Journal of Network and Computer Applications 225 (2024) 103866

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Research paper

SplitStream: Distributed and workload-adaptive video analytics at the edge

Yu Liang [a,*], Sheng Zhang [b], Jie Wu [c]

[a] School of Computer and Electronic Information/School of Artificial Intelligence, Nanjing Normal University, China
[b] State Key Laboratory for Novel Software Technology, Nanjing University, China
[c] Center for Networked Computing, Temple University, USA

ARTICLE INFO

Keywords:
Edge computing
Video analytics
Workload balancing

ABSTRACT

Deep learning-based video analytics is computation-intensive. Manufacturers such as Nvidia have launched many embedded deep learning accelerators and are rapidly gaining market share. However, the computing resources of such accelerators are still limited and heterogeneous. Although existing systems aim at optimizing video query tasks from a variety of perspectives, they rarely consider the general cooperation between heterogeneous edge devices and the dynamic workload of video content. In this work, we present SplitStream, a distributed system for accelerating video query tasks across heterogeneous edge devices, which is able to fully utilize the resources on each device and adapt to the workload dynamics. The key to achieving this is the data parallelism brought by the multi-instance mechanism and the dynamic workload adaptability brought by the two-level workload balancing mechanism. Evaluation results show SplitStream reduces the result retrieval time by up to 19% and improves the resource utilization by up to 234%.

1. Introduction

Video cameras are ubiquitous nowadays. Cities and organizations are steadily increasing the size and reach of their camera deployments, generating massive video data. A survey of tens of organizations shows that their average number of cameras has increased by almost 70% from 2015 to 2018 (International trends, 2018). Due to the breakthrough in deep learning (Voulodimos et al., 2018), we are able to extract useful information from video feeds using deep learning tools (Preeti and Sri, 2021; Zhou et al., 2022; Zhao et al., 2017), a.k.a. video query (Bolle et al., 1998; Wang et al., 2020). Performing video queries on retrospective videos generates query results, including the class, location, and confidence of each object in each frame of a video. These results are increasingly analyzed to support lots of tasks including traffic monitoring, surveillance, customer tracking, and so on.

A key problem is, *how can we analyze video feeds to acquire as many query results as possible in a given time?* (Xu et al., 2021). As we know, timely processing video queries require heavy computing resources that are far beyond the traditional video surveillance infrastructures (Teutsch et al., 2010). Besides, modern video query pipelines tend to use cascaded architectures (Jang et al., 2021; Kleinrouweler et al., 2021; Elgamal et al., 2020), which usually contain multiple Deep Neural Networks (DNNs) and thus require even more resources.

One possible solution to achieve high throughput of video query is to run video analytics tasks on the remote cloud. However, on one hand, the retrospective video data is usually stored in local cameras or storage servers, hence the data flow, storage cost, and processing cost of cloud services rise as the video data grows (Public, 2022; Elastic GPU, 2022); on the other hand, transferring the video data to the public cloud is sometimes forbidden due to data privacy (Shi et al., 2016).

An alternative way is to move video analytics tasks from the remote cloud to the edges, where data resides (Shi et al., 2016). A typical edge environment consists of many heterogeneous devices such as Internet of Things (IoT) devices, intelligent cameras, and other edge servers. These heterogeneous devices have different computing and memory capacities (Pasteris et al., 2019; Luo et al., 2021). There are several factors that make the local edge processing of the query task possible. Firstly, manufacturers such as Nvidia and Google are rapidly iterating on their embedded machine learning accelerators (Nvidia, 2022; Google, 2022; Intel, 2022), and existing research shows that cameras equipped with these accelerators are gaining market share at a high rate, which makes it possible to deploy simple deep learning-based video analytics tasks on a single camera. Secondly, an edge system usually shares a relatively large network bandwidth in its local networks through switches, private access points, and routers, which saves the cost of data transmission and reduces network traffic in the backbone network compared with cloud processing (Mao et al., 2017; Mach and Becvar, 2017).

