FlexPatch: Fast and Accurate Object Detection for On-device High-Resolution Live Video Analytics

Kichang Yang, Juheon Yi, Kyungjin Lee, Youngki Lee
Seoul National University
{kichang96, johnyi0606, jin11542, youngkilee}@snu.ac.kr

***Abstract*—We present FlexPatch, a novel mobile system to enable accurate and real-time object detection over high-resolution video streams. A widely-used approach for real-time video analysis is detection-based tracking (DBT), i.e., running the heavy-but-accurate detector every few frames and applying a lightweight tracker for in-between frames. However, the approach is limited for real-time processing of high-resolution videos in that i) a lightweight tracker fails to handle occlusion, object appearance changes, and occurrences of new objects, and ii) the detection results do not effectively offset tracking errors due to the high detection latency. We propose *tracking-aware patching* technique to address such limitations of the DBT frameworks. It effectively identifies a set of subareas where the tracker likely fails and tightly packs them into a small-sized rectangular area where the detection can be efficiently performed at low latency. This prevents the accumulation of tracking errors and offsets the tracking errors with frequent fresh detection results. Our extensive evaluation shows that FlexPatch not only enables real-time and power-efficient analysis of high-resolution frames on mobile devices but also improves the overall accuracy by 146% compared to baseline DBT frameworks.**

***Index Terms*—live video analytics, On-device AI, high-resolution video, object detection, object tracking**

I. Introduction

On-device live video analytics enables various useful services, including AR person identification [1], visual support for the blind [2], and drone surveillance [3]. Especially, it becomes increasingly crucial to accurately track distant small objects from high-resolution videos (e.g., 1080p). For instance, an augmented reality application for pedestrian safety should accurately detect and track high-speed vehicles from a long distance and raise alerts in advance. On-device object tracking systems are compelling over cloud-aided systems, considering the large data size of high-resolution videos and bandwidth fluctuation in outdoor use cases, along with privacy concerns.

The key challenge for on-device high-resolution video analytics lies in the high object detection latency. For example, it takes $\approx$1,029 ms to process a 1080p frame with Tiny YOLO-v4 [4], a widely-used lightweight object detector, on a high-end smartphone (i.e., LG V50 with Qualcomm Adreno 640 GPU). Such high latency makes the detection results stale and inaccurate, especially when objects are small and move fast (example shown in Fig. 3). To overcome the challenge, recent works [5]–[8] adopt the *Detection-Based-Tracking* (DBT) approach. They periodically run detectors every N frames while processing in-between frames using a lightweight object tracker (based on optical flow or motion vectors). Despite its effectiveness, we identify that prior techniques are still critically limited for high-resolution videos (e.g., 1080p) with distant objects. The primary sources of errors are two folds: i) object tracker frequently fails due to occlusion, appearance changes, or new appearance of target objects and ii) tracking error quickly accumulates due to the long detection latency (Section II).

This paper proposes FlexPatch, a fast and accurate on-device object detection and tracking technique for high-resolution live video analytics. Our key idea is *tracking-aware patching* to combine detection and tracking in a highly synergistic way. In particular, it identifies small subareas, i.e., *patches*, where lightweight tracking is likely to fail and creates a *patch cluster*, a small-sized rectangle (e.g., 360p) by carefully arranging variable-sized patches. Then, it runs the detector over the patch cluster to quickly offset the tracking errors. The detection latency over a patch cluster is small (e.g., 139 ms for 360p), providing an opportunity to amend tracking errors with fresh detection outcomes while preventing the long accumulation of tracking errors.

Our approach is significantly advantageous over prior DBT frameworks performing *tracking-agnostic* detection [5]–[8]. Unlike our approach, they execute a detector on full high-resolution frames, resulting in high detection latency (e.g., $>$1 sec). Such high latency makes it difficult to fix tracking errors since objects may have moved to different positions (See Section II). Also, our approach is distinguished from prior RoI (Region-of-Interest)-based detection methods (e.g., removing background regions [1], regions where object motion is not significant [9], or running high resolution detection only on regions proposed by a separate DNN [10]) in that i) we actively reduce RoIs for detection by focusing on the tracking-failing subareas, and ii) we aggregate them into a single patch cluster to minimize the detection overhead, whereas prior works separately run the detection over multiple RoIs.

There are multiple challenges and design considerations in realizing our *tracking-aware patching* approach. First, we need a clear understanding of the failure cases of trackers and should efficiently identify the patches where the trackers have a high probability of failing. It is essential to identify these patches accurately with minimal overhead for resource-constrained mobile devices. Second, it is vital to form a

This work was supported by Samsung Research Funding Incubation Center of Samsung Electronics under project number SRFC-IT2001-03. Youngki Lee is the corresponding author of this work.

small patch cluster based on variable-sized patches to run the detector with low latency. The large cluster size would increase the detection latency, making it challenging to offset tracking errors. Also, running the detector on each patch is inefficient since the latency gain gets smaller at a certain input size due to the under-utilization of the processor.

We develop a suite of techniques to address the challenges. We first develop a fast and accurate *Patch Recommender* that effectively finds patches with i) objects suffering from low tracking accuracy and ii) newly appeared objects that are neither detected nor tracked yet. To identify the objects with low tracking accuracy, we first identify a set of useful features (see Section IV-B for details) to estimate such failures and train a machine learning classifier that flags the priority (i.e., high, medium, low). Then, we generate candidate patches that can include those objects individually. To find newly appearing objects from regions outside the tracked patches, we divide the frame into small-sized cells and allocate the priority based on the two following factors: i) edge intensity and ii) refresh interval, which indicates how long it has been since its last detection. Then, we generate candidate patches by grouping neighboring cells that have high priority.

Second, we develop a highly efficient *Patch Aggregator*. We model the patch aggregation problem as the two-dimensional bin packing problem (i.e., packing the variable-sized candidate patches into a rectangular region). Then, we employ the *Guillotine algorithm* [11] to efficiently obtain a good approximate solution since bin packing is a well-known NP-hard problem [12]. We also thoroughly study the tradeoffs of the approximation algorithm and design various aggregation policies (e.g., cluster sizes, weights on the new object detection) that can be adapted to various datasets.

Our contributions can be summarized as follows:

- We develop FlexPatch, a novel technique that enables on-device real-time object detection for live high-resolution videos.
- We propose the *tracking-aware patching* approach that synergistically integrates detection and tracking capabilities. It significantly enhances the prior DBT frameworks that alternate detection and tracking in a simple manner.
- We devise a suite of techniques, i.e., Patch Recommender and Patch Aggregator. They efficiently identify the patches where the tracking likely fails and offset the tracking errors by quickly running the detection on a small cluster of multiple patches.
- We implement FlexPatch on two commodity smartphones (LG V50 and Samsung Galaxy S20) and evaluate its performance on object tracking benchmark datasets. FlexPatch achieves up to 146% accuracy gain in terms of AP compared to the state-of-the-art DBT baseline [6]. Also, it consumes only 37% power compared to the DBT baseline.

