MobiEye: An Efficient Cloud-based Video Detection System for Real-time Mobile Applications

Jiachen Mao, Qing Yang, Ang Li, Hai Li, Yiran Chen
Duke University
{jiachen.mao,qing.yang21,ang.li630,hai.li,yiran.chen}@duke.edu

ABSTRACT

In recent years, machine learning research has largely shifted focus from the cloud to the edge. While the resulting algorithm- and hardware-level optimizations have enabled local execution for the majority of deep neural networks (DNNs) on edge devices, the sheer magnitude of DNNs associated with real-time video detection workloads has forced them to remain relegated to remote execution in the cloud. This problematic when combined with the strict latency requirements that are coupled with these workloads, and imposes a unique set of challenges not directly addressed in prior works. In this work, we design *MobiEye*, a cloud-based video detection system optimized for deployment in real-time mobile applications. *MobiEye* is able to achieve up to a 32% reduction in latency when compared to a conventional implementation of video detection system with only a marginal reduction in accuracy.

1 INTRODUCTION

Deep neural networks (DNNs) are now utilized in many applications on mobile, including automated speech recognition, natural language processing, object detection and classification, facial recognition, and etc. Of these, real-time video detection systems are among the most computationally-demanding tasks.

Input frames received by a video detection system may contain multiple objects. This possibility imposes a significant increase in resource requirements for two main reasons: 1) With the potential for multiple objects within a single frame, the resolution of the input image must be increased in order to reliably identify the characteristic details within each object. 2) As an individual object may occupy only a small section of an input frame, real-time video detection systems actually involve the two separate tasks of object localization and object classification.

Many attempts have been made to minimize "data-to-decision-availability" latency in real-time video detection on mobile. For local execution on the device itself, *model compression* has been explored. By reducing the size of DNN models with various methodologies, e.g. weight quantization [5], group lasso [16], or low rank [2], execution speed can be increased. However, these techniques have individually been shown to result in loss of classification accuracy for the underlying network. When it is considered that they would need to be used in unison in order to allow for real-time computation on a mobile device, these solutions remain unideal.

The best existing solution for real-time video detection systems on mobile relies on *cloud computation*, in which the mobile device is only responsible for data collection, with all evaluative tasks (e.g., DNN inference) being performed server-side following transmission of the data [6]. However, a remaining challenge is that such a client-server paradigm depends heavily on the capabilities of the remote system, making it difficult to guarantee latency because of dynamic server workload.

In this paper, we propose *MobiEye*, a cloud-based, real-time video detection system optimized around reducing latency in *cloud computation* contexts. *MobiEye* achieves this through novel advances in both system- and algorithm-level design:

- We adopt Deep Feature Flow (DFF) in *MobiEye* and utilize asynchronous computing methods, namely ADFF, to optimize the execution pipeline of DFF from system-level.
- We propose Video-based Dynamic Scheduling (VDS) scheme, which utilizes the motion vector in video decoding procedure to dynamically adjust the inference frequency.
- We propose Spatial Sparsity Inference (SSI), which further accelerates the inference time for each video frame base on designed computation masks.
- We implement *MobiEye* components on both mobile and server and evaluate it with state-of-the-art DNNs.

2 PRELIMINARY

2.1 DNN Model Profiling on Mobile Devices

Although DNNs allow for state-of-the-art accuracy when running on high-performance platforms (e.g., FPGA, GPU, TPU), the limited computing resources available on embedded platforms (e.g., smartphones) limit DNN accuracy and/or viability in those contexts [12] [11]. Many prior works have focused on the speed-accuracy tradeoff between different DNN model structures. Table 1 illustrates the inference time of two representative DNNs [8][14] when run on flagship smartphones, as well as their realized Top-1 accuracy on the ImageNet dataset [9]. MobileNet [8] is an efficient network structure designed for mobile devices, while Inception-v4 [14] represents state-of-the-art in accuracy. It can be seen that even the highly-optimized MobileNet is limited to only 6fps on Pixel 2. Note that the inference times in Table 1 are measured from image classification tasks, where the input image size and network complexity is greatly reduced when compared to video detection.

Table 1: Profiling of 2 state-of-the-art DNN models.

