Continuous, Real-Time Object Detection on Mobile Devices without Offloading

Miaomiao Liu, Xianzhong Ding, Wan Du
Department of Computer Science and Engineering
University of California, Merced, USA
{mliu71, xding5, wdu3}@ucmerced.edu

***Abstract*—This paper presents AdaVP, a continuous and real-time video processing system for mobile devices without offloading. AdaVP uses Deep Neural Network (DNN) based tools like YOLOv3 for object detection. Since DNN computation is time-consuming, multiple frames may be captured by the camera during the processing of one frame. To support real-time video processing, we develop a mobile parallel detection and tracking (MPDT) pipeline that executes object detection and tracking in parallel. When the object detector is processing a new frame, a light-weight object tracker is used to track the objects in the accumulated frames. As the tracking accuracy decreases gradually, due to the accumulation of tracking error and the appearance of new objects, new object detection results are used to calibrate the tracking accuracy periodically. In addition, a large DNN model produces high accuracy, but requires long processing latency, resulting in a great degradation for tracking accuracy. Based on our experiments, we find that the tracking accuracy degradation is also related to the variation of video content, e.g., for a dynamically changing video, the tracking accuracy degrades fast. A model adaptation algorithm is thus developed to adapt the DNN models according to the change rate of video content. We implement AdaVP on Jetson TX2 and conduct a variety of experiments on a large video dataset. The experiment results reveal that AdaVP improves the accuracy of the state-of-the-art solution by up to 43.9%.**

I. Introduction

Continuous and real-time object detection is essential for many mobile applications, like traffic monitoring [8] and augmented reality (AR) [6]. For example, real-time warnings can be sent to road users automatically by a camera installed on top of a highway road if any reckless driving maneuvers are detected. AR-based videos are promising in many applications, such as tourism, navigation and entertainment [9], which require to detect and track objects in videos on mobile devices continuously and in real time, like 30 or 60 Frames Per Second (FPS). Deep learning has shown superior performance in object detection and many deep neural networks have been developed, like YOLO [2] and SSD [10]. Albeit high accuracy, they normally involve intensive computation that cannot be fully supported by the constrained hardware resource of mobile devices, i.e., they cannot accomplish the processing of one frame before the next frame is captured, i.e., 33 ms for 30 FPS. Some solutions offload a part of computation from mobile devices to the cloud [5], [11], [12]. However, offloading suffers from privacy concerns and unpredictable network latency [3].

Recently, some compressed DNN models have been developed to do object detection task efficiently on mobile devices without offloading, e.g., YOLOv3-tiny [2] and Faster R-CNN based on MobileNets [13]. Our experiments show that YOLOv3-tiny can process a frame on Nvidia Jetson TX2 within 60 milliseconds, but its detection accuracy is low. At the same time, some light-weight deep learning frameworks have been developed, such as DeepMon [3] and NestDNN [14]; whereas they cannot meet the real-time requirement of video processing. For example, DeepMon achieves continuous video processing at 1-2 frames per second [3].

In this paper, we develop AdaVP, an accurate and real-time video processing system on mobile devices. AdaVP is based on a novel parallel object detection and tracking pipeline, named as Mobile Parallel Detection and Tracking (MPDT). Nowadays, many of mobile devices have GPU, like Samsung Galaxy S10 and Apple iPhone 11, which allow us to implement DNN-based object detection on GPU and object tracking on CPU. The two types of operations are executed independently on two different hardware resources.

At the beginning, we use a general DNN-based object detector (i.e., YOLOv3 [15] in our current implementation) to process one frame (frame 0). After the processing of that frame (e.g., 330 ms), the camera may have already captured several frames in the buffer (e.g., 11 frames a capture rate 30 FPS). The object detector will start processing the newest frame in the buffer (e.g., the 12th frame). At the same time, based on the objects identified by the object detector, an object tracking algorithm will localize these objects in the accumulated frames (the 1st-11th frames). The tracking accuracy degrades gradually due to the accumulation of tracking error and the appearance of new objects. Before the tracking accuracy drops to too low, the object detector will provide the objects in a new frame (e.g., the 12th frame) with a high detection accuracy. By parallel detection and tracking, we can obtain the object detection results at the maximum frequency and use them to calibrate the object tracking.

To further improve the accuracy of proposed parallel detection and tracking pipeline, we adjust the setting of DNN object detection model at runtime, based on the tradeoff of initial detection accuracy and tracking accuracy degradation. On the one hand, a high detection accuracy normally requires intensive computation (i.e., a heavy-weight DNN model) and long processing time. The latter means more frames accumulated

TABLE I
COMPARISON OF ADAVP AND THE EXISTING WORK

Features \ System	Glimpse [1]	TinyYOLO [2]	DeepMon [3]	DeepCache [4]	DeepDecision [5]	Liu et al. [6]	MARLIN [7]	AdaVP
Real-time updates	✓	✓				✓	✓	✓
No offloading		✓	✓	✓			✓	✓
Localizes multiple objects		✓	✓	✓	✓	✓	✓	✓
Using DNN		✓	✓	✓	✓	✓	✓	✓
Model adaptation								✓
High accuracy			✓	✓		✓		✓

in the buffer. One the other hand, the object tracking accuracy degrades sharply if the video content changes fast, since the tracking error accumulates fast and many new objects appear in the accumulated frames. In this case, we need to run a light-weight object detector in order to calibrate the object tracker more frequently, although the detection accuracy is relatively low. On the contrary, if the video content change slowly, we prefer to use a heavy object detector that provides high detection accuracy. Although the detection latency is long, it does not cause the tracking accuracy to degrade much.

In this work, we adapt the DNN model settings by changing the frame size of YOLOv3 at runtime. YOLOv3 allows us to change the frame size at runtime without reloading the model. A large frame size indicates long computation latency and high detection accuracy. In AdaVP, we measure the video content changing rate based on the intermediate results of object tracking. We learn the quantified relationship between the best frame size and the video content changing rate, based on a large amount of training data. We then develop a DNN model setting adaptation algorithm that decides whether to switch to a different frame size after each object detection.

To perform object tracking, we use the standard *good features to track* [16] method to extract good features in the last DNN detected frame. And then we track these features in the following frames by the *Lucas-Kanade* optical flow method [17]. Due to the tracking and rendering latency of one frame (from 57 to 70 ms) is larger than the frame interval of a video (e.g., 33 ms), we also design a scheme to skip some frames from tracking without impacting the synchronization with the operations of object detection.

We implement AdaVP on an open mobile platform, Nvidia Jetson TX2, based on Pytorch deep learning framework. We use both standard video dataset [18] and some videos downloaded from public websites [19], [20] to do experiments. The data we used to train our DNN model adaptation module contains 105205 frames, and the videos for validation contain 141213 frames. The evaluation results show that AdaVP improves the accuracy of the state-of-the-art solution by up to 43.9%. In particular, the parallel detection and tracking pipeline (MPDT) improves accuracy by up to 21.95%, and the model setting adaptation algorithm in AdaVP further improves accuracy by up to 34.1% on top of MPDT.

In summary, the contributions of this work are as follows:

- We develop AdaVP, a mobile video processing system that achieves high detection accuracy in real time on mobile devices without offloading.
- We develop a parallel detection and tracking pipeline to fully utilize the computation resource on current mobile devices for high detection accuracy.
- We further increase the detection accuracy by adjusting the DNN model setting at runtime according to the variation of video content.
- We implement the system on an open mobile platform, Nvidia Jetson TX2 and extensively evaluate the system using different types of videos.