II. Motivational Studies

To motivate FlexPatch, we first analyze the limitations of prior DBT frameworks for high-resolution videos. Fig. 1 depicts the operation. The object detector results arrive at every t-th frame due to its inference latency. The lightweight object tracker runs on every frame to track the detected objects, where tracking is done by calculating the optical flow between extracted feature points. The tracker performs one of the following two tasks: i) Detection Propagation: when the fresh detection result becomes available (e.g., the result of 0-th frame arrives at t-th frame), track the bounding boxes onto the current frame with optical flow between the frame that the detector processed and the current frame (e.g., between 0-th and t-th), and ii) Successive Propagation: while detection is running, track the bounding boxes of the previous frame onto the current frame (e.g., between t-th and $t+1$-th). As a reference, we implement the object detector (Tiny YOLO-v4 [13]) and the object tracker (ORB feature point extractor [14] + Lucas-Kanade optical flow estimator [15]) on LG V50 smartphone, where the object detection and tracking latency on a 1080p frame is 1,029 and 10 ms, respectively.

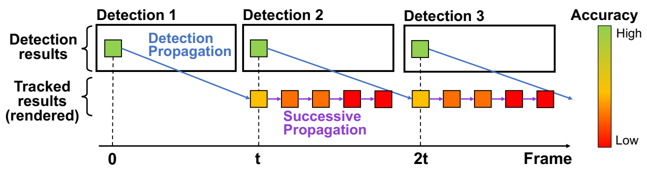


Fig. 1. Real-time Detection-Based-Tracking framework.

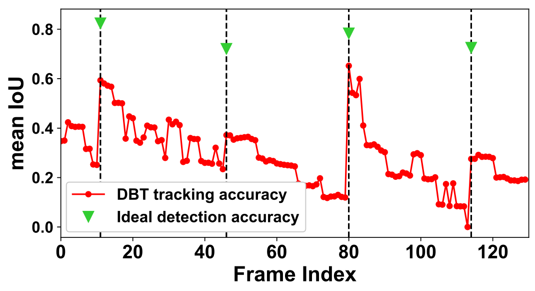
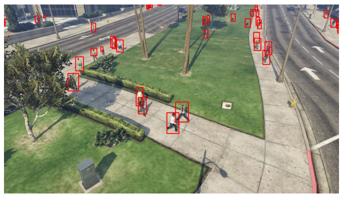


Fig. 2. Tracking accuracy of Detection-Based-Tracking framework over time.

Fig. 2 shows the accuracy of the aforementioned DBT framework in terms of mean IoU over a 4-second window (120 frames) on the Okutama-Action dataset [16]. We observe that the ideal detection accuracy (i.e., accuracy on the frame that the detector processed) is sufficiently high (0.62 on average). However, the actual tracking accuracy is much lower (0.28 on average). In particular, the accuracy continuously decreases between consecutive detection result arrivals (1,029ms apart) due to accumulated tracking errors from successive propagation. Also, the detection results periodically offset the tracking errors, but the tracking accuracy does not fully recover to the ideal accuracy since long detection latency makes the detection result stale and incurs high detection propagation error.

Through an in-depth analysis, we identified two root causes of the low accuracy:

• **Object Tracking Failure.** We observe that the tracker is more prone to failure on high-resolution urban scenes with distant and fast-moving objects than prior works focusing on tracking large objects in close vicinity (e.g., tracking a face or person right in front of the camera [5], [7]). Specifically, we


(a) Ideal detection result.

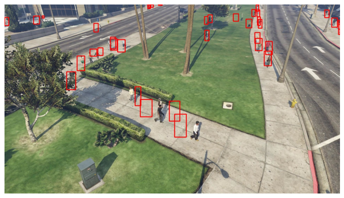

(b) Delayed rendering result (1s).

Fig. 3. Example of object tracking error due to detection latency.

categorize the three main cases of tracking failure: i) inaccurate feature point extraction (e.g., point in the background is extracted as a feature), ii) inaccurate optical flow estimation due to object occlusion (Fig. 6), and iii) optical flow estimation error due to the appearance change (e.g., rotation) of the object (Fig. 7b). Also, it is an inherent limitation that the tracker will never see newly appearing objects.

• **High Object Detection Latency.** Due to the high inference latency of the object detector (i.e., 1,029 ms), the detection result is already stale when it is rendered as shown in Fig. 3; this is especially problematic for distant, small objects, as the detected bounding box on the previous frame can have no overlap with the object in the current frame. Even if we apply detection propagation, the problem is not fully solved. The objects are too far and different in the current frame for the tracker to be accurate. Furthermore, the successive propagation needs to run over a long period ($\approx$30 frames), whose tracking accuracy inherently drops over time.

III. FlexPatch System Overview

A. Our Approach: Tracking-Aware Patching

The DBT frameworks effectively enable real-time continuous object tracking by combining accurate-but-slow detector and fast-but-less-accurate tracker. However, it is not trivial to fuse them over high-resolution videos as the detection latency increases significantly, making it difficult to offset the accumulated tracking errors.

We propose *tracking-aware patching* to address this core challenge of the DBT frameworks. It effectively identifies the flexible-sized patches where the tracking has failed and tightly packs them into a small-sized patch cluster to minimize the detection latency. This prevents the long accumulation of tracking errors and offsets the tracking errors with frequent fresh detection results. Thus, it revives the original intention of the DBT frameworks to combine the unique advantages of detectors and trackers synergistically.

There are other plausible approaches to reducing the detection latency, but they are mostly unsuitable for high-resolution videos. The most simple method is to down-sample the input frame. However, this significantly degrades the accuracy for complex scenes with several distant and small objects. A more sophisticated method is to run the detector on the Regions of Interests (RoIs). It selects sub-areas of the high-resolution frame by training a DNN model to estimate important regions or dividing the frame into fixed-size grids and removing unnecessary regions. This method, however, still suffers from

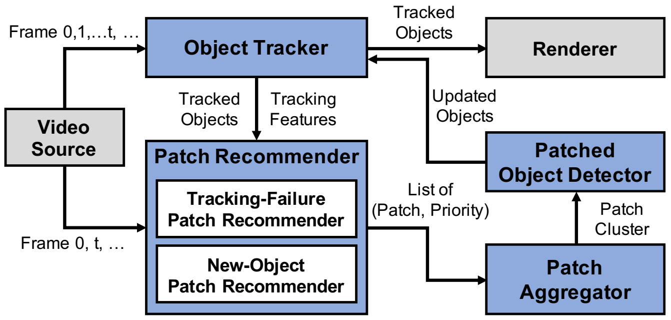


Fig. 4. System architecture of FlexPatch.

two following challenges. First, using RoIs that are tracking-agnostic is inefficient: existing solutions are limited to removing backgrounds [1] or objects with no significant motion [9], which still waste computation on regions where the tracking can work well. Another approach uses a separate DNN trained to output RoI [10], [17], but incurs an additional overhead making it unsuitable for mobile devices. Second, using a content-agnostic fixed-size grid is inefficient: setting the grid size too large would result in unnecessary computation caused by the detector running on background regions, whereas setting it too small makes it hard to detect large objects.

