Scalable Object Tracking in Smart Cities

Jose Stovall, Austin Harris, Amanda O'Grady, Mina Sartipi
College of Engineering and Computer Science
Center for Urban Informatics and Progress
University of Tennessee at Chattanooga

***Abstract*—In smart cities equipped with cameras, one desirable use-case is to detect and track objects. While object detection has been implemented using various methods, object tracking poses a different problem; to track an object requires object permanence to be established between each frame of video. While many technologies have been proposed as a solution for problem, an implementation with scalability in mind has not been developed and poses many new challenges. This paper proposes e-SORT, a solution for scalable object tracking using an enhanced version of the Simple Online and Realtime Tracking (SORT) algorithm. Beyond its scalability, e-SORT stores a mapping of each objects' locations such that the full path of each object is available and several metrics (such as velocity and acceleration) can be calculated. Both e-SORT's abilities and our proposed solution to scalable object tracking are tested and evaluated on Chattanooga Tennessee's live urban testbed.**

***Index Terms*—Smart City, Object Detection, Object Tracking, Machine Learning, Smart Infrastructure**

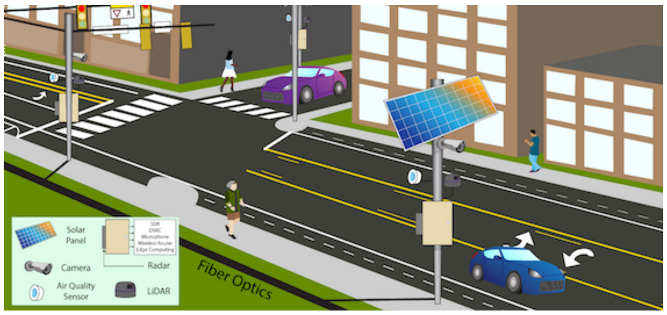

Figure 1: Graphical Representation of the MLK Smart Corridor's Testbed and its features.

I. Introduction

Throughout the last decade, smart cities have grown in popularity. A smart city is a city in which the city and its occupants live with better connectivity to each other. Smart cities often include a wide variety of Internet of Things (IoT) devices and sensors, such as air quality sensors, network activity sensors, Vehicle-to-Infrastructure connectivity and more. Video cameras are often found among these devices, and allow for many forms of analytics via machine learning such as object detection, object tracking and more.

Object tracking is one of many applications for video cameras in a smart city, and provides important data to the city's occupants. In most object detection algorithms, an image (be it a video frame or single image) is fed into the model and the detections are made with their corresponding bounding boxes and labels. While this addresses many challenges, a detection model fails to maintain object permanence in between frames. This poses the issue that object tracking must solve: how to maintain object permanence in between frames. Due to the simplicity of the problem object trackers are usually used in conjunction with object detection models instead of developing their own model to detect objects, and are only tasked to help correlate detections between frames to maintain object permanence. Unlike object detection alone, object tracking offers the ability to store all previous object locations, predict future ones, and calculate many variables and statistics based on an *individual* event. This can provide the city with helpful information on traffic trends, pedestrian density, jay-walking zones and more, allowing it to use this data to make the city more intelligent.

The complexity of making a city "smart" scales with the size of the city; all software and hardware design must be designed with scalability in mind. While many object tracking solutions have already been developed, they do not address the challenge of scalability, therefore any proposed solution must be capable of scaling out to infinitely many cameras provided that the hardware is available. This paper shall address these concerns as they relate to object tracking. Additionally, it shall discuss our scalable object tracking implementation that has been deployed on the testbed on Chattanooga Tennessee's MLK Smart Corridor (illustrated in Figure 1) . Our implementation on this testbed provides the benefit of a real-time live urban environment, as opposed to artificial or simulated environments.