II. RELATED WORK

We compare AdaVP with some prior works in terms of some common features of video processing in Table I.

Real-Time Mobile Vision without Offloading. MARLIN [7] is the latest work about real-time mobile vision without offloading. It runs object detector and object tracker in a sequential order. When object tracker starts tracking the objects in the accumulated frames, object detector stops its detecting task. MARLIN is inefficient when the video content becomes complex and varies fast. And it always uses the same DNN model setting during video processing. Different from MARLIN, AdaVP runs object detector and object tracker in parallel and switches among different model settings according to the changing rate of video content.

Real-Time Mobile Vision with Offloading. Due to the limited computation, storage and power of mobile devices, offloading intensive tasks to the cloud to process is a popular way [1], [5], [12], [21]–[23]. However, mobile vision offloading approaches are easily affected by network conditions and usually receive stale results which degrade processing accuracy of mobile continuous vision. MCDNN [12] and DeepDecision [5] design a framework which decides to offload tasks to the cloud or execute locally according to the network condition. Glimpse [1] and Liu et al. [6] offload some key frames to the cloud to do detection and tracks the detected objects on mobile devices. However, they all suffer from long transmission latency and privacy concerns. In contrast, AdaVP focuses on executing object detection task on mobile devices without offloading.

Mobile Deep Learning. With the development of deep learning and deep reinforcement learning [24], [25], many works take effort to reduce latency and computation time of deep learning algorithms on mobile devices. DeepX [26] designs a pair of resource control algorithms for deep learning inference. DeepMon [3] designs a suite of optimization techniques to accelerate the processing of DNNs on mobile

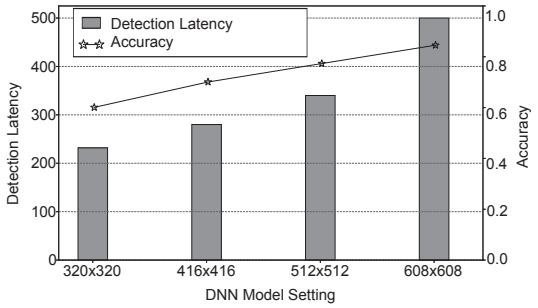


Fig. 1. Detection latency and accuracy per frame for different frame sizes. The latency data is presented by the bars, and the accuracy data is shown by the lined stars.

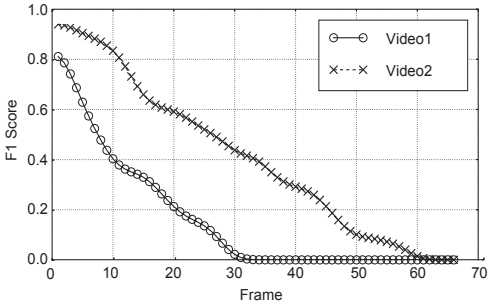


Fig. 2. Tracking accuracy of two different videos. The content of Video1 changes faster than that of Video2. YOLOv3-608 is used to detect the objects in the first frame.

devices. DeepEye [27] proposes a novel inference software pipeline that enables multiple CNN models to execute locally without offloading. DeepCache [4] presents a principled cache design for deep learning inference in continuous mobile vision. NestDNN [14] takes the dynamic runtime resources into account to enable resource-aware on-device deep learning for mobile vision systems. Naderiparizi et al. designs [28] a novel architecture that gates wearable vision using low-power vision modalities to reduce mobile power and data usage. AdaDeep [29] develops a usage-driven selection framework to automatically select a combination of compression techniques for a given DNN. Our work is orthogonal and complementary to these prior works, it can work on the top of the above works to make the camera-based mobile applications more efficiently.

III. Motivation

In this section, we conduct experiments on real mobile devices to evaluate the performance of state-of-the-art detection and tracking models. From the experiment results, we derive four observations and challenges that motivate our design.

A. Experimental Setting

DNN-based Object Detection. Two DNN-based object detection pipelines are widely used. The first type detects and classifies objects by a single DNN object detection model, e.g., Faster RCNN [30], SSD [10] and YOLO [2]. The second type first detects the regions of interest using background subtraction and then classifies each small region using a DNN classification model, e.g., ResNet [31] and InceptionV4 [32]. The latter sometimes is insufficient if there are many regions of interests to detect [33]. In this work, we thus adopt a single DNN model for both detection and classification.

In particular, we use YOLOv3 [15] based on the following considerations. 1) YOLOv3 has an optimal overall performance (i.e., accuracy and latency) among all the object detection architectures [15]. On the same hardware, YOLOv3 runs significantly faster than other detection models such as SSD and R-FCN, and still provides comparable accuracy [34]. 2) YOLOv3 scales with a set of input frame sizes, which further determines the processing time of one frame. We can change the input frame size of YOLOv3 during the processing of one video without changing the weights of the model. This feature allows us to adjust its accuracy and detection latency without loading a new model.

Object Tracking Algorithms. When the DNN model is detecting the objects in one frame, *good features to track* algorithm is used to extract good feature points from the detected frame. And then we implement object tracking algorithm based on standard *Lucas-Kanade* method [17]. The details of object tracking can be found in IV. The *Lucas-Kanade* method is widely used in some prior works [1], [7]. It can estimate the positions of feature points in next frame by a local image flow (velocity) vector (V_x ,V_y).

Besides tracking objects, it is unique in our work to use the intermediate results of the *Lucas-Kanade* method to adapt the DNN model setting. We need to detect how fast does the video content changes in time. The average moving velocity of key points between frames is a lightweight and good indicator to describe the changing rate of video content.

Hardware. We use Nvidia Jetson TX2 mobile development board as mobile device in this work. TX2 is a representative open-source mobile platform that can easily establish a mobile development environment with Jetson TX2 development kit, like configuring cuDNN, CUDA Toolkit. It includes 6 CPU cores, 8-GB DRAM and an integrated 256-core Pascal GPU. In our implementation, we run Ubuntu 16.04 OS on a Nvidia Jetson TX2 platform. We implement our system based on Nvidia JetPack [35], Nvidia TensorRT [36] and Nvidia Video Codec SDK [37].

Performance Metric. We use F1 score to measure the detection or tracking accuracy of a single frame. F1 score is the harmonic mean of precision and recall, calculated as:

$$F1\ Score = 2\ (\frac{1}{Precision} + \frac{1}{Recall}), \tag{1}$$

where precision is the ratio of the number of true positives to the total number of objects detected by the scheme, and recall is the ratio of the number of true positives to the total number of objects in the ground truth. When the detected bounding box has the same label and sufficient spatial overlap with the ground truth box, this object will be identified as a true positive. We use intersection over union (IoU) to measure the spatial overlap.

$$IoU = \frac{area(Y \cap G)}{area(Y \cup G)}, \tag{2}$$

TABLE II
THE LATENCY OF DETECTION AND TRACKING FOR ONE FRAME.

Component	Time (ms)
YOLOv3 detection latency	230-500
Good feature extraction	40
Tracking latency	7-20
Overlay latency	50

where Y is the detected area of objects from the scheme under evaluation and G is the ground truth area of objects. The value of IoU is set to 0.5 in this work.

Generally, a high F1 score indicates better detection and tracking accuracy, e.g., an object detector is considered to be perfect when its F1 score is 1. To calculate the false positive and false negative, we need to know the ground truth, i.e., labels and accurate locations of objects in frame. In our experiment, we use the detection results of YOLOv3-704 as the ground truth. Since the large frame size 704x704 provides a high detection accuracy.

B. Experiment Results and Observations

YOLOv3 has two versions, i.e., a full version [15] and a lightweight version YOLOv3-tiny. YOLOv3-tiny is tailored for mobile devices by trading detection accuracy for short processing latency [2]. Based on our experiments on 13 video clips with 141213 frames, YOLOv3-tiny still cannot provide real-time (30 fps) video processing on mobile device. What is worse, its average F1 score per frame is as low as 0.3. Only 0.7% frames achieve a F1 score higher than 0.7. The official website of YOLO [2] also shows that full YOLOv3 can provide more than 55.6% higher accuracy than YOLOv3-tiny. Therefore, we use YOLOv3 not YOLOv3-tiny as the object detector in this work.

Detection Accuracy and Latency. Figure 1 depicts the detection latency and accuracy of YOLOv3 under different settings of frame sizes. In this experiment, we use YOLOv3 to process 4000 frames one by one. We record the F1 score and the processing latency of each frame. Both the average processing time and detection accuracy per frame increases as the frame size of DNN model increases. The processing time changes from 230 ms to 500 ms. The F1 score per frame augments from 0.62 to 0.88, if the largest frame size 608x608 among our settings is used.

Observation 1: Even with the lightest model setting (i.e., YOLOv3-320 in our implementation), the DNN-based object detector cannot process a video in real time. To capture the speed of mobile cameras (like 30 or 60 FPS), after processing one frame, the object detector must process the newest frame in the frame buffer that was captured by the camera. The frames between two processed frames will be skipped by the object detector. The objects detected by the first processed frame will be used as the reference by the tracking algorithm to track these objects until the object detector accomplish processing the next frame.

Observation 2: For one frame, a larger YOLOv3 frame size produces higher detection accuracy, but with longer processing latency, and vice versa. If a large frame size is used, the detection accuracy is high, which provides the tracking algorithm with a high initial tracking accuracy; but it also needs a long processing latency, which means the number of frames between two YOLOv3 detected frames is large.

Tracking Accuracy. Between two frames that are processed by the object detector, we track the objects in these frames based on *Lucas-Kanade* method. To study the tracking accuracy, we use YOLOv3-608 to detect the objects in one frame, and then run the tracking algorithm to track these objects in the following frames. We do such an experiment of detection and tracking 10 times on two different videos respectively. Figure 2 shows the average tracking performance of those two videos. The content of Video1 changes faster than that of Video2. For both videos, the initial tracking accuracy is high, as it is based on the detection results of YOLOv3-608. However, the tracking accuracy drops below 0.5 after 9 frames for video1 and 27 frames for video2. The tracking accuracy of video1 degrades faster. Because it is hard to accurately estimate the positions of detected objects in the following fast-changing frames and more new objects also appear in these frames.

Observation 3: The tracking accuracy drops fast for the videos in which the content varies fast. Before the tracking accuracy drops to a low level (e.g., F1 score is below 0.5), we need to accomplish another object detection to calibrate the tracking performance to the initial tracking accuracy level. If the video content varies fast, a small frame size may be used in YOLOv3 detection, so that its processing time of one frame is short. Therefore, the YOLOv3 frame size determines both the initial tracking accuracy and the number of frames the tracking algorithm needs to track before the next calibration. It needs to be carefully set according to the online changing rate of video content.

Tracking Latency. Table II shows the processing time of tracking one frame. It takes 40 ms on average to extract good feature points for tracking. We do not need to extract good features for each frame, but only the DNN detected frames. It takes 7 ms to 20 ms on average to track all the feature points from one frame to another. The latency depends on the number of objects and good features in the frame. Finally, for each frame, it takes 50 ms to find a good feature for each object and overlay the bounding boxes on top of all objects and display the image on the screen.

Observation 4: The tracking latency of one frame is larger than the frame interval of normal camera video streams. To provide real-time video processing, we have to skip some frames from tracking process to catch up the frame capture speed of mobile cameras.

IV. DESIGN OF ADAVP

In this section, we describe the design of AdaVP. After a brief overview of AdaVP's architecture, we will introduce three key components of AdaVP.

A. System Overview

Figure 3 shows the system architecture of AdaVP. The frames taken by a mobile camera are first stored in a frame

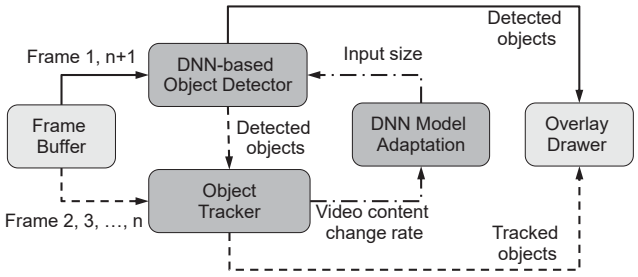


Fig. 3. Architecture of AdaVP. Each frame is either processed by the object detector or by the object tracker. The object tracker takes the objects detected by the object detector as input. The object detector uses the results of the object tracker to calculate the video content change rate and further adapt its DNN model setting. Finally, the processed frame will be passed to the overlay drawer module to draw the bounding boxes and display the frame on screen.

buffer. The task of AdaVP is to process the frames in the buffer one by one in real time, so that there is no frame accumulated in the frame buffer. After AdaVP's processing, the objects of each frame will be identified. The results will be passed to the overlay drawer module to draw the bounding boxes. The overlaid frames are the views with overlaid augmented objects, which will be finally displayed on the mobile screen.

AdaVP is mainly composed of three components, i.e., DNN-based object detector, object tracker and DNN model adaptation module. The object detector and the object tracker form a *parallel detection and tracking pipeline*. A frame in the frame buffer is either processed by the object detector or by the object tracker. When the object detector is processing a new frame, the object tracker handles all the accumulated frames before that frame in the buffer. In addition, the object detector uses the results of the object tracker to *calculate the video content changing rate and adapts its DNN model setting*.

With the parallel detection and tracking pipeline, when object detector accomplishes the processing of one frame, object tracker takes the objects detected by detector as input to track these objects in the following frames. At the same time, the object detector fetches the newest frame from the buffer and starts detecting the objects in that frame. To support real-time video processing, when object detector finishes the processing of the newly fetched frame, object tracker needs to accomplish the processing of all frames before that frame (details can be found in IV-B).

To enable an efficient pipeline of parallel detection and tracking, we will adapt DNN model setting according to the changing rate of video content. Our DNN model adaptation module takes the intermediate results of object tracker to calculate the changing rate of video content. Based on *Observation 3*, if the video content changing rate is high, the tracking accuracy degrades sharply. A new model adaptation algorithm is developed to adapt the DNN model setting according to the video content changing rate.

B. Parallel detection and tracking pipeline

Figure 4 illustrates the workflow of our proposed *parallel detection and tracking pipeline* and a baseline system. The baseline system in Figure 4 is a simple implementation of the latest mobile video processing work, MARLIN [7]. To avoid offloading, the baseline system executes object detector and object tracker on mobile devices sequentially. At the beginning, the object detector fetches $frame\ m_0$ from frame buffer and detects the objects in this frame. It takes hundreds of milliseconds for the DNN model to finish processing of one frame. During this process, k frames have been accumulated in the buffer. When the object detector completes its detection, it delivers the detection results to the object tracker. The latter will track the objects in the following j frames (from $frame\ m_0+1$ to $frame\ m_1-1$) to catch up. When the system detects significant scene changes, object detector will be triggered to detect a new frame, the $frame\ m_1$. In this system, to catch up camera feeds, j must be larger than k. If the system is required to achieve real-time processing, the object tracking time should be larger than the object detection time. However, if the system detects significant scene changes before object tracker catches up, it will trigger DNN object detector immediately and the system latency will be accumulated. The accumulated latency will further hurt accuracy.

To reduce the accumulated latency and improve processing accuracy, we propose MPDT (Mobile Parallel Detection and Tracking). It is a component in AdaVP that executes object detector and object tracker in parallel. For MPDT, after the object detector delivers the detection result of $frame\ n_0$ to object tracker, k frames have been accumulated in the buffer. Object detector fetches the newest frame from the buffer to do the object detection task, which is $frame\ n_1$. At the same time, object tracker start tracking the objects in the frames from $frame\ n_0+1$ to $frame\ n_1-1$ using the detection results (object locations, object labels) of $frame\ n_0$ received from object detector. While the object detector is detecting $frame\ n_1$, object tracker is tracking $frame\ n_0+1$ to $frame\ n_1-1$. In this way, the object detector and object tracker keep working and working in parallel, the object detection time and object tracking time are basically same.

To implement MPDT, we use `multithreading` techniques. There are two technical problems. The first one is to prevent multiple threads to read/write the shared data at the same time. The second is the communication among multiple threads. We use three threads in our system, object detector thread, object tracker thread and main thread. The main thread is responsible for scheduling the other two threads and displaying images. The shared data among these threads are frame buffer, detected results from object detector and display image. The shared data cannot be operated by multiple threads at the same time, we use `lock` to prevent data from being operated at the same time. The threads also need to be notified when they can operate the shared data. We use `event` to do the communication among different threads. To synchronize the object detector and object tracker threads, once the object detector fetched a new frame, the object tracker will cancel its tracking tasks after finishing the current task if there is. But it does not display the current task. This is because the current task object tracker is doing is the prior frame to the frame is fetched by object detector, if displays it, the displayed results will go backwards.

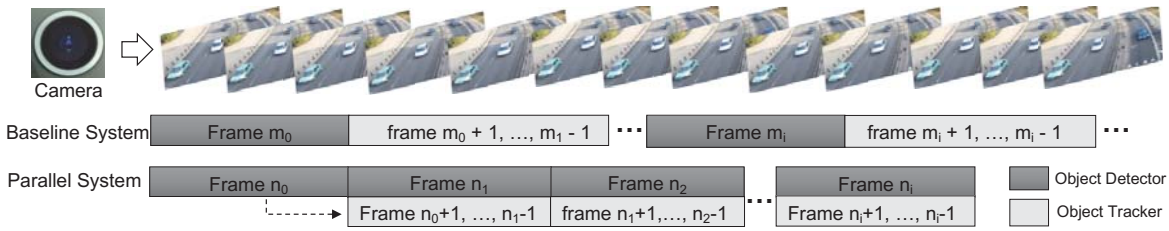


Fig. 4. Two different video processing systems, i.e., a baseline system and the pipeline of parallel detection and tracking.

C. Object Tracker

MPDT needs to continuously track objects (detected by object detector) across the frames in between two DNN executions. The number of frames to be handled could be large, e.g., 20 frames for YOLOv3 with a frame size of 608x608 (YOLOv3-608). To maintain the detection accuracy and achieve real-time performance, the object tracker needs to be accurate and lightweight. Typically, there are two steps in the object tracker, *Feature Extraction* and *Object Tracking*.

Feature Extraction. We first extract some features in the last DNN detected frame and then track these features in the following accumulated frames. By tracking these features, we can estimate the moving speed of the objects in the frame. For a frame, its features can be extracted using a feature detector and descriptor such as SIFT (Scale-invariant feature transform), SURF (Speeded-Up Robust Features), good features to track, FAST (Features from Accelerated Segment Test) and ORB (Oriented FAST and Rotated BRIEF) [16], [38], [39]. Generally, the more accurate a feature descriptor is, the longer processing latency it needs. After evaluating the overall performance of all the above feature descriptors, we use the standard *good features to track* [16] method to extract the good feature points in the DNN detected frames.

Object Tracking. We then track the extracted good features in the following frames between two DNN detected frames using the Optical Flow Based Object Tracking method. Optical flow captures the pattern of apparent motion of objects, surfaces, and edges among frames. We use the well-known Lucas-Kanade [17] optical flow method to track good features in the following frames. Because only the feature points inside the bounding boxes detected by YOLOv3 are useful for the system to track objects. We only detect and extract feature points inside the bounding boxes. The feature points extracted from the same object should have similar moving vector (distance and direction) between frames, so the moving vector of these feature points can describe the moving vector of the object.

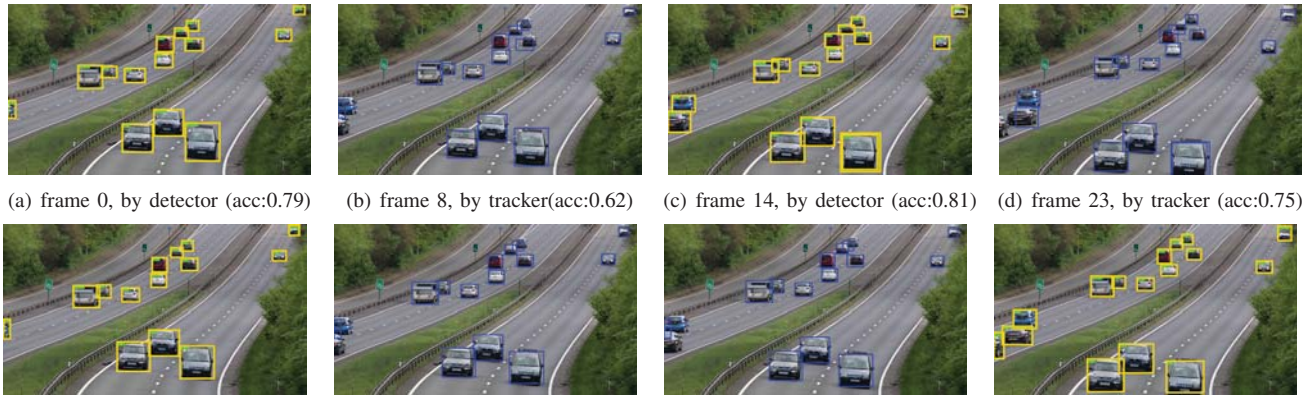
There are usually multiple objects in one video frame. Different objects may have different moving vectors. To achieve high tracking performance, instead of calculating an average moving vector of all objects, we calculate the moving vector for each object. As a result, the tracking time per frame is related to the numbers of objects in this frame, i.e., the more objects a frame has, the longer time it takes to find good features and calculate the moving vector for each object.

Tracking Frame Selection: The intermediate frames between two DNN detected frames will be fetched from the frame buffer into a temporary buffer. In this work, we call the time of one DNN detection execution as a detection or tracking cycle. From the motivation experiments and the *observation 4*, we know that it is not practical to track all the frames in the temporary buffer, as the feature tracking and overlay drawing of one frame take more than 33ms (if the frame rate of the video is 30 FPS). We can leverage the temporal correlation between adjacent frames in a video, i.e., adjacent frames usually contain similar content [1], [4] to select a certain number of frames at regular intervals in the temporary buffer to do object tracking.

To decide the number of frames to track, we need to know the processing time of the feature tracking per frame. However, as explained above, the tracking time per frame varies according to the number of objects in one frame. MPDT uses the prior tracking experience to find this number. We assume the number of objects in two adjacent tracking cycles does not change much. MPDT counts the number of frames h_{t-1} were tracked and the total number of frames f_{t-1} in the buffer during the last cycle and calculate the tracking frame fraction $p = \frac{h_{t-1}}{f_{t-1}}$. Then it gets the total number of frames f_t in the buffer during the current cycle and estimates how many frames can be tracked during this cycle $h_t = p * f_t$. After getting the predicted h_t, the system knows how to select frames at regular intervals. The frames that are not selected by the tracker use the location and label of objects from the previous tracked or detected frame [33].

The object tracker uses the last DNN detected frame as the reference frame, including the labels and bounding box positions (locations) of all the objects in the reference frame. It will process the frames selected by the tracking frame selection method. It outputs the labels and locations of the objects in these tracked frames. A label is a class as which the DNN identified the object (e.g., person or dog). A bounding box represents the position of an object in the frame, which is represented by a 4-tuple vector (left, top, width, height).

Workflow of Our Object Tracker. Putting all of these components together, the workflow of our object tracker is as follows: 1) Receive the detection results (labels and bounding box positions) of $frame\ n_0$ from object detector and fetch the $frame\ n_0 + i$ selected by tracking frame selection method. 2) Extracts all the good feature points inside all the bounding boxes in $frame\ n_0$ using good features to track method. 3) Finds one feature point for each bounding box. 4) Use Lucas-Kanade method to estimate the optical flow from $frame\ n_0$ to $frame\ n_0 + i$. 5) Calculate moving vector for each good



(a) frame 0, by detector (acc:0.79) (b) frame 8, by tracker(acc:0.62) (c) frame 14, by detector (acc:0.81) (d) frame 23, by tracker (acc:0.75)

(e) frame 0, by detector (acc:1) (f) frame 8, by tracker (acc:0.83) (g) frame 14, by tracker (acc:0.73) (h) frame 23, by detector (acc:1)

Fig. 5. Frame accuracy of MPDT using two different model settings (First row: MPDT-YOLOv3-320; second row: MPDT-YOLOv3-608).

feature between $frame\ n_0$ to $frame\ n_0 + i$ and uses this value to shift the old bounding box positions to the current positions in $frame\ n_0 + i$. 6) Select a new frame in frame buffer to track.

D. Model Adaptation

From the *observation 1-3* obtained in Section III, we know that different frame sizes of YOLOv3 result in different detection accuracy and tracking performance. In this section, we adjust frame size to achieve better accuracy of the proposed parallel object detection and tracking pipeline (MPDT). After some preliminary results of MPDT, we introduce the change rate detection of video content and our adaptation algorithm.

1) Preliminary experiment results: Figure 5 depicts an example of the detection accuracy of MPDT under two different DNN model settings (MPDT-YOLOv3-608 and MPDT-YOLOv3-320) on the same video clip. We show the results of 4 frames each.

- **Frame 0.** Both settings perform object detection for Frame 0. The detection accuracy of MPDT-YOLOv3-608 is 1, and the accuracy of MPDT-YOLOv3-320 is 0.79. Because the latter has 3 false positive cases, i.e., it identifies 2 cars as trucks and 1 truck as car.
- **Frame 8.** Both settings execute object tracking by taking the detection results of Frame 0 as reference. The tracking accuracy of MPDT-YOLOv3-320 drops to 0.62; whereas the accuracy of MPDT-YOLOv3-608 is still 0.83, as the latter has an initial detection accuracy of 1.
- **Frame 14.** MPDT-YOLOv3-320 fetched this frame to do detection and its accuracy improves to 0.81. MPDT-YOLOv3-608 is still doing tracking using the detection results from frame 0, and its accuracy drops to 0.73, because new vehicles appear.
- **Frame 23.** MPDT-YOLOv3-320 is performing tracking and its accuracy drops to 0.75. MPDT-YOLOv3-608 fetched this frame to do detection and its accuracy is calibrated to 1.

From the above example, we see that MPDT-YOLOv3-608 has a high initial detection accuracy, but its long detection latency results in a large number of frames to be tracked (i.e., low tracking accuracy for the last few frames). On the other hand, MPDT-YOLOv3-320 has a relatively lower initial detection accuracy, but it calibrates its tracking accuracy more frequently by performing light-weight object detection. For some frames, MPDT-YOLOv3-320 has a higher accuracy; but for the others, MPDT-YOLOv3-608's performance is better.

From the experiment results in Figure 2, we know that when the video content changes slowly, the tracking accuracy degrades slowly. In this situation, MPDT-YOLOv3-608 should be used to have a high initial detection accuracy. On the other hand, when the video scenes change fast, MPDT-YOLOv3-320 should be used, as the tracking accuracy drops fast and needs to be calibrated more frequently. Therefore, we propose to dynamically switch the frame size of YOLOv3 at runtime according to the video content changing rate to achieve the best performance all the time. We first design a metric to measure the changing rate of video content. Based on that, we develop an adaptation algorithm to adjust the frame size at runtime.

2) Video Content Changing Rate: The metric to evaluate the video content changing rate must be lightweight so that its computation will not impact the tracking and detection operation of the real-time system. We propose to leverage the intermediate result from tracking to measure the changing rate of video content. By doing so, it almost adds no extra computation. We use the average motion velocity of all good features extracted from two adjacent frames (i and $i+1$) as the changing rate of video content. It is calculated as follows.

$$v_{i,i+1} = \frac{\left|\sum_{k=1}^{M} f_i^k(x,y) - f_j^k(x,y)\right|}{M * (j-i)} \tag{3}$$

where $f_i^k(x,y)$ and $f_j^k(x,y)$ are the pixel positions of the kth feature in the ith frame and the jth frame respectively. We have M features extracted from these frames. Since we skip some frames during object tracking, i.e., $j - i \neq 1$, we normalize the motion velocity of features to the velocity between two adjacent frames by dividing the results by the number of frames between the ith and jth frame.

We use the pixel coordinates of features to calculate the motion velocity. For different cameras with different capture distance and angles, our motion velocity metric can measure how fast the objects move in the pixel coordinates of a frame.

A high motion velocity means the video content is changing fast, i.e., the existing objects moves out of the frame fast and new objects may appear frequently.

3) DNN Model Setting Adaptation: We design a lightweight DNN model adaptation module to find the relationship between the motion velocity and different frame sizes (4 settings in our current implementation, i.e., 320x320, 416x416, 512x512 and 608x608). At runtime, the model adaptation module decides whether to switch to another frame size (model setting). Our adaptation scheme also works for selecting the right model not just model setting at runtime, as long as these DNN models have complementary detection accuracy and latency [33]. However, in order to use multiple DNN models simultaneously, we must pre-load these models, which requires large memory cost and cannot be supported by mobile devices. Therefore, we focus on switching to different model settings in this work. Different model settings have similar performance as different DNN models.

Generally speaking, a high motion velocity indicates high video content changing rate and in turn sharp degradation of object tracking; as a result, a small frame size is necessary to keep the detection latency small and calibrate the object tracking more frequently. We assume the relationship between the motion velocity and 4 frame sizes is linear, i.e., high velocity requires small frame size. To quantify the relationship, we need to find 3 velocity thresholds, i.e., v_1, v_2 and v_3. If $v \leq v_1$, the frame size 608x608 will be used. If $v_1 < v \leq v_2$, $v_2 < v \leq v_3$ or $v_3 \geq v$, the frame size 512x512, 416x416 or 320x320 will be used respectively.

It is a typical classification problem to find the three velocity thresholds. We first generate a large amount of training data and then learn the training data to find the thresholds. In our current implementation, 32 videos, corresponding to 105205 frames, are used for finding the threshold. The videos include 14 scenarios, including surveillance videos at highway, intersection, city street, train station, bus station, and residential area. Car-mounted videos driving on highway or around downtown. Mobile camera videos about airplanes, boat, animals in the wild, racetrack, meeting room and skating rink.

To collect training data, we divide each video into a sequence of chunks. Each chunk is 1 second. We run MPDT to process each video with 4 frame sizes independently. We calculate an average detection accuracy and an average motion velocity every second. For each video, we obtain 4 sequences of pairs (detection accuracy and motion velocity). For each chunk, by comparing the detection accuracy from 4 frame sizes, we can find the frame size that provides the highest detection accuracy. Finally, we generate a training dataset that is composed of a large number of vectors (motion velocity and the best frame size). The best frame size is the label of the corresponding motion velocity. We then use the motion velocities and their labels to train a classification model to find the three thresholds under a certain frame size.

To use the adaptation module, we use the motion velocity measured in the current detection cycle to decide which frame size of YOLOv3 will be used for the next cycle. The motion velocity measured in the current detection cycle is measured based on the current frame size. In our experiments, we find that for the same chunk of the video, the motion velocity measured under different frame size settings are similar, but not exactly the same. It may be because the object bounding boxes detected by 4 frame sizes are not exactly the same. The feature points are extracted within the bounding boxes, thus the extracted feature points are not exactly the same. To solve this problem, we find the three thresholds for each frame size. For online adaptation, we use the correct thresholds based on the frame size of current detection cycle. After training, the DNN model setting adaptation module can be used to guide the system to adapt to the change of video content. At runtime, this module takes the motion velocity and current DNN model setting as input and outputs the next DNN model setting.

It only takes 8.49×10^{-2} ms to extract the motion features from object tracker and 1.89×10^{-2} ms to switch to a different DNN model setting. Compared to other components, our motion feature extraction time and DNN setting switching time is negligible, but we can improve the system performance significantly by using these two components. Since we change the setting of DNN model at runtime, the latency of AdaVP is not fixed. It varies from 200 ms to 470 ms (one DNN detection time subtract one frame time). This latency is inevitable in DNN-based video processing system.

V. Implementation

Framework: We use PyTorch [40] as the deep learning framework, because it supports dynamic computational graph building. As we know, a computational graph is normally built to represent some complex computation in DNN models. PyTorch can build and compute the computational graph at the same time, which is different from Tensorflow or Darknet. Tensorflow or Darknet builds the computational graph in a static way before computations start. Because our system intends to be adaptive to the video content, and it will switch the DNN model setting dynamically during video processing according to the change of video content. The computational graph will be built according to the DNN model setting of the model. When the setting changes, the computational graph changes as well. So static computation graph building cannot meet the system requirements.

System implementation: We use multithreaded programming to implement our system. *Object detector* and *Object tracker* are implemented as two threads. `CUDA` is set up for the object detector thread, so it is able to use GPU resource at runtime. The *Frame Buffer* is implemented by using `Queue` data structure. Both object detector and object tracker have access to it. For object detector, we get the `weights` file and `cfg` file from official darknet [2] website. For object tracker, we use the `good features to track` function provided in OpenCV [41] to detect and extract good feature points. Because only the feature points inside the bounding boxes detected by YOLOv3 are useful to track objects, we use `mask` for the detected bounding boxes and only detect and extract feature points inside masks. Compared to extracting

the features across the whole image, only extracting features within masks saves computation and energy. We use the `calcOpticalFlowPyrLK` function to track these feature points in the following frames. To reduce latency, for each bounding box, we find one point inside it and calculate the moving vector of this point to shift the bounding box.

Energy consumption: We use the shell file `Power_Monitor.sh` to get the power of GPU, CPU, DDR and SoC of TX2. We record the power when the TX2 is running AdaVP or other baseline systems and the power when it does not run anything. The difference between these two records is the power of AdaVP or other baseline systems. Then we can calculate the energy consumption by multiplying power and running time.

Data storage: We save some data at runtime, including frame number, object class labels and object locations, motions of video from object detector and object tracker. We use these data to train our DNN model setting adaptation module and compute the evaluation accuracy offline. Saving data at runtime adds extra computational overhead to our video processing system, which impacts the system performance slightly. The real performance of AdaVP should be better than that in our evaluation.

VI. Evaluation

In this section, we conduct a variety of experiments to evaluate the performance of AdaVP comprehensively.

A. Experiment Setting

Dataset. To evaluate our system completely, we use the ImageNet and Videezy video datasets [18], [19] and also collect some real-world public videos from YouTube [20]. Our dataset includes 45 indoor or outdoor videos that are recorded by static, moving or car-mounted cameras. These videos contain various scenarios with multiple objects (e.g., cars, trucks, trains, persons, airplanes, animals). Compared to live camera feed, these videos are much more challenging. Most of the videos are 30 FPS at a resolution of 1280 x 720 pixels. The length of each video ranges from 15 seconds to 34 minutes. These videos contain 246418 frames in total. We use 105205 frames to train the adaptation module which explores the relationship between DNN model setting and motion of video content and 141213 frames to evaluate the system performance. We conduct all the evaluation experiments on the entire testing dataset.

Baselines. We compare the performance of AdaVP with three baseline solutions.

- **MPDT.** MPDT uses fixed DNN model setting for all types of videos all the time. We use 4 settings (320x320, 416x416, 512x512 and 608x608) to do the experiments. These four settings refer to YOLOv3-320, YOLOv3-416, YOLOv3-512 and YOLOv3-608. AdaVP switches the model setting of DNN model at runtime according to the changing rate of video content.
- **MARLIN.** MARLIN [7] is the latest work that executes object detector and tracker sequentially, without parallel computing scheme. Object detector will be triggered when significant changes detected by video content changing detector. We implement the idea of MARLIN in our framework by the same DNN detector, object tracker and video content change detector as AdaVP. For video content change detector, we conduct a set of experiments to find a motion velocity threshold that provides the best detection accuracy for MARLIN.
- **Without Tracking.** In this scheme, there is no object tracker to do tracking. We only use the DNN model to do detection. The DNN model is always going to fetch the current video frame. For the skipped frames between two DNN executions. We use the detection result from the previous frame [33].

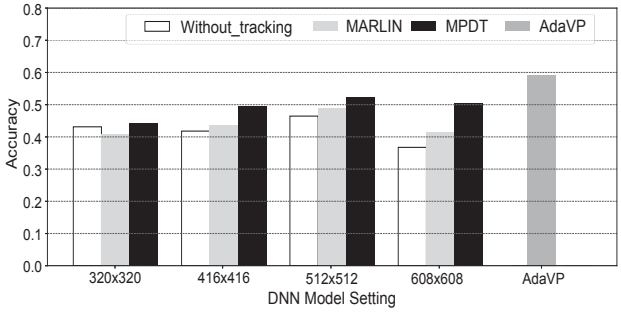


Fig. 6. Performance comparison of AdaVP and baseline systems.

Detection Accuracy. We use F1 score to measure the detection or tracking accuracy of a single frame. We use the percentage of frames with certain F1 score threshold to measure the accuracy of a video [1]. The F1 score threshold is set as 0.7 as default. For example, if the accuracy of a video is 0.6, it means there are 60% frames with F1 score higher than 0.7. For the video set, we use the average percentage per video as accuracy to demonstrate the evaluation.

B. Overall Performance

Figure 6 depicts the performance of AdaVP and the baseline systems on the whole testing dataset. From the experiments, we find AdaVP increases 20.4% to 43.9% accuracy compared to MARLIN and increases 13.4% to 34.1% accuracy compared to MPDT under different DNN model settings. The experimental results show that YOLOv3-512-based system achieves the best performance compared to other model settings for both MPDT and MARLIN.

Number of cycles per DNN model setting switching. Figure 7 presents the cumulative probability of number of cycles per DNN model setting switching. It is near 50% the system switches model setting after one cycle. Since AdaVP switches the DNN model setting at runtime, the duration of one cycle is not fixed, the number of frames within one cycle is not fixed (e.g., 10 to 25 frames per cycle). For 90% cases, the number of cycles per switching is below 20. There are 5% cases that AdaVP switches to another model setting after 40 cycles. For these cases, the video content changing detector does not detect significant change of the video content. Thus, AdaVP keeps using the same DNN model setting.

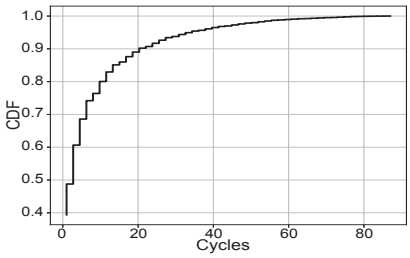


Fig. 7. Cumulative probability of number of cycles per DNN model setting switching

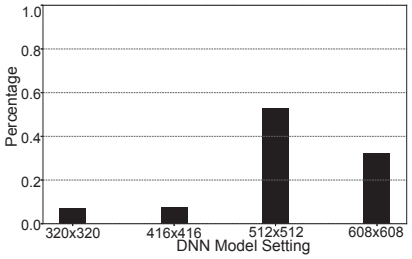


Fig. 8. Trigger percentage of every DNN model setting from AdaVP

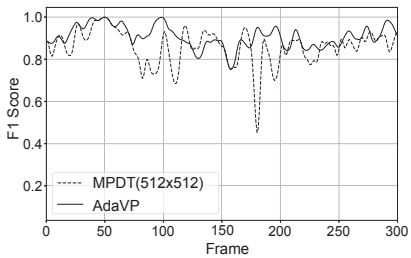


Fig. 9. Frame accuracy comparison of AdaVP and MPDT-YOLOv3-512 (the best baseline)

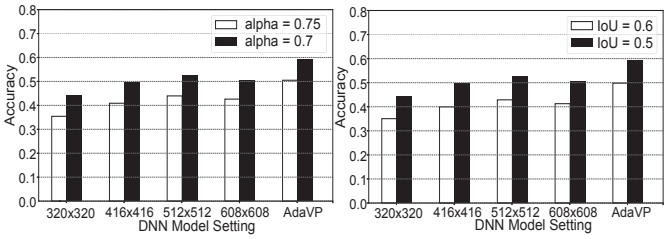


Fig. 10. Performance comparison under different thresholds of F1 score

Fig. 11. Performance comparison under different IoU value

Usage of different model settings. Figure 8 shows the trigger percentage of different DNN model settings of AdaVP. From the experiment results, we know that all of the model settings have been triggered at runtime according to the video content changing rate. The frame sizes of 512x512 and 608x608 are mostly being used. The usage of the other two model settings is around 10%.

Frame Accuracy Comparison. Figure 9 demonstrates the accuracy of AdaVP in frame level. We use MPDT by YOLOv3-512 as a comparison since it is better than other model settings we used. Most of time, AdaVP achieves higher accuracy than MPDT-YOLOv3-512. Around frame 180, the detection accuracy of MPDT-YOLOv3-512 drops heavily. But the accuracy of AdaVP is still high, this is because the DNN model adaptation module decides not to use YOLOv3-512 for this cycle according to detected change of video content. From the long run, AdaVP can combine the benefits of different model settings and achieve higher accuracy.

C. Performance Gain of Parallel Detection and Tracking

From Figure 6, we also know MPDT outperforms MARLIN and without tracking scheme under each model setting. MPDT achieves 7.1% to 21.95% higher accuracy than MARLIN and 2.3% to 37.3% higher accuracy than without tracking scheme under different model settings. This is because MPDT keeps doing object detection and tracking concurrently and calibrates the object tracking by running object detector at the maximum frequency. However, MARLIN does object detection and tracking sequentially, which is inefficient for complex and challenging video scenes.

D. Parameter Setting in AdaVP

F1 Score Threshold. Figure 10 presents the performance under different accuracy threshold α at 30 FPS. As introduced in the experiment setting, we use the percentage of frames with certain F1 score threshold as the accuracy metric for a video. When we change the threshold α from 0.7 to 0.75, the accuracy is stricter. But AdaVP still outperforms the baseline system MPDT. When α is set as 0.75, AdaVP increases the accuracy of MPDT by 14.9% to 42.6%. The performance gain is even larger than the case when α is 0.7. From these two accuracy thresholds, we know AdaVP has more frames with higher accuracy than the baseline system.

IoU Threshold. We also compare the accuracy with different IoU values under 4 different model settings at 30 FPS. The widely-used IoU is 0.5 in the computer vision community. Here, we use a stricter IoU threshold, which is 0.6 for comparison. Higher IoU value means true positives are identified stricter. So, the F1 score per frame decreases and the overall accuracy decreases. Figure 11 reveals that AdaVP consistently outperforms the baseline when IoU is 0.6. It increases the accuracy by 16.1% to 41.8% compared to MPDT. The performance gain is even higher when IoU is 0.6, compared with the default 0.5.

E. Energy Consumption and Accuracy

Table III shows the energy consumption of different hardware components (GPU, CPU, SoC and DDR) from different video processing methods. We choose MPDT and MARLIN based on both YOLOV3-512 and YOLOv3-320, since they have the best real-time performance under the setting of 512x512, and they are most energy-efficient under 320x320. We choose YOLOv3-tiny-320 because it is almost real-time on TX2 without tracking or skipping any frames. For comparison, we also execute YOLOv3-320 and YOLOv3-608 continuously without frame skipping. If we do not consider latency and execute DNN for every frame, YOLOv3-320 is the most energy-efficient, and YOLOv3-608 can provide the highest accuracy for each frame among the model settings we used.

From Table III, we found AdaVP increases by 20.4% accuracy compared to MARLIN-YOLOv3-512 at the cost of 14.9% more energy. This is because AdaVP targets at improving accuracy, but MARLIN focuses on energy efficiency. They have different design goals. We also found compared to the best baseline MPDT-YOLOv3-512, AdaVP increases the accuracy by 13.4% with 2.3% less energy consumption. AdaVP even achieves 3.5% more accuracy with 7.95x less energy compared to YOLOv3-320. We do not consider the 7x latency from YOLOv3-320 into the accuracy calculation. If we take the latency into consideration, the accuracy of YOLOv3-320 even much worse. Though YOLOv3-608 without frame

TABLE III
COMPARISON OF ENERGY CONSUMPTION AND ACCURACY FROM DIFFERENT METHODS

	AdaVP	MPDT-YOLOv3-320	MARLIN-YOLOv3-320	YOLOv3-tiny-320 (1.8x latency)	YOLOv3-320 (7x latency)	MPDT-YOLOv3-512	MARLIN-YOLOv3-512	YOLOv3-608 (10.3x latency)
GPU ($w \cdot h$)	3.65	2.85	2.22	4.09	36.25	3.53	3.03	68.84
CPU ($w \cdot h$)	1.88	2.08	1.25	3.14	6.64	2.14	1.84	6.24
SoC ($w \cdot h$)	0.39	0.34	0.24	0.53	3.60	0.40	0.32	6.62
DDR ($w \cdot h$)	1.34	1.18	0.82	1.66	11.25	1.36	1.13	20.17
Total ($w \cdot h$)	7.26	6.45	4.53	9.42	57.74	7.43	6.32	101.87
Accuracy	0.59	0.44	0.41	0.07	0.57	0.52	0.48	0.89

skipping achieves the highest accuracy, it has 10.3x latency and consumes 14x more energy than AdaVP.

VII. CONCLUSION

This paper presents a continuous and real-time video processing system that incorporates object detection and object tracking on mobile devices without offloading. We develop MPDT, a parallel detection and tracking pipeline that executes object detection and tracking concurrently. On top of that, we design a DNN model setting adaptation module. This module switches the DNN model settings at runtime according to the detected video content changes. Experiments show that AdaVP outperforms the state-of-the-art methods.

ACKNOWLEDGMENT

We gratefully acknowledge the support of NVIDIA Corporation with the donation of Jetson TX2 used for this research.

REFERENCES

[1] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan, "Glimpse: Continuous, real-time object recognition on mobile devices," in *ACM SenSys*, 2015.
[2] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.
[3] L. N. Huynh, Y. Lee, and R. K. Balan, "Deepmon: Mobile gpu-based deep learning framework for continuous vision applications," in *ACM MobiSys*, 2017.
[4] M. Xu, M. Zhu, Y. Liu, F. X. Lin, and X. Liu, "Deepcache: Principled cache for mobile deep vision," in *ACM MobiCom*, 2018.
[5] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen, "Deepdecision: A mobile deep learning framework for edge video analytics," in *IEEE INFOCOM*, 2018.
[6] L. Liu, H. Li, and M. Gruteser, "Edge assisted real-time object detection for mobile augmented reality," in *ACM MobiCom*, 2019.
[7] K. Apicharttrisorn, X. Ran, J. Chen, S. V. Krishnamurthy, and A. K. Roy-Chowdhury, "Frugal following: Power thrifty object detection and tracking for mobile augmented reality," in *ACM SenSys*, 2019.
[8] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *Computer*, 2017.
[9] D. Chatzopoulos, C. Bermejo, Z. Huang, and P. Hui, "Mobile augmented reality survey: From where we are to where we go," *Ieee Access*, vol. 5, pp. 6917–6950, 2017.
[10] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *ECCV*, 2016.
[11] J. Wu, J. B. Tenenbaum, and P. Kohli, "Neural scene de-rendering," in *IEEE CVPR*, 2017.
[12] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints," in *ACM MobiSys*, 2016.
[13] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.
[14] B. Fang, X. Zeng, and M. Zhang, "Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision," in *ACM MobiCom*, 2018.
[15] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.
[16] J. Shi and C. Tomasi, "Good features to track," Cornell University, Tech. Rep., 1993.
[17] B. D. Lucas, T. Kanade *et al.*, "An iterative image registration technique with an application to stereo vision," in *IJCAI*, 1981.
[18] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *IJCV*, 2015.
[19] "Videezy," https://www.videezy.com/free-video/traffic//.
[20] "Youtube," https://www.youtube.com//.
[21] J. Gu, J. Wang, Z. Yu, and K. Shen, "Walls have ears: Traffic-based side-channel attack in video streaming," in *IEEE INFOCOM*, 2018.
[22] L. Cheng and J. Wang, "Vitrack: Efficient tracking on the edge for commodity video surveillance systems," in *IEEE INFOCOM*, 2018.
[23] Q. Liu, S. Huang, J. Opadere, and T. Han, "An edge network orchestrator for mobile augmented reality," in *IEEE INFOCOM*, 2018.
[24] X. Ding, W. Du, and A. Cerpa, "Octopus: Deep reinforcement learning for holistic smart building control," in *Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, 2019, pp. 326–335.
[25] Z. Shen, K. Yang, W. Du, X. Zhao, and J. Zou, "Deepapp: a deep reinforcement learning framework for mobile application usage prediction," in *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, 2019, pp. 153–165.
[26] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *IEEE/ACM IPSN*, 2016.
[27] A. Mathur, N. D. Lane, S. Bhattacharya, A. Boran, C. Forlivesi, and F. Kawsar, "Deepeye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware," in *ACM MobiSys*, 2017.
[28] S. Naderiparizi, P. Zhang, M. Philipose, B. Priyantha, J. Liu, and D. Ganesan, "Glimpse: A programmable early-discard camera architecture for continuous mobile vision," in *ACM SenSys*, 2017.
[29] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du, "On-demand deep model compression for mobile devices: A usage-driven model selection framework," in *ACM MobiSys*, 2018.
[30] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *NeurIPS*, 2015.
[31] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016.
[32] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *AAAI*, 2017.
[33] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica, "Chameleon: scalable adaptation of video analytics," in *ACM SIGCOMM*, 2018.
[34] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *ICCV*, 2017.
[35] P. Baranyi, "Nvidia jetpack," https://developer.nvidia.com/embedded/jetpack.
[36] "Nvidia tensorrt," https://developer.nvidia.com/tensorrt.
[37] P. Baranyi, "Nvidia video codec sdk," https://developer.nvidia.com/nvidia-video-codec-sdk.
[38] P. Guo and W. Hu, "Potluck: Cross-application approximate deduplication for computation-intensive mobile applications," in *ACM ASPLOS*, 2018.
[39] L.-Y. Duan, V. Chandrasekhar, S. Wang, Y. Lou, J. Lin, Y. Bai, T. Huang, A. C. Kot, and W. Gao, "Compact descriptors for video analysis: The emerging mpeg standard," *IEEE MultiMedia*, 2018.
[40] "Pytorch," https://pytorch.org/.
[41] "Opencv," https://https://opencv.org//.