Toward Efficient and Adaptive Design of Video Detection System with Deep Neural Networks

JIACHEN MAO, QING YANG, ANG LI, KENT W. NIXON, HAI LI, and YIRAN CHEN,
Duke University, United States

In the past decade, Deep Neural Networks (DNNs), e.g., Convolutional Neural Networks, achieved human-level performance in vision tasks such as object classification and detection. However, DNNs are known to be computationally expensive and thus hard to be deployed in real-time and edge applications. Many previous works have focused on DNN model compression to obtain smaller parameter sizes and consequently, less computational cost. Such methods, however, often introduce noticeable accuracy degradation. In this work, we optimize a state-of-the-art DNN-based video detection framework—Deep Feature Flow (DFF) from the cloud end using three proposed ideas. First, we propose Asynchronous DFF (ADFF) to asynchronously execute the neural networks. Second, we propose a Video-based Dynamic Scheduling (VDS) method that decides the detection frequency based on the magnitude of movement between video frames. Last, we propose Spatial Sparsity Inference, which only performs the inference on part of the video frame and thus reduces the computation cost. According to our experimental results, ADFF can reduce the bottleneck latency from 89 to 19 ms. VDS increases the detection accuracy by 0.6% mAP without increasing computation cost. And SSI further saves 0.2 ms with a 0.6% mAP degradation of detection accuracy.

CCS Concepts: • **Computing methodologies** → **Image and video acquisition**; *Parallel computing methodologies*; • **Computer systems organization** → *Real-time systems;*

Additional Key Words and Phrases: Video detection, neural networks, model pruning, embedded software, parallel computing, real-time system, mobile computing

ACM Reference format:
Jiachen Mao, Qing Yang, Ang Li, Kent W. Nixon, Hai Li, and Yiran Chen. 2022. Toward Efficient and Adaptive Design of Video Detection System with Deep Neural Networks. *ACM Trans. Embedd. Comput. Syst.* 21, 3, Article 33 (July 2022), 21 pages.
https://doi.org/10.1145/3484946

1 INTRODUCTION

1.1 Motivation

Deep Neural Networks (DNNs) are now widely utilized in many cognitive tasks, such as automated speech translation [1], natural language processing [2], object detection [3], facial recognition [4], and so on. In this work, we focus on the optimization of DNN-based video detection

This work was supported in part by Grants No. NSF CNS-1717657, No. CCF-1937435, No. CNS-1822085, and No. NSF IUCRC for ASIC memberships from Unisound, etc.
Authors' address: J. Mao, Q. Yang, A. Li, K. W. Nixon, H. Li, and Y. Chen, Duke University, 100 Science Drive, Durham, NC, 27708, United States; emails: {jiachen.mao, qing.yang21, ang.li630, kwn8, hai.li, yiran.chen}@duke.edu.

1539-9087/2022/07-ART33 $15.00
https://doi.org/10.1145/3484946

application. Video detection has been popularly adopted in the mobile context and aims at detecting the objects in continuous computer vision captured by mobile cameras for the purposes such as video analytics. Video detection is one of the most computationally demanding DNN applications because of many reasons. First, compared to language processing, the input (image) size of video detection system is usually larger. Second, convolutional layers are commonly used in video detection and are more computation-intensive than other DNN layers. Last, compared to object detection, video detection involves a sequence of video frames to be detected in real-time. Many attempts have been made to accelerate DNN-based video detection, which can be categorized into *model compression* and *system optimization*. *Model compression* tries to find a compact model by removing weights from the over-parameterized DNNs. Representative methods include structure search [5–7], weight pruning [8], low rank approximation [9], and quantization [10]. Another solution is *system optimization*, which accelerates the DNN-based system through application-oriented methods. In Reference [11], Hauswald et al. divide the computation pipeline of computer vision tasks into a pipeline and examine the tradeoff between different workload partition schemes on local mobile devices and outside server; in Reference [12], Zhu et al. utilize the motion data from **Image Signal Processor (ISP)** to relax the number of expensive **Convolutional Neural Networks (CNN)** inferences. Because *system optimization* focuses on specific applications (e.g., video detection), it can obtain better performance, in general, than *model compression*.

1.2 Contributions

Our work falls in the category of *system optimization*. The contribution of our work can be summarized as follows:

- We adopt ***Deep Feature Flow (DFF)*** as video detection framework and utilize ***Asynchronous DFF (ADFF)*** to parallelize the execution of *DFF*;
- We propose ***Video-based Dynamic Scheduling (VDS)*** scheme, which utilizes motion vectors in video decoding procedure to dynamically adjust the inference frequency for adaptive computing;
- We propose ***Spatial Sparsity Inference (SSI)*** to further accelerate the inference time for each video frame base on either dynamic or static computation masks;
- We dig into the implementation details of *SSI* on GPU and compare *SSI* with two related spatial sparsity works;
- We implement our proposed system optimizations on both mobile and server side and evaluate them in detail.

1.3 Organization of the Article

The remainder of our article is organized as follows: In Section 2, we first give a brief preliminary of DNN profiling results on mobile devices, which show the necessity of executing video detection with the help of cloud. Then, we describe the motion vector in video codec, which is tightly related to our proposed optimization methods. In Section 3, we present the framework of the DNN-based detection system along with our optimization using asynchronous computing. In Section 4, we discuss our proposed dynamic scheduling method based on the motion vector, which can adaptively choose the inference frequency. Section 5 presents our optimization, which dynamically executes a spatial part of the input video frame. Extended from our preliminary work [13], we also implement the proposed methods on GPU and quantitatively analyze the pros and cons of the proposed spatial sparsity inference. In Section 6, we set up and conduct experiments on our video detection system and show how each proposed design optimizes the whole system. In Section 7, we

Table 1. Profiling of Two 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%	17 MB	224×224
Inception-v4	3180 ms	611 ms	80.2%	171 MB	229×229

summarize the related system optimization works on DNN-based video detection and their difference from our work. Section 8 concludes our article.

2 PRELIMINARY

2.1 DNN Model Profiling on Mobile Devices

Although DNNs can achieve state-of-the-art accuracy when running on high-performance platforms [14–16], the low availability of computing resources on embedded platforms limits accuracy and/or viability of the DNNs in those contexts [17–19]. Table 1 illustrates the inference time of two representative DNNs [20, 21] when running on flagship smartphones, as well as their corresponding Top-1 accuracy on the ImageNet dataset [22]. Those data are gathered from the official document of TensorFlow [23], which is a popular toolkit for embedded platforms. In Table 1, MobileNet [20] is an efficient network structure designed for mobile devices, while Inception-v4 [21] represents state-of-the-art accuracy. It can be seen that even the highly optimized MobileNet still cost a long inference latency, leading to 6fps on Pixel 2. Note that the inference times in Table 1 are measured from image classification tasks, the input image resolution of which is around 200×200. While the typical input image resolution of video detection is about $600 \times 1{,}000$, the corresponding computation cost will increase by more than 10 times. Because deploying large-scale video detection systems on local mobile devices suffers from overwhelming computation and energy costs, our detection system targets at a cloud computing setup to achieve both high accuracy and low latency.

2.2 Motion Vector in H.264 Video Codec

Considering the limitation of network bandwidth, video contents captured from mobile camera are compressed before being distributed. H.264 is a widely used video format, which is also known as MPEG-4 Part 10, Advanced Video Coding [24]. One key reason of the fierce compression rate of H.264 is its utilization of *motion compensation*. During encoding, H.264 protocol gathers certain successive video frames in a single group, which is named as **Group of Pictures (GoP)**. Each GoP is made up of one I frame ("I" for Intra) and multiple P and B frames ("P" for Prediction, "B" for Bidirectional). I frame independently encodes a complete frame, which serves as the reference point for P and B frames in the same GoP. P frames are predicted by analyzing the difference between themselves and the previous P frames or I frames. Similar to P frames, B frames are also predicted by the frame difference except that they are compared with the later frames. Hence, P and B frames store less data with high resolution. The video frames are divided into several macroblocks, serving as the basic unit for predicting the frame difference. The difference between frames is described by *motion vector*, including the source and destination of all macroblocks. We adopt H.264 codec in the communication between mobile and cloud under wireless networks such as Wi-Fi or LTE networks.

3 SYSTEM FRAMEWORK

Figure 1 shows an overview of our DNN-based video detection system. The upper part of Figure 1 represents the edge users, which capture the frames, encode them to video, transmit them to the cloud, and wait for the final detection result. The bottom part denotes the execution flow of the cloud. The received video is first decoded and then fed to the DNN-based detector. Figure 1 also depicts the position where we add our three system-level optimization methods:

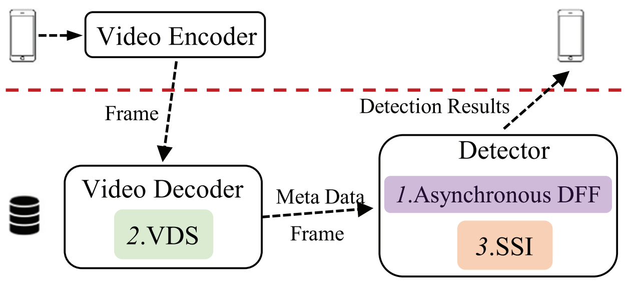

Fig. 1. Overview of our video detection system.

(1) **Asynchronous Deep Feature Flow (ADFF):** A system optimization of *DFF* video detection framework [25], which uses multi-threading to asynchronously execute the DNNs.

(2) **Video-based Dynamic Scheduling (VDS):** A system optimization that dynamically adjusts key-frame detection frequency based on the metadata (motion vector) that exists in H.264 encoded video. The video decoder component extracts the metadata and sends them to the detector.

(3) **Spatial Sparsity Inference (SSI):** A DNN-related system optimization which accelerates DNN inference by focusing on only visual saliency areas. Specifically, SSI accelerates DNN by re-implementing the convolutional layers so that only the designated spatial positions are convoluted.

3.1 Deep Feature Flow (DFF)

We adopt *DFF* [25] as the video detection framework. The essence of *DFF* lies in that it divides video frames into two kinds: key frames and inter frames. Key frames are detected using a traditional object detection DNN, i.e., *R-FCN*, which consists of a feature extractor network, **Region Proposal Network (RPN)**, and classification network. For the detection of inter frames, *DFF* compares these frames with previous key frame using a smaller optical flow network and then generates their feature map based on the interpolation from the cached key-frame feature map. According to the default setting of *DFF*, a key frame occurs every 10 frames, while the other 9 frames are inter frames. The feature extractor network is substituted by an optical flow network for all the inter frames, saving much for inference time per frame. Figure 2 shows the execution flow of *DFF*. The red arrow shows the execution flow of the key frame, while the green arrow shows the execution flow of the inter frames. In general, *DFF* includes four network components:

- **Feature Extraction Network (FeatNet)**: Feature extraction network extracts the high-level features from raw images through stacks of convolutional, activation, and pooling layers. FeatNet is the most computation-intensive component in *DFF*.
- Optical Flow Network (FlowNet): Optical flow network computes the optical flow between the previous key frame and themselves for the detection of inter frames. To enable end-to-end training, such an optical flow is also realized by DNN called FlowNet [26].
- Propagation Function (Propagation): Propagation function takes the optical flow between the key frame and the inter frames as well as the feature map of the key frame as input. Then, propagation function interpolates the feature map of the key frame using optical flow via bi-linear interpolation.
- Region Proposal Network (RPN) and Classification Network: Region proposal network is applied on the feature map of each frame to locate the potential areas of the objectiveness in the frames. The candidate **Regions of Interest (RoIs)** are then fed into the classification network to get the final object category by pooling and voting.

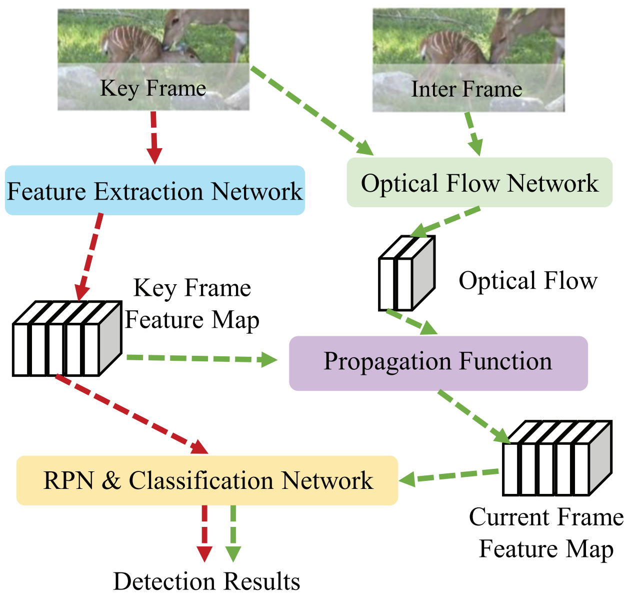

Fig. 2. Execution flow in *DFF* detection framework [25].

ALGORITHM 1: Original Computation Procedure in *DFF*.

```
1: Init Keyframe Interval: $i$
2: for each Current Frame $f_{cur}$ with Index $idx$ do
3:     if $f_{cur}$ is keyframe then
4:         $Dets, feat_{key}$ = FeatNet($f_{cur}$)
5:     else if $f_{cur}$ is interframe then
6:         $Dets$ = FlowNet($f_{cur}$, $f_{key}$,$feat_{key}$)
7:     end if
8: end for
```

3.2 Asynchronous DFF (ADFF)

One limitation of *DFF* is that although it achieves a high frame rate on average, the inference time varies depending on whether the current frame is a key frame or an inter frame. Due to the different networks utilized by each frame type, when running the original *DFF* code on GTX 1080 GPU, the detection latency of the key frames is 89 ms, while that of the inter frames is 19 s. If the key frame interval is set to 7 frames, then the first key frame detection latency is 89 ms followed by 6 inter frames that consume 19 ms per frame. Such sequential *DFF* is described in Algorithm 1. It leads to unbalanced inference time of the frames that is not desirable in real-time applications.

We propose to optimize the system by executing FeatNet and FlowNet asynchronously. Two threads are utilized here: one to execute key-frame inference and the other to execute inter-frame inference. Figure 3 compares the original sequential *DFF* (top) with the proposed *ADFF* (bottom) when the key-frame interval is 7. In the sequential *DFF*, the 1st and the 8th frames take 0.6 s, respectively, while asynchronous execution allows for an inference time of 0.2 s for all the frames.

Before the current key-frame feature map is generated, every inter frame uses the feature map of the last key frame and thus asynchronously gets the detection results. By performing computation asynchronously, detection latency only depends on the execution of FlowNet and video

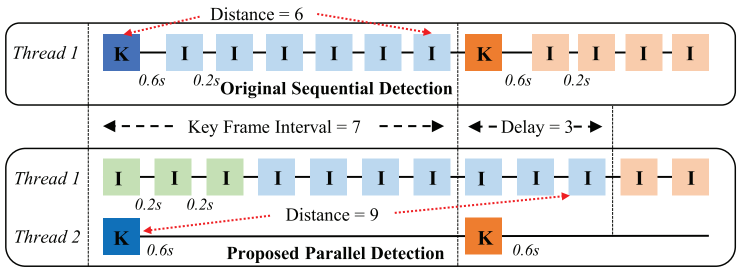

Fig. 3. Original sequential *DFF* (top) vs. proposed asynchronous execution *ADFF* (bottom).

transmission. As shown in the bottom of Figure 3, the first frame and the eighth frame are the key frame and inter frame for both two parallel threads. In the example of Figure 3, the feature map delay is 3 frames as FeatNet is nearly 3× slower then FlowNet. 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 D_f is the feature map delay in terms of frame count. In practice, D_f may be even smaller because of the communication overhead.

The details of *ADFF* are illustrated in Algorithm 2. We can tell that the correctness of multi-thread execution is assured by utilizing a mutex to indicate when the current key-frame feature map is available. Concretely, if the current frame to be detected is key frame, then a new thread will be created to execute the FeatNet. Once the thread is created, it will acquire a thread lock to prevent the later inter frames from getting the wrong key-frame feature map. In the mean time,

ALGORITHM 2: *Asynchronous Deep Feature Flow* (*ADFF*).

```
1:  KeyframeThread (FeatNet, $f_{cur}$):
2:          _ , $feat_{key}^{temp}$ = FeatNet($f_{cur}$), $threadLock$.acquire()
3:          $feat_{key} = feat_{key}^{temp}$, $threadLock$.release()
4:  Init Thread Lock: $threadLock$, Keyframe Interval: $i$, Delay: $d$
5:  for each Current Frame $f_{cur}$ with Index $idx$ do
6:      if $f_{cur}$ is the first frame in video then
7:          $Dets, feat_{key}$ = FeatNet($f_{cur}$)
8:      else
9:          if $f_{cur}$ is keyframe then
10:             $thread$ = new KeyframeThread()
11:             $thread$.start(FeatNet, $f_{cur}$),
12:             $threadLock$.acquire()
13:         else if $idx \% i == d$ then $thread$.join()
14:         else if $idx \% i == d - 1$ then $threadLock$.release()
15:         end if
16:         $Dets$ = FlowNet($f_{cur}$, $f_{key}$,$feat_{key}$)
17:     end if
18: end for
```

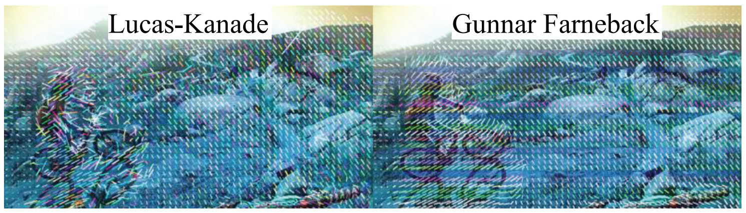

Fig. 4. Optical flow using Lucas Kanade (left) and Gunnar Farneback (right).

the current frame is also treated as an inter frame to get the detection results immediately. When the FeatNet of the newest key frame is generated, as shown in line 3 of Algorithm 2, the thread lock will be released so that the subsequent inter frames acquire the lock to compare themselves with the newest feature map.

Another difference between sequential and asynchronous *DFF* is that, to reach the same accuracy, asynchronous *DFF* requires a smaller key-frame interval. This is due to an increased interval between key frames and the inter frames that utilize their feature maps, and it is an unavoidable result of D_f. For example, the key-frame interval of sequential *DFF* in Figure 3 is 6, while it is 9 for *ADFF*. In our experiments, we find that two threads work better when they are deployed on different GPUs than on a single GPU. The reason is that if GPU memory is shared by two threads executing two networks, the total number of GPU kernels for each thread is dynamically changed, resulting in the fluctuation of the inference time. If we execute two threads on two separated GPUs, then the inter-frame inference time will not be negatively affected by the parallel execution of key frame.

4 VIDEO-BASED DYNAMIC SCHEDULING

As part of our detection system, we target the elimination of redundant calculations within the video frames. Our system attempts such a goal by dividing the input frames into key frames and inter frames, only performing higher-cost inference on key frames when necessary. In original *DFF*, key frames are selected at a predetermined interval, regardless of the underlying video data [25]. 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 expensive key-frame inference calculations. In our system, we design a *VDS* scheme to dynamically determine whether the current frame is a key frame or not.

4.1 Motion Vector and Optical Flow

The idea of *VDS* is inspired by the similarity between motion vector and optical flow. Motion vectors in H.264 work in the discrete domain, representing the displacement of sub-blocks in the video frame. Optical flows are the motions of image brightness in the continuous domain, which are caused by brightness change. In two-dimensional images, optical flow is the same as motion vector in the ideal case [27].

One straightforward idea is directly substituting motion vector in H.264 for the FlowNet in our system. However, the resolutions of motion vectors of different video frames in H.264 are not identical. Moreover, the accuracy of the H.264 motion vectors is much lower than that of the FlowNet, which prevents us from using the motion vectors directly. Additionally, we also tried to replace FlowNet with a more light-weighted optical realization such as Lucas Kanade optical flow [28] or Gunnar Farneback optical flow [29]. As can be seen in Figure 4, the accuracy of Lucas

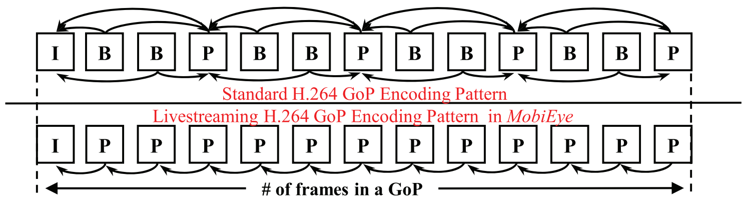
I B B P B B P B B P B B P
Standard H.264 GoP Encoding Pattern
Livestreaming H.264 GoP Encoding Pattern in MobiEye
I P P P P P P P P P P P P
# of frames in a GoP

Fig. 5. Standard H.264 GoP pattern (top) and adopted H.264 GoP pattern in our system (bottom).

Kanade optical flow is worse than that of Gunnar Farneback optical flow although Lucas Kanade method is faster. Even if we utilize Gunnar Farneback optical flow to replace original FlowNet, the **Mean Average Precision (mAP)** still drops by 9.6% when the frame interval is set to 10, which is unacceptable.

Although we could not directly utilize the motion vector, we leverage it to evaluate the displacement between video frames for dynamically scheduling the frame interval for inference. Figure 5 shows how we encode the videos with H.264 format to be uploaded to the server (bottom). Comparing with the original H.264 video encoding pattern in the top of Figure 5, the encoding pattern in our real-time video detection system discards all the B frames. Moreover, the number of reference frames is set to 1 so that all P frames motion vectors are calculated based on a single prior frame. In such a way, the motion value means the displacement between frame pairs.

4.2 Video-based Dynamic Scheduling (VDS)

In *VDS*, we pre-define a frame interval range and a motion vector range. The current frame interval is dynamically assigned based on the current motion vector magnitude of the current frame. 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. Till now, *VDS* only considers the motion between the current frame and its previous frame. To consider the video change over a sequence of video frames, we define the accumulated motion vector mv_{acc} to denote the total motion from the last key frame until the current frame. As depicted in lines 7 and 8 of Algorithm 3, when the magnitude of the accumulated motion vector mv_{acc} is larger than mv_{max}, the minimum frame interval (fi_{min}) will be assigned to the current frame interval (fi_{cur}).

ALGORITHM 3: *Video-based Dynamic Scheduling* (*VDS*).

```
1:  Init Dynamic key-frame interval range : $fi_{min}, fi_{max}$
2:  Init Motion vector value range : $mv_{min}, mv_{max}$
3:  Init Accumulated motion value : $mv_{acc} = 0$
4:  for each new frame motion vector $mv_{cur}$ from key frame do
5:      $mv_{mean} = \text{Mean}(\text{Abs}(mv_{cur}))$
6:      $mv_{acc} += mv_{mean}$
7:      if $mv_{acc} > mv_{max}$ then
8:          $fi_{cur} = fi_{min}$
9:      else
10:         $fi_{cur} = fi_{min} + (fi_{max} - fi_{min}) * \frac{mv_{max} - mv_{mean}}{mv_{max} - mv_{min}}$
11:     end if
12: end for
```

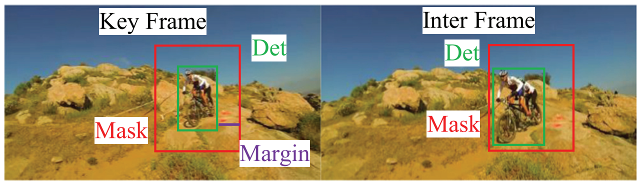

Fig. 6. Feedback computation mask.