	Pixel 2	iPhone 8	Top-1 Acc	Param	Input Size
MobileNet	166.5 ms	32.2 ms	70.9%	17MB	224×224
Inception-v4	3180 ms	611 ms	80.2%	171MB	229×229

DAC '19, June 2–6, 2019, Las Vegas, NV, USA

ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317865

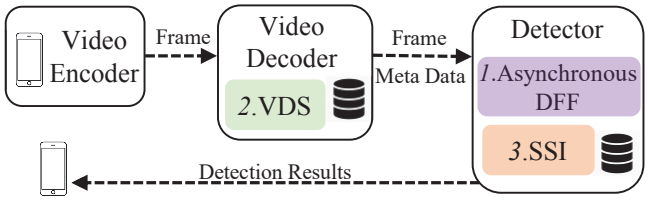

Figure 1: System overview of *MobiEye*.

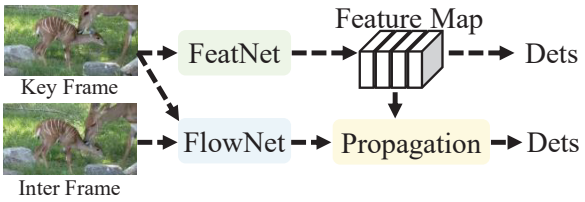

Figure 2: Network architecture of *DFF* [17].

2.2 Feature Map Sparsity in DNNs

Different from traditional model compression which focuses on weight compression, feature map sparsity leverages the zeros in feature maps to accelerate DNN inference. Many priors works accelerate DNN execution based on the feature map sparsity derived from the semantic content of input images. In [10], Li *et al.* convert a DNN model into a cascaded structure for image segmentation so that some simple objects can be identified in earlier cascaded layers. In [4], Figurnov *et al.* proposed *PerforatedCNNs*, which skips convolution operations utilizing a perforated mask, utilizing interpolation to restore skipped feature maps. In SBNet [13], gather and scatter operators are designed to compute only the sub-blocks when a non-zero number in the feature map surpasses a threshold.

Compared to model compression, the advantage of feature map sparsity with semantic properties (e.g., saliency) lies in the reduced amount of modification to the underlying DNN structure. In *MobiEye*, we design Spatial Sparsity Inference (SSI), which spatially skips the feature map based on the proposed computation masks to achieve speedup target with tiny accuracy loss.

2.3 Motion Vector in H.264 Video Codec

Considering the network bandwidth, video contents captured from mobile camera are compressed before being distributed. H.264 is a widely-used format, which is also known as MPEG-4 Part 10, Advanced Video Coding [15]. The fierce compression rate of H.264 lies in its *motion compensation* scheme. H.264 gather several continuous video frames in a group, referred as Group of Pictures (GoP). Each GoP consists of one I frame ("I" for Intra) and multiple P and B frames ("P" for Prediction or "B" for Bidirectional). I frame independently encodes a complete frame, which serves as the reference point for P and B frame in the same GoP. P and B frame only encode the difference between frames and thus need less data size. The video frame are divided into several macroblocks, serving as the basic unit for predicting the frame difference. The size of macroblock are typically set to 16×16 in H.264. The difference between frames are described by *motion vector*, including the source and destination of all macroblocks. In this work, we adopt H.264 codec for efficient communication between mobile camera and cloud execution under wireless network such as Wi-Fi or LTE networks.

3 SYSTEM FRAMEWORK OF MOBIEYE

Figure 1 depicts an overview of *MobiEye*, which includes three system-level and algorithm-level optimization methods:

(1) **Asynchronous Deep Feature Flow (ADFF)**: A system-level optimization of the DFF video detection framework [17] using multi-threading method, which will be detailed in Section 3.2;

(2) **Video-based Dynamic Scheduling (VDS)**: An algorithm-level optimization which dynamically adjusts keyframe detection rate based on metadata already present in H.264 encoded video, which will be detailed in Section 4;

(3) **Spatial Sparsity Inference (SSI)**: An algorithm-level optimization which accelerates DNN inference by focusing only on visual saliency areas, which will be detailed in Section 5.

3.1 Video Detection Framework (DFF)

We adopt a state-of-the-art video detection system: Deep Feature Flow (*DFF*) [17] in *MobiEye*. *DFF* divides video frames into key frames and inter frames. Key frames are detected with DNN while inter frames use a comparatively smaller DNN to compare themselves with key frame, generating their feature map based on interpolation from the key frame feature map.

