A Distributed Scheme for Accelerating Semantic Video Segmentation on An Embedded Cluster

Hsuan-Kung Yang*, Tsu-Jui Fu*, Po-Han Chiang†, Kuan-Wei Ho†, and Chun-Yi Lee
Elsa Lab, Department of Computer Science
National Tsing Hua Unversity
Hsinchu City, Taiwan
{hellochick, rayfu1996ozig, ymmoy999, kuan.wei.ho, cylee}@gapp.nthu.edu.tw

***Abstract*—We present a methodology for enhancing the throughput of semantic video segmentation tasks on an embedded cluster containing multiple embedded processing elements (ePEs). The methodology embraces a scalable master-slave hierarchy and features a global and local key management scheme for allocating video frames to different ePEs. The master ePE divides each video frame into frame regions, and dynamically distributes different regions to different slave ePEs. Each slave ePE executes either a segmentation path or a flow path: the former is highly accurate but slower, while the latter is faster but less accurate. A lightweight decision network is employed to determine the execution path for each slave ePE. We propose a global and local key management scheme to facilitate the execution of the embedded cluster, such that the average processing latency of each frame is significantly reduced. We evaluate the performance of our methodology on a real embedded cluster in terms of accuracy and frame rate, and validate its effectiveness and efficiency for various ePE configurations. We further provide a detailed latency analysis for different configurations of ePEs.**

***Index Terms*—embedded cluster, semantic video segmentation, optical flow, decision network, global and local key management.**

I. Introduction

Deep convolutional neural networks (DCNNs) have achieved great success in a number of research areas of computer vision (CV), including image classification [1], [2], object detection [3], [4], and semantic segmentation [5], [6]. Semantic segmentation has been considered as a much more difficult task than the former two areas, as it requires performing dense pixel-level predictions for input images. Although recent advances in DCNN-based semantic segmentation techniques have achieved high accuracies [5]–[8], these techniques are still not directly applicable to embedded systems because of their significantly longer execution latency and heavier computational workloads.

On the other hand, applications requiring semantic video segmentation has emerged recently in a wide range of domains, including autonomous vehicles, unmanned aerial vehicles (UAVs), virtual reality, etc. These applications demand both high frame rates per second (*fps*) (typically up to 20 *fps* or even more for real-time applications) and high pixel-level accuracy, while requiring low computational overheads for the models such that they are able to be executed on embedded processing elements (ePEs). Although several approaches for real-time semantic segmentation [9]–[12] have been proposed, they usually suffer from accuracy degradation. In addition, these techniques similarly incur expensive computational workloads, as they are not specifically developed and tailored for ePEs.

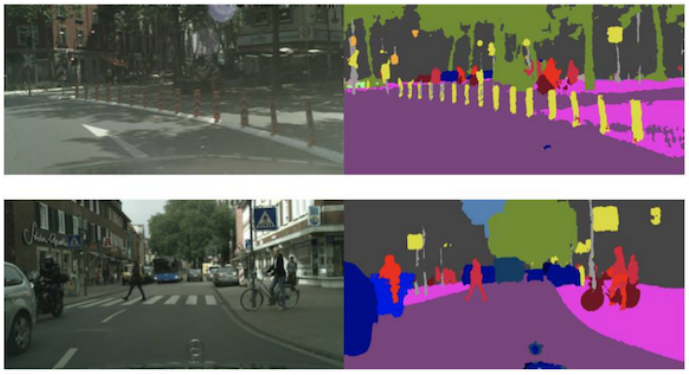

Fig. 1. Example qualitative results generated by the proposed methodology. The left-hand side and the right-hand side figures illustrate the sample input frames and the corresponding output semantic segmentations, respectively.

Motivated by the requirements of executing DCNN models on embedded systems, a number of emerging devices and accelerators targeting at DCNN applications have also emerged in the past few years. These devices are capable of carrying out computations for simple DCNN models. However, delivering satisfactory results for complex models in real time is still challenging for them due to their limited ePEs, memories, and battery budgets. Researchers in recent years have proposed different approaches to accelerate the computation of DCNNs deployed on devices other than graphics processing units (GPUs). These approaches include distributed programming [13], [14], parallel data fusion [15], and utilization of high-level tokens to represent computation nodes [14]. Efficient architectures [9]–[12], [16] and parameter pruning methods have also been proposed to reduce the size of DCNNs. Although these methods are effective in reducing the latencies of DCNNs, most of them still suffer from inevitable accuracy degradation.

In order to address the issues mentioned above, we propose a distributed methodology for allocating the heavy computational workloads of semantic video segmentation to a cluster

* and † indicate equal contributions.

containing multiple ePEs. The proposed methodology is built atop the framework of dynamic video segmentation network (DVSNet) [17], and is able to adapt to different numbers of ePEs in the embedded cluster. Instead of separating a DCNN model and distributing the model to multiple ePEs, the proposed methodology adaptively apply different segmentation strategies to different video frame regions via two types of paths: a segmentation path and a flow path. The former is composed of a highly accurate DCNN model with a longer execution latency. The latter is developed based on the concept of optical flow estimation, and is shorter in its latency but less accurate. As different frame regions in a video sequence typically experience different extents of changes, the two segmentation strategies can be applied to different frame regions to maximally exploit the spatial and temporal redundancies in consecutive frames.

In this paper, we implement the proposed methodology on an embedded cluster in a distributed framework. The embedded cluster contains a master ePE and a number of slave ePEs. The number of slave ePEs is scalable. The master ePE divides video frames into frame regions, and dynamically distributes these frame regions to the available slave ePEs via the assistance of a workload manager. Each slave ePE executes either of the two paths mentioned above. In order to balance the workloads and make the most efficient use of the two paths, we further propose a global and local key management scheme. The scheme continuously maintains a key buffer in each ePE and a key table in the master ePE, such that the slave ePEs can execute their flow paths as frequently as possible without significant accuracy degradation. The proposed methodology is compatible with contemporary embedded platforms whose ePEs are capable of executing the two execution paths. To validate the effectiveness of the proposed methodology, we perform extensive experiments on real embedded clusters containing multiple NVIDIA Jetson TX2's as our ePEs. Jetson TX2 has been widely employed in embedded applications, and proven to offer sufficient computing power with limited memory capacity and battery budget [18]–[20]. We investigate various cluster configurations, and demonstrate that our methodology does offer higher *fps*, superior speedup, and negligible accuracy drop when compared with several representative baseline methods. Moreover, we comprehensively analyze the execution latencies of the ePEs for a number of different cluster configurations.