This paper shall break down our enhancements and modifications to a "Simple Online Realtime Tracker" [1] to create e-SORT, and discuss our implementations in a live urban environment and how they validate the necessity of e-SORT. This paper comprises the following sections: Section II surveys popular implementations of object detection and tracking. Section III discusses in more depth what the current issues are, and section IV discusses our proposed approach. Section V discusses our findings in different metrics for our proposed tracking solution, and section VI concludes the paper and discusses future plans and improvements to object tracking in smart cities.

II. Related Works

In this section, we will explore the two major concepts that object tracking tasks are composed of. In general, the architecture of a tracking solution begins with the detection of an object or objects, followed by feeding those detection(s) into an object tracker to correlate detections to previous frames. As such, it is generally possible to choose nearly any object detection solution and fit it to any object tracking solution.

A. Object Detection

Object detection has been studied extensively. In this section, only a few of the most unique (be it accuracy, implementation methods or speed) will be discussed in this paper; You Only Look Once (YOLO) (v1 [2], v2 [3] and v3 [4]) will be discussed in detail for its speed, MaskRCNN [5] for its overall high accuracy, and Objects as Points [6] for its speed and unique implementation.

The authors of [2], [3], and [4] have introduced an incrementally improved real-time object detection model. The authors have proposed a Convolutional Neural Network (CNN), which solves the object detection problem by treating it similar to a regression problem. Their model provides both labels and bounding boxes, and comes in many variants. The original YOLOv1 is limited to only 20 different detectable objects, and had two variants: YOLO and Fast YOLO. YOLO would perform at 45 FPS, and Fast YOLO would perform at 155 FPS. YOLOv2 is capable of detecting 80 different types of objects, and achieves over 40 FPS (depending on the variant). YOLOv3 brought about a series of smaller quality-of-life changes, such as utilizing the GPU for more tasks than before. All variants of YOLO provide a balance between performance and accuracy that makes it a good fit for a large-scale problem such as the one we address in this paper.

The authors of [5], an extension to Faster-RCNN [7], not only detects objects but also predicts each object's mask. Mask R-CNN outperforms all other existing single-model detectors (at the time the authors wrote their paper), and is designed for use beyond its original scope as an object detection model - such as human pose estimation. While Mask R-CNN is an excellent candidate for the detection component in a real-time scalable object tracking solution for smart cities, it lacks speed - one of the most critical components for real-time applications. At a peak of 5 Frame Per Seconds (FPS), it is incapable of keeping up with the real-time environment that a smart city encompasses. The extra 25 frames that most cameras produce will be wasted, and such poor frame-rate may lead to issues with some object tracking algorithms (such as SORT [1]) and poses a risk of missing important event details; a near-miss may be undetectable at frame-rates so low.

The authors of [6] propose that axis-aligned bounding boxes are highly inefficient. They have introduced the concept of using points to determine detections, as they are much more efficient. These points are found using keypoint estimation, and are located in the center of the object's bounding box. The additional information (such as the size, bounds, three-dimensional location, pose, and more) are found using regression. The authors claim that they have obtained the "best speed-accuracy trade-off" in their model implementation of this idea, CenterNet. Results from testing with the Pascal VOC 2007 test prove this to be true; CenterNET achieved as high as 142FPS average using CenterNet-Res101 (at a 512x512 input resolution) and still managed to achieve a 72.6 mean average precision (mAP) with an Intersection of Unions (IoU) threshold of 0.5 (mAP@0.5). This approach proves to have the frame-rate and accuracy that is necessary for the large-scale real-time requirements discussed later in this paper.

B. Object Tracking

As discussed, tracking an object poses a problem different from that of detection; in general, off-the-shelf object detection algorithms do not offer any means to maintain object persistence between the frames of a video. There are two common approaches within the subject of object tracking: single-object, and multi-object. Single-object tracking focuses on tracking a single object within the camera view, where multi-object tracking will track all given objects with the camera view. For the purposes of a smart city there will almost always be more than a single object within the camera's view, so this paper has also been written to address a multiple-object environment.