However, the resource requirement of a cascading query pipeline usually far exceeds one single device's capability, it is then necessary to distribute the pipeline across multiple devices, since there are many

* Corresponding author.
E-mail addresses: liangyu@njnu.edu.cn (Y. Liang), sheng@nju.edu.cn (S. Zhang), jiewu@temple.edu (J. Wu).

https://doi.org/10.1016/j.jnca.2024.103866
Received 12 September 2023; Received in revised form 31 December 2023; Accepted 5 March 2024
Available online 19 March 2024

devices with scattered resources in an edge environment. However, there are several challenges to achieve this. Firstly, *it is hard to fully utilize scattered edge resources.* As mentioned before, there are more and more edge devices equipped with deep learning accelerators. In an edge network, the number of such devices is highly possible to far exceed the number of the pipeline primitives. If we deploy one primitive on exactly one device (Zhang et al., 2021), a large number of devices will be idle because they cannot participate in the processing, and many resources will be wasted. Secondly, *orchestrating heterogeneous devices needs careful design.* Due to the heterogeneity of edge devices, a task runs at different speeds at different edges. If one primitive has multiple instances running parallel, the instance on a relatively weak device could become a straggler and suffer from backlog. If we adjust the data input rate at the data source according to the slowest instance, the other instances may become relatively idle, causing resource waste. Finally, *video analytics workload is time-varying.* The third obstacle is the workload dynamics due to time-varying video content. For some query tasks, the complexity of video content may affect the computing workload. For example, if a query task needs to run classification for every object in a frame, then the more objects appear in a frame, the greater the amount of computation. In such a case, each primitive may experience workload dynamics, affecting the overall throughput. The common solution is to adjust task share for each instance and scale up/down the instances. However, deploying new DNN tasks is very slow and may impair other deployed instances (Singh et al., 2021; Yeh et al., 2019; Zhao et al., 2020) and the dynamics may have temporary spikes, misleading the balancing decisions.

Our key insight is that, video analytics tasks should be adaptively distributed to edge devices. Here, "adaptively" means the analytics workload of an edge device should be proportional to its available computational capacity, besides, we should adjust the analytics workload between edge devices from time to time if necessary.

In this paper, we develop SplitStream, a distributed and workload-adaptive video query processing system in the edge environment that addresses these challenges. In SplitStream, each query pipeline is abstracted as a Directed Acyclic Graph (DAG) in which each node represents a pipeline primitive (i.e., processing step). SplitStream intelligently deploys these primitives on different edge devices and adaptively adjusts the deployment when necessary. For the first challenge, SplitStream scales the number of instances of each primitive. By assigning one primitive's task to multiple instances, more resources from idle devices can be utilized. For the second challenge, we adopt a stochastic task partition strategy that assigns different amounts of tasks to different instances using stochastic sampling according to the devices' computing powers. For the third challenge, SplitStream applies a two-level workload balancing mechanism that adjusts the task partition in a short time interval and scales up or scales down the number of instances in a long time interval using a multi-queue filtering method. Evaluation results show SplitStream reduces at most 40% time in video query processing and doubles the utilization of devices.

The rest of this paper is organized as follows. Section 2 shows the motivation. Section 3 provides the overview of SplitStream. Section 4 provides the details of SplitStream. Section 5 evaluates the performance of SplitStream. We provide related works in Section 6 and conclude the paper in Section 7.

2. Motivation

In this section, we motivate the design of SplitStream. We firstly show video query pipelines are complex and resource-intensive. Then we show edge devices are heterogeneous and thus they have different processing capacities for video feeds. Finally, we show video analytics workloads are not static; instead, they are time-varying, requiring periodical workload balancing.

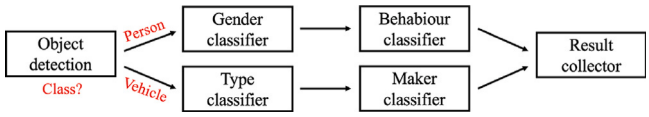

Fig. 1. A representative video query pipeline used in this paper. Based on the type of the detected object (vehicle or person), the task-specific module selects one of the two branches, each of which employs a cascaded sequence of classifiers to further identify the attributes of the object. For example, if the detected object is a vehicle, then it goes to the lower branch: classify its type (car, truck, etc.) and then recognize its maker.