FlexPatch overcomes such problems by i) aggressively reducing the RoI by leveraging the tracking results and ii) using flexible-sized patches for dynamically changing content while tightly packing them to a single rectangular cluster to enable highly efficient detection.

B. System Architecture

Fig. 4 shows the overall system architecture and the operational flow of FlexPatch to realize our approach. First, the Object Tracker calculates the optical flow for every video source frame and tracks the result of the latest detected objects to the current input frame. Since the results of the tracker can be erroneous, the Patch Recommender analyzes the current frame to generate two types of candidate patches: i) tracking-failure patch, where objects are already detected and tracked but might have failed, and ii) new-object patch, where new objects likely appeared. When the Patch Recommender generates a list of patches with different sizes and priorities, the Patch Aggregator tightly packs high priority patches into a patch cluster, which is much smaller than the original frame size. Finally, the Patched Object Detector runs the detection on this patch cluster with low latency.

IV. FlexPatch: Operation Pipeline

In this section, we provide the technical details of FlexPatch in the order of the operational flow shown in Fig. 5.

A. Continuous Object Tracking

Following the conventional DBT framework in Section II, the object tracker runs on every frame to calculate the optical flow and update the previously detected bounding boxes of the objects. We employ two optimizations to enhance performance for high-resolution video processing further. First, we adopt

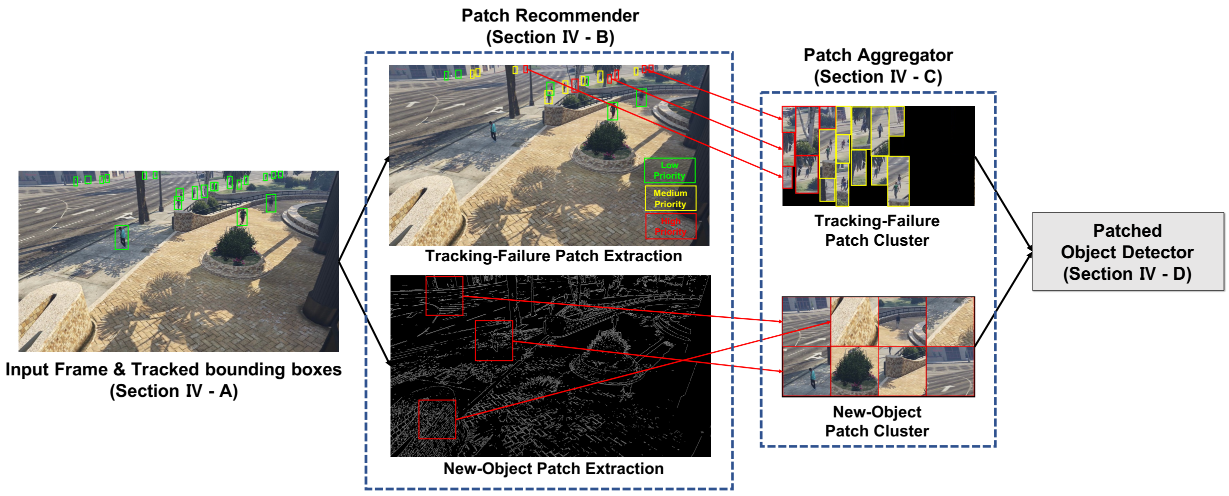


Fig. 5. Operation pipeline of FlexPatch's tracking-aware patching.

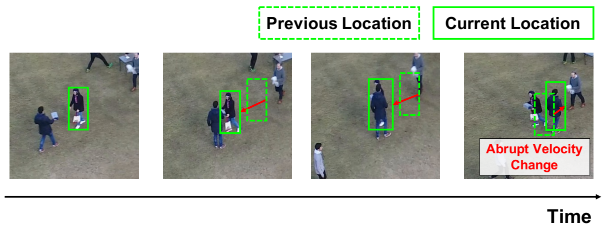
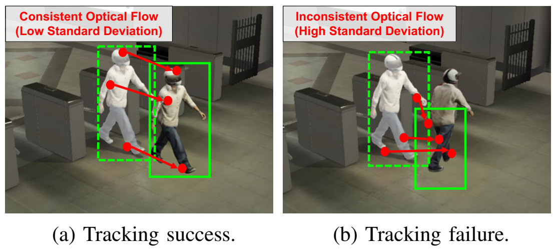


Fig. 6. Tracking failure due to occlusion leads to large acceleration.



(a) Tracking success. (b) Tracking failure.

Fig. 7. Tracking failure due to appearance change or bad feature points can be captured by inconsistent optical flow vectors (high standard deviation).

Incremental Detection Propagation (IDP), which propagates the detection result incrementally to the current frame by running the tracker on intermittent frames cached until the current frame [5]. Second, we utilize an association technique proposed in [18] to prevent flickering effects of the detected objects. Some objects that were detected a few frames ago may fail to be detected even under minor changes such as the lighting condition. To reduce these false negatives, we first associate the previous bounding boxes with new detection results based on the IoU. The boxes that were associated successfully are replaced with the corresponding new detection result. The boxes that failed to be associated are not removed immediately, but their age is increased. Boxes are removed only when the age becomes larger than a predefined threshold.

B. Patch Recommender

Now, we provide the details of our novel Patch Recommender, which analyzes the results of the Object Tracker to generate a list of patches with different priorities (i.e., high, medium, low).

The Patch Recommender conducts lightweight analysis to extract patches where the Object Tracker failed or new objects that are not tracked by the Object Tracker exist and hence should be included in the input for new detection. Since the patch cluster has limited space, it should determine the priority of the patches and the optimal size for the individual patches.

Two types of patches are extracted based on the results of the Object Tracker: i) tracking-failure patch, where objects are already detected and tracked but with severe errors, and ii) new-object patch, where new objects likely appeared. We develop different techniques for each patch type.