The authors of [8] have proposed an MIL object tracker with online boosting. Their proposed tracking system does not require object detection (though it *can* be augmented with it), as it only requires an initial bounding box which can be drawn by multiple sources (such as a human or a model). While this is particularly effective for less complex views, it is not an ideal solution for camera views with many objects entering and leaving the scene. Therefore, for the purpose of this paper we have considered this tracker *only* in conjunction with a model, whose detections can be used to update the tracker's tracklets (i.e. re-seed the tracker). Since new objects within the camera view can only be tracked once the tracker is fed in a new bounding box, a compromise must be made in how often to re-seed the tracker with model detections. One approach would be to re-seed the tracker with model detections every other frame, though we have found this to result in a higher number of instances where an object's ID is changed to a completely new one. Another approach would be to wait longer to re-seed the tracker with model detections (for example, every 10 frames). While this approach is a good compromise, it still allows for objects to be re-assigned to an ID of an object which left the camera's view anywhere between 0 and x frames ago, where x is the number of frames between re-seeding the tracker with model detections.

The authors of [9] introduce a correlation-filter based tracking algorithm which uses their proposed Minimum Output Sum of Squared Error (MOSSE) filter. This filter allows for stabilization of correlation filters upon initialization. This method is robust enough to maintain its performance through changes of multiple conditions such as scale, lighting, and more, all-the-while performing at over 600FPS. As it is a correlation-filter based tracker, it also suffers from the same issues and compromises that [8] does.

The authors of [1] have proposed a “Simple Online Real-time Tracker”. This tracker works by taking detections from some model, and associating a tracklet to these detections using Intersection of Unions (IoU) and a Kalman Filter. The Kalman Filter is used to predict the next location of a tracklet, which is used with IoU and the current detections to associate a detection back to the tracklet. This tracker requires a model’s detections *every frame*, as these detections are how the tracker performs its reassociation to tracklets. The tracker manages old tracklets by determining how recently the tracklet has been reassociated. If the tracklet has not been reassociated within some threshold of frames (user-configurable), it is deleted for memory conservation purposes. One shortcoming of this tracker is that it requires a model which is not provided in the paper. Additionally, the tracker’s tracklets do not store some useful information such as a history of that tracklet’s locations (and including timestamps would also prove useful), and obsolete tracklets (those tracklets which cannot be reassociated with detections) are simply deleted. This tracker is also heavily frame-rate dependent; the higher the frame-rate of the video and detections, the fewer instances of ID reassignment and higher the precision of the tracker.

The authors of [10] have proposed a multiple object tracking solution which uses a K-Shortest-Path algorithm in conjunction with a linear equation to perform tracking. While this solution is highly accurate, it fails to perform with the accuracy and efficiency required to run the many cameras along the MLK Smart Corridor. Between the linear functions and the K-Shortest-Path algorithm, this object tracking model is enable to meet our 30FPS requirement, and would be problematic to scale with the smart city.

C. Motivations and Contributions

While the combination of SORT [1] and an off-the-shelf object detection algorithm solves the challenge of tracking an object in real-time, it fails to address the issue of scalability in smart cities. We must design a scalable software architecture for tracking objects in real-time, allowing for new cameras to be added with ease to increase maintainability for the long term. As a part of this scalable workflow, we will need to submit trackers to a real-time publish/subscribe database for graphing, analysis and more. In order to do so, SORT [1] will require modifications to optimize certain elements for this software architecture.

III. Problem Statement

Based on the other works discussed in II, challenges for e-SORT have been established. e-SORT must be capable of submitting data from obsolete tracklets (tracklets which no longer appear within the frame) to our real-time publish/subscribe database. While using an object detection algorithm with SORT [1] will reduce the complexity of this problem, the following challenges still remain:

1) This combination of detection and tracking needs to be scalable.
2) Once a tracklet is obsolete, it must be captured and submitted to our real-time publish/subscribe database, requiring that:
    a) The tracklet stores its own label.
    b) The tracklet stores a mapping of its history of locations in the frame with their corresponding timestamps (in UTC).
    c) SORT [1] is modified to allow access to the obsolete trackers.
3) The tracklet map of location to timestamps is reduced to prevent overly large messages.
4) Tracklet data is formed as a JSON to submit to our real-time publish/subscribe database.