The main contributions of the paper are as the following:

- An embedded cluster architecture consisting of multiple ePEs for enhancing the overall frame throughput and execution efficiency of semantic video segmentation tasks.
- A distributed version of DVSNet framework [17] for the master ePE to dynamically allocate workloads (i.e., different regions in a video frame) to available slave ePEs, such that video frames are able to be segmented in parallel.
- A global and local key management scheme for regulating and maintaining the segmentation accuracy (i.e., $mIoU$) and frame rate (i.e., *fps*) of the proposed methodology.
- An extensive set of experiments, analysis, and discussion of the proposed methodology on real embedded clusters.

The remainder of this paper is organized as follows. Section II reviews the background material related to this work. Section III walks through the proposed methodology, its framework and components, as well as the key management scheme (in terms of pseudocodes) in detail. Section IV presents the experimental results and analyzes their implications. Section V concludes.

II. Background

In this section, we provide background material necessary for understanding the technical contents of this paper. We first briefly introduce the fundamental concepts and related works of semantic segmentation and optical flow estimation. Then, we describe the architecture and working procedure of DVSNet.

A. Semantic Segmentation

Semantic segmentation is one of the key research directions in CV, which aims at performing pixel-level predictions (i.e., dense predictions) for an image. An example of semantic segmentation is depicted in Fig. 1. A semantic segmentation model typically predicts a *segmentation logit* for each pixel first, and then classifies each pixel based on it. The segmentation logit of a pixel represents the probabilities of a predefined set of classes that the pixel may belong to. The accuracy of a semantic segmentation model is commonly measured by a metric called **m**ean **I**ntersection-**o**ver-**U**nion ($mIoU$), given by:

$$mIoU = \frac{1}{N_{class}} \sum_{i=1}^{N_{class}} \frac{TP(i)}{TP(i) + FP(i) + FN(i)}, \quad (1)$$

where N_{class} is the number of classes. $TP(i)$, $FP(i)$, and $FN(i)$ are the pixel counts of true positive, false positive, and false negative pixels of the i^{th} class in the image, respectively.

Training a highly accurate semantic segmentation model for performing such dense pixel-wise predictions has long been considered much more challenging than image classification and object detection tasks. State-of-the-art semantic segmentation models [5], [6] typically demand significant amount of computational workloads due to their deep architectures. Therefore, most of them are inappropriate for embedded systems. A number of lightweight DCNN architectures have been proposed as candidate solutions in the past few years. For instance, ERFNet [11] employs a layer structure that uses residual connections and factorized convolutions for enhancing the efficiency of the DCNN model. ENet [10] and ESPNet [12] are based on specialized convolutional modules which are efficient in terms of computation, memory, and power consumption, and thus are more suitable for edge devices. However, they generally suffer from serious drops in segmentation accuracy, which are essential for video applications. Although a highly efficient DCNN architecture called *MobileNet* [16] has been applied to semantic segmentation tasks [5] and has achieved significant improvements in $mIoU$ and *fps* in recent years, the *fps* delivered by [5] is still insufficient for embedded devices to perform real-time applications. A comparison of the above techniques with our methodology are presented in Section IV.

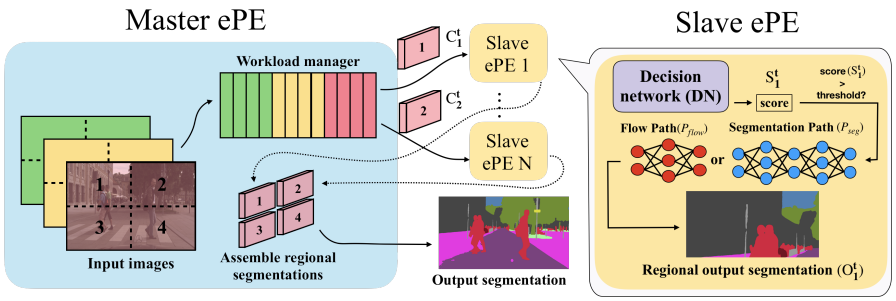

Fig. 2. An overview of the framework.

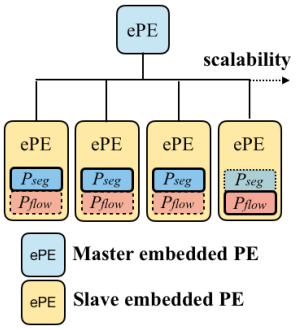

Fig. 3. The master-slave hierarchy. Different slave ePEs can execute different execution paths.

B. Optical Flow Estimation

Optical flow estimation [21] is a technique for evaluating the motion of objects between a reference image and a target image. It is usually represented as a sparse or dense vector field, where displacement vectors are assigned to certain pixel positions of the reference image. These vectors point to where those pixels can potentially be found in the target image. Mainstream approaches for estimating the optical flow between two consecutive image frames are based on the work of Horn and Schunck [21]. A few recent research works [24] are proposed to efficiently deal with large displacement problems occurring in realistic videos. These techniques are, unfortunately, mostly designed to run on central processing units (CPUs), and are thus unable to leverage the computational efficiency offered by contemporary embedded systems equipped with GPUs. Different from the approaches mentioned above, FlowNet [22] is the first DCNN model specifically developed for performing optical flow estimation on GPUs. It then soon evolved into another enhanced model called FlowNet 2.0 [23]. FlowNet 2.0 features a stacked architecture, a better learning schedule than FlowNet, as well as a special sub-network design focusing on small displacement extraction for obtaining a finner flow estimation than the former version. As a result, we incorporate the building blocks from FlowNet 2.0 into our methodology.

C. Dynamic Video Segmentation Network (DVSNet)

DVSNet [17] is a framework which incorporates two distinct DCNNs for enhancing the frame rates of semantic video segmentation tasks while maintaining the accuracy of them. DVSNet achieves such improvements by exploiting spatial and temporal redundancies in consecutive video frames as much as possible and then adaptively processing different frame regions using different DCNNs. The first DCNN employed by DVSNet is called the *segmentation network*, which generates highly accurate semantic segmentations, but is deeper and slower. The second DCNN is called the *flow network*, which employs a warping function [23] to generate approximated semantic segmentations and is much shallower and faster than the segmentation network. DVSNet takes advantage of the fact that different regions in a video sequence experience different extents of changes to avoid reprocessing every single pixels in consecutive frames. Frame regions with huge differences in pixels between consecutive frames, where the contents may have changed significantly, have to pass through the segmentation network. Otherwise, they are processed by the flow network. In other words, different frame regions in a frame may traverse different networks of different lengths when they are presented to DVSNet. In order to determine whether an input frame region has to traverse the segmentation network or not, DVSNet further employs a lightweight *decision network (DN)* to evaluate a confidence score for each frame region. A confidence score lower than a pre-defined decision threshold indicates that the corresponding frame region is required to be processed by the segmentation network. DVSNet allows the decision threshold for the confidence score to be customizable.

III. Methodology

In this section, we present the architecture and the implementation details of the proposed methodology. We first provide an overview of the entire framework, followed by a detailed description of the components in it. Next, we introduce the global and local key management scheme. Finally, we walk through the inference procedure as well as its pseudocodes.

A. Overview of the Framework

Fig. 2 illustrates the proposed framework, which contains a master ePE and N slave ePEs, where N is a scalable integer. Fig. 3 illustrates the master-slave hierarchy of the framework. The notations used in this paper are summarized in Table I. The main objective of the framework is to enhance the throughput (i.e., the frame rates) of semantic video segmentation by multiple ePEs, while maintaining the $mIoU$ accuracy of the system by exploiting the benefits offered by DVSNet. The master ePE divides each input frame into four frame regions, and allocates each frame region to an available slave ePE by dynamic scheduling to generate the semantic segmentation of the regions. The unallocated frame regions are stored in a queue managed by a workload manager, which is responsible for selecting an appropriate slave ePE to process the region stored at the head of the queue. The information exchanged between the master ePE and the slave ePEs is regulated by the global and local key management scheme, which is discussed in Section III-C. The master ePE is also responsible for gathering

TABLE I
THE NOTATIONS USED IN THIS PAPER.

t	The current timestep index
r	Region index (ranges from 1 to 4)
T_r	Timestep of keyframe region r stored in keytable
C_r^t	Frame region r of the **current frame** at timestep t
K_r^t	Frame region r of the **key frame** at timestep t
O_r^t	**Segmented result** of frame region r at timestep t
S_r^t	**Confidence score** of frame region r at timestep t
L_r^t	**Segmentation logits** of frame region r at timestep t

the semantic segmentations of the frame regions that belong to the same frame from the slave ePEs, and assembling them to generate the final semantic segmentation of the frame. The above procedures are illustrated in the blue region of Fig. 2.

The slave ePE contains three major components: a segmentation path P_{seg}, a flow path P_{flow}, and a *DN*. Each slave ePE has the same architecture, except that the execution paths of different slave ePEs are allowed to be different, as depicted in Fig. 3. A frame region C_r^t forwarded to a slave ePE is processed by either P_{seg} or P_{flow}. The primary function of P_{seg} is to directly generate high-quality semantic segmentation O_r^t from C_r^t by DCNN models such as [5], [6]. It requires a longer period of processing time, but is able to deliver a higher $mIoU$ accuracy for the generated O_r^t. Please note that the frame region C_r^t which travels through P_{seg} is referred to as the key frame region K_r^T, where T is used to record the timestep t (i.e., $T := t$ and $K_r := C_r$ when C_r^t is processed by P_{seg}). On the other hand, P_{flow} generates O_r^t from (C_r^t, K_r^T), as well as the segmentation logits L_r^T of K_r^T by optical flow estimation, where t is always larger than or equal to T, and r is required to be the same for the inputs (C_r^t, K_r^T) of P_{flow}. The processing latency of P_{flow} is much lower than that of P_{seg}, however, the $mIoU$ accuracy achievable by P_{flow} decreases as $(t - T)$ increases. In other words, increasing the time span between C_r^t and K_r^T may lead to an O_r^t with a lower $mIoU$ when C_r^t is processed by P_{flow}. *DN* is a neural network trained to evaluate a *confidence score* S_r^t from C_r^t. The role of *DN* is to determine the execution path according to S_r^t. If S_r^t is higher than a predefined threshold δ, P_{flow} is selected to process C_r^t. Otherwise, P_{seg} is used to process C_r^t and update K_r^T as C_r^t. The above mechanism enables different slave ePEs to execute frame regions with different latencies.

B. Components of the Framework

In this section, we describe the implementation details of P_{seg}, P_{flow}, and *DN* in the slave ePEs, as well as the workload manager in the master ePE in separate paragraphs.

Segmentation path (P_{seg}). The architecture of P_{seg} is flexible in the proposed framework, as long as the selected DCNN model is able to generate high-quality O_r^t's from C_r^t's. In this paper, we employ DeepLabv3+ [5] as our P_{seg}. The main advantage of DeepLabv3+ is that one implementation of its backbone feature extraction networks is based on MobileNet-v2 [16], a highly efficient and accurate DCNN architecture specifically developed for vision applications on embedded

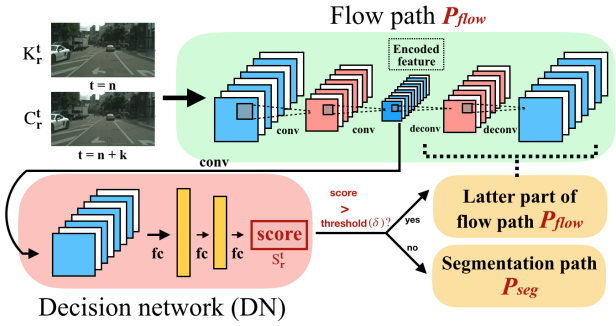

Fig. 4. The architecture of the decision network (*DN*).

systems. In this work, P_{seg} is pre-trained on the MS-COCO dataset [25], and fine-tuned on the Cityscapes dataset [26].

Flow path (P_{flow}). As described in Section III-A, P_{flow} estimates the optical flow between C_r^t and K_r^T for frame region r, where $T \leq t$. The estimated optical flow along with the segmentation logits L_r^T of K_r^T are then processed by a warping function [23] to generate O_r^t. The quality of O_r^t generated by P_{flow} may decrease as $(t-T)$ increases [17]. We adopt FlowNet 2.0 [23] as P_{flow}. FlowNet 2.0 is a lightweight stacked DCNN containing multiple sub-networks for estimating small and large displacements with high accuracy. The latency of P_{flow} is significantly lower than that of P_{seg}. Please note that P_{flow} is executed once for every single input frame region allocated to the slave ePE, as the encoded feature map of P_{flow} is also used as the input for *DN*, as illustrated in Fig. 4.

Decision network (*DN*). *DN* is a shallow regression model pre-trained for determining whether C_r^t has to go through P_{seg} or P_{flow}. Fig. 4 illustrates the architecture of *DN* (the red part). It is composed of one convolutional layer and three fully-connected layers. *DN* takes as input the feature map extracted from the former part of P_{flow} to evaluate S_r^t, which is used as a reference to decide the execution path of C_r^t. An S_r^t satisfying the condition $(S_r^t > \delta)$ implies that it is likely for P_{flow} to generate high-quality O_r^t. As a result, P_{flow} is set as the execution path when $(S_r^t > \delta)$ is met. Otherwise, P_{seg} is selected as the execution path to maintain the $mIoU$ of O_r^t.

Workload manager. The workload manager resides in the master ePE and is responsible for allocating frame regions to the slave ePEs. It maintains a first-in, first out (FIFO) queue for accommodating unallocated frame regions. In addition, it keeps another FIFO queue to continuously track the indexes of the idle slave ePEs. When a frame region is to be allocated, the second queue pops out an index if it is not empty. The master ePE then uses that index to forward the frame region to the idle slave ePE. If the queue is empty, the master ePE then waits until a slave ePE becomes available. When a slave ePE completes its task, its index is pushed back into the queue.

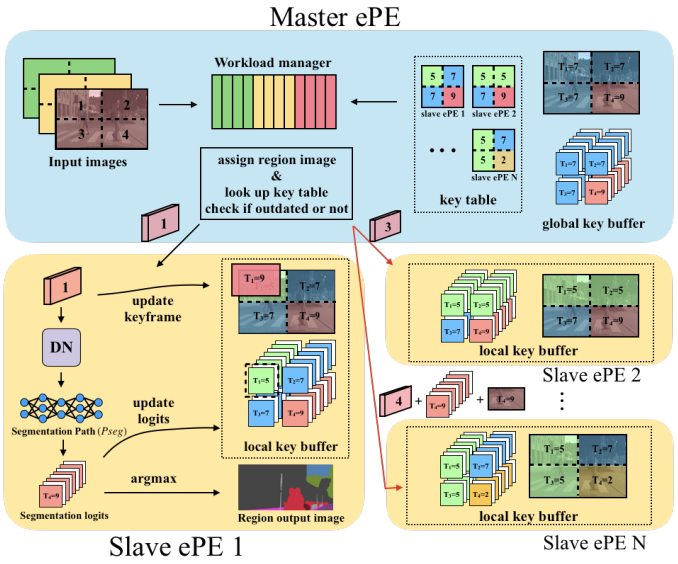

Fig. 5. The global and local key management scheme.

C. Global and Local Key Management Scheme

Fig. 5 illustrates the global and local key management scheme. The primary objective of the scheme is to maintain a *key buffer* in each ePE to store a 4-tuple ($L_1^{T_1}$, $L_2^{T_2}$, $L_3^{T_3}$, $L_4^{T_4}$) of segmentation logits and another 4-tuple ($K_1^{T_1}$, $K_2^{T_2}$, $K_3^{T_3}$, $K_4^{T_4}$) of key frame regions, such that they can be used when P_{flow} is selected as the execution path. The key buffer in the master ePE is called the *global key buffer*, while that in the slave ePE is called the *local key buffer*. The contents of the key buffers from different ePEs are not necessarily the same. However, *global key buffer* always keeps the tuples containing the newest L_r's and K_r's respectively for the four frame regions. The master ePE additionally maintains a *key table* for each of the slave ePEs to monitor (T_1, T_2, T_3, T_4) of their *local key buffers*, as depicted in Fig. 5. When a slave ePE is selected by the workload manager to process a frame region C_r^t, the master ePE first compares the difference between T_r of its *global key buffer* and that stored in the *key table* corresponding to the slave ePE. If the difference is larger than a threshold $\mathcal{T}$ (e.g., slave ePE **N**, assume that $\mathcal{T} = 5$), the master ePE then forwards its $L_r^{T_r}$ and $K_r^{T_r}$ to the slave ePE to update the corresponding entry in the *local key buffer*. This ensures that the slave ePE has the newest $L_r^{T_r}$ and $K_r^{T_r}$ to perform P_{flow}. If a slave ePE performs P_{seg} for a frame region C_r^t (e.g., slave ePE **1**), the newly generated $L_r^{T_r}$ is transmitted back to the master ePE to update $L_r^{T_r}$ and $K_r^{T_r}$ of the *global key buffer*, as well as T_r of the corresponding *key table*. This ensures that the *global key buffer* maintains the latest L_r's and K_r's for the four frame regions, and allows the master ePE to keep track of the status of the slave ePEs. Please note that in the proposed key management scheme, the ePEs communicate in a peer-to-peer fashion, rather than broadcasting to all of the ePEs. Peer-to-peer communication enables the interconnects to be efficiently utilized, alleviating potential congestions in the communication channels of the embedded cluster.

Algorithm 1 Master ePE

$InputQueue$: A FIFO queue for accommodating unallocated frame regions.
$IdleQueue$: A FIFO queue for tracking the indexes of idle slave ePEs.

$KeyTable(m, i)$: A table for monitoring T_i on slave ePE m.
$GlobalKeyBuffer(i)$: A buffer for storing the newest K_i, L_i, and T_i on master ePE.

function MASTERSENDER
  while $True$ **do**
    for $i = 1$ to 4 **do**
      C_i^t, t = $InputQueue$.front()
      m = $IdleQueue$.front() ▷ Obtain the index of an idle slave ePE

      T_i = $KeyTable(m, i)$
      K_i, L_i, T_i' = $GlobalKeyBuffer(i)$
      if T_i' - $T_i > \mathcal{T}$ **then**
        SENDTOSLAVE(m, i, t, $True$, C_i^t, K_i, L_i)
      else
        SENDTOSLAVE(m, i, t, $False$, C_i^t, $null$, $null$)

function MASTERRECEIVER
  while $True$ **do**
    m, i, t, is_change, O_i^t, L_i^t = RECEIVEFROMSLAVE
    $IdleQueue$.push(m)

    if is_change is $True$ **then**
      $KeyTable(m, i)$ = t
      $GlobalKeyBuffer(i)$ = C_i^t, L_i^t, t

    $Outputs(t, i)$ = O_i^t
    Assemble the segmentation result if all $Outputs(t, \cdot)$ are ready

Algorithm 2 Slave ePE

P_{seg}: The segmentation path path that directly generates high-quality O_i^t.
P_{flow}: The flow path that generates O_i^t by optical flow estimation.

$LocalKeyBuffer(i)$: A buffer for storing the locally newest K_i and L_i.
$DN(EncodedFeature)$: A regression model for predicting S_i^t using the encoded feature from the former part of P_{flow}.

function SLAVE
  while $True$ **do**
    m, i, t, is_change, C_i^t, K_i, L_i = RECEIVEFROMMASTER

    if is_change is $True$ **then** ▷ *Global* forces to update K_i and L_i
      $LocalKeyBuffer(i)$ = K_i, L_i

    K_i, L_i = $LocalKeyBuffer(i)$ ▷ Obtain the local newest K_i and L_i

    $EncodedFeature$ = $FormerPartOfP_{flow}(K_i, C_i^t)$
    S_i^t = DN($EncodedFeature$)
    if $S_i^t > \delta$ **then**
      O_i^t, L_i^t = $LatterPartOfP_{flow}(EncodedFeature, L_i)$
      is_change = $False$
    else
      O_i^t, L_i^t = $P_{seg}(C_i^t)$
      is_change = $True$
      $LocalKeyBuffer(i)$ = C_i^t, L_i^t

    SENDTOMASTER(m, i, t, is_change, O_i^t, L_i^t)

D. Inference Procedure and Pseudocodes

We summarize the proposed global and local key management scheme in Algorithms 1 and 2. Algorithm 1 presents the procedures of the master ePE, while Algorithm 2 describes the procedures for each of the slave ePEs. In these Algorithms, the primary function of the master ePE is to distribute frame regions to the slave ePEs, and gather the predicted results from them. The master ePE maintains a *global key buffer* (denoted as *GlobalKeyBuffer*) and *key table* (denoted as *KeyTable*) to keep

TABLE II
SPECIFICATION OF THE EMBEDDED PE (NVIDIA JETSON TX2) ADOPTED IN OUR EXPERIMENTS

Item	Specification
CPU	Denver + Cortex-A57
CPU cores	6
GPU	256 CUDA cores (Pascal architecture)
GPU Frequency	0.85 Ghz
Memory	8GB DDR4
Network bandwidth	1 Gbits/sec
Power consumption	7.5W (Average) / 15W (Maximum)

TABLE III
COMPARISON OF THE QUANTITATIVE RESULTS OF THE BASELINES AND THE PROPOSED METHODOLOGY. THE FIRST FOUR ROWS CORRESPOND TO THE BASELINE METHODS, WHILE THE REMAINING SEVEN ROWS CORRESPOND TO OUR METHODOLOGY. PLEASE NOTE THAT THE RESULTS OF ENET, ERFNET, AND ESPNET ARE OBTAINED FROM [12].

Configurations	mIoU	FPS	Speedup
SegPath (DeepLabv3+) [5]	**72.63**	1.52	-
ENet [10]	58.3	9.2	-
ERFNet [11]	68.0	3.6	-
ESPNet [12]	60.3	9.0	-
Single (DVSNet) [17]	67.81	10.67	1x
1+1	67.81	8.74	0.82x
1+2	67.42	17.59	1.65x
1+3	66.24	22.04	2.07x
1+4	65.92	28.16	2.64x
1+5	65.22	**31.75**	**2.98x**
1+6	65.49	28.22	2.64x

track of the latest key frame regions ($K_1^{T_1}$, $K_2^{T_2}$, $K_3^{T_3}$, $K_4^{T_4}$) and segmentation logits ($L_1^{T_1}$, $L_2^{T_2}$, $L_3^{T_3}$, $L_4^{T_4}$), as discussed in Section III-C. On the other hand, each slave ePE receives frame regions sent from the master ePE, performs segmentation, and also maintains a *local key buffer* (denoted as *LocalKeyBuffer*) to keep track of the key frame regions and segmentation logits.

Algorithm 1 defines the procedures performed by the master ePE, which includes a *MasterSender* function and a *MasterReceiver* function (lines 20 to 27). Lines 1 to 2 define the two FIFO queues managed by the workload manager: *InputQueue* and *IdleQueue*, as described in Section III-B. Lines 4 to 5 define the *KeyTable* for storing the timesteps of the key frame regions for each slave ePE, and *GlobalKeyBuffer* for storing the global key frame regions as well as the corresponding segmentation logits. From line 7 to 18, *MasterSender* first obtains a frame region C_i^t (where i denotes the frame region number) and an idle slave ePE m from *InputQueue* and *IdleQueue*, respectively. It then queries *KeyTable* for the stored T_i for the corresponding tuple (m, i), as well as *GlobalKeyBuffer* for the stored K_i, L_i, and T_i' according to the frame region i. If the difference $(T_i' - T_i)$ is larger than $\mathcal{T}$, *MasterSender* forwards the 3-tuple (C_i^t, K_i, L_i) to slave ePE m. Otherwise, it only sends C_i^t to m. From line 20 to 27, *MasterReceiver* first receives O_i^t and L_i^t from slave PE m, and pushes m back to the *IdleQueue*. If m updates its *LocalKeyBuffer* (i.e., $is\_change == True$ and m performs P_{seg}), the tuple (m, i, t) is used for updating *KeyTable*. Meanwhile, (C_i^t, L_i, i, t) is used for updating *GlobalKeyBuffer*. Line 30 indicates that when O_i^t of all the regions ($i = 1$ to 4) of the same frame are available, the master ePE then assembles them to generate the final segmentation of the entire frame.

Algorithm 2 presents the workflow of a slave ePE. Lines 1 to 2 define the two execution paths for generating O_i^t: P_{seg} and P_{flow} , as described in Section III-B. Lines 4 to 5 define the *LocalKeyBuffer* for storing the locally newest K_i and L_i, and *DN* for evaluating S_i^t which is used as a reference to decide the execution path. Line 9 defines the parameters received from the master ePE. If the master ePE requires the slave ePE to update its *LocalKeyBuffer* (i.e., $is\_change == True$ at line 11), the latter then replaces the local key frame region and segmentation logits with the global one, as defined in lines 11 and 12. The slave ePE then obtains K_i and L_i from its *LocalKeyBuffer*. Next, from lines 16 to 24, the slave ePE executes DVSNet to obtain the predicted O_i^t and L_i^t, and sent them back to the master ePE by the *SendToMaster* function (line 26). If the slave ePE performs P_{seg} its *LocalKeyBuffer* is updated with (C_i^t, L_i^t) (line 24). It also notifies the master ePE through *SendToMaster* by setting is_change to $True$.

IV. EXPERIMENTAL RESULTS

In this section, we present the experimental results and discuss their implications. Fig. 1 presents a few qualitative results generated by our framework, where the left-hand side and the right-hand side figures are the sample input frames and the corresponding output semantic segmentations, respectively. These qualitative results serve as examples for justification and demonstration of the proposed methodology presented in Section III. We organize the presentation of our experimental results as the following. First, we introduce our experimental setup in Section IV-A, which covers the specifications of the ePE, our embedded clusters, as well as a description of the training dataset used. Then, we analyze the quantitative results measured from the real embedded clusters, compare them against the baseline approaches mentioned in Section II, and discuss the impacts of our methodology on $mIoU$ and *fps* for a number of cluster configurations in Section IV-B. Finally, we investigate into the breakdowns of the execution latencies for the ePEs for various cluster configurations in Section IV-C.

A. Experimental Setup

a) System configuration: We adopt NVIDIA Jetson TX2 as our ePE for all of our experiments due to its ease of programmability and wide availability. Each Jetson TX2 incorporates a Pascal GPU with 256 CUDA cores and 8 GB memory shared by the GPU and a hexa-core ARMv8 64-bit CPU complex. The detailed specification is provided in Table II. The embedded cluster is implemented by connecting multiple TX2's via a router with up to 1 Gbits/sec transmission rate. Jetson TX2 is an excellent option for embedded systems, as its power consumption is only 7.5 Watts on average, significantly lower than that of commercial desktop or server grade GPUs. Although Jetson TX2 is remarkable for a number of embedded

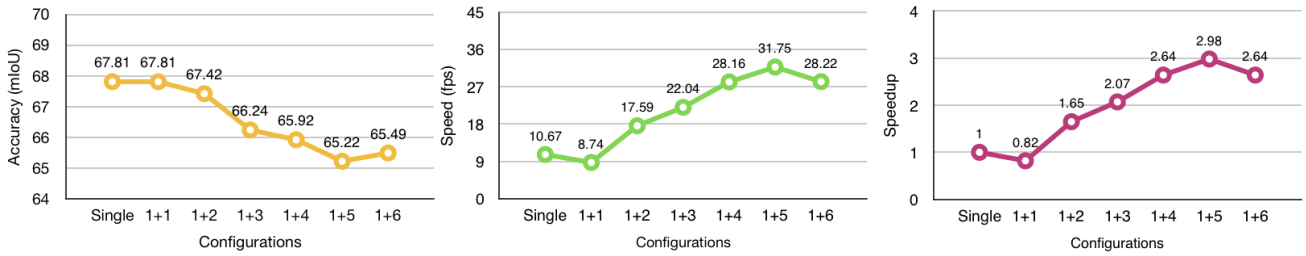

Fig. 6. Performance comparison of $mIoU$, *fps*, and $Speedup$ for different configurations.

system applications, however, it still fails to meet both the accuracy and real-time requirements in most semantic video segmentation tasks (please refer to the first four rows of Table III for the quantitative results and Section IV-B for a more detailed discussion). In order to validate the proposed distributed methodology on different cluster configurations, we employ up to seven TX2's in our experiments. Please note that contemporary embedded DCNN accelerators or ePEs other than NVIDIA Jetson TX2 can also fit in the proposed framework.

b) Dataset: We perform our experiments on the Cityscapes dataset [26], which is composed of urban street scenes from 50 different cities. There are totally 2,975 training images and 500 validation images in the dataset. The dataset contains pixel-level annotations with 19 classes. The annotation is provided on the 20^{th} frame of each 30-frame video snippet. We evaluate our framework on each Cityscapes validation snippet from the 1^{st} frame to the 20^{th} frame. We set the 1^{st} frame as our initial key frame and measure $mIoU$ on the annotated 20^{th} frame. All of the experiments are conducted on image frames of a fixed dimension of $512 \times 1,024$ pixels.

B. Comparison of the Quantitative Results

Table III presents and compares the quantitative results of the proposed methodology with the baseline approaches discussed in Section II for a number of system configurations. The first four rows correspond to the baseline approaches measured on a single TX2, while the remaining seven rows correspond to our methodology. The results of ENet, ERFNet, and ESPNet are obtained from [12]. The rest of the results are directly measured on our embedded systems. Please note that the resolutions of the input frames are the same (i.e., $512 \times 1,024$ pixels) for all of the configurations listed in Table III. Three quantities are compared in our experiments: $mIoU$, *fps*, and the speedup ratio. The baseline entry *SegPath* denotes that P_{seg} is used for processing every input frame region. The entry *Single* serves as the reference entry corresponding to the case where the default DVSNet [17] is performed on a single ePE. Each of the rest entries represents the configuration of one master ePE and the corresponding number of slave ePEs. For example, *1+3* means that one master ePE and three slave ePEs are used. The speedup ratio *Speedup* is the ratio of the entry's *fps* with respect to that of *Single*. The threshold $\mathcal{T}$ is set to 15 in all cases, which is the maximum interval for the master ePE to update a slave ePE's *local key buffer*, as discussed in Section III-C.

From Table III, it is observed that for the *Single* reference entry, the values of $mIoU$ and *fps* are 67.81% and 10.67, respectively. Although the $mIoU$ accuracy of *Single* is slightly less than that of *SegPath*, the former's *fps* is $7.02\times$ as compared to that of the latter. When the *Single* reference entry is compared with the other baselines, its *fps* is $1.16\times$, $2.96\times$, and $1.19\times$ as high as those of ENet, ERFNet, and ESPNet, respectively. The $mIoU$ accuracy of *Single* is $0.93\times$, $1.16\times$, $1.00\times$, and $1.12\times$ as compared with those of *SegPath*, ENet, ERFNet, and ESPNet, respectively, indicating that the accuracy of *Single* is comparable to the baseline methods. The speedup of *Single* mainly results from the advantages offered by DVSNet [17].

Nonetheless, an *fps* of 10.67 is still far from satisfactory for real-time needs, which typically require an *fps* up to 20 or even more. To demonstrate the effectiveness of the proposed distributed framework, a number of configurations with multiple slave ePEs are evaluated in our experiments. Table III and Fig. 6 compare the performance of different configurations. It can be seen from Fig. 6 that for most configurations, increasing the number of slave ePEs tend to deliver higher *fps*, with only a slight decrease in $mIoU$. Table 3 shows that the proposed framework achieves $1.65\times$, $2.07\times$, $2.64\times$, and $2.98\times$ *Speedup* for configurations *1+2, 1+3, 1+4*, and *1+5*, respectively. Configurations *1+1* and *1+6* do not follow the above increasing trend due to different reasons. For *1+1*, the *fps* is merely 8.74, which is even slower than that of *Single*. The decreased performance in *fps* is primarily caused by the data transmission overhead between the master ePE and the slave ePE. In other words, configuration *1+1* does not provide sufficient parallelism for the proposed framework to outweigh the communication latency between the ePEs. For *1+6*, the *fps* drops from 31.75 to 28.22, as compared with that of *1+5*. The decrease in *fps* is due to the fact that more slave ePEs may result in higher chances for the frame regions allocated to the slave ePEs to deviate from the key frame regions. As a result, the slave PEs tend to execute P_{seg} more often, leading to an overall performance drop in *fps*. Based on the above observations, we conclude that configuration *1+5* is the most ideal option in this experiment. Please note that different specifications of P_{seg}, P_{flow}, $\mathcal{T}$, and system implementations may lead to different optimal configurations.

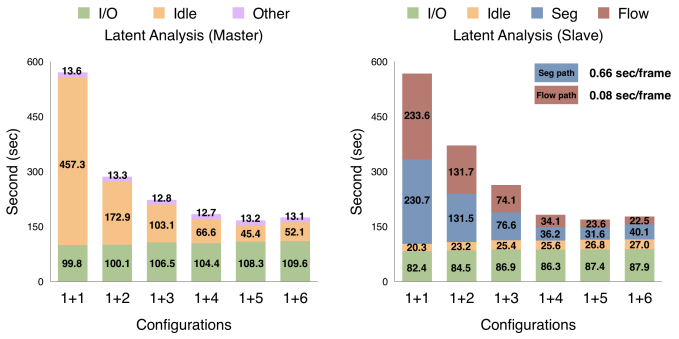

Fig. 7. Latency analysis for the master ePE and the slave ePE.

Table III and Fig. 6 also reveal that the $mIoU$ accuracy do not decrease significantly as the number of the slave ePEs increases. Even under configurations *1+5* and *1+6*, the $mIoU$ accuracies only decrease by 2.59% and 2.32% as compared with that of *Single*, respectively. The decreases in $mIoU$ are due to the fact that more slave ePEs may increase the average timestep interval between the current frame regions and the key frame regions, potentially decreasing the $mIoU$ accuracy of P_{flow} of the slave PEs, as described in Section III-A. The above observations validate that the proposed distributed framework does lead to performance enhancement in terms of *fps* and *Speedup*, with little degradation in $mIoU$. The results also justify the motivation for performing semantic video segmentation tasks on embedded clusters, especially when a single ePE is unable to deliver sufficient performance.

C. Latency Analysis for the Master and Slave ePEs

To further examine the proposed methodology, we conduct a latency analysis and plot breakdowns for the master ePE and the slave ePE under different cluster configurations. The breakdowns of the master ePE and the slave ePE are presented on the left-hand side and right-hand side of Fig. 7, respectively. The components of the breakdowns are different, as we intend to highlight the differences in functionality between the master ePE and the slave ePEs. The breakdowns of the master ePE focus on the time cost in data transmission (denoted as *I/O*) and the time spent on waiting for available slave ePEs (denoted as *Idle*), as the master ePE is mainly responsible for pre-processing the input frames as well as transmitting and aggregating data to/from slave ePEs. On the other hand, the breakdowns of the slave ePE focus on the time cost in data transmission, the time spent on waiting the master ePE for frame region allocation, as well as the time spent on the two execution paths (i.e., P_{seg} and P_{flow}). The numbers in Fig. 7 are derived by summing the latencies of the distinct components spent on the frames of the entire validation set.

From Fig. 7, it is observed that for the master ePE, the *I/O* time spent on data transmission (mainly contributed by frame region allocation) is similar under different configurations. This is because the number of input frame regions is identical for all of the configurations. In addition, these frame regions can only be transmitted one by one by the master ePE, no matter how many slave ePEs are employed in the system. On the other hand, the *Idle* time of the master ePE decreases drastically as the number of slave ePEs increases. It can be seen that under configuration *1+1*, the master ePE wastes most of its time waiting for the sole slave ePE to finish its tasks. The ratio of the *Idle* time to the *I/O* time is about $4.5\times$. With more slave ePEs, however, the average latency that the master ePE has to wait for an available slave ePE becomes significantly shorter, as it is more likely for the master ePE to acquire an idle slave ePE right after a frame region transmission. The *Idle*/*I/O* ratios of configurations *1+5* and *1+1* are only about one-tenth of that of configuration *1+1*. The above observations suggest that more slave ePEs tend to improve the efficiency of the master ePE, and partially validate the trend of *fps* plotted in Fig. 6.