1) Tracking-Failure Patch Recommendation: For tracking-failure patches, we aim to find the bounding boxes with severe tracking error and assign them high priority. **Patch Priority Estimation.** The tracking errors mainly come from the inaccuracy of the optical flow calculation. However, it is challenging to figure out whether the estimated optical flow is accurate or not. There are existing features that indicate the optical flow's confidence, such as the eigenvalue of the spatial gradient matrix. However, the confidence value itself is often inaccurate due to its simplicity. Therefore, through an in-depth analysis of the fail cases, we identify three main causes: i) bad feature points (e.g., in the background) are extracted, ii) the appearance of the object changes, and iii) occlusion takes place. We estimate the occurrence of these events.

Based on this observation, we choose multiple features that should be highly correlated to these error cases as follows:

- **Minimum eigenvalue of spatial gradient matrix.** Obtained during the calculation process of the Lucas-Kanade method, this feature indicates the quality of the feature points.
- **Normalized Cross Correlation (NCC).** NCC between the pixel values of the original bounding box and that of the currently tracked bounding box indicates the extent of appearance change and occlusion.
- **Bounding Box Acceleration.** When the object gets occluded by obstacles or other objects, the velocity tends to change abruptly as illustrated in Fig. 6, because no corresponding feature point is found, or the box gets associated with the wrong object.

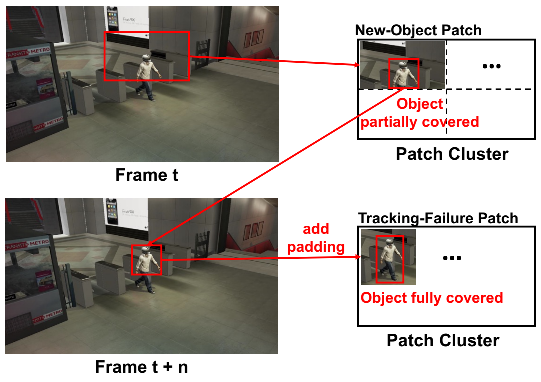


Fig. 8. Example operation of tracking-failure patch extraction with padding.

• **Standard deviation of optical flow vectors.** When the appearance of the object changes (e.g., rotating on its axis), the standard deviation of optical flow vectors is likely to be large as in Fig. 7b. Also, bad feature points in the background area lead to high standard deviation since these points do not move along with the object.

• **Confidence score of the detector.** This feature measures the failure of the detector. A low confidence score may indicate that the box may be a false positive, in which case there is no point in further tracking the box.

Our custom decision tree classifier then uses these features to determine the priority of each patch. The decision tree model is trained to estimate the IoU (Intersection over Union) between the currently tracked object's bounding box and the ground truth object. Specifically, the IoU values of the boxes are divided into three classes: 0 (high priority), 0 to 0.5 (medium priority), and higher than 0.5 (low priority). The classifier estimates the class of each box, which will be its final priority. Among the boxes with the same priority class, we sort the order by the detector confidence score. In addition, in case some high priority boxes are misclassified as low priority, the priority of the box is increased after a predefined amount of time has passed after its last inclusion in patch cluster for detection.

Patch Extraction. To extract the final patches, we add padding for each bounding box since it may not fully cover the object due to the tracking error. In our current version, we add padding size equal to the height and width of the box for simplicity. However, the padding size can be optimized based on the estimated priority (e.g., larger padding for high priority box as it indicates that the tracking is failing significantly, and vice versa).

2) New Object Patch Recommendation: To detect newly appearing objects, we analyze and extract the new-object patches that are likely to contain new objects and assign them high priorities.

Patch Priority Estimation. Since there are no prior hints for new objects from the Object Tracker, we first divide the frame into equal-sized cells (e.g., 8x8 pixels). Then, we assign the priority for each cell based on the following two criteria: i) *Edge Intensity*: the number of edge pixels in a given cell detected by edge detection (e.g., Canny [19]), which is normally high for cells that contain objects [1], and ii) *Refresh Interval*: the time (in number of frames) since the cell was last included in the patch cluster and provided as input to the detector (larger refresh interval indicates that the cell has not been processed for a long time).

By aggregating the two values, the final priority of the cell is calculated as follows:

$$priority = \min(50, RI) + W \times \mathbf{1}_{EI>T}, \qquad (1)$$

where RI and EI are refresh interval and edge intensity, respectively, T is the threshold for edge intensity, and W is the weight to balance the two features. Refresh interval is clipped to have a maximum value of 50, since all the cells would need a new detection similarly after a long time. The priority values of the cells within the existing bounding boxes are set to 0 since the tracking-failure patch recommender handles them.

Patch Extraction. Treating the individual cells as separate patches will not work well since objects are usually larger than a single cell. Therefore $m \times n$ cells (total size of $8m \times 8n$ pixels) are treated as a single patch. We allocate high priority to the patches where the included cells have the largest priority value in summation. The size of the patch (determined by the values of m, n) is set empirically depending on the size of the objects in the dataset. For example, we choose 20x22 cells for the two representative datasets we used, Okutama-Action and MTA dataset. Then 4x2=8 new-object patches will compose a 640x360 sized patch cluster.

However, the fixed patch size might not fully cover some bigger-sized objects. In that case, it can be recovered by the tracking-failure patch recommender. Fig. 8 shows an example of the recovery process: when the new-object patch at *Frame t* partially covers the object and results in inaccurate bounding box, the padding of tracking-failure patch at *Frame t+n* enables the tracking-failure patch to include the whole object, and thus the full bounding box can be recovered in the subsequent detection.

C. Flexible Patch Aggregator

Once the tracking-failure patches, new-object patches and their corresponding priorities have been analyzed by the Patch Recommender, the Patch Aggregator selects the patches starting from the ones with high priority and packs them into a single patch cluster to be processed by the Patched Object Detector. Specific aggregation policy and packing algorithm are detailed as follows:

1) Patch Aggregation Policy: The aggregation policy is determined by the size of the patch cluster and how frequently the two types of patches are aggregated into the patch cluster.

Patch Cluster Size. The patch cluster size is determined by considering the trade-off between the detection latency and the number of high-priority patches that can fit in the patch cluster. Small patch cluster size means that the low detection latency allows more frequent detection updates, but only a few high-priority patches can be included. If the patch cluster size is bigger, more patches can be included, but the detection latency

TABLE I
DETECTOR LATENCY FOR VARIOUS INPUT SIZES MEASURED IN LG V50.

Input Size	320x240	480x270	640x360	720x480	1920x1080
Latency (ms)	70.16	91.91	139.48	206.64	1029.45

```
1:  procedure GUILLOTINE BIN PACKING(list of patches)
2:      Sort the patches by priority
3:      Set $F = \{(W, H)\}$                ▷ list of free rectangles
4:      for patch in patches do
5:          Let $w, h$ = width, height of the patch
6:          $F_p = \{f \in F \mid (w_f \geq w) \cap (h_f \geq h)\}$
7:          if $F_p \neq \varnothing$ then
8:              Choose $f$ from $F_p$
9:              Pack the patch at the bottom left of $f$
10:             Split $f$ into $f'$ and $f''$ on a shorter axis
11:             Set $F = F \cup \{f', f''\} \setminus f$
12:         end if
13:     end for
14: end procedure
```

Fig. 9. Patch aggregation algorithm.

increases. As shown in Table I, the latency gain gets smaller at a certain input size due to the under-utilization issue of the processors, so we can choose the optimal patch cluster size. By default, for a 1080p full-frame, we set the patch cluster size as 360p. However, the optimal patch cluster size can differ depending on the characteristic of the data.

