorithm (§II-C). We start with ranking the visual association score among all candidates without any additional processing at all (NONE). Then we add each component one by one in this order: preprocessing (PRE), direction filter (DIR), spatio-temporal association (ET), background subtraction (GrabCut). This process is cumulative: for example, DIR also includes PRE.

Figure 17 shows the precision and recall of each successive combination. Generally, given an instance at any location at any time, Kestrel can almost always ($> 97\%$) find the same object in a neighbor camera, if there were one to be found, within the top 3 returned instances. Comparing the performance of each component combination, PRE effectively narrows down the search space and brings precision and recall to nearly 90%. DIR further rules out false positives, as cars moving in wrong direction can confound visual association. Both DIR and ET increase recall and precision by moving true positives higher in the rank (which increases Top 1 precision). Grabcut also increases precision and recall noticeably.

Figure 18 shows an example of how various steps of the association algorithm can reduce the number of candidates for the association. In this specific query, Kestrel is able to filter down to 3 candidates from an initial set of 342, and finds the correct target vehicle by ranking the histogram correlation offset from among these three, even though the two cameras capture the vehicle from completely different perspectives. When averaged over all associations, each target has an average of 385 candidates to match, PRE prunes the search space to about 20, DIR removes about 8 cars on average going in the wrong direction, ET is able to further narrow down to 6 candidates. Finally, Kestrel ranks the correlation offset of the remaining candidates.

Choice of Descriptor: To validate our choice of color histogram, Figure 14 compares different features in terms of the average data size and computation latency per object. Local features like SIFT and SURF incur both high computation overhead and large size (and we have earlier shown that they are sensitive to perspective differences, and so not a good choice for Kestrel for that reason), while the lightweight color histogram incurs minimal latency and small size with reasonable performance.

E. Object Detection Performance

Performance of CPU offload: In Table I we summarize the timing results of running the various strategies for GPU memory optimization. The *Original network* is already optimized (compared to the original network that runs on server class GPUs) in that it does not allocate memory for redundant variables not used in the testing phase. Running the computation on the CPU takes close to *half a minute* per frame, which is extremely slow, so it is imperative to leverage the GPU cores on the board for accelerated computation. Moreover, we see that for the original network, memory constraints do not allow running it on GPU at all; our subsequent optimizations to reduce the memory footprint make this possible.

CPU offload brings the computation time to under 1s. Offloading the FC layer allows us to read the weight file only *once* for all the frames and store the weights in CPU virtual memory. Finally, CPU offload along with pipelining gives the best results; it brings down the computation time to about 0.42s or almost *60 times faster* than running it on the CPU. We see this speed up because although the FC layer is memory intensive, the running time even on the CPU is not a bottleneck—in other words, the CPU processing completes within the time that the GPU is processing the convolutional layers of the next frame. This is interesting because it is achieved without compromising accuracy.

Other optimizations: Other optimizations are less effective. *Split* allows running the original network on GPUs but it is still slow as it incurs a weight-file read overhead every frame. Reducing the size of the network, by reducing the size of the

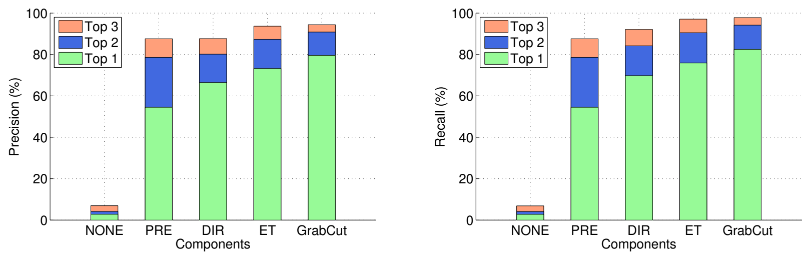

Fig. 17—Precision and Recall Performance of Pairwise Association using Different Kestrel Component Combinations

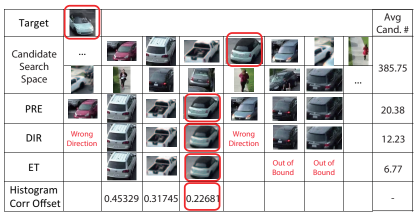

Fig. 18—Shrinking the Candidate Size at Each Step Matching a Target

	Memory Requirement	FC layer: resulting size of matrix multiplication	CPU	GPU	Split	CPU Offload	Pipeline
Original	2.8 GB	$1 \times 50176 * 50176 \times 4096$	25.026524	N/A	10.249577	0.703449	0.416910
Medium	1.6 GB	$1 \times 25088 * 25088 \times 2048$	24.315950	0.272191	0.573386	0.400366	0.261985
Small	1.3 GB	$1 \times 12544 * 12544 \times 1024$	24.003447	0.259000	0.394960	0.299940	0.261144

TABLE I—Average Time Taken to Run Detection per Image (seconds).

weight matrix in the FC layer to one fourth of the original size (*Medium network*) and 1/16 the original size (*Small network*), enables YOLO to run faster and follow similar trend as *Original network*, with one exception: for *Small network* the GPU is the fastest alternative as the FC layer is small enough that overhead of pipelining negates its benefit. However, these networks incur accuracy loss. On the Pascal VOC 2012 test dataset [10], we lose about 2-3% in mAP at each step of the reduction from Original to Medium to Small.

Power measurement: Interestingly, we found that CPU offloading based schemes are also the most energy-efficient. This is because these schemes optimize speed and so the circuitry is used for much shorter duration. They reduce the energy requirement by 20× the Original network (107J/frame for running on CPU, 3.78J with CPU offload, 3.95J with CPU Offload + Pipelining). Smaller CNNs do save energy (2.25 J/frame on Medium network for CPU Offload + Pipelining) but at the expense of accuracy.

F. Attribute Extraction

Tracking Accuracy vs Processing Latency: Most existing state-of-the-art multi-object tracking techniques assume offline processing for stored video and can incur significant overhead. To demonstrate this, we pick a representative of this class, MDP [48], which is regarded as the algorithm with the best accuracy with reasonable processing overhead. Also, we extend a robust single-object tracking tool, OpenCMT [13] to perform multi-object tracking for the comparison with Kestrel.

Method	Rcll	Prcn	FN	MOTP	TM	TD
Kestrel	76.4	88.2	152	76.4	0.081	0.057
MDP	70.1	87.1	192	75.4	N/A	0.146
CMT	75.9	88.6	155	76.3	1.599	0.907

TABLE II—Tracking Accuracy and Latency on TK1 and Desktop

We use a standard multi-object tracking benchmark [24] to compare our tracker against these approaches. To level the playing field, we use YOLO for detection for all three approaches. In addition to precision, recall, and false negatives, we use a metric called MOTP [24] that measures the tightness of the bounding boxes generated by the tracking algorithm.

We run the trackers on our vehicle dataset to compare the tracking performance as well as the average processing latency per frame, both on the TK1 (TM) and a desktop server (TD). Table II shows that Kestrel can achieve a slightly higher recall, comparable precision, lower false negatives and a comparable MOTP score compared to more sophisticated algorithms, while incurring an order of magnitude lower processing latency on the TK1 (one of the algorithms, MDP, cannot even be executed on TK1). The primary reason for the improved tracker performance is our design choice to periodically invoke YOLO, which can refine the box generated by the tracker. Between two YOLO frames, tracking and association is easier than the kinds of continuous tracking performed by modern trackers.

We also measure the energy consumption on TK1 using the same setup, and we find that our tracking requires 0.42 J/frame, but OpenCMT requires energy as high as 8.4 J/frame.

Period	Rcll	Prcn	TTRK	TYOLO	TTOT
1	76.4	88.2	0.081	0.259	0.340
3	74.7	82.5	0.081	0.086	0.167
5	64.4	74.2	0.081	0.051	0.132

TABLE III—Tracking Performance and Latency Tradeoff on TK1

Next, we explore the tradeoff between tracking accuracy and total latency by calling YOLO every 1, 3, 5 frames, and tracking only on non-YOLO frames. This evaluation uses a series of consecutive frames that have moving objects and pass the motion filter. In other words, we try to examine the best performance when every frame has objects to detect and track. Intuitively, the more frames between YOLO detection, the larger chance that the tracker may be led astray. As shown in Table III, less frequent YOLO detection effectively reduces the total average processing latency (TTOT) per frame, while maintaining a reasonable precision and recall (in some cases, even better than MDP and CMT, not shown) for attribute extraction. Specifically, when YOLO is invoked every 3 frames, Kestrel's mobile pipeline can sustain about 6 fps without noticeable loss in tracking performance. With further optimization, and on more recent GPUs such as TX1 [33], query latency could be further reduced to achieve 10 fps. Finally, our camera movement compensation algorithm has nearly identical tracking precision and recall at walking speeds, when compared with a stationary camera.

Sensitivity to Camera Motion: To examine the tracking robustness of Kestrel for mobile cameras, we collected a set of videos with a mobile camera while walking, biking, and driving on our campus. In these videos, we manually annotated the bounding boxes of vehicles appearing in every frame over nearly 6000 frames. Then, we ran our camera motion compensation algorithm, and our tracker, to evaluate its accuracy. Our results show a similar performance to Table II, with an average precision of 74.9%, an average recall of 70.9%, and an MOTP of 78.7%. This indicates Kestrel's algorithms can accurately track vehicles even when there is significant camera motion, with an accuracy comparable to a fixed camera.

Bandwidth: To quantify Kestrel's bandwidth-efficiency, in our dataset a typical 10-minute video file recorded at 1920×1080 (30 fps) is around 1.4GB. Streaming every frame in real time requires ~20Mbps. Instead, Kestrel only sends attributes. In our dataset, after the cloud pipeline, the average size of a query sent to the mobile is only 1.52KB. The re-ranked result and its corresponding instance metadata average 0.92KB.

IV. Related Work

DNNs on mobile devices: Recent work has explored neural nets on mobile devices for audio sensing activity detection, emotion recognition and speaker identification [31]. Their networks use only a small number of layers and are much simpler than the networks required for image recognition tasks. DeepX [32] is able to achieve reduced memory footprint of the deep models via using compression, at the cost of a small loss in accuracy. LEO [36] schedules multiple sensing applications on mobile platforms efficiently. MCDNN [41] explores cloud offloading and on-device versus cloud execution tradeoff, but on models smaller than the ones required for our work. Finally, DeepMon [26] proposes offloading the FC layers to the GPU for low power, lower frame rate (1-2 fps) applications, while using a tensor decomposition technique to reduce the memory footprint, at the expense of accuracy.

Object Detection: Early object detectors use deformable part models [35] or cascade classifiers [37], but perform relatively poorly compared to recent CNN based classification-only schemes which achieve high accuracy at real-time speeds. However, top detection systems like Fast R-CNN [38] exhibit less than real-time performance even on server-class machines. YOLO is a one-shot CNN-based detection algorithm that predicts bounding boxes and classifies objects, and we have used a mobile GPU on a deep CNN like YOLO.

Object Tracking: Prior tracking approaches like blob tracking [19] work well in static camera networks, but not for mobile cameras. Many other trackers are targeted to single object tracking, *e.g.* OpenCMT [13], and some at multiple objects [48]. However, most of these trackers are not targeted at execution on resource-constrained devices. Glimpse [46] achieves tracking on mobile devices using a combination of offloading and keeping an active cache of frames to work on, once a stale result is received from the server. By contrast, our tracking does not rely on offloading. We have compared the performance of our tracker with OpenCMT [13] and MDP [48] and the sensitivity to camera motion in §III.

Video Surveillance Systems and Object Association across Cameras: Vigil [43] is a wireless surveillance system that performs simpler vision tasks on powerful edge devices while offloading more complex computations to the cloud. However, it does not specifically address the re-identification problem we consider. Prior work [42] tries to associate people *etc.* across different cameras using a query retrieval framework by ranking nodes in their camera network. Other work [27] proposes a centralized multi-hypothesis model to track a vehicle through a multi-camera network. While Kestrel can support such applications, our focus is to enable mobile camera based surveillance, so our architectural and design choices are different from this line of work. [47] optimizes the efficiency of data storage and object detection queries for multiple videos, but does not consider multi-camera tracking.

V. Conclusion and Future Work

This paper explores whether it is possible to perform complex visual detection and tracking by leveraging recent improvements in mobile device capabilities. Our system, Kestrel, tracks vehicles across a heterogeneous multi-camera network by carefully designing a combination of vision and sensor processing algorithms to detect, track, and associate vehicles across multiple cameras. Kestrel achieves $> 90\%$ precision and recall on vehicle path inference, while significantly reducing energy consumption on the mobile device.

Future work includes experimenting with Kestrel at scale with different traffic densities and camera placement densities than the ones we have explored in this paper. We anticipate that our results will be insensitive to camera placement density (because our approach only detects paths through the camera network), but may perform differently at different traffic densities because of inaccuracies in the underlying computer vision tools. Other future work includes extending Kestrel to support more queries, and different types of objects, so it can be used as a general visual analytics platform.

Bibliography

[1] *Agent VI.* http://www.agentvi.com/.

[2] *Asus ZenFone AR.* https://www.asus.com/us/Phone/ZenFone-AR-ZS571KL.

[3] *Avigilon Video Analytics.* http://avigilon.com/products/video-analytics/solutions/.

[4] A.Viterbi. "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm". In: *IEEE Trans. Inf. Theor.* 13.2 (Sept. 2006).

[5] B.Lucas and T.Kanade. "An Iterative Image Registration Technique with an Application to Stereo Vision". In: IJCAI'81.

[6] C.Rother, V.Kolmogorov, and A.Blake. ""GrabCut": Interactive Foreground Extraction Using Iterated Graph Cuts". In: SIGGRAPH '04.

[7] *Data Acquisition Kit.* http://www.dataq.com/products/di-149/.

[8] D.Chu et al. "Balancing Energy, Latency and Accuracy for Mobile Sensor Data Classification". In: *Proc. of Sensys '11.*

[9] D.Lowe. “Distinctive Image Features from Scale-Invariant Keypoints”. In: *Int. J. Comput. Vision* 60.2 (Nov. 2004).
[10] E.Mark et al. “The Pascal Visual Object Classes (VOC) Challenge”. In: *Int. J. Comput. Vision* 88.2 (June 2010).
[11] *Every Chicago patrol officer to wear a body camera by 2018*.
[12] G.Chen, C.Parada, and G.Heigold. “Small Footprint keyword spotting using deep neural networks”. In: *Proc. of IEEE ICASSP ’14*.
[13] G.Nebehay and R.Pflugfelder. “Clustering of Static-Adaptive Correspondences for Deformable Object Tracking”. In: *CVPR ’15*. IEEE.
[14] *Google Maps Directions API*.
[15] *Google Tango*. https://www.google.com/atap/project-tango/.
[16] *Google’s Pixel C*. https://store.google.com/product/pixel_c.
[17] H.Bay, T.Tuytelaars, and L.Van Gool. “SURF: Speeded Up Robust Features”. In: ECCV’06.
[18] H.Eom et al. “Machine Learning-Based Runtime Scheduler for Mobile Offloading Framework”. In: *Proc. of UCC ’13*.
[19] H.Marko and P.Matti. “A texture-based method for modeling the background and detecting moving objects”. In: *IEEE Trans. on PAMI* 28.4 (2006), pp. 657–662.
[20] H.Zhang et al. “Live Video Analytics at Scale with Approximation and Delay-Tolerance”. In: *NSDI ’17*.
[21] J.Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: *CVPR ’16*.
[22] J.Shi and C.Tomasi. “Good features to track”. In: *CVPR*. IEEE, 1994, pp. 593–600.
[23] K.Alex. “One weird trick for parallelizing convolutional neural networks”. In: *CoRR* abs/1404.5997 (2014).
[24] Leal-Taixé et al. “MOTChallenge 2015: Towards a Benchmark for Multi-Target Tracking”. In: *arXiv:1504.01942 [cs]* (Apr. 2015).
[25] *Lenovo Phab 2 Pro*. http://www3.lenovo.com/us/en/smart-devices/-lenovo-smartphones/phab-series/Lenovo-Phab-2-Pro.
[26] L.N.Huynh, Y.Lee, and R.K.Balan. “DeepMon: Mobile GPU-based Deep Learning Framework for Continuous Vision Applications”. In: MobiSys ’17.
[27] B. C. Matei, H. S. Sawhney, and S. Samarasekera. “Vehicle tracking across nonoverlapping cameras using joint kinematic and appearance features”. In: *Proc. of IEEE CVPR ’11*.
[28] M.Fischler and R.Bolles. “Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography”. In: *Commun. ACM* 24.6 (June 1981).
[29] M.Moazzami et al. “ORBIT: A Smartphone-based Platform for Data-intensive Embedded Sensing Applications”. In: *Proc. of ACM IPSN ’15*.
[30] *Multi, Social Media Play Huge Role in Solving Boston Bombing*. https://www.voanews.com/a/multi-social-media-play-huge-role-in-solving-boston-bombing/1649774.html.
[31] N.Lane, P.Georgiev, and L.Qendro. “DeepEar: robust smartphone audio sensing in unconstrained acoustic environments using deep learning”. In: *Proc. of UbiComp ’15*.
[32] N.Lane et al. “DeepX: A Software Accelerator for Low-power Deep Learning Inference on Mobile Devices”. In: IPSN ’16.
[33] *nVidia Jetson TX1*. http://www.nvidia.com/object/jetson-tx1-module.html.
[34] *NYPD Plans to Put Body Cameras on All 23,000 Patrol Officers by 2019*. http://www.nbcnewyork.com/news/local/NYPD-Plans-to-Put-Body-Cameras-on-All-23000-Patrol-Officers-by-2019-413487003.html.
[35] F. Pedro et al. “Object detection with discriminatively trained part-based models”. In: *Trans. on PAMI* 32.9 (2010), pp. 1627–1645.
[36] P.Georgiev et al. “LEO: Scheduling Sensor Inference Algorithms Across Heterogeneous Mobile Processors and Network Resources”. In: MobiCom ’16.
[37] P.Viola and M.Jones. “Robust Real-Time Face Detection”. In: *Int. J. Comput. Vision* 57.2 (May 2004).
[38] R.Girshick. “Fast r-cnn”. In: ICCV ’15.
[39] R.Lindsay, L.Cooke, and T.Jackson. “The impact of mobile technology on a UK police force and their knowledge sharing”. In: *Journal of Information & Knowledge Management* 8.02 (2009), pp. 101–112.
[40] S.Bhattacharya and N.D.and Lane. “Sparsification and Separation of Deep Learning Layers for Constrained Resource Inference on Wearables”. In: SenSys ’16.
[41] S.Han et al. “MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints”. In: MobiSys ’16.
[42] S. Sunderrajan, J.Xu, and B. S. Manjunath. “Context-aware graph modeling for object search and retrieval in a wide area camera network”. In: *Proc. of ICSDC ’13*.
[43] T.Zhang et al. “The Design and Implementation of a Wireless Video Surveillance System”. In: MobiCom ’15.
[44] *Vista WiFi*. https://watchguardvideo.com/body-cameras/vista-wifi.
[45] *WatchGuard: Law Enforcement Video Systems*. http://watchguardvideo.com/.
[46] Y.Chen et al. “Glimpse: Continuous, Real-Time Object Recognition on Mobile Devices”. In: Sensys ’15.
[47] Y.Wu and G.Cao. “VideoMec: a metadata-enhanced crowdsourcing system for mobile videos.” In: *IPSN*. 2017, pp. 143–154.
[48] Y.Xiang, A.Alahi, and S.Savarese. “Learning to Track: Online Multi-Object Tracking by Decision Making”. In: *Proc. of the IEEE ICCV*. 2015.
[49] Z.Zhang. “A Flexible New Technique for Camera Calibration”. In: *IEEE Trans. Pattern Anal. Mach. Intell.* 22.11 (Nov. 2000).