For the slave ePE, the *I/O* time spent on data transmission is roughly the same for different configurations. The reason is similar to that of the master ePE. It is also observed that as the number of slave ePEs increases, the total time of each ePE spent on the execution paths decreases. This is because the number of input frames allocated to each slave PE decreases. For both of the breakdowns, the total amount of time also decreases as the number of slave ePEs increases. The above evidences suggest that the proposed distributed framework does improve the throughput and efficiency of embedded clusters for performing semantic video segmentation tasks, especially when multiple slave ePEs are incorporated into the system.

V. Conclusions

In this paper, we presented a framework for performing semantic video segmentation tasks on an embedded cluster. We embraced the advantages provided by DVSNet, and developed a distributed scheme for allocating different frame regions to different PEs. The PEs in the framework are coordinated in a master-slave hierarchy, and are regulated by a global and local key management scheme. Our experimental results demonstrated that the proposed methodology does lead to enhanced performance in terms of *fps* and *Speedup*, with little degradation in $mIoU$. The proposed framework is flexible in its implementations, and is especially suitable for embedded systems with limited computational resources such as self-driving cars, drones, or autonomous machines.

Acknowledgment

This work was supported by the Ministry of Science and Technology (MOST) in Taiwan under grant nos. MOST 108-2636-E-007-010 (Young Scholar Fellowship Program) and MOST 108-2634-F-007-002. H.-K. Yang acknowledges the student fellowship support from Novatek Microelectronics Corporation. T.-J. Fu, P.-H. Chiang, and C.-Y. Lee acknowledge the financial support from MediaTek Inc. The authors acknowledge the donation of NVIDIA Jetson TX2's from NVIDIA Corporation and NVIDIA AI Technology Center (NVAITC) used in this research.

REFERENCES

[1] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," *arXiv:1707.07012*, Aug. 2017.

[2] K. He *et al.*, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR)*, pp. 770-778, Jun. 2016.

[3] S. Ren *et al.*, "Faster R-CNN: Towards real-time object detection with region proposal networks," *IEEE Trans. Pattern analysis and Machine Intelligence (TPAMI)*, pp. 1137-1149, Apr. 2017.

[4] W. Liu *et al.*, "SSD: Single shot multibox detector," in *Proc. European Conf. Computer Vision (ECCV)*, pp. 21-37, Sep. 2016.

[5] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, "Encoder-decoder with atrous separable convolution for semantic image segmentation," *arXiv:1802.02611*, Mar. 2018.

[6] H. Zhao, J. Shi, X. Qi, X. Wang, and J. Jia, "Pyramid scene parsing network," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR)*, pp. 6230-6239, Jul. 2017.

[7] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Semantic image segmentation with deep convolutional nets and fully connected CRFs," *arXiv:1412.7062*, Jun. 2016.

[8] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, "Rethinking atrous convolution for semantic image segmentation," *arXiv:1706.05587*, Jun. 2017.

[9] M. Siam *et al.*, "RTSeg: Real-time semantic segmentation comparative study," *arXiv:1803.02758*, Aug. 2018.

[10] A. Paszke, A. Chaurasia, S. Kim, and E. Culurciello, "ENet: A Deep Neural Network Architecture for Real-Time Semantic Segmentation," *arXiv:1606.02147*, Jun. 2016.

[11] E. Romera, J. M. Alvarez, L. M. Bergasa and R. Arroyo, "ERFNet: Efficient Residual Factorized ConvNet for Real-Time Semantic Segmentation," *IEEE Trans. Intelligent Transportation Systems*, pp. 263-272, Jan. 2018.

[12] J. S. Mehta, M. Rastegari, A. Caspi, L. G. Shapiro and H. Hajishirzi, "ESPNet: Efficient Spatial Pyramid of Dilated Convolutions for Semantic Segmentation," in *Proc. European Conference on Computer Vision (ECCV)*," pp. 561-580, Oct. 2018.

[13] J. Dean *et al.*, "Large scale distributed deep networks," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pp. 1232-1240, Dec. 2012.

[14] A. Dundar, J. Jin, B. Martini, and E. Culurciello, "Embedded streaming deep neural networks accelerator with applications," *IEEE Trans. Neural Networks and Learning Systems*, pp. 1572-1583, Jun. 2017.

[15] P. G. López *et al.*, "Edge-centric computing: Vision and challenges," in *Computer Communication Review*, pp. 23-42, Oct. 2015.

[16] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov and, L.-C. Chen, "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation,"*arXiv:1801.04381*, Feb. 2018.

[17] Y.-S. Xu, T.-J. Fu, H.-K. Yang, and C.-Y. Lee, "Dynamic video segmentation network," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR)*, pp. 6556-6565, Jun. 2018.

[18] D. Maturana, S. Arora, and S. Scherer, "Looking forward: A semantic mapping system for scouting with micro-aerial vehicles," in *IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, pp. 6691-6698, Sep. 2017.

[19] I. Baek *et al.*, "Real-time detection, tracking, and classification of moving and stationary objects using multiple fisheye images," *arXiv:1803.06077*, Apr. 2018.

[20] T. Amert *et al.*, "GPU scheduling on the NVIDIA TX2: hidden details revealed," *IEEE Real-Time Systems Symposium (RTSS)*, pp. 104-115, Feb. 2018.

[21] B. K. P. Horn and B. G. Schunck, "Determining optical flow," *J. Artificial intelligence*, pp. 185-203, Aug. 1981.

[22] A. Dosovitskiy *et al.*,"FlowNet: Learning optical flow with convolutional networks," in *Proc. IEEE Conf. International Conference on Computer Vision (ICCV)*, pp. 2758-2766, Apr. 2015.

[23] E. Ilg *et al.*, "FlowNet 2.0: Evolution of optical flow estimation with deep networks," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR)*, pp. 1647-1655, Nov. 2017.

[24] P. Weinzaepfel, J. Revaud, Z. Harchaoui, and C. Schmid., "DeepFlow: Large displacement optical flow with deep matching," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR)*, pp. 1385-1392, Dec. 2013.

[25] T.-Y. Lin *et al.*, "Microsoft COCO: Common objects in context," in *Proc. European Conf. Computer Vision (ECCV)*, pp. 740-755, Sep. 2014.

[26] M. Cordts *et al.*, "The Cityscapes dataset for semantic urban scene understanding," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR)*, pp. 3213-3223, Jun. 2016.