Aggregation Frequency Ratio of Patch Types. Next, the proportion of the two types of patches that are packed into the patch cluster should also be determined. It is inevitable for this policy to adapt to different datasets. If tracking the existing objects accurately is more important, we can set the proportion of the tracking-failure patches higher. If detecting new objects quickly is more important, new-object patches can be included in the patch cluster more frequently.

We set the default setting to alternate between the two types of patches with adequate frequency ratio for simplicity. In other words, since only a single patch cluster can be executed at once by the Patched Object Detector, when the detector becomes available the Patch Aggregator alternates between creating the following two types of patch cluster: i) tracking-failure patch cluster, which consists of all the high to medium-priority tracking-failure patches and additional new-object patches if the patch cluster has unfilled spaces, and ii) new-object patch cluster, which consists of high-priority new-object patches (see Patch Aggregator of Fig. 5).

2) *Patch Aggregation Algorithm:*

Based on the parameters from the aggregation policy, the Patch Aggregator has to concatenate prioritized patches to generate a single patch cluster. The challenge here is to pack as many patches as possible into a fixed-size rectangular space, starting with high-priority ones. Also, the fact that the sizes of the patches are irregular complicates the problem. On top of that, to run this algorithm for every patch aggregation, we need an effective and lightweight algorithm.

Filling a two-dimensional patch cluster with the given patches and priority can be formulated as a Two-dimensional Bin Packing problem with the number of bins limited to 1. The objective is to pack the high-priority patches compactly. Since two-dimensional bin packing is a well-known NP-hard problem [12], we adopt a simple Guillotine algorithm [11] that is effective enough with minimal overhead on the system. The algorithm (Fig. 9) works as follows. Every iteration keeps a list of free rectangles F, which is at the beginning comprised of a single rectangle with the size of a patch cluster (line 3). Then, starting from the patches with higher priority, it finds a free rectangle large enough to fit the patch (line 6). After packing the patch into the chosen free rectangle, it splits the remaining space into two new free rectangles (line 10) and updates the list (line 11). The splitting is done horizontally if the height of the original free rectangle was larger than the width and vertically otherwise. The final output is the matched pairs of patch and free rectangle, which indicates the location of each patch in the patch cluster.

In order to pack even more patches, if the width or height of the tracking-failure patch is larger than a certain size, the patch is down-sampled. The down-sampling here has little impact on accuracy compared to down-sampling the entire frame because a large tracking-failure patch indicates a large object, and a large object will be correctly detected even if it is down-sampled. This guarantees that at least a minimum number of patches will be packed into the patch cluster. At the same time, it ensures that even objects larger than the patch cluster can fit into the patch cluster.

D. *Patched Object Detector and Renderer*

At the final stage, the patch cluster is fed to the object detector. Since the size of the patch cluster is smaller than the full-frame, the latency of the patched object detection is also smaller (e.g., 1,029 ms for full-frame, 139 ms for 360p patch cluster). The detection results are fed to the tracker and the renderer to be displayed on the screen output.

V. EVALUATION

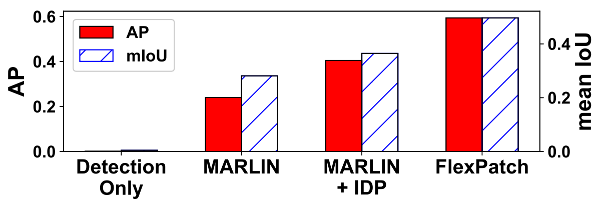
A. *Experiment Setup*

1) *Implementation:*

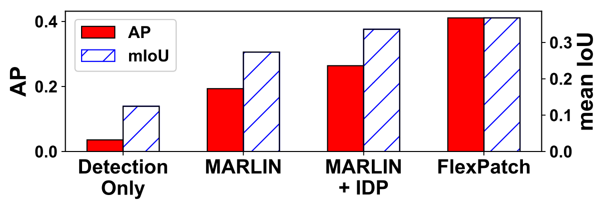
We implement FlexPatch on two commodity smartphones: LG V50 (Qualcomm Snapdragon 855 SoC and Adreno 640 GPU) and Samsung Galaxy S20 (Snapdragon 865 and Adreno 650 GPU). Unless specified, the evaluation result on LG V50 is reported. We use the Tiny YOLO-v4 [13] for the object detector, which is implemented and trained with Darknet framework and converted to TensorFlow-Lite for mobile inference. We implement the optical flow based object tracker and other image processing functions using JavaCV Android 1.5.4. Note that any detectors and trackers can be plug-and-played into our system. The decision tree model for priority estimation of tracking-failure patches is implemented and trained with Python Scikit-Learn library.

2) *Evaluation Datasets:*

We use two benchmark video datasets composed of urban scenes with pedestrians for repeatable evaluation: the Okutama-Action dataset [16] and the MTA dataset [20]. All

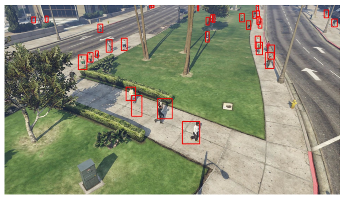


(a) Okutama-Action dataset.



(b) MTA dataset.

Fig. 10. Overall tracking accuracy of FlexPatch and baselines.


(a) MARLIN+IDP.

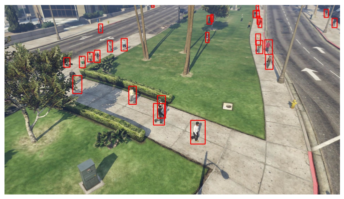
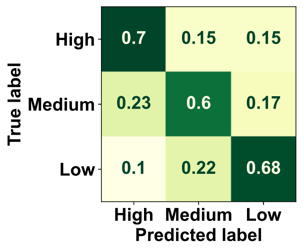
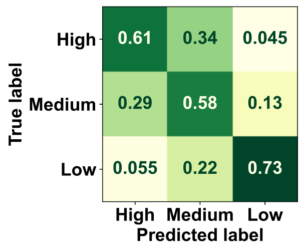

(b) FlexPatch.

Fig. 11. Visual example of MARLIN and FlexPatch's tracking accuracy.



	High	Medium	Low
High	0.7	0.15	0.15
Medium	0.23	0.6	0.17
Low	0.1	0.22	0.68

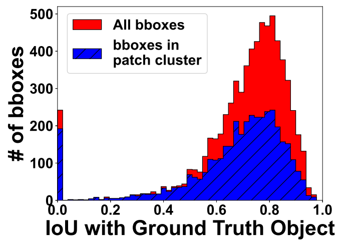
(a) Okutama-Action dataset.



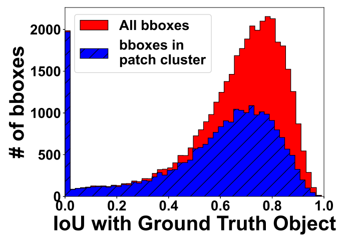
	High	Medium	Low
High	0.61	0.34	0.045
Medium	0.29	0.58	0.13
Low	0.055	0.22	0.73

(b) MTA dataset.

Fig. 12. Confusion matrix of the tracking-failure patch priority estimator.



(a) Okutama-Action dataset.



(b) MTA dataset.

Fig. 13. IoU distribution of the tracked bounding boxes.

the videos are re-encoded to be 1920x1080@30fps and fed to the application to emulate the live video stream. The average number of objects in the Okutama-Action and MTA dataset are 5 and 24, respectively, with sizes less than 25 to 380 pixels in height.

3) Baselines:
We compare FlexPatch against the following baselines.
• **Detection-Only** runs the detector continuously without the object tracker. For the frames in between the detections, it renders the most recent detection result.
• **MARLIN [6]** is a state-of-the-art DBT framework that runs the detector only when there is a significant change in the scene to optimize energy consumption. For fair comparison with FlexPatch, we implemented MARLIN to run the detector continuously to maximize the accuracy.
• **MARLIN + IDP** is an enhanced version of MARLIN that employs Incremental Detection Propagation (Section IV-A).

4) Evaluation Metrics: We evaluate the tracking accuracy in terms of Average Precision (AP@0.5, determining that the object is detected when the IoU is higher than 0.5) and mean Intersection over Union (mIoU) to measure the detection accuracy [21]. Both metrics are calculated per each frame and averaged over the entire video.

B. Performance Overview

We first compare the end-to-end performance of FlexPatch with the baselines. Fig. 10 shows the results. FlexPatch significantly outperforms MARLIN by up to 146% (from 0.24 to 0.59 in terms of AP on the Okutama-Action dataset) by frequently providing new detection results and re-calibrating the bounding boxes. Fig. 11 and Fig. 3 show the visual example comparison of the tracking results of FlexPatch and baselines, clearly showing the superior tracking accuracy gain. Detection-Only shows a poor tracking accuracy as high detection latency ($\approx$ 1 s) leads to stale detection results (i.e., the object locations significantly change from the frame processed by the object detector). MARLIN boosts the accuracy by incorporating the object tracker but still suffers low performance as the tracking accuracy degrades significantly due to large detection latency. MARLIN+IDP alleviates this error to some extent but also suffers from the tracking error accumulation over time, especially when the objects move fast. FlexPatch can achieve significant accuracy gain even compared to the enhanced version of MARLIN due to small detector latency (139 ms for 640x360 patch cluster).

C. Performance of Patch Recommender

1) Tracking-Failure Patch Extraction:
Priority Estimation Accuracy. Fig. 12 shows the confusion matrix of the decision tree-based patch priority classifier. The overall classification accuracy is 0.68 and 0.70 for the Okutama-Action and MTA dataset, respectively. Note that the misclassifications mainly occur between the medium and the high-priority; this is not critical as the Patch Aggregator can effectively pack all the medium and high-priority patches in most cases. Even for cases where the priority of a patch is misclassified as low, the error is quickly recovered as the estimator increases the priority proportional to the time elapsed since it has been included in the patch cluster for detection.
Effectiveness of Patch Extraction. We also show how the percentage of patches included in the patch cluster is distributed for different IoU values in Figure 13. Among the boxes with IoU lower than 0.5 (i.e., high and medium-priority), 85% and

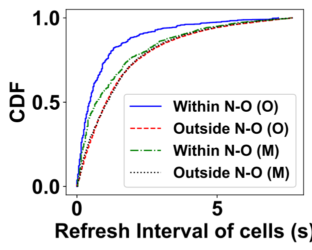


Fig. 14. CDF of refresh interval of the cells within/outside New-Objects (N-O) (O: Okutama-Action, M: MTA).

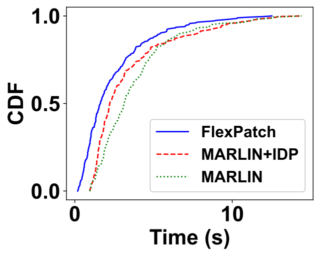
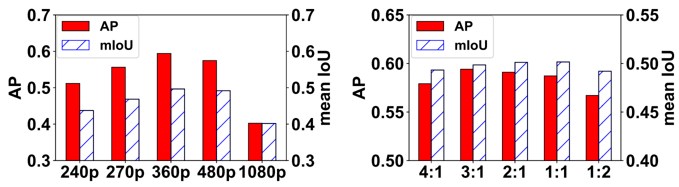


Fig. 15. New object detection delay.



(a) Patch cluster size.

(b) Frequency ratio of patch types.

Fig. 16. Performance of different aggregation policies.

90% are included in the patch cluster for Okutama-Action and MTA datasets, respectively. For boxes with IoU higher than 0.5 (i.e., low priority), the percentage is 53% and 52%. The result indicates that the priority estimator successfully assigns high priority to the bounding boxes with high error.

2) New Object Patch Extraction:

Priority Estimation Accuracy. Fig. 14 shows how well the priorities of new-object patches are estimated. We observe that the average refresh interval is lower for the cells within new objects than those outside new objects (by 0.88 and 0.44 s for Okutama-Action and MTA dataset, respectively), indicating that the new objects are more frequently included in the patch cluster than the background regions without any objects. The difference was lower for the MTA dataset because the scenes contain many background edges (e.g., trees, cars).

New Object Detection Delay. Next, we evaluate the effectiveness of the new-object patch extraction by showing the time elapsed between when the new object appears in the video and when it is successfully detected (i.e., IoU>0.5 with the ground truth). Fig. 15 shows that FlexPatch achieves the lowest average delay (3.11 s) compared to MARLIN (3.97 s) and MARLIN+IDP (4.04 s). The delay is mainly due to the high object detection latency (≈1 s) for the baselines, which bounds the delay and incurs large tracking error. FlexPatch significantly alleviates the issue by running detection on small-sized 360p patch cluster (which takes 139 ms on average) and effectively extracting the new-object patches.