2.1. Video query pipelines are directed acyclic graphs (DAGs)

The breakthrough in deep neural networks results in the emergence of massive accurate primitives of video analysis in the computer vision community. Modern live video analytics pipelines typically adopt a cascaded architecture which consists of a front-end object detector followed by a back-end task-specific module to perform a variety of analytics tasks on each of the detected object of interest within a video frame. Object detection primitive (e.g., Faster-RCNN Ren et al., 2015, YOLO Redmon et al., 2016, SSD Liu et al., 2016, and RetinaNet) is the core in many video query systems (Zhang et al., 2021). The task-specific primitives including DNN based image classification such as Resnet (He et al., 2016), VGG (Simonyan and Zisserman, 2014), and GoogleNet (Szegedy et al., 2015). Such primitives are usually resource-intensive.

Fig. 1 displays a representative video query pipeline discussed in this paper. It detects the objects in each video frame and then extracts specific information using image classification primitives according to the class of each object. The task-specific DAG consists of two branches. Based on the type of the detected object (vehicle or person), the task-specific module selects one of the two branches, each of which employs a cascaded sequence of classifiers to further identify the attributes of the object. For example, if the detected object is a vehicle, then it goes to the lower branch: classify its type (car, truck, etc.) and then recognize its maker.

This query pipeline has widespread usage in many scenes. For example, the gathered vehicle information can help manage traffic by analyzing the number and types (e.g., car, truck) of vehicles, while the personal information can help search for missing people or criminal suspects with specific characteristics. In the pipeline shown in Fig. 1, video data are extracted into frames by the video data source, and then one frame is sent into an object detector, which extracts the regions of interest (ROI[1]) using the bounding boxes and classes, and encodes them into small pictures. At this point, one frame is turned into multiple pictures containing exactly one object of person or vehicle. Each ROI will be sent to the following classifiers (e.g., gender classifier, behavior classifier) independently according to its class. Each classifier classifies an attribution such as the gender of a person and adds it into the intermediate result. Finally, the results collector will collect the analytics results and return the results to the user.

2.2. Edge devices are heterogeneous

Recent surveys (Fortinet, 2022; Hsieh et al., 2018) show that edge storage costs are getting lower and lower in recent years. The cost per gigabyte of SD cards drops from $0.35 in 2017 to $0.13 in 2020. New video analytic schemes store video files on the local storage of cameras and stream them to the cloud when a query task is committed (Xu et al., 2021). However, transmitting large video files consumes a lot of bandwidths of public network (Hung et al., 2018) and may violate data privacy. At the same time, the edge network is usually connected by private switches, APs, and routers (Shi et al., 2016), providing large bandwidth resources. If the query can be done locally, the traffic in the

[1] ROI means the boundaries of an object of interest on an image.

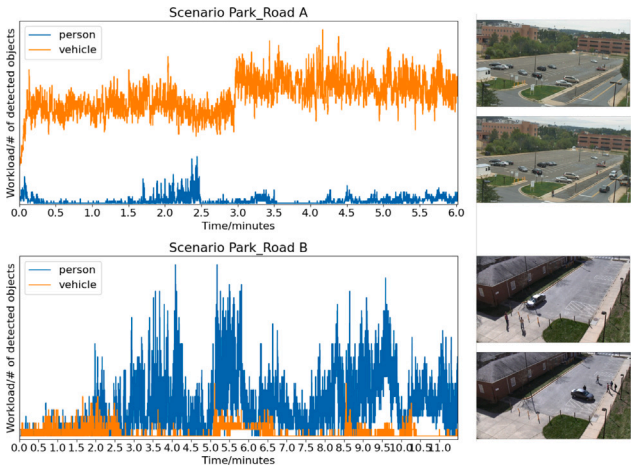

Fig. 2. Time-varying workloads. Scenario A is a large parking lot with dozens of vehicles next to an intersection. Vehicles and people enter or leave the parking lot. Scenario B is a vacant lot behind a house. Many pedestrians stay in the scene for a long time while a few pedestrians and vehicles pass through the road quickly.