Figure 2 shows the architecture of *DFF*, which consists of three networks: (1) Feature Network (FeatNet): FeatNet extracts the feature maps of the input frame utilizing ResNet-101 [7]. (2) Optical Flow Network (FlowNet): The most recent key frame and the current inter frame are provided as input to FlowNet [3], which calculates the spatial difference between the two frames in terms of optical flow. (3) Propagation Function (Propagation): The key frame feature map is propagated from the calculated optical flow using bilinear interpolation, generating the feature map for the current inter frame. Note that the computational cost of FlowNet and the propagation function are much smaller than that of FeatNet, causing average time consumption per frame to be most heavily influenced by the key frame interval.

3.2 Asynchronous Computation (ADFF)

One limitation of *DFF* is that although it can achieve high framerate on average, inference requires different computation time depending on whether the current frame is a key frame or an inter frame. This is a result of the different network structures utilized by each type of data. Such sequential execution is described in Algorithm 1,

Algorithm 1 Original Computation Procedure in *DFF*.

```
Init Keyframe Interval: i
for each Current Frame f_cur with Index idx do
    if f_cur is keyframe then
        Dets, feat_key = FeatNet(f_cur)
    else if f_cur is interframe then
        Dets = FlowNet(f_cur, f_key, feat_key)
    end if
end for
```

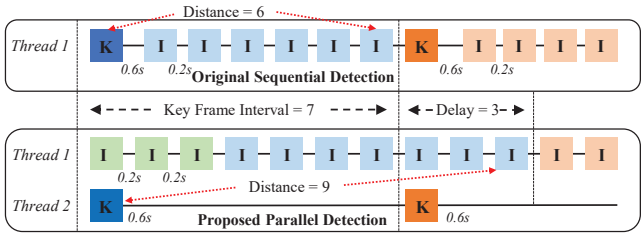

Figure 3: Original sequential execution of DFF (top) vs proposed asynchronous execution of DFF (bottom).

and leads to unbalanced inference time, which is not desirable for real-time applications.

Therefore, in *MobiEye*, we execute the feature network and optical flow network asynchronously. Figure 3 compares the original sequential (top) with proposed asynchronous (bottom) execution of *DFF* when the key frame interval is set as 7. In sequential execution, the first and eighth frame require 0.6*s*, while asynchronous execution allows for an inference time of 0.2*s* for every frame. In asynchronous execution, we set utilize two threads: one to execute key frame inference and the other to execute inter frame inference. Before the current key frame feature map is generated, the inter frame uses the feature map of the last key frame and thus asynchronously gets the detection results. By performing computations asynchronously, detection latency only depends on the execution time of the optical flow network and the transmission time of video frames. Correctness is assured by utilizing a mutex to indicate when the current key frame feature map is available. In the example of Figure 3, the feature map delay is 3 frames as FeatNet is approximately 3× slower then FlowNet. More formally,

$$D_f = \left\lceil \frac{T_{FeatNet}}{T_{FlowNet}} \right\rceil \tag{1}$$

where $T_{FeatNet}$ and $T_{FlowNet}$ are the inference times for FeatNet and FlowNet, respectively, $\lceil \cdot \rceil$ rounds up to the nearest integer, and

Algorithm 2 Asynchronous Computation in *MobiEye* (ADFF).

KeyframeThread (FeatNet, f_{cur}):
  $\_, feat_{key}^{temp}$ = FeatNet(f_{cur}), *threadLock*.acquire()
  $feat_{key} = feat_{key}^{temp}$, *threadLock*.release()
Init Thread Lock: *threadLock*, Keyframe Interval: i, Delay: d
for each Current Frame f_{cur} with Index idx **do**
  if f_{cur} is the first frame in video **then**
    $Dets, feat_{key}$ = FeatNet(f_{cur})
  else
    if f_{cur} is keyframe **then**
      thread = new **KeyframeThread()**
      thread.start(FeatNet, f), *threadLock*.acquire()
    else if $idx \% i == d$ **then** *thread*.join()
    else if $idx \% i == d - 1$ **then** *threadLock*.release()
    end if
    $Dets$ = FlowNet(f_{cur}, f_{key},$feat_{key}$)
  end if
end for

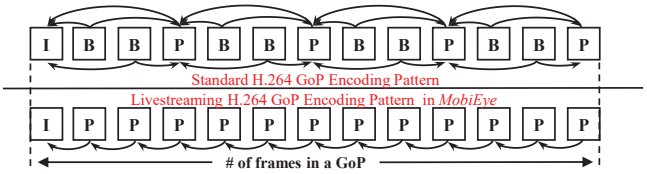

Figure 4: Standard H.264 GoP pattern (top) and adopted H.264 GoP pattern in *MobiEye* (bottom).