D. Patch Aggregator

1) Aggregation Latency and Efficiency:

The Guillotine packing algorithm runs on average 2 ms for the evaluation datasets, indicating its high computational efficiency. For the Okutama dataset, on average 3.11 medium and high-priority tracking-failure patches (out of the entire 5.40 objects) were compactly packed in the 640×360 patch cluster, occupying 21% of the space, leaving sufficient room

TABLE II
ENERGY CONSUMPTION OF FlexPatch AND MARLIN.

Idle	Read video & Rendering	MARLIN	FlexPatch
0.46 W	2.20 W	4.16 W	2.79 W

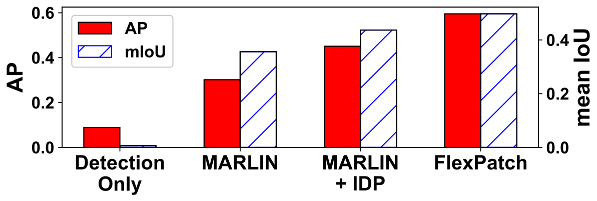


Fig. 17. Overall tracking accuracy on Samsung Galaxy S20.

for the new-object patches to be packed in. Similarly, for the MTA dataset, on average 9.68 out of 15.74 were packed while occupying 35% of the area, indicating high packing efficiency.

2) Performance for Various Aggregation Policies:

We evaluate how different aggregation policies affect the overall performance on the Okutama-Action dataset. Fig. 16a shows that while the performance is consistently high for various patch cluster sizes, 640×360 achieves the best accuracy. When the cluster size gets smaller, the detection result for high-priority patches of both types can be delivered to the tracker faster, increasing the accuracy. However, if the size gets too small to pack most of the high-priority patches, the accuracy decreases. Our result indicates that a cluster size of 640×360 is the minimum size that can pack most high-priority patches. This can be extended to other datasets by considering the number of objects and their sizes.

Next, Fig. 16b shows that the patching frequency ratio of 3:1 achieves the best accuracy. This is because the tracking failure is frequently occurring, thus requiring more frequent detection for accurate tracking. However, if the ratio is too high (e.g., 4:1), the new object detection delay also increases, offsetting the benefit. Thus, the appropriate frequency ratio should be chosen by comparing the importance of tracking the existing boxes accurately and detecting new objects quickly.

For the MTA dataset, we found that the best aggregation policy uses the same 640×360 patch cluster, but with 2:1 patching frequency ratio; This is because the MTA dataset has smaller but more number of objects, and they are relatively slow moving, resulting in less tracking failure.

E. Energy Consumption

We evaluate the energy consumption of FlexPatch in Table II. We decompose the energy consumption of FlexPatch from the baseline app power (i.e., video decoding and rendering). FlexPatch consumes 37% power compared to MARLIN. The main source for the energy consumption is running the object detector continuously on the GPU. We conjecture that FlexPatch's energy consumption gain mainly comes from running the detector on smaller-sized input (e.g., 360p vs. 1080p), resulting in smaller GPU utilization.

F. Performance Scalability on Other Mobile Devices

Finally, we evaluate the performance of FlexPatch on Samsung Galaxy S20, where the average detector latency on 1080p and 360p images are 824 ms and 126 ms, respectively. Due to space limit, we only show the end-to-end performance for Okutama-Action dataset. Fig. 17 shows that FlexPatch achives similar performance gain over the baselines (e.g., 21% accuracy gain compared to MARLIN+IDP), validating FlexPatch's scalability on various mobile devices.

VI. Discussion

A. Generality

Generalizability to Other Detection Tasks. While FlexPatch is mainly evaluated on pedestrian detection datasets, it can be generalized to other detection tasks on high-resolution videos. For example, it can be used for vehicle detection in drone video surveillance or face detection in AR person identification. We expect similar performance gains in such scenarios, as FlexPatch can be applied similarly to fuse the detection and tracking in a highly synergistic manner to optimize tracking accuracy.

Extension to Offloading Systems. While FlexPatch currently assumes full on-device processing, FlexPatch can also be easily extended to offloading systems for live video analytics [5], [7], [22]. Our tracking-aware patching approach can be used to identify the patches of the frame to offload, significantly reducing the network bandwidth consumption and transmission latency, especially in outdoor environments with constrained and fluctuating network bandwidths.

B. Limitations and Future Work

In this work, we identified the parameters for our patching technique individually for each dataset. A system that can automatically adjust the parameters depending on the characteristics of the current data remains as our future work. Based on our observation, the following characteristics of the input data can be considered:

Target Object Size. If some objects are much larger than the predefined new-object patch size, our patching method would not be able to catch those objects. With an additional technique that can identify these failures in run-time, we can enlarge the patch size adaptively.

Extent of Tracking Failure. Depending on how much the object tracker is failing and the cause of the failure (i.e., actual tracking failure on the detected objects or failure on new objects), the patch aggregation policy (i.e., the patch cluster size and the aggregation frequency ratio of patch types) can be dynamically adjusted.

VII. Related Works

A. Live Video Analytics Systems

Live video analytic systems have been actively studied in recent years for various applications, such as face recognition [1], pedestrian detection [23], or vehicle detection [24]. Most systems, however, are focused on object detection [2], [3], [25]. Despite active research in this area, there are no systems that can meet the requirement for high-resolution videos on mobile devices.

B. Lightweight Object Detection Models

With the recent advancements in deep learning, several high accuracy object detection models have been proposed (e.g., Faster-RCNN [26], R-FCN [27], and Cascade-RCNN [28]). However, such models are often computationally heavy for mobile inference. Recently, several lightweight object detectors have also been proposed; YOLO [4], [29]–[31], SSD [32] and RetinaNet [33] are some of the most popular detectors that are highly efficient with decent accuracy. FlexPatch takes a complementary approach to optimize the object detection latency on high-resolution videos.

C. On-device Deep Learning Systems

There has been a streamline of works on on-device deep learning systems for real-time object detection on mobile devices. One approach is to adjust the input frame size and the deep learning model adaptively depending on the video content and resource contention [34]–[37]. Some works apply caching on partial results of DNN layers to reduce repetitive computation [9], [38]. However, these works only consider features in frame scale and therefore may not benefit much if each region of the frames has different characteristics and priorities.

D. ROI-based Object Detection

There have been prior attempts to extract region-of-interest (RoI) to reduce the computational load by running inference only on the partial area. Several works remove background regions with edge detector [1]. Others apply encoding techniques to compress the background regions heavily [7], [22]. However, RoI extraction in these works are tracking-agnostic, incurring computational waste. There are more advanced approaches to narrow down RoIs. Some works run a DNN on down-sampled images to select difficult regions [10], [39]. Another prior work employs reinforcement learning to select RoIs [17]. However, the region selection processes in these works are not only tracking-agnostic but also compute-intensive to run in real-time on mobile device. FlexPatch utilizes a lightweight estimator to predict tracking failure and optimize the detection latency efficiently.