Additionally, e-SORT must be implemented with scalability in mind so that it may perform real-time object tracking across the testbed’s cameras (of which 24 have been deployed already, with up to 8 more to be deployed across two additional intersections). Since each camera runs at a resolution of 1920×1080 at 30 frames per second, this task is computationally expensive as well and thus hardware considerations must also be made. Since no publicly available solutions for scalable object tracking exist, it is necessary to create the entire architecture which utilizes e-SORT (among many other technologies).

IV. Proposed Approach

A. Proposed Architecture

We propose a multi-processed architecture, seen in Figure 2, which is designed with scalability in mind. A block queue is used to transfer data between the four different types of processes, and there is a queue dedicated for raw frames, processed frames and tracking results. Each of the queues’ placements within the architecture can be seen in Figure 2. The four aforementioned types of processes are:

1) Capture Processor: This architecture contains n Capture Processors (represented in blue in Figure 2, where n is the number of cameras in the Smart City. Each of these is a dedicated process for capturing video frames from one specific camera on the infrastructure. Each Capture Processor contains its own instance of SORT [1], as there are $n/3$ Frame Processors. $n/3$ was found to be the most efficient distribution of our limited computational resources. The Frame Processor which processes the frame resulting from this process is random as a result of the shared queue logic that allows each processor to share data.

2) Frame Processor: This architecture contains $n/3$ Frame Processors (represented in red in Figure 2), where n is the number of cameras in the Smart City. This number was a result of testing, where the Frame Processor was allocated another Capture Processor until the Frame Processor could not keep up. The Frame Processor is responsible for performing object detection via **any model**, although some models may require modification of the Frame Processor’s algorithm if their output

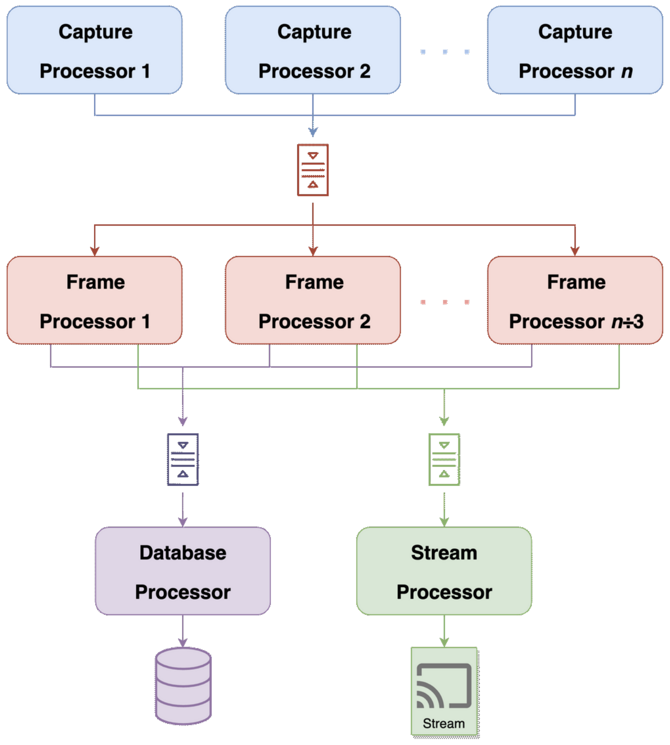

Figure 2: Architecture Overview, where n is the number of cameras in the Smart City. Blue indicates n number of Capture Processors, whose purpose is solely to capture video footage. Red indicates $n/3$ number of Frame Processors, whose purpose is to detect and track objects. Purple indicates the Database Processor, whose purpose is to form data from the Frame Processor into a JSON format and submit to our real-time publish/subscribe database. Green indicates the Stream Processor, whose purpose is to stream all n streams for demonstration purposes.

is not (or cannot be parsed to be) of form $[x_1, y_1, x_2, y_2]$. The results from detection are fed into the SORT instance which corresponds to the frame. Obsolete tracklets from the results of SORT are then fed into the Submission Queue, the original video frame is destroyed for privacy purposes, and the frame from the Frame Processor has overlays drawn around objects, which is fed into the Stream Queue.