D_f is the feature map delay in terms of frame count. In practice, D_f can be even smaller because of the communication overhead.

The details of *ADFF* are illustrated in Algorithm 2.

4 VIDEO-BASED DYNAMIC SCHEDULING

In *DFF*, key frames are selected at a set interval, regardless of the underlying video data [17]. This overlooks the case where video clips may contain long runs where there is very little change from one frame to the next. In such a case, key frame interval can be greatly increased without loss of accuracy, reducing the requirement for expensive key frame inference calculations. In *MobiEye*, we design a Video-based Dynamic Scheduling (VDS) scheme to dynamically determine whether the current frame is a key frame or not. By adopting VDS, the feature map network is frequently executed when the movement in video is fast, and rarely executed when movement is slow.

4.1 H.264 Motion Vector Extraction

The key idea of capturing inter-frame movement quantity information is based on the H.264 video codec. During server-side video decoding, the motion vector of the frame is extracted at the same time. The H.264 GoP pattern adopted in *MobiEye* is described in the bottom of Figure 4, the top part of which shows a standard GoP pattern. The GoP pattern of *MobiEye* does not include B frames as the nature of live video streaming means the required information from future frames is never available. Targeting dynamic key frame scheduling, the number of reference frames is set to 1. In this way, all P frames motion vectors are calculated based on a single prior frame, as shown in the bottom of Figure 4. Note that motion vector extraction incurs no additional computation cost as the motion vector is already encoded as part of the H.264 video stream.

4.2 Video-based Dynamic Scheduling (VDS)

In order to achieve high efficiency, VDS is a simple scheduling scheme. As described in Algorithm 3, VDS initializes the dynamic frame interval ranging from fi_{min} to fi_{max}, and the motion vector magnitude from mv_{min} to mv_{max}. For each video frame, VDS calculates a scalar mv_{mean}, representing the mean of the magnitude of the incoming mv_{cur}. Following this, VDS maps mv_{mean} to a new key frame interval (fi_{cur}) using Min-Max linear mapping. We utilize mv_{acc}, the accumulated motion vector over several continuous frames, to denote the total motion from the last key frame until the current frame. When mv_{acc} is larger then mv_{max}, the minimum frame interval (fi_{min}) will be assigned to the current frame interval (fi_{cur}).

Algorithm 3 Video-based Dynamic Scheduling Scheme (VDS).

Init Dynamic key frame interval range : fi_{min}, fi_{max}
Init Motion vector value range : mv_{min}, mv_{max}
Init Accumulated motion value : $mv_{acc} = 0$
for each new frame motion vector mv_{cur} from key frame **do**
  mv_{mean}+ =Mean(Abs(mv_{cur})), mv_{acc}+ = mv_{mean}
  if $mv_{acc} > mv_{max}$ **then** $fi_{cur} = fi_{min}$
  end if
  $fi_{cur} = fi_{min} + (fi_{max} - fi_{min}) * \frac{mv_{max} - mv_{mean}}{mv_{max} - mv_{min}}$
end for

5 INFERENCE WITH SPATIAL SPARSITY

In Section 4, temporal redundancy in video detection system is eliminated via VDS. In this section, a method for removal of spacial redundancy in video detection system is also detailed.

5.1 Spatial Sparsity Inference (SSI)

The key idea of Spatial Sparsity Inference (SSI) lies in skipping unimportant pixels in order to accelerate DNN execution. In *MobiEye*, we design two computation masks, indicating the spatial position to be skipped: (1) **Computation mask based on feedback detection results:** Note that, in *DFF*, the inter frame is detected based on the key frame and the optical flow between these two frames. Therefore, the background part of both frames need not be examined by FlowNet in *DFF*, as background movement would not affect final detection results. Figure 5 shows an example of a feedback computation mask, where only the region within the red bounding box is executed. The red bounding box is defined by the detection results from the key frame (green box), with an additional margin of constant size. In *MobiEye*, we set $Margin = 64$ due to the fact that the ratio between the spatial size of the input image and the feature map of the last layer in FlowNet is 64 : 1. For the feedback computation mask in SSI, the non-zero mask area grows with the increase in interval frames between the last key frame and current inter frame due to the expected increase in movement. (2) **Computation mask based on brightness error:** Feedback computation masks reduce computational redundancies due to reprocessing of the image background. However, they can not be applied when feedback detection results cover the entire video frame. To deal with such a case, we design a computation mask based on brightness error calculated by subtracting inter frame from key frame, which is defined as:

$$Mask = 0 \; where \; |B_{key} - B_{inter}| < thd, else \; 1; \tag{2}$$

where thd stands for threshold of brightness error to be skipped and B_{key} and B_{inter} represent pixel value of key frame and inter frame, respectively. Figure 6 shows the brightness error mask of an elephant video with a brightness threshold of 15. It can be seen that a large region is unnecessary for re-calculation even though it falls within the detection bounding box.

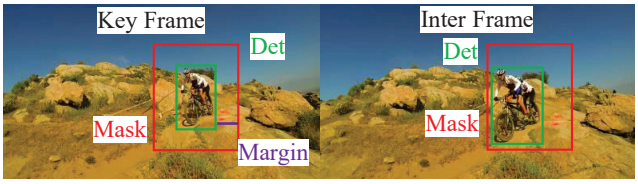

Figure 5: Feedback computation mask.

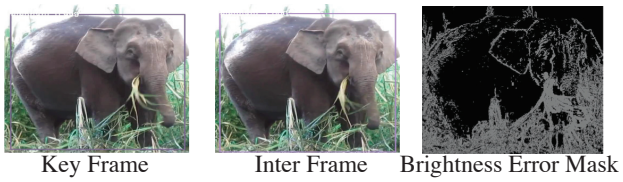

Figure 6: Brightness error computation mask.

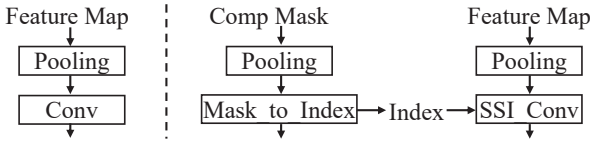

Figure 7: Original layers (left) and layers in SSI (right).

5.2 Realization of SSI

The realization of SSI is jointly inspired from *PerforatedCNNs* [4] and *SBNet* [13], with necessary modifications. The reason why we combine the idea of *PerforatedCNNs* with *SBNet* in *MobiEye* is due to their functional limitations when taken separately. If individually adopted, (1) *PerforatedCNNs* support efficient pixel-wise skipping but the skip index is static with the same computation mask for all input images, while (2) *SBNet* supports dynamic computation mask but only realizes convolution operation speedup in block-wise situations.

Figure 7 presents the original layer modules (left) and their corresponding SSI modules (right), except that SSI skips the spatial pixels for fast execution. Compared with the original layer module with feature map as input, the SSI module is fed the same spatial size computation mask (as described in Section 5.1) and feature map. To deal with the scale change in different DNN layers, we borrow the idea from SBNet [13] which uses a pooling operation followed by a threshold to downsample the input computation mask. The convolution in the original layer module is replaced with a SSI_Conv Layer, which is similar to the perforated convolutional layer in *PerforatedCNNs*.

Because SSI belongs to structured sparsity, it can better accelerate DNN execution with fully-optimized, dense GEMM in both CPU and GPU mode. Figure 8 details the procedures of the SSI_Conv operation layer. Before matrix to matrix multiplication in step 2, the input feature map is first expanded from a 3-dimension to a

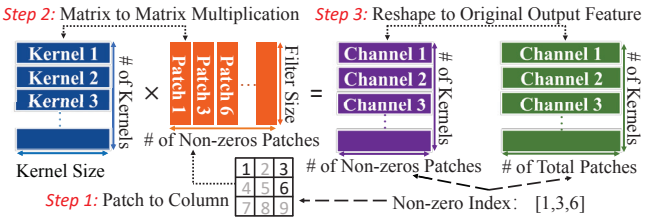

Figure 8: SSI convolution operation.

2-dimension tensor with a patch to column operation. Originally, the each patch in the feature map are flatten to a single matrix column iteratively through X-axis and Y-axis. In SSI_Conv, patch to column operation take the non-zero index as input and iteratively assigns the column base on the patches with the non-zero index.

6 EXPERIMENTS

6.1 Experimental Setup

Test Benches: We implement *MobiEye* using the *DFF* [17] framework, with all optimization schemes in *MobiEye* compared to *DFF* as baseline. For fairness of comparison, we directly utilize trained models from *DFF* without fine-tuning for all experiments. We adopt the ImageNet VID dataset for evaluation, which includes 5354 annotated videos. The model structures adopted for the feature extraction network and optical flow network are ResNet101 [7] and FlowNet [3], respectively. Video frames are resized to 600 pixels on the shorter side as the input of the feature network, and 300 pixels on the shorter side for the optical flow network. In our experiments, accuracy-related performance of the video detection system is reported using a mean average precision (mAP), speed-related performance is evaluated in terms of sparsity and milliseconds (ms).

System Environment: For the client side of *MobiEye*, we adopt the Nexus 5 and and the Pixel 2, representing two popular Android-based smartphones. The Nexus 5 is powered by a Quad-core Krait CPU with an Adreno 330 GPU and 2GB of RAM. The Pixel 2 is equipped with an Octa-core Kryo CPU with an Adreno 540 GPU and 4 GB RAM. We implement and deploy a H.264 video encoding application on both devices which utilizes the EGL interface for efficient GPU-accelerated encoding. For the server side, we deploy *MobiEye* on a server running Ubuntu 16.04, with a 16-core, 2.4GHz Intel Xeon CPU, two NVIDIA GPU's (GeForce GTX 1080 and GeForce GTX TITAN), and 128GB RAM. Corresponding to the video encoding performed on the Android devices, we establish a video decoding application on the server with low-level ffmpeg and x264 libraries. For the DNN video frame inference module on the server side, we extend the existing *DFF* project with our optimizations from *MobiEye*. *DFF* itself is based on MXNet [1], a powerful deep learning framework developed by DMLC team. Figure 9 (a) shows the histogram of the motion value derived from ImageNet VID. For ease of visualization, all motion values larger than 10 are truncated. From the figure, it can be seen that the distribution of motion values is polarized: 29% of the video motion values are smaller than 1 while 23% of the values are larger than 9. The mean of the motion values is 6.83, and the median is 3.05. Figure 9 (b) shows the relationship between different image scales and their corresponding encoding/decoding time. Generally, Pixel 2 is faster than Nexus 5 for video encoding, which consumes 14.9*ms* and 22.8*ms* on average, respectively. When the image scale reaches 246K on the Pixel 2, the encoding time becomes shorter than when encoding lower-scale images. This is due to the highly-parallel nature of both the encoding operation and of the mobile GPU itself, resulting in more efficient computation when performed at scale. The video decoding time from server side is 3.9*ms* per frame on average.

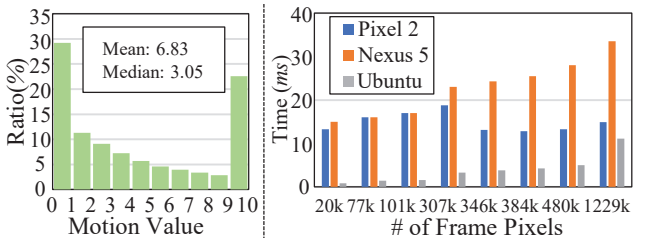

Figure 9: (a) Motion value histogram on ImageNet VID with H.264, and (b) encode/decode time on tested hardware.

6.2 Evaluation of ADFF

Figure 10 compares DNN inference time between original sequential *DFF* and our proposed *ADFF*. As indicated in Figure 10, the time consumption of each video frame always equals the time for inter frame in *ADFF*. We find that the inference time for key frame and inter frame are 93*ms* and 20*ms* for GTX TITAN, and 89*ms* and 19*ms* for GTX 1080. Given this, we deploy the key frame thread on GTX TITAN and the inter frame on GTX 1080, as the latter card results in lower latency for *ADFF*. In all cases, the time consumption per frame in the original sequential *DFF* baseline is much higher than *ADFF*, which is due to the bottleneck of key frame execution. Note that *delay* in *ADFF* is set as 3, and that *delay* should be no bigger than the key frame interval. Hence, Figure 10 shows the key frame interval from 6. From the dotted line in Figure 10, we find that *ADFF* achieves 1.05 – 1.47× and 1.1 – 1.5× speedup compared with sequential *DFF* on RTX 1080 and RTX TITAN, respectively.

6.3 Evaluation of VDS

Figure 11 presents the results of VDS scheme in *MobiEye*, where the gray dots show the baseline speed-accuracy tradeoff with a static key frame interval. For VDS, we try three max motion values: 10, 20, and 30, the results of which are colored as orange, yellow, and green. For each max motion value setting, we set the minimum key frame interval as 5, 10, 15 and max key frame interval as 20, 30, 40, forming totally 9 dynamic interval ranges. As shown in Figure 11, when the key frame interval is small (e.g., <12), VDS does not show an advantage compared to static key frame scheduling. The reason for this is that the dynamic interval range is small and thus there exists little optimization space for higher accuracy. The maximum accuracy that can be reached is the situation where all the video frames are considered as key frame. With the increase of key frame interval, we derive a higher accuracy than baseline when adopting VDS. For example, when the average key frame interval is 21.6, the mAP reaches 70.6%. Meanwhile, the mAP of static key frame

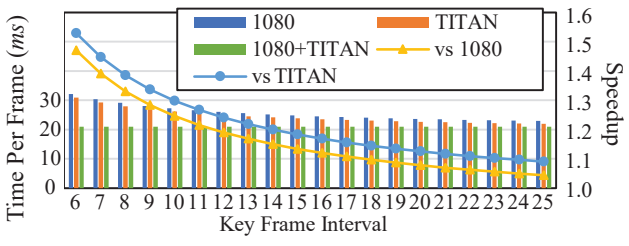

Figure 10: Computation time comparison between sequential and asynchronous execution of *DFF*.

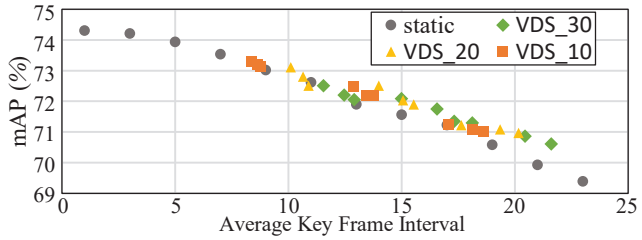

Figure 11: mAP under different average key frame interval with static scheme and VDS scheme.

scheduling only achieves 69.9% when the key frame interval is 21. Therefore, we find that VDS does help the prediction of key frame scheduling to achieve better accuracy with lower computation cost.

6.4 Evaluation of SSI

To evaluate SSI, we set three brightness error mask thresholds: 10, 20, and 30 to show the tradeoff between the spatial sparsity and accuracy. Figure 12 shows the sparsity-accuracy tradeoff when adopting SSI in *MobiEye*. The average spatial sparsity equals 52.8% when only applying the feedback computation mask and the mAP drop is controlled within 1% for all the key frame interval settings. Furthermore, the mAP drop is negligible (e.g. $< 0.5\%$) when the key frame interval is small (e.g.< 9). With the increase of the key frame interval, the spatial sparsity keeps decreasing because more computation is needed due to feedback computation mask. The reason lies in that larger key frame interval incurs larger movement between key frame and inter frame and thus the bounding box in the feedback computation mask is larger. As illustrated in Figure 12, the spatial sparsity increases with the increase of brightness error mask thresholds. Take the key frame interval 9 as an example, the 53%, 68%, and 75% for threshold 10, 20, and 30, respectively. Their corresponding mAP is 72.53%, 72.18%, and 71.85%, incurring 0.35% and 0.68% mAP drop. Therefore, we find that setting the brightness error mask threshold as a small number(e.g. 10) leads to a better sparsity-accuracy tradeoff. One limitation of SSI is that it could not achieve speedup when the input feature map is large in height and width. So, we apply SSI after 2 pooling operations in FlowNet. In our experiments, SSI achieves 1.06× – 4.25× speedup on convolutional layers of FlowNet with 60% – 90% spatial sparsity. Because FlowNet only contains 12 convolutional layers after 2 pooling operations, the impact of SSI on speedup of the whole inference operation is limited to 1.01× – 1.15× under 60% – 90% spatial sparsity due to operations such as region proposal network, deconvolution, etc.