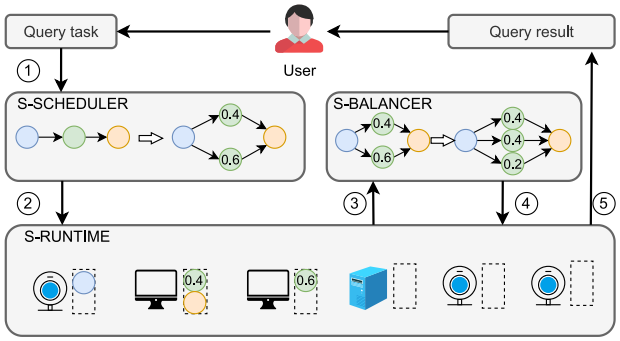

Fig. 3. SplitStream Architecture. A user commits a query task to S-Scheduler. S-Scheduler abstracts the query task into a DAG and generates an initial deployment scheme (Section 4.1), then informs the devices in S-Runtime to run instances. During running time, S-Balancer performs workload balancing periodically (Section 4.2).

public network can be reduced, and so as the data flow cost. Due to the growth of edge deep learning accelerators and the cheap storage with abundant bandwidths, it is promising to process the video query task locally. To achieve that goal, the scattered resources on different devices should be fully utilized.

Due to the popularity of deep learning, hardware manufacturers such as Nvidia and Google are speeding up iteration and promotion of their embedded accelerators, which can significantly enhance the computing power of edge devices, such as surveillance cameras (Meel, 2021). Research shows that the market share of cameras equipped with such accelerators increases from 355 million dollars in 2018 to 1,120,000 million dollars in 2023 (Softtek, 2020). This means that there will be more devices with deep learning capability in the edge environment such as smart cameras.

Due to different manufacturers and product lines, these devices may have heterogeneous computing and memory resources, which makes the same DNN task run at different speeds on different devices. Moreover, some complex DNN tasks need more memory to store parameters and intermediate result, and cannot be deployed on devices with limited memory. Table 1 shows inference times of some typical DNNs for object detection and image classification on several edge devices. We can see the inference time varies a lot across devices due to their heterogeneities. Specifically, when then computational capacity of an edge device increases, we can see that the inference time decreases.

Table 1
Inference time on typical devices.

Device	Yolo-v5m	Resnet18	Resnet34	Resnet50
Jetson Nano[a]	0.574 s	0.0231 s	0.045 s	0.061 s
AGX Xavier[b] 15 W	0.132 s	0.0167 s	0.029 s	0.043 s
AGX Xavier 30 W	0.105 s	0.0169 s	0.025 s	0.032 s
AGX Xavier MAXN	0.071 s	0.0081 s	0.015 s	0.018 s
E5 2640v4 Server	0.158 s	0.0214 s	0.041 s	0.047 s

[a] 72 GFLOPs, 4 GB/2 GB.
[b] 2.8 TFLOPs, 64 GB/32 GB.

2.3. Inference workloads are time-varying

To explain how video contents affect the workload, we choose two typical surveillance video clips selected from the VIRAT Ground Dataset (Oh et al., 2011). Scenario A is a large parking lot, scenario B is a small parking lot, and both are next to a road. The evaluated pipeline is shown in Fig. 1. Fig. 2 shows the results in which we use the number of detected objects to represent the workload. We can see that the number of vehicles or persons detected by the object detector directly affects the subsequent classifiers' workloads. There are 2 factors that affect the workload: (i) video content is changing over time. In a period of time, there may be more or fewer vehicles/persons that enter/leave the view. (ii) some objects may appear at the corner of the view and stay in the view only for a short time. Moreover, there may be many objects detected by the object detector incorrectly, causing workload spikes.

Such workload dynamics may have significant impacts on the user's experience. Users are mainly concerned about throughput, i.e., the number of results acquired per second. The workload spikes may result in some pipeline primitives being overloaded and becoming bottlenecks, reducing the overall throughput.

3. SplitStream overview

We present SplitStream, a distributed and workload adaptive system for scheduling video query tasks in the edge environment, which can fully utilize resources scattered on different devices, orchestrate heterogeneous devices to reduce bottlenecks, and handle workload dynamics to further improve the throughput.

3.1. SplitStream architecture

SplitStream mainly contains *S-Scheduler* (Section 4.1), *S-Balancer* (Section 4.2), and *S-Runtime*, as shown in Fig. 3. *S-Scheduler* helps abstract the query task into a DAG and generate an initial deployment scheme, including the number of instances for each pipeline primitive, the workload share of each instance, and the deployment device of each instance. *S-Balancer* targets to handle workload dynamics. It gathers workload information of each instance and performs workload balancing algorithms. *S-Runtime*, located on smart cameras, PCs, or servers, holds one or more instances with specific workload shares according to the initial deployment scheme.