3) Submission Processor: This architecture contains 1 Submission Processor (represented in purple in Figure 2). The Submission Processor uses in the obsolete tracklets from the Submission Queue to create a JSON message containing critical information. This JSON message contains the following:

- ID: A UUID created for the tracklet.
- Label: The label of the tracklet, given by the detection algorithm.
- Hit Count: The number of locations within the frame that the tracklet was found.
- Locations: A mapping of $timestamp : boundingbox$ for that tracklet, which stores every single bounding box (a list containing x_1, y_1, x_2, y_2) and maps it to the time (in UTC) that it was detected.

Once the JSON object is created, it is published to our real-time publish/subscribe database for further consumption. This data allows us to perform several useful analytics such as near-miss detection without sacrificing the privacy of the city's occupants.

4) Stream Processor: This architecture contains 1 Stream Processor (represented in green in Figure 2). The purpose of the Stream Processor is purely visual; it enables us to show the results of our tracking on a web interface, which has backend authentication for privacy purposes. The Stream Processor takes frames from the Stream Queue and streams them using an asynchronous web-hosting library.

B. e-SORT Modifications

This architecture requires some modifications to the original SORT algorithm. We propose an enhanced version of the SORT algorithm called e-SORT. The next three sections shall discuss in detail what modifications were made and how they are implemented to create e-SORT.

1) Obsolete Tracklets: The original SORT algorithm, for the purposes of memory management, *deletes* obsolete tracklets upon calling the $update$ function. Since our publish/subscribe database is not designed to handle the updating of an entry, it is required that the JSON submission contain *all* of the data for the tracklet, as opposed to submitting the data as it comes in. As a result, keeping the obsolete tracklets was a clear solution to this problem, as it would allow us to have all of the information at once and will not be updated (since the tracker is *obsolete* and won't be updated again within SORT). Therefore, modifications to the $update$ function have been made to allow it to also return these obsolete tracklets. After enqueueing the tracklet to the Submission Queue, the obsolete tracklet is deleted. This retains the much needed memory management while meeting our needs for the publish/subscribe database.

2) Object Labels: Each tracklet in SORT does not store its label. This is an important trait as it simplifies the workflow (as detailed in the Obsolete Tracklets portion above). To resolve this, modifications to SORT's $update$ method had to be made. The $update$ function now requires a $labels$ argument, such that when the tracklet is reassociated to a detection, it can also be reassociated with the correct label. For further improvement, model detection labels may vary for the same tracklet. Instead of overwriting the label each time, a list of all assigned labels from the detection model are kept within the tracklet. When the tracklet's label is then requested using the get_label function, the label which is most frequently found is returned. This prevents any anomalies from the labelling process to be corrected in most instances.

3) History of Locations: Another trait required from SORT was a way for each tracklet to retain knowledge of "when" and "where" it was. This takes form as a Python *dict* whose keys are timestamps (in UTC) and whose values are a bounding box (of form x_1, y_1, x_2, y_2). A new key-value pair are appended to the tracklet's locations dict when *update* is called to the SORT instance. This is also the time at which the timestamp is created which delays the timestamp less than a second more than the actual event occurring, which for most use-cases in a smart city is negligible. This history allows for a user to subscribe to the publish/subscribe database later and perform later analysis and calculate properties such as velocity and acceleration.