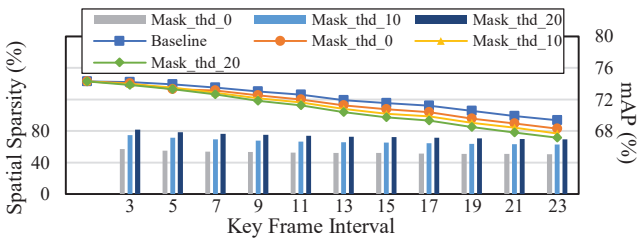

Figure 12: Sparsity-accuracy tradeoff with SSI.

6.5 Overall Evaluation of MobiEye

We evaluate *MobiEye* under the setting of key frame interval = 20, video scale = 800 × 600, network throughput = $10Mbps$, video size per frame = $47KB$, and H.264 Bitrate = $2Mbit/s$. The corresponding communication and codec latency sums up to $30ms$.

The baseline inference latency from the server side is $89.8ms$ for FeatNet and $19ms$ for FlowNet. After adopting *ADFF*, the execution of FeatNet is hidden so that the execution bottleneck becomes only FlowNet latency. With the same key frame interval, VDS increase the mAP by 0.6%. Finally, If we want to further exchange accuracy with computation cost, SSI saves another $0.2ms$ on FlowNet by sacrificing 0.6% mAP.

7 CONCLUSION

In this work, we propose *MobiEye*, an efficient cloud-based video detection system for real-time applications. *MobiEye* adopts the state-of-the-art video detection framework *DFF* with several optimizations. From system level, we utilize multi-thread technology to asynchronously execute the DNNs in *DFF* while insure the functionality correctness via thread lock. From algorithm level, we propose VDS to dynamically decide the key frame based on H.264 motion vector and design SSI for spatially partial inference based on both feedback detection results and brightness error. *MobiEye* is able to reach real-time requirement of video detection applications on mobile platforms with marginal accuracy drop.

8 ACKNOWLEDGMENT

This work was supported in part by NSF CNS-1717657, SPX-1725456 and DOE DE-SC0018064.

REFERENCES

[1] Tianqi Chen et al. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015).
[2] Misha Denil et al. 2013. Predicting parameters in deep learning. In *NIPS*. 2148–2156.
[3] Alexey Dosovitskiy et al. 2015. Flownet: Learning optical flow with convolutional networks. In *ICCV*. 2758–2766.
[4] Mikhail Figurnov et al. 2016. Perforatedcnns: Acceleration through elimination of redundant convolutions. In *NIPS*. 947–955.
[5] Song Han et al. 2015. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR* abs/1510.00149 (2015).
[6] J. Hauswald et al. 2014. A hybrid approach to offloading mobile image classification. 8375–8379. https://doi.org/10.1109/ICASSP.2014.6855235
[7] Kaiming He et al. 2016. Deep residual learning for image recognition. In *ICCV*. 770–778.
[8] Andrew G. Howard et al. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017).
[9] Alex Krizhevsky et al. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*. 1097–1105.
[10] Xiaoxiao Li et al. 2017. Not all pixels are equal: Difficulty-aware semantic segmentation via deep layer cascade. In *CVPR*. 2.
[11] Jiachen Mao et al. 2017. AdaLearner: An adaptive distributed mobile learning system for neural networks. In *ICCAD*. IEEE, 291–296.
[12] Jiachen Mao et al. 2017. Mednn: A distributed mobile system with enhanced partition and deployment for large-scale dnns. In *ICCAD*.
[13] Mengye Ren et al. 2018. SBNet: Sparse Blocks Network for Fast Inference. In *CVPR*. 8711–8720.
[14] Christian Szegedy et al. 2016. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. *CoRR* abs/1602.07261 (2016).
[15] H-YC Tourapis et al. 2003. Fast motion estimation within the H. 264 codec. In *ICME*, Vol. 3. IEEE, III–517.
[16] Wei Wen et al. 2016. Learning Structured Sparsity in Deep Neural Networks. In *NIPS*.
[17] Xizhou Zhu et al. 2017. Deep feature flow for video recognition. In *Proc. CVPR*, Vol. 2. 7.