VIII. Conclusion

In this paper, we presented FlexPatch that enables high-resolution live video analytics on resource-constrained mobile devices. We designed a novel tracking-aware patching technique which extracts dynamically-sized patches where tracking is likely to fail and runs the detection only on these patches by aggregating them into a single small-sized rectangular patch cluster. Our results showed that FlexPatch achieved up to 146% accuracy gain in terms of AP compared to the state-of-the-art DBT baselines while consuming only 37% power.

References

[1] J. Yi, S. Choi, and Y. Lee, "Eagleeye: Wearable camera-based person identification in crowded urban spaces," in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, 2020, pp. 1–14.

[2] J. K. Mahendran, D. T. Barry, A. K. Nivedha, and S. M. Bhandarkar, "Computer vision-based assistance system for the visually impaired using mobile edge artificial intelligence," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2021, pp. 2418–2427.

[3] J. Wang, Z. Feng, Z. Chen, S. George, M. Bala, P. Pillai, S.-W. Yang, and M. Satyanarayanan, "Bandwidth-efficient live video analytics for drones via edge computing," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 2018, pp. 159–173.

[4] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," *arXiv preprint arXiv:2004.10934*, 2020.

[5] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan, "Glimpse: Continuous, real-time object recognition on mobile devices," in *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, 2015, pp. 155–168.

[6] 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 *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, 2019, pp. 96–109.

[7] L. Liu, H. Li, and M. Gruteser, "Edge assisted real-time object detection for mobile augmented reality," in *The 25th Annual International Conference on Mobile Computing and Networking*, 2019, pp. 1–16.

[8] M. Liu, X. Ding, and W. Du, "Continuous, real-time object detection on mobile devices without offloading," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2020, pp. 976–986.

[9] L. N. Huynh, Y. Lee, and R. K. Balan, "Deepmon: Mobile gpu-based deep learning framework for continuous vision applications," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, 2017, pp. 82–95.

[10] M. Gao, R. Yu, A. Li, V. I. Morariu, and L. S. Davis, "Dynamic zoom-in network for fast object detection in large images," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 6926–6935.

[11] J. Jylänki, "A thousand ways to pack the bin-a practical approach to two-dimensional rectangle bin packing," *retrived from http://clb. demon. fi/files/RectangleBinPack. pdf*, 2010.

[12] J. Hartmanis, "Computers and intractability: a guide to the theory of np-completeness (michael r. garey and david s. johnson)," *Siam Review*, vol. 24, no. 1, p. 90, 1982.

[13] Z. Jiang, L. Zhao, S. Li, and Y. Jia, "Real-time object detection method based on improved yolov4-tiny," *arXiv preprint arXiv:2011.04244*, 2020.

[14] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: An efficient alternative to sift or surf," in *2011 International conference on computer vision*. Ieee, 2011, pp. 2564–2571.

[15] B. D. Lucas, T. Kanade *et al.*, "An iterative image registration technique with an application to stereo vision." Vancouver, British Columbia, 1981.

[16] M. Barekatain, M. Martí, H.-F. Shih, S. Murray, K. Nakayama, Y. Matsuo, and H. Prendinger, "Okutama-action: An aerial view video dataset for concurrent human action detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2017, pp. 28–35.

[17] Y. Chai, "Patchwork: A patch-wise attention network for efficient object detection and segmentation in video streams," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 3415–3424.

[18] N. Wojke, A. Bewley, and D. Paulus, "Simple online and realtime tracking with a deep association metric," in *2017 IEEE international conference on image processing (ICIP)*. IEEE, 2017, pp. 3645–3649.

[19] J. Canny, "A computational approach to edge detection," *IEEE Transactions on pattern analysis and machine intelligence*, no. 6, pp. 679–698, 1986.

[20] P. Kohl, A. Specker, A. Schumann, and J. Beyerer, "The mta dataset for multi-target multi-camera pedestrian tracking by weighted distance aggregation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, 2020, pp. 1042–1043.

[21] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International journal of computer vision*, vol. 88, no. 2, pp. 303–338, 2010.

[22] X. Wang, Z. Yang, J. Wu, Y. Zhao, and Z. Zhou, "Edgeduet: Tiling small object detection for edge assisted autonomous mobile vision," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2021.

[23] J. Li and W. Gong, "Real time pedestrian tracking using thermal infrared imagery." *J. Comput.*, vol. 5, no. 10, pp. 1606–1613, 2010.

[24] A. Gomaa, M. M. Abdelwahab, and M. Abo-Zahhad, "Efficient vehicle detection and tracking strategy in aerial videos by employing morphological operations and feature points motion analysis," *Multimedia Tools and Applications*, vol. 79, no. 35, pp. 26 023–26 043, 2020.

[25] K. Chen, T. Li, H.-S. Kim, D. E. Culler, and R. H. Katz, "Marvel: Enabling mobile augmented reality with low energy and low latency," in *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, 2018, pp. 292–304.

[26] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *Advances in neural information processing systems*, vol. 28, pp. 91–99, 2015.

[27] J. Dai, Y. Li, K. He, and J. Sun, "R-fcn: Object detection via region-based fully convolutional networks," in *Advances in neural information processing systems*, 2016, pp. 379–387.

[28] Z. Cai and N. Vasconcelos, "Cascade r-cnn: Delving into high quality object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 6154–6162.

[29] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[30] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7263–7271.

[31] ——, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[32] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European conference on computer vision*. Springer, 2016, pp. 21–37.

[33] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2980–2988.

[34] R. Xu, C.-l. Zhang, P. Wang, J. Lee, S. Mitra, S. Chaterji, Y. Li, and S. Bagchi, "Approxdet: content and contention-aware approximate object detection for mobiles," in *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, 2020, pp. 449–462.

[35] R. Xu, R. Kumar, P. Wang, P. Bai, G. Meghanath, S. Chaterji, S. Mitra, and S. Bagchi, "Approxnet: Content and contention-aware video object classification system for embedded clients," *ACM Transactions on Sensor Networks (TOSN)*, 2021.

[36] T.-W. Chin, R. Ding, and D. Marculescu, "Adascale: Towards real-time video object detection using adaptive scaling," *arXiv preprint arXiv:1902.02910*, 2019.

[37] B. Fang, X. Zeng, and M. Zhang, "Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, 2018, pp. 115–127.

[38] M. Xu, M. Zhu, Y. Liu, F. X. Lin, and X. Liu, "Deepcache: Principled cache for mobile deep vision," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, 2018, pp. 129–144.

[39] Y. Zeng, P. Zhang, J. Zhang, Z. Lin, and H. Lu, "Towards high-resolution salient object detection," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 7234–7243.