As mentioned in Section III, the tracklet's history of locations must be reduced before publishing to our real-time publish/subscribe database. This can be done by applying a threshold for the euclidean distances between two timestamps. This can be seen in the algorithm below. This algorithm is important as it prevents a still car from creating *thousands* of entries to flood into the database, while retaining important details such as how long it stayed in that location. In testing, a 30 pixel threshold has been found to be most optimal within our environment.

```
Input: locations, threshold
for timestamp, bbox in locations do
    if distance(last_bbox, bbox) >= threshold then
        ret_val[timestamp] = bbox
        last_bbox = bbox
    end
end
return ret_val
```

Algorithm 1: History Simplification

V. Numerical Results

To test e-SORT and the scalable architecture discussed in Section IV, we used the testbed located on Chattanooga's MLK Smart Corridor (located in Tennessee). The testbed is an open platform which permits research problems to be tested in a live urban environment. At the time of writing, the testbed contains 27 cameras across 9 major intersections. All IoT devices within the testbed have a 10-gigabit fiber backbone which supports real-time data transfers to and from each pole. The results described in this section come from real-time video stream from this testbed. More information on the testbed's capabilities can be found on the testbed's website [11], and a representative figure can be seen in Figure 1.

Determining the accuracy of our approach poses new challenges, as there are multiple metrics by which accuracy can be determined. This is made possible by the login-protected live streaming of our processed video streams, which allow an analyst to view what the algorithm is detecting. The method of accuracy determination used and discussed in this paper was a manual process in which an analyst would count the number of times an event would occur, then calculate the accuracy using total count of objects from the beginning and end of their session. The types of events counted for were multiple-ID instances, no ID instances, and mislabeled instances, each of which will be discussed in further detail below. Each experiment was performed on differing video data and results, as it is against our privacy policy to store any video from the testbed for any time period longer than 30 minutes.

A. Multiple-ID Instances

Multiple-ID instances are cases in which one object gets multiple IDs, meaning its tracklet was lost too early. In observing the object tracking algorithm, on average for every 115 objects there would be five cases of ID reassignment. Considering that this count of objects is gathered from the visualization, the object count should actually be approximately five ID reassignments per 110 objects. This can be represented as a 95.45% ID retention rate or a 4.55% ID reassignment rate.

The importance of ID reassignment is beyond aesthetic; ID reassignment is a direct result of tracklet reassignment. This indicates that the tracklet prior to reassignment will be considered as obsolete before the object actually is, resulting in a single object represented by two tracklets. This is problematic for a few reasons, one of which being the real-time publish/subscribe database. Since we submit the data from tracking upon retrieval of obsolete tracklets, one object will be represented by two events in the publish/subscribe database. This means that traffic counts may be higher (albeit only by a difference of 5-10 objects), but more importantly the future use of the dataset being created by our approach may be problematic, as two objects' paths may be shorter than they should be.

B. No ID Instances

No ID instances are cases in which an object is not given an ID at all, indicating that our detection algorithm did not *detect* the object to begin with. We have monitored these events and have found that we have a 2.978% likelihood for improperly tracking / detecting an object; that is to say, we have a 97.022% accuracy in this category. While the number of objects this was analyzed for was 6421 according to the visualization, this is technically incorrect. Considering that 6421 was the count for an average of 97.022% of all *actual* objects through the scene, then the actual count should be approximately 6612 objects.

C. Mislabeled Instances

Mislabeled instances: cases in which the label from our object detection algorithm failed to give the correct label for the object. We have gathered this metric for 8112 different objects, revealing that only 2.79% of the time is an object mislabeled; that is to say, we have a 97.21% accuracy in this category.