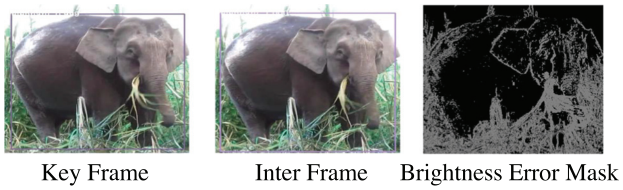

Fig. 7. Brightness error computation mask.

We can find that the only computation in *VDS* is vector summation and averaging, incurring very marginal computation cost compared with DNN inference. Furthermore, unlike [30], *VDS* does not require extra learning procedures. By adopting *VDS*, the FeatNet is frequently executed when the movement in the video is fast and rarely executed when the movement is slow.

5 INFERENCE WITH SPATIAL SPARSITY

5.1 Spatial Sparsity Inference (SSI)

In Section 4, temporal redundancy is eliminated via *VDS*. In this section, *Spatial Sparsity Inference* (*SSI*) is proposed for removal of spacial redundancy in videos. *SSI* skips unimportant pixels to accelerate DNN execution. In *SSI*, we design two computation masks to denote the unimportant pixels 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, as background movement would not affect final detection results. Figure 6 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 a constant size. Here, we set the margin as 64, because in FlowNet, the ratio between the spatial size of the feature map before the first convolution and the smallest feature map after convolution is 64:1. The ratio can be inferred from Figure 11, which is $w/2$ or $w/2$ divided by $w/128$ or $h/128$. (2) **Computation mask based on brightness error:** Feedback computation mask reduces the spatial redundancy of the frame background. However, they cannot be applied when feedback detection results cover the entire video frame. To deal with such a case, brightness error computation mask is designed, which is defined as the subtraction between inter-frame brightness (B_{inter}) and key-frame brightness (B_{key}):

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

where *thd* stands for the threshold of the brightness error to be skipped. Figure 7 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 to be re-calculated even though it falls within the detection bounding box.

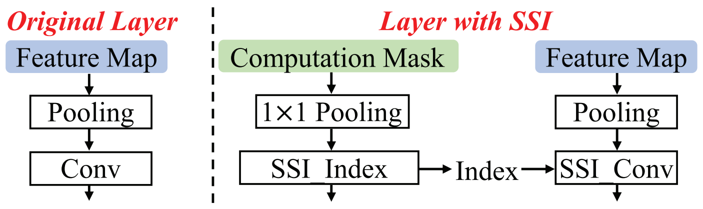

Fig. 8. Original layers (left) and layers with *SSI* functionality (right).

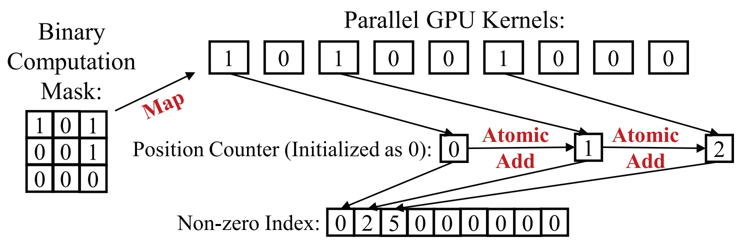

Fig. 9. Realization of SSI index operation on GPU.

5.2 Realization of SSI

Figure 8 presents the original layers (left) and their corresponding SSI layers (right). Compared with the original layers with feature map as input, SSI layers are fed with the feature map and the computation mask as described in Section 5.1. To deal with the scale change in different DNN layers, we borrow the idea from SBNet [31], 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 SSI_Conv and SSI_Index Layer. Because *SSI* belongs to structured sparsity, it can better accelerate DNN execution with fully optimized, dense GEMM in both CPU and GPU modes.

5.3 Realization of SSI Index Layer

To skip the designated pixels, we need to first get the indices of the pixels to be skipped, which is realized by SSI index layer. SSI index layer converts a binary computation mask to a non-zero index array as well as the total size of non-zero numbers. Figure 9 depicts how to realize an efficient SSI index operation on GPU in parallel. Each position of the binary computation mask is assigned to one GPU kernel for parallel execution. Then, for each GPU kernel, it decides whether the assigned number is 0 or 1. If the assigned number is 0, then the kernel will end its execution thread. Otherwise (the assigned number is 1), the kernel will first do an atomic addition to the global position counter. Atomic addition assures the correctness of parallel execution of SSI index operation so that two GPU kernels will not access and update the same position in the output non-zero index array. After that, the thread will get the old value before the atomic addition, indicating the position that can be filled for the current GPU kernel. Last, such a GPU kernel will update the position in the non-zero index array with the indices of the non-zero numbers in the original computation mask. As shown in the example of Figure 9, the final non-zero index array contains 0, 2, and 5, which are the position indices of the input binary computation mask.

5.4 Realization of SSI Convolution Layer

Figure 10 details the procedure of the SSI_Conv operation. Before the matrix-by-matrix multiplication in step 2, the input feature map is first expanded from a three-dimensional tensor to a

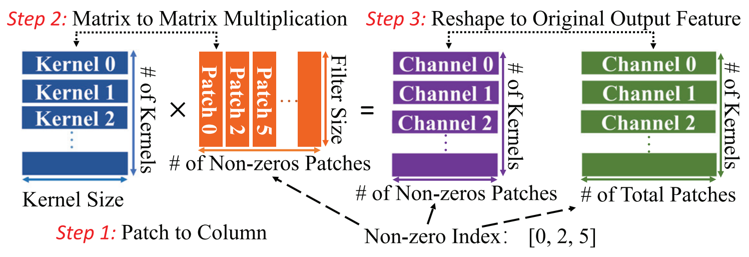

Fig. 10. Procedures of SSI convolution operation.

two-dimensional tensor with a patch to column operation. Patch here means a subset of feature map that is of the same dimension of the convolution kernels. Originally, each patch in the feature map is flattened to a single matrix column iteratively along the x axis and y axis. In SSI_Conv, patch to column operation takes the non-zero indices from the SSI_Index layer as input and iteratively assigns the columns to GPU kernels based on the patches with the non-zero indices. Comparing to the original patch to column implementation, the output matrix is smaller with fewer columns, leading to less memory consumption. Take step 1 in Figure 10 as an example, the number of the output matrix column is 3, because the non-zero index only includes three indices (0,2,5). Consequently, step 2 in Figure 10 does the matrix multiplication with the smaller matrix. Step 3 is an extra step for SSI_Conv operation, which expands the output matrix to the original size based on the non-zero indices.

5.5 Applying *SSI* on FlowNet

After applying *ADFF*, the latency bottleneck becomes the inference time of FlowNet. Thus, *SSI* is applied on FlowNet to further accelerate the inference latency. The detailed network architecture of FlowNet is shown in Figure 11. The input of FlowNet is a frame with height (h) and width (w), which is first compressed by half in both dimensions. There are six convolution groups, each of which decreases the input feature map side length by half. There is one convolution operation in the groups of *Conv1*, *Conv2*, while there are two convolution operations in the groups of *Conv3*, *Conv4*, *Conv5*, *Conv6*. After six convolution groups, the output feature map dimension becomes $h/128 \times w/128$, which is fed into the refinement layers. The refinement layers generate the optical flow with the spatial dimension of $h/16 \times w/16$ with two channels, representing the displacement of two frames from the x axis and y axis. As indicated in Figure 9, SSI index operation contains atomic addition, which needs to be executed sequentially even on GPU. The time consumption of SSI index operation increases quadratically with the increase of computation mask spatial scale. As a result, applying *SSI* on *Conv1* and *Conv2* does not achieve speedup in practice. Therefore, *SSI* is applied after two convolution groups.

5.6 Comparison of *SSI* with Related Works

The realization of *SSI* is jointly inspired from *PerforatedCNNs* [32] and *SBNet* [31], with necessary modifications. The reason why we combine the ideas of *PerforatedCNNs* and *SBNet* in *SSI* is due to their functional limitations when taken separately:

- *PerforatedCNNs* supports pixel-wise skipping but the skip index is static with the same computation mask for all input images. *PerforatedCNNs* could not be directly applied to our system, because in our system, the computation mask is dynamically changing.
- *SBNet* supports dynamic computation mask but only realizes convolution speedup in block-wise situations. However, *SBNet* could not achieve speedup with a small scale feature map.