The component that actually runs the query task is *S-Runtime*. Fig. 4 illustrates *S-Runtime* on each device. The aggregator layer provides a uniform interface for each device, and the dispatcher layer enables result dispatch to different downstream instances with different workload assignments. Both of them provide a uniform interface that enables direct data transfer. The task queues buffer the tasks to be processed by primitive instances and provide information such as processing rate, data incoming rate, and processing rate for *S-Balancer* to perform workload balancing.

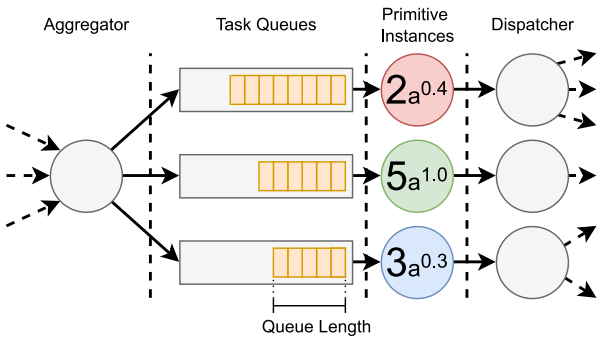

Fig. 4. *S-Runtime* on each device. The aggregator and dispatcher provide a uniform interface for direct data transfer between devices; the task queues buffer tasks of each primitive instance and monitor workload for balancing.

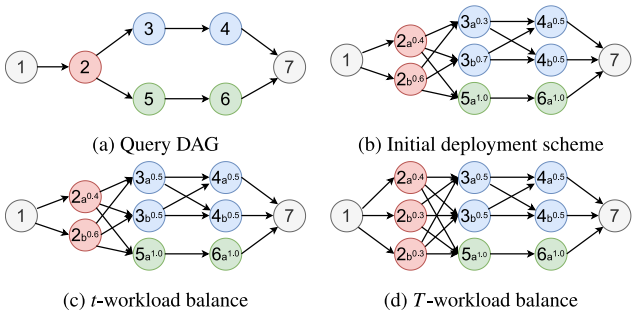

Fig. 5. Example of Query DAG and deployment. (a) the query DAG of the pipeline in Fig. 1; (b) the initial deployment generated by *S-Scheduler* (e.g., $2_{a^{0.4}}$ means it is an instance of 2 and instance 1 should send 40% of its processing results to $2_{a^{0.4}}$); (c) the deployment after performing t-workload balance, which changed the workload assignments of 2_a and 2_b; (d) the deployment after performing T-workload balance, which adds a new instance of 2.

3.2. Workflow

The workflow from a user submitting its video query request to the user obtaining the query results is shown in Fig. 3: ① A user submits the task configuration to *S-Scheduler*. The configuration describes the DAG pipeline of the query task (e.g., Fig. 5(a) shows a DAG pipeline). It contains information of each pipeline primitive, including its upstream and downstream primitives and the resource requirements (i.e., computation and memory). ② *S-Scheduler* uses the task configuration and the specifications of all devices (i.e., computing power and memory size) to generate an initial deployment scheme as shown in Fig. 5(b). Each node in the scheme is an instance of a pipeline primitive and one primitive can have multiple instances with different workload assignments. For example, instance $3_{a^{0.3}}$ in Fig. 5(b) is responsible for processing 30% of the workloads received from the upstream instances (i.e., $2_{a^{0.4}}$ and $2_{b^{0.6}}$), which means $2_{a^{0.4}}$ and $2_{b^{0.6}}$ need to send 30% of their processing results to $3_{a^{0.3}}$. After obtaining the deployment scheme, *S-Scheduler* informs *S-Runtime* on each device to start the instances and begin video processing. ③ To accommodate the workload dynamics and reduce bottlenecks, *S-Runtime* sends instance workload states to *S-Balancer* within heartbeat packets. *S-Balancer* performs t-workload balancing every t time and T-workload balancing every T time, in which $t < T$. The result is a new deployment scheme as shown in Figs. 5(c) and 5(d). The t-workload balancing adjusts workload partitions for the instances of a primitive and the T-workload balancing adds or removes instances of a primitive. ④ *S-Balancer* then informs *S-Runtime* to apply the scheme changes. ⑤ Query results are sent to the user finally.