D. Real-time Visualizations

An easy to understand use for the real-time tracking data coming into our real-time publish/subscribe database takes form through visualizations. While there are many options to use, there are two that have already been implemented and receive regular use:

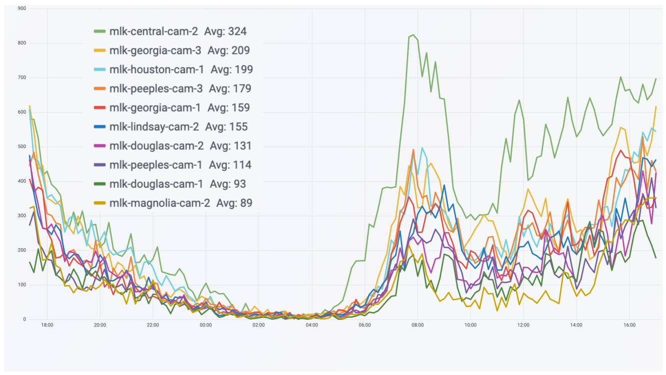

Figure 3: Graph of Aggregated Counts of Objects from Tracking for 24 hours, where each different color represents a different camera's object counts, e.g., mlk-central-cam-2 Avg: 251 indicates that an average of 251 objects have passed through that camera's view in a 10-minute interval

1) Graphing Dashboard: As a part of our server's software stack, we have deployed a graphic dashboard to read the data published to our real-time publish/subscribe database. This allows us to visualize the data easily, making some traffic trends quite clear. Figure 3 shows a sample for 10 cameras on Chattanooga Tennessee's up and coming smart city.

2) Overlay Generator: Written using the OpenCV library for Python, another visualization that uses the real-time tracking data is an Overlay Generator. This overlay generator uses an existing image of the camera view requested, and overlays paths of objects (retrieved from our real-time publish/subscribe database) in different colors for each object. This allows us to show off what the data looks like in an anonymous but still important way, and makes more visible. For example, this imagery makes it possible to see jay-walking trends, vehicles in the bike lanes and more. This can be seen in Figure 4.

Utilizing only ten of the 27 cameras on the MLK Smart Corridor's testbed (due to hardware limitations), our proposed solution to scalable object tracking produces approximately 200,000 results per day. As the accuracy of our proposed solution shows improvement, this number will more accurately reflect a day's worth of traffic in the live urban environment we have performed tests on.

VI. Conclusion and Future Work

In this work, we have discussed our solution to object tracking which can scale to fit the size of a smart city. This approach allows for each type of detectable object (determined solely by the object detection model used) to be treated as an event with unique attributes, such as object label, path taken, timestamp of each location, and more. With these metrics, it is possible to perform trajectory prediction using known velocity and acceleration for the event. If the predicted trajectories of any two objects overlap, their trajectories and velocities are used to calculate time 'til collision (TTC). If the TTC of these two objects falls below a certain threshold and the two objects

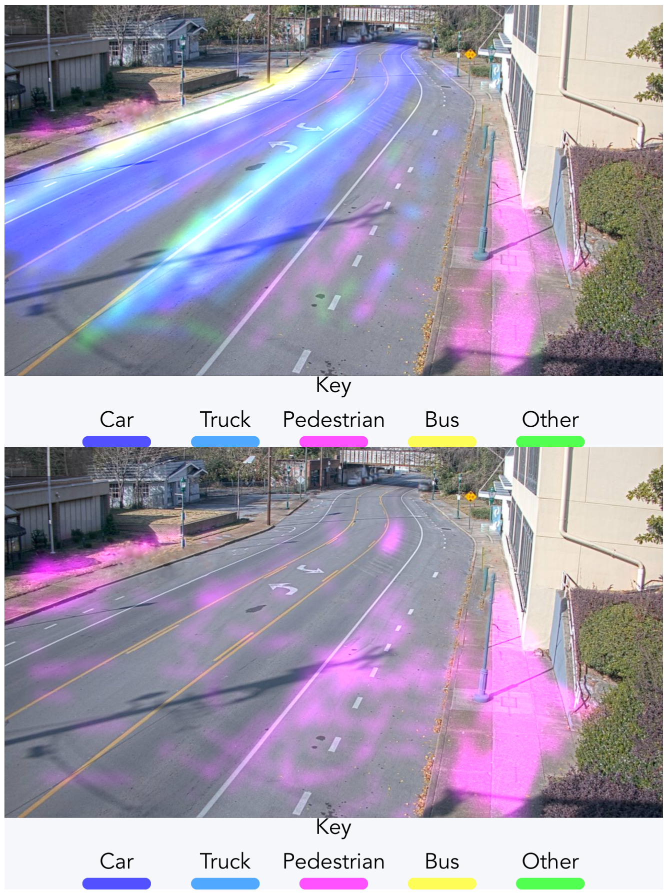

Figure 4: Heat map of the tracked objects from one camera. Top: All objects' paths; Bottom: Objects paths for objects with label "Pedestrian". The bottom image makes pedestrian walking trends clear.

do not collide, a near-miss event has just occurred. This near-miss use-case will be explored in a future work.

There are a few methods which can be applied to improve the proposed solution for scalable object tracking, the first of which has to do with frame-rate. Due to the nature of how SORT [1] works, it is very frame-rate dependent; since it uses a Kalman Filter to predict the next location, a higher frame-rate will increase the accuracy of this prediction. The testbed our approach has been deployed on is equipped with cameras which stream two types of video stream in parallel: H.264, and MJPEG. MJPEG is the most convenient solution to use in code, but comes at the cost of low frame-rate (6 to 8 FPS) as a result of the camera itself. This is believed to be the main contributor to the re-assigned ID issue discussed in Section V. Since the H.264 stream does maintain 30FPS with no issue it would is the ideal solution, but includes a decoding overhead which is not present when using the MJPEG stream. A solution is being implemented now to offload the H.264 decoding overhead.

Another optimization yet to be deployed is to reduce overall CPU usage, allowing for more instances of the tracking algorithm to be run on the same hardware. This source of the

CPU savings was the video streaming, used as a visualization and for demonstration purposes. The prior video streaming implementation used a synchronous, threaded approach, in which every new stream-viewing client would be processed on a newly spawned thread. The new approach uses an asynchronous method, which does not require new threads for each user.

Acknowledgment

The project team would like to extend their gratitude to the NSF US Ignite (Award #1647161) for partially funding this project.

References

[1] A. Bewley, Z. Ge, L. Ott, F. Ramos, and B. Upcroft, “Simple online and realtime tracking,” *CoRR*, vol. abs/1602.00763, 2016. [Online]. Available: http://arxiv.org/abs/1602.00763

[2] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” *CoRR*, vol. abs/1506.02640, 2015. [Online]. Available: http://arxiv.org/abs/1506.02640

[3] J. Redmon and A. Farhadi, “Yolo9000: Better, faster, stronger,” *arXiv preprint arXiv:1612.08242*, 2016.

[4] ——, “Yolov3: An incremental improvement,” *arXiv*, 2018.

[5] K. He, G. Gkioxari, P. Dollár, and R. B. Girshick, “Mask R-CNN,” *CoRR*, vol. abs/1703.06870, 2017. [Online]. Available: http://arxiv.org/abs/1703.06870

[6] X. Zhou, D. Wang, and P. Krähenbühl, “Objects as points,” *CoRR*, vol. abs/1904.07850, 2019. [Online]. Available: http://arxiv.org/abs/1904.07850

[7] S. Ren, K. He, R. B. Girshick, and J. Sun, “Faster R-CNN: towards real-time object detection with region proposal networks,” *CoRR*, vol. abs/1506.01497, 2015. [Online]. Available: http://arxiv.org/abs/1506.01497

[8] B. Babenko, M. Yang, and S. Belongie, “Robust object tracking with online multiple instance learning,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 8, pp. 1619–1632, Aug. 2011.

[9] D. S. Bolme, J. R. Beveridge, B. A. Draper, and Y. M. Lui, “Visual object tracking using adaptive correlation filters,” in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, June 2010, pp. 2544–2550.

[10] J. Berclaz, F. Fleuret, E. Turetken, and P. Fua, “Multiple object tracking using k-shortest paths optimization,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 9, pp. 1806–1819, Sep. 2011.

[11] “CUIP.” [Online]. Available: https://utccuip.com/