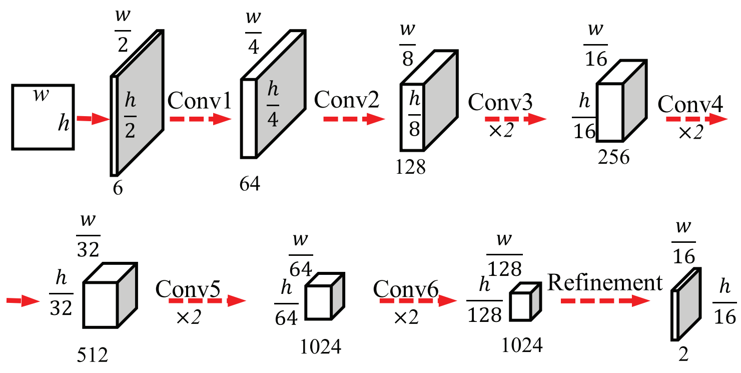

Fig. 11. Network architecture of FlowNet in *DFF*.

This is due to the computation overhead incurred by *Gather* and *Scatter* operations when the feature map is of low dimension and large channel size.

Therefore, *SSI* adopts the dynamic computation mask method from *SBNet* as well as the efficient convolution from *PerforatedCNNs*, yielding an efficient convolution inference with spatial sparsity.

6 EXPERIMENTS

6.1 Experimental Setup

Test Benches: We implement our video detection system using the *DFF* [25] framework, with all proposed optimization schemes. *DFF* also serves as the baseline. For the fairness of comparison, we directly utilize the trained models from *DFF* without fine-tuning in all experiments. We adopt the ImageNet VID dataset for evaluation, which includes 5,354 annotated videos. The model structures adopted for the FeatNet and FlowNet are ResNet101 [33] and FlowNet [26], respectively. Video frames are resized to 600 pixels on the shorter side as the input of FeatNet, and 300 pixels on the shorter side for FlowNet. In our experiments, the accuracy-related performance of the video detection system is reported using mean average precision (mAP), speed-related performance is evaluated in terms of sparsity and milliseconds (ms).

System Environment: For the client side, we adopt the Nexus 5 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 2 GB 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 an H.264 video encoding application on both devices, which utilizes the EGL interface for efficient GPU-accelerated encoding. For the server side, we deploy our system on a server running Ubuntu 16.04, with a 16-core, 2.4 GHz Intel Xeon CPU, two NVIDIA GPUs (GeForce GTX 1080 and GeForce GTX TITAN), and 128 GB 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 ×264 libraries. For the DNN video frame inference module on the server side, we extend the existing *DFF* project with our optimizations on MXNet [34], a powerful deep learning framework developed by DMLC team. Figure 12(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 12(b) shows the relationship between different image scales and their corresponding encoding/decoding time. Generally, Pixel 2 is faster than Nexus 5 for

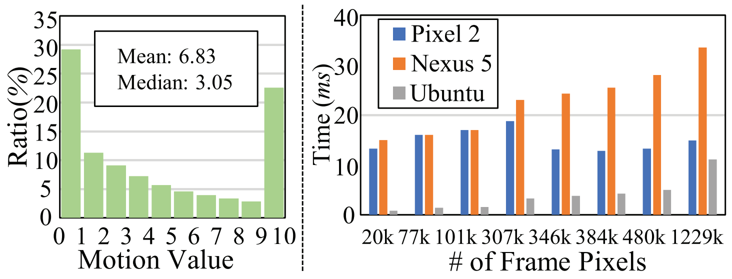

Fig. 12. (a) Motion value histogram on ImageNet VID with H.264 and (b) encode/decode time on tested hardware.

Table 2. Ablation Study of Proposed Optimization Methods

	Key frame	Inter frame	Ave FPS	mAP
DFF [25]	89.8 ms	19 ms	44.37	69.92
ADFF	19 ms	19 ms	52.63	69.92
DFF+VDS	89.8	19 ms	44.37	**70.56**
DFF+SSI	89.8	18.8 ms	44.74	69.32
ADFF+VDS+SSI	**18.8 ms**	**18.8 ms**	**53.19**	69.92

video encoding, which consume 14.9 and 22.8 ms on average, respectively. When the image scale reaches 246K on Pixel 2, the encoding time becomes shorter than that of lower-scale images. This is due to the highly parallel nature of both the encoding operation and the mobile GPU. The video decoding time from server side is 3.9 ms per frame on average.

6.2 Overall Evaluation

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

Table 2 demonstrates the ablation study about the inference latency of key frame, inter frame, the average FPS, and the performance with or without our proposed methods. The baseline inference latency from the server side is 89.8 ms for key frame and 19 ms for inter frame. After adopting *ADFF*, the execution of FeatNet is hidden so that the execution bottleneck becomes only FlowNet latency. In such a case, the inference latency becomes 19 ms for both key frame and inter frame. When combining *DFF* with *VDS*, a higher mAP is realized under the same frame interval because of the content-based scheduling. When combining *DFF* with *SSI*, the inference latency of inter frame is further shortened by sacrificing 0.6% mAP The average time saving of 0.2 ms is derived from both SSI index and SSI convolution. If the computation mask is asynchronously generated, then *SSI* will achieve a higher speedup. As shown in the last line of Table 2, when combining all the methods, the inference FPS achieves 53.19 compared to the original 44.37 under the same mAP.

6.3 Evaluation of Communication Cost

Figure 13 illustrates the average communication time per frame of transmitting the H.264 encoded videos of different resolutions under different uploading bandwidths. The uploading bandwidth we choose in Figure 13 is from 6 to 21 Mbps, which is the common uploading bandwidth range in LTE standard [35]. From Figure 13, we find that H.264 could fiercely compress the video frame with high resolution: For 1,280×960 video resolution, the average frame size can be compressed down to

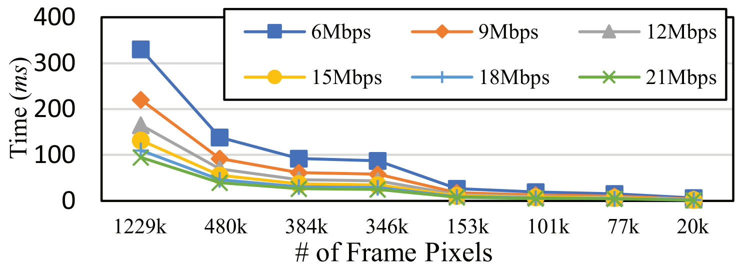

Fig. 13. Communication time per frame under different bandwidths and resolutions.

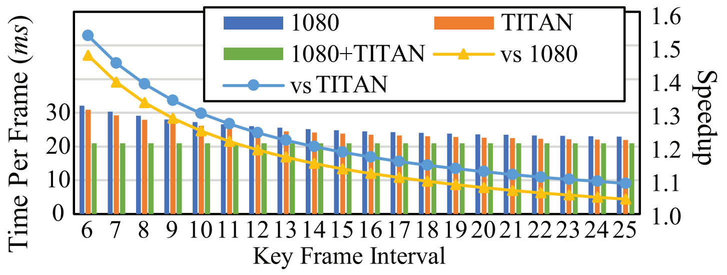

Fig. 14. Computation time comparison between sequential *DFF* and *ADFF*.

1.98 Mb from an original raw data of 9.83 Mb. The communication latency varies under different uploading bandwidths. For the 1,280 × 960 video resolution, it takes 330 ms to communicate under 6 Mbps bandwidth while 94 ms under 21 Mbps bandwidth. Because the communication latency is roughly proportional to image resolution, different resolutions could be used for different user requirement under different bandwidths.

6.4 Evaluation of ADFF

Figure 14 compares DNN inference time between original sequential *DFF* and our proposed *ADFF*. As indicated in Figure 14, the inference time of each video frame always equals the one of the inter frame in *ADFF*. We find that the inference time for key frame and inter frame are 93 and 20 ms for GTX TITAN, and 89 and 19 ms for GTX 1080, respectively. Based on this observation, we deploy the key-frame thread on GTX TITAN and the inter-frame thread on GTX 1080, as the execution time of inter frame is the bottleneck latency in *ADFF*. The inference time per frame in the original sequential *DFF* baseline is always much higher than that of *ADFF*, which is due to the bottleneck of key-frame execution. Note that the *delay* in *ADFF* is set as 3, and that *delay* should be no bigger than the key-frame interval. Hence, Figure 14 shows the key-frame interval from 6 to demonstrate the advantage of *ADFF*. As the dotted line in Figure 14 shows, *ADFF* achieves 1.05–1.47× and 1.1–1.5× speedup compared with sequential *DFF* on GTX 1080 and GTX TITAN, respectively. Another benefit of *ADFF* is its uniform frame timings, whereas sequential *DFF* is highly fluctuating. It will tremendously improve user experience.

6.5 Evaluation of VDS

Figure 15 presents the results of VDS, where the gray dots show the baseline speed-accuracy trade-off 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

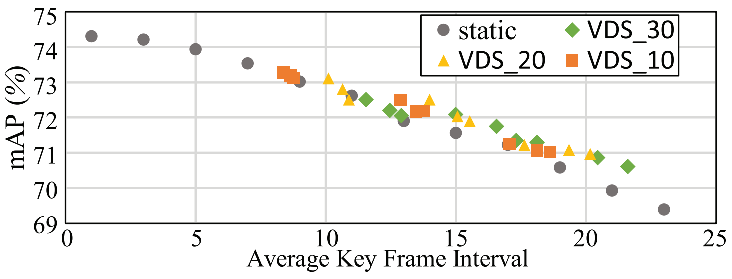

Fig. 15. mAP under different average key-frame interval with static scheme and VDS scheme.

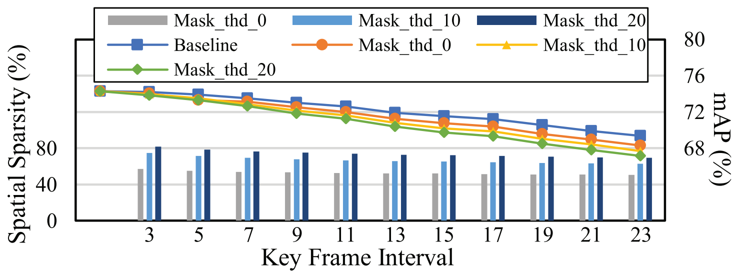

Fig. 16. Sparsity-accuracy tradeoff with *SSI*.

set the minimum key-frame interval as 5, 10, 15 and max key-frame interval as 20, 30, 40, forming nine dynamic interval ranges in total. As shown in Figure 15, when the key-frame interval is small (e.g., <12), VDS does not show an advantage compared to static key-frame scheduling. It is because the dynamic interval range is small and thus there exists little optimization space for higher accuracy. 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 scheduling only achieves 69.9% when the key-frame interval is 21. Therefore, VDS does help the prediction of the key frame, achieving a better accuracy with less computation.

6.6 Sparsity-accuracy Tradeoff of *SSI*

We set three brightness thresholds: 0, 10, and 20 to demonstrate sparsity-accuracy tradeoff of *SSI*. Figure 16 shows the sparsity-accuracy tradeoff when adopting *SSI* in our system. The average spatial sparsity equals 52.8% when only applying the feedback computation mask. 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. The reason lies in that a larger key-frame interval incurs larger movement between the key frame and the inter frames and thus the bounding box area in the feedback computation mask becomes larger. As illustrated in Figure 16, the spatial sparsity increases with the increase of brightness error mask thresholds. Take the key-frame interval 9 as an example, the sparsities are 53%, 68%, and 75% for the thresholds 0, 10, and 20, respectively. Their corresponding mAP results are 72.53%, 72.18%, and 71.85%. 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 two pooling operations in FlowNet. In our experiments, *SSI*

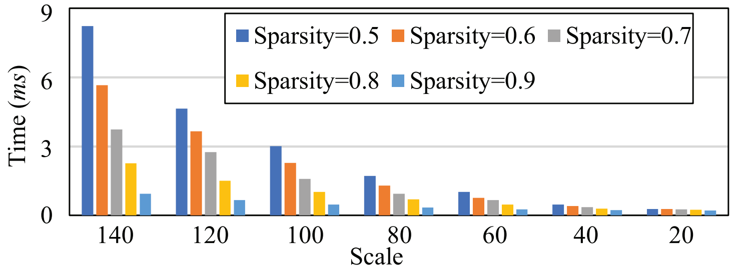

Fig. 17. Speedup of SSI index operation.

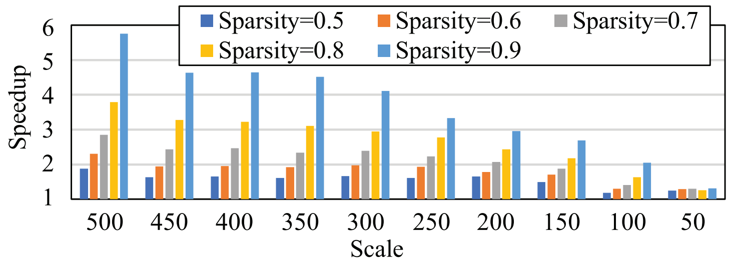

Fig. 18. Speedup of SSI convolution operation.

achieves 1.06×–4.25× speedup on convolutional layers of FlowNet with 60%–90% spatial sparsity. Because FlowNet only contains 12 convolutional layers after two 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. The rest operations such as RPN, deconvolution, and so on, were not accelerated.

6.7 Speed Evaluation of SSI Index

Figure 17 depicts the time consumption of SSI index operation with different scales of computation mask and spatial sparsities using GeForce GTX 1080. As illustrated in Figure 17, the time consumption of SSI index operation reduces with the sparsity and increases with the scale of the input image. Here, the scale of the input image denotes the side length of the input frame. In general, the time consumption of SSI index operation is proportional to the number of non-zeros in the computation mask, which is due to the atomic addition in SSI index operation. For example, the time consumption of SSI index operation reaches 0.94 ms when the computation mask is 80×80 with 70% sparsity. Such a long operation time is unacceptable, because the standard convolution operation just costs about 1 ms on GPU. To get rid of the computational overhead of SSI index operation in the FlowNet inference, *SSI* is not adopted for the first two layers that have large-scale computation masks.

6.8 Speed Evaluation of SSI Convolution

Figure 18 demonstrates the speedup of SSI convolution operation using GeForce GTX 1080. Figure 18 is derived from running standalone layer-wise speedup tests, where the layer setting includes $3 \times 3 \times 128 \times 128$ convolution kernels with 1 padding and stride. The implementation details are similar to *PerforatedCNNs*. So, in our video detection system, if the computation mask is static or pre-defined, the speedup in Figure 18 is the final speedup that *SSI* could achieve. *SBNet* claims

Table 3. Configuration of Convolution Layers in FlowNet

	Input Dim	Output Dim	Kernel	Stride	Filter Num
Conv1	300 × 500	150 × 250	7	2	64
Conv2	150 × 250	75 × 125	5	2	128
Conv3	75 × 125	38 × 63	5	2	256
Conv3_1	38 × 63	38 × 63	3	1	256
Conv4	38 × 63	19 × 32	3	2	512
Conv4_1	19 × 32	19 × 32	3	1	512
Conv5	19 × 32	10 × 16	3	2	512
Conv5_1	10 × 16	10 × 16	3	1	512
Conv6	10 × 16	5 × 8	3	2	1,024
Conv6_1	5 × 8	5 × 8	3	1	1,024

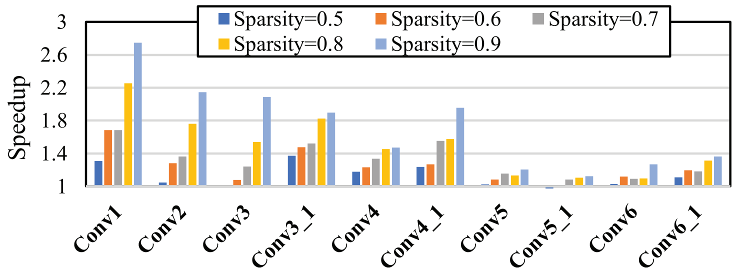

Fig. 19. Layer-wise speedup on FlowNet with SSI convolution.

that it achieves 0.88–3.39× speedup under 90% sparsity with different input scales [31]. Compared with *SBNet*, SSI convolution achieves higher speedups under the same input scale, e.g., 1.25–3.79× speedup under 80% sparsity and 1.32–5.76× speedup under 90% sparsity, respectively, when the input scale is ranging from 50 × 50 to 500 × 500. The reason is that *SSI* requires less memory allocation and memory copy than *SBNet*. Similar to *SBNet*, with the decrease of the scale, the speedup of SSI convolution becomes lower. This phenomenon is because the total parallel GPU threads could cover the parallel capacity of GEMM operation when the input scale is small [36]. In such a case, a higher spatial sparsity could no longer achieve proportional speedup.

6.9 Evaluation of FlowNet with *SSI*

Table 3 shows the configuration of each convolution layer in FlowNet with input video frame dimension of 600 × 1,000. Figure 19 shows their corresponding speedup with different sparsities of static computation mask on GeForce GTX 1080. Later layers get lower input dimensions of the feature map. So, their corresponding speedups are limited compared with the prior layers. Furthermore, *SSI* achieves marginal speedup with comparatively low sparsity such as 50%. As a result, it is recommended that *SSI* could be used when the spatial sparsity is high. Mind that the speedup results shown in Figure 19 only considers SSI convolution operations. Therefore, if we adopt a dynamic computation mask for each video frame, then the overall speedup will be lower because of the overhead of the SSI index operations. Fortunately, the SSI index operations could be done once the feedback bounding boxes of the last video frame are available, which will not affect the detection latency.

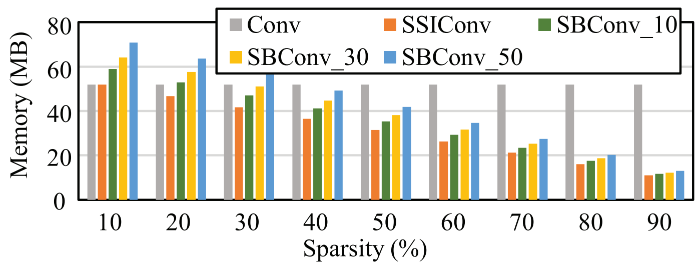

Fig. 20. Memory consumption of original, SSI, and SBNet Convolution.

6.10 Memory Evaluation of *SSI*

Figure 20 shows the memory consumption of original convolution, SSI convolution, and SBNet convolution, respectively. We calculate the memory consumption with different non-zero sparsities under the setting of 128 3×3 convolution kernels and a 100×100 feature map of 128 channels as input. To show the memory consumption of *SBNet* with different block granularities, we shows three sparsity levels, e.g., *SBConv10*, *SBConv30*, and *SBConv50*, respectively. Here 10, 30, and 50 represent the number of blocks from each side of the feature map. For example, *SBConv10* means that the feature map is divided into $10 \times 10 = 100$ blocks. We could find from Figure 20 that *SSI* and *SBNet* require similar or even more memory than the original convolution when the spatial sparsity is under 40%. Hence, neither *SSI* nor *SBNet* achieves satisfactory speedup under low spatial sparsity. Additionally, *SSI* requires less memory than *SBNet*, because *SBNet* needs to make a copy of the input feature map during the *Gather* operation. Moreover, with the decrease of block granularity in *SBNet*, the memory consumption keeps increasing due to the increase of total blocks and the overlaps between blocks.

7 RELATED WORKS

7.1 Impact on Video Detection System

In this work, we focus on the optimization of video detection—an application of continuous computer vision for object detection and analytics purpose. Many prior works tried to accelerate DNN-based video detection on mobile platforms by leveraging cache reuse [37, 38], model compression [39, 40], dynamic execution paradigm [41, 42], communication deduction [43, 44], and so on.

The contributions that extinguish our work from the prior-arts can be summarized as follows: First, our proposed optimization methods (*ADFF* and *SSI*) are specific for *DFF* framework. *ADFF* is the asynchronous version of original *DFF* and *SSI* is only used for Optical Flow estimation. In this way, a better performance can be achieved thanks to the based cutting-edge video detection framework. Second, we emphasize the low coupling character of our proposed *VDS* method. *VDS* could be easily ported to any DNN-based video detection system without touching the other function modules and no extra computation is incurred.

7.2 Multi-path Execution of DNNs

Multi-path execution of DNNs is a way to accelerate the detection pipeline. In Reference [41], the authors propose NoScope, where large DNNs are only executed when specialized models that perform classification tasks fail. Similar to NoScope, in Reference [42], the authors propose Frugal Following, where small networks are used for object tracking and large DNNs are executed only when needed. Unlike these works, *ADFF* decouples the original sequential execution of two DNNs into two parallel parts. In our experiments, R-FCN (ResNet101) is adopted as the FeatNet [45].

Mind that the FeatNet could be substituted by any other object detection framework like SSD or Yolo [46, 47]. Thanks to *ADFF*, the bottleneck of the original *DFF* is transferred from the FeatNet to FlowNet. Therefore, the execution time of FeatNet will not affect the latency of our video detection system.

7.3 Temporal Redundancy in Video Detection

Removing temporal redundancy in video detection is also an efficient way to accelerate the execution. Many previous works utilize a smaller model or light-weight algorithms (e.g., logistic regression, Lucas-Kanade optical flow, or diamond search) to detect the frames with small displacement and thus avoid executing the large DNN model on those frames [37, 38, 41–44, 48]. Following the same idea, *VDS* also utilizes the idea of low displacement between frames to dynamically decide the frame interval in *DFF*. However, there exist two main differences that make *VDS* outperform the related works. First, no extra computation is required for *VDS*, because the motion vector is directly derived from H.264. Such an idea is also used in Reference [44] for alleviating communication costs. Second, unlike the previous works, the motion vector in *VDS* is not directly applied to the video frame and thus will not affect the detection performance. For example, in Reference [48], the motion vector is applied in the receptive fields while in *VDS*, the H.264 motion vector is accumulated by a single value to guide the key-frame selection.

To skip the unchanged areas in each video frame, DeepCache [37] and DeepMon [38] cache the previous feature map and utilize Chi-Square distance and diamond search to detect the change of each grid block of the frame. Similarly, *DFF* also caches the feature map of the previous key frame and our proposed *SSI* selects the areas using two computation masks. Compared with DeepMon, *SSI* can tolerate scene variation. Compared with DeepCache, our proposed computation masks require less computation.

8 CONCLUSION

In this work, we propose an efficient cloud-based video detection system for real-time applications. Our system adopts the state-of-the-art video detection framework *DFF* with several optimizations. First, we design *Asynchronous DFF (ADFF)*, which utilizes multi-thread technology to asynchronously execute the DNNs in *DFF* while ensuring the functional correctness via Mutex. Second, we propose *Video-based Dynamic Scheduling (VDS)* to dynamically decide the key frame based on H.264 motion vector. Third, We propose *Spatial Sparsity Inference (SSI)* for spatially partial inference based on both feedback detection results and brightness error. Our video detection system is able to satisfy real-time requirement of video detection applications on popular mobile platforms with marginal accuracy degradation.

REFERENCES

[1] Ashish Vaswani et al. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, 5998–6008. Retrieved from http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. Retrieved from http://arxiv.org/abs/1810.04805.

[3] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems*. MIT Press, 91–99.

[4] Yi Sun, Ding Liang, Xiaogang Wang, and Xiaoou Tang. 2015. DeepID3: Face recognition with very deep neural networks. Retrieved from http://arxiv.org/abs/1502.00873.

[5] Barret Zoph and Quoc V. Le. 2016. Neural architecture search with reinforcement learning. Retrieved from http://arxiv.org/abs/1611.01578.

[6] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. DARTS: Differentiable architecture search. Retrieved from http://arxiv.org/abs/1806.09055.

[7] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. 2018. AMC: AutoML for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV'18)*.

[8] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems.* MIT Press, 2074–2082.

[9] Patrick Chen, Si Si, Yang Li, Ciprian Chelba, and Cho-Jui Hsieh. 2018. GroupReduce: Block-wise low-rank approximation for neural language model shrinking. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, 10988–10998.

[10] Hsin-Pai Cheng, Yuanjun Huang, Xuyang Guo, Yifei Huang, Feng Yan, Hai Li, and Yiran Chen. 2018. Differentiable fine-grained quantization for deep neural network compression. Retrieved from http://arxiv.org/abs/1810.10351.

[11] Johann Hauswald, Thomas Manville, Qi Zheng, Ronald Dreslinski, Chaitali Chakrabarti, and Trevor Mudge. 2014. A hybrid approach to offloading mobile image classification. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'14)*. IEEE, 8375–8379.

[12] Yuhao Zhu, Anand Samajdar, Matthew Mattina, and Paul N. Whatmough. 2018. Euphrates: Algorithm-SoC co-design for low-power mobile continuous vision. Retrieved from http://arxiv.org/abs/1803.11232.

[13] Jiachen Mao, Qing Yang, Ang Li, Hai Li, and Yiran Chen. 2019. MobiEye: An efficient cloud-based video detection system for real-time mobile applications. In *Proceedings of the 56th Annual Design Automation Conference.* 1–6.

[14] Chuhan Min, Jiachen Mao, Hai Li, and Yiran Chen. 2019. NeuralHMC: An efficient HMC-based accelerator for deep neural networks. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference (ASPDAC'19)*. ACM, New York, NY, 394–399. DOI: http://dx.doi.org/10.1145/3287624.3287642

[15] Linghao Song, Jiachen Mao, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2019. HyPar: Towards hybrid parallelism for deep learning accelerator array. Retrieved from http://arxiv.org/abs/1901.02067.

[16] Bing Li, Wei Wen, Jiachen Mao, Sicheng Li, Yiran Chen, and Hai Helen Li. 2018. Running sparse and low-precision neural network: When algorithm meets hardware. In *Proceedings of the 23rd Asia and South Pacific Design Automation Conference (ASP-DAC'18)*. IEEE, 534–539.

[17] Jiachen Mao, Xiang Chen, Kent W. Nixon, Christopher Krieger, and Yiran Chen. 2017. MoDNN: Local distributed mobile computing system for deep neural network. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'17)*. IEEE, 1396–1401.

[18] Jiachen Mao, Zhongda Yang, Wei Wen, Chunpeng Wu, Linghao Song, Kent W. Nixon, Xiang Chen, Hai Li, and Yiran Chen. 2017. MeDNN: A distributed mobile system with enhanced partition and deployment for large-scale DNNs. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'17)*. IEEE, 751–756.

[19] J. Mao, Z. Qin, Z. Xu, K. W. Nixon, X. Chen, H. Li, and Y. Chen. 2017. AdaLearner: An adaptive distributed mobile learning system for neural networks. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'17)*. 291–296. DOI: http://dx.doi.org/10.1109/ICCAD.2017.8203791

[20] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. Retrieved from http://arxiv.org/abs/1704.04861.

[21] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. 2017. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*.

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, 1097–1105. Retrieved from http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[23] Martín Abadi et al. 2016. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 265–283.

[24] H.-Y. C. Tourapis et al. 2003. Fast motion estimation within the H. 264 codec. In *Proceedings of the International Conference on Multimedia and Expo (ICME'03)*, Vol. 3. IEEE, III–517.

[25] Xizhou Zhu, Yuwen Xiong, Jifeng Dai, Lu Yuan, and Yichen Wei. 2017. Deep feature flow for video recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17)*.

[26] Alexey Dosovitskiy et al. 2015. Flownet: Learning optical flow with convolutional networks. In *Proceedings of the International Conference on Computer Vision (ICCV'15)*. 2758–2766.

[27] Anita Sellent, Daniel Kondermann, Stephan Simon, Simon Baker, Goksel Dedeoglu, Oliver Erdler, Phil Parsonage, Christoph Unger, and Wolfgang Niehsen. 2012. Optical flow estimation versus motion estimation. https://archiv.ub.uni-heidelberg.de/volltextserver/13641/.

[28] Aurélien Plyer, Guy Le Besnerais, and Frédéric Champagnat. 2016. Massively parallel Lucas Kanade optical flow for real-time video processing applications. *J. Real-Time Image Process.* 11, 4 (2016), 713–730.

[29] Gunnar Farnebäck. 2003. Two-frame motion estimation based on polynomial expansion. In *Proceedings of the Scandinavian Conference on Image Analysis*. Springer, 363–370.

[30] Xizhou Zhu, Jifeng Dai, Lu Yuan, and Yichen Wei. 2018. Towards high performance video object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 7210–7218.

[31] Mengye Ren et al. 2018. SBNet: Sparse blocks network for fast inference. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR'18)*. 8711–8720.

[32] Mikhail Figurnov et al. 2016. PerforatedCNNs: Acceleration through elimination of redundant convolutions. In *Proceedings of the International Conference on Neural Information Processing Systems (NIPS'16)*. 947–955.

[33] Kaiming He et al. 2016. Deep residual learning for image recognition. In *Proceedings of the International Conference on Computer Vision (ICCV'16)*. 770–778.

[34] Tianqi Chen et al. 2015. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. Retrieved from http://arxiv.org/abs/1512.01274.

[35] Yihua Guo, Feng Qian, Qi Alfred Chen, Zhuoqing Morley Mao, and Subhabrata Sen. 2016. Understanding on-device bufferbloat for cellular upload. In *Proceedings of the Internet Measurement Conference*. ACM, 303–317.

[36] David Kirk et al. 2007. NVIDIA CUDA software and GPU parallel computing architecture. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM'07)*, Vol. 7. 103–104.

[37] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. 2018. DeepCache: Principled cache for mobile deep vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*. 129–144.

[38] Loc N. Huynh, Youngki Lee, and Rajesh Krishna Balan. 2017. 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*. 82–95.

[39] Sicong Liu, Yingyan Lin, Zimu Zhou, Kaiming Nan, Hui Liu, and Junzhao Du. 2018. On-demand deep model compression for mobile devices: A usage-driven model selection framework. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. 389–400.

[40] Biyi Fang, Xiao Zeng, and Mi Zhang. 2018. 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*. 115–127.

[41] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. Noscope: Optimizing neural network queries over video at scale. Retrieved from https://arXiv:1703.02529.

[42] Kittipat Apicharttrisorn, Xukan Ran, Jiasi Chen, Srikanth V. Krishnamurthy, and Amit K. Roy-Chowdhury. 2019. Frugal following: Power thrifty object detection and tracking for mobile augmented reality. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*. 96–109.

[43] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. 2015. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*. 155–168.

[44] Luyang Liu, Hongyu Li, and Marco Gruteser. 2019. Edge assisted real-time object detection for mobile augmented reality. In *Proceedings of the 25th Annual International Conference on Mobile Computing and Networking*. 1–16.

[45] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. 2016. R-fcn: Object detection via region-based fully convolutional networks. In *Advances in Neural Information Processing Systems*. MIT Press, 379–387.

[46] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. SSD: Single shot multibox detector. In *Proceedings of the European Conference on Computer Vision*. Springer, 21–37.

[47] Joseph Redmon et al. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 779–788.

[48] Mark Buckler, Philip Bedoukian, Suren Jayasuriya, and Adrian Sampson. 2018. EVA^2: Exploiting temporal redundancy in live computer vision. In *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE, 533–546.

Received February 2021; revised May 2021; accepted September 2021