3.3. Deployment problem formulation

In this subsection, we formally present the pipeline deployment problem. Main notations are listed in Table 2 for quick reference.

Denote the set of edge devices by V. Without loss of generality, we assume edge devices are connected through switches by using wired cables, thus pairwise bandwidth is relatively sufficient. We model the query pipeline as a directed acyclic graph $D = (M, E)$, where a node $g \in M$ represents a pipeline primitive (e.g., object detection or gender classification) as well as the video data source and the result collector.

We want to maximize the number of frames processed per second. To achieve the goal, we want to fully utilize the heterogeneous resources across all edge devices. We break the pipeline DAG into primitives with each of them having a fixed memory consumption M_g.

Let C_g be the computation workload of primitive g in a single video frame, which is quantified by the amount of computations (e.g., FLOPS). We assume for now that each primitive has a fixed workload according to a given throughput f, then the workload of g per second at throughput f is $C_g \cdot f$. As shown in Fig. 5(b), each primitive can have multiple instances that run on different edge devices, and the maximum possible number of instances of each primitive is $|V|$. Let the number of all possible deployment schemes (also DAGs) be N. We use $c_i \in \{0,1\}$, $\forall i \in N$, to indicate whether the ith deployment scheme is used.

The corresponding DAG i is $D_i = (M_i, E_i)$. For the ith DAG, each instance m in M_i should undertake a part of the computing workload of the original primitive. The specific portion depends on which device it will be deployed on and the instances already deployed on that device. Let s^i_m denote m's workload share of the original primitive g. $x^i_{m,g}$ serves as an indicator of whether m is an instance of g. $p^i_{m,u}$ indicates that if m will be placed on device u. Then the computation constraints can be written as follows:

$$\sum_{g\in V}\sum_{i\in N}\sum_{m\in M_i} c_i C_g f s^i_m x^i_{m,g} p^i_{m,u} \le C_u, \quad \forall u \in V, \tag{1}$$

$$\sum_{g\in V}\sum_{i\in N}\sum_{m\in M_i} c_i M_g x^i_{m,g} p^i_{m,u} \le M_u, \quad \forall u \in V, \tag{2}$$

$$x^i_{m,g} \in \{0,1\}, \quad \forall m \in M_i, \forall g \in V, \forall i \in N, \tag{3}$$

$$p^i_{m,u} \in \{0,1\}, \quad \forall m \in M_i, \forall i \in N, \forall u \in M, \tag{4}$$

$$s^i_m \in [0,1], \quad \forall m \in M_i, \forall i \in N, \tag{5}$$

where C_u and M_u represent the computing and memory capacity of device u respectively. Eqs. (1) and (2) ensure that the computation and memory requirements of instances deployed on device u cannot exceed its capacity C_u and M_u, respectively.

The workload share of all instances of a primitive g should be exactly 1, and each instance m must be deployed on exactly one device. We have the following constraints:

$$\sum_{m=1}^{M_i} s^i_m x^i_{m,g} = 1, \quad \forall g \in V, \forall i \in N, \tag{6}$$

$$\sum_{u=1}^{V} p^i_{m,u} = 1, \quad \forall m \in M_i, \forall i \in N. \tag{7}$$

And our objective is

$$\max f. \tag{8}$$

Combining Eqs. (1) and (8) forms the pipeline deployment problem. The deployment scheme consists of c_i, $p^i_{m,u}$ and s^i_m. Fig. 5(b) shows an example of a deployment, in which $2_{a^{0.4}}$ means it is an instance of primitive 2 and instance 1 should send 40% of its processing results to $2_{a^{0.4}}$.

Table 2
Main notations for reference.

Notation	Description
V	Set of edge devices
D	Query pipeline of DAG structure
M	Set of pipeline primitives
E	Set of data dependencies among primitives
M_g	Memory consumption of primitive $g \in M$
C_g	Computation workload of primitive $g \in M$
f	Number of frames processed per second
N	Number of all the possible deployment schemes
c_i	Whether the ith deployment scheme will be used
D_i	The ith deployment scheme
M_i	Set of primitive instances in D_i
E_i	Set of data dependencies among instances in D_i
s^i_m	Workload share of instance $m \in M_i$
$x^i_{m,g}$	Whether m is an instance of primitive g
$p^i_{m,u}$	Whether instance m will be placed on device $u \in V$
C_u	Computation capacity of device $u \in V$
M_u	Memory capacity of device $u \in V$

Algorithm 1: Initial deployment generation

Input: Primitive confgs., device confgs.
$scheme \leftarrow \emptyset$; $f \leftarrow 0$; $i \leftarrow 1$;
while $i \leq k$ **do**
  $f \leftarrow f + \delta$; $scheme' \leftarrow$GetDeploymentScheme(f);
  if $scheme' \neq \emptyset$ **then**
    $scheme \leftarrow scheme'$;
    break;
  $i \leftarrow i + 1$;
if $scheme = \emptyset$ **then**
  $scheme \leftarrow$randomly deployment;
return *scheme*;

4. SplitStream details

In this section, we provide details of S-Scheduler and S-Balancer, which generates the initial deployment scheme and handles workload dynamics, respectively.

4.1. S-scheduler: initial deployment generation

As the mixed integer linear programming (MILP) problem is NP-Hard and its solution space grows exponentially, it is hard to search all possible solutions to find the optimal one. Besides, achieving the optimal solution is not cost-efficient due to the workload dynamics, thus, we adopt a simple but efficient way.

Simultaneously considering f, c_i, $p^i_{m,u}$ and s^i_m is very difficult. However, we find it easy to find a feasible deployment scheme under a fixed throughput f. Once f is given, the computation resource C_g for each primitive g can be estimated. The user can manually set the number of tasks to be processed in a single frame.

Alg. 1 describes the idea. We continuously increase the throughput f with a small step δ and calculate the computation and memory requirement of each primitive, then try GetDeploymentScheme at most k times to achieve a feasible scheme. The loop terminates when all k times of trials fail to achieve a feasible scheme.

The GetDeploymentScheme algorithm depicted in Alg. 2 traverses primitives of the pipeline (e.g., Fig. 5(a)) except for the data source and the result collector in topological order. For each primitive, it continuously selects available devices that have enough resources. It deploys instances on these available devices one by one and accordingly assigns computing tasks to them until the requirement of workload computation is satisfied. If the throughput cannot be satisfied, the algorithm will return the empty set.

Algorithm 2: GetDeploymentScheme(f)

Input: Throughput f
$M \leftarrow$ sort the primitives in topological order;
$scheme \leftarrow \emptyset$;
for each primitive g in M **do**
  $V \leftarrow$ get the set of edge devices;
  $w \leftarrow C_g \cdot f$;
  while $V \neq \emptyset$ and $w > 0$ **do**
    $u \leftarrow$ a randomly selected device from V;
    if u has enough resources to deploy g **then**
      $\theta \leftarrow \min\{w$,available resources of $u\}$;
      allocate θ resources of u to create an instance of g;
      $w \leftarrow w - \theta$;
      update the available resources of u;
      update *scheme*;
    $V \leftarrow V - \{u\}$;
  if $w > 0$ **then**
    return $\emptyset$;
return *scheme*;

Pre-profiling. The key to implement Alg. 1 is estimating the workload C_g of primitive g incurred by one frame. However, it is not trivial to achieve appropriate estimations due to the workload dynamics. A common way is manually setting an average C_g according to previous experiences, and it may not achieve a good result because of the dynamics of future video clips. Instead, we use a pre-profiling technique to handle this. During the video capture time or before running Alg. 1, the pre-profiler samples video clips to be queried with fixed time intervals, generating results and averaging them as the estimations of C_g and M_g.

4.2. S-Balancer: adaptive balancer

S-Balancer in Fig. 3 gathers workload states from *S-Runtime* and applies the new scheme to *S-Runtime*. However, balancing the workload on each device is not trivial due to the complexity of mutual dependencies: one device can have instances of different primitives, which means adjusting the workload of a primitive (including multiple instances) may easily affect co-located instances of other primitives.

Building an accurate model is too expensive to implement, so we adopt a simple but efficient solution: two-level workload balancing. The first level is to adjust the workload assignments of the instances belonging to the same primitive, and the second level is to add or remove instances of a primitive.

We observe that the second level balancing should be adopted carefully for the following reasons. Firstly, creating a new instance for some primitive is costly. Secondly, the workload dynamics contain many spikes, which may easily mislead the decisions of *S-Balancer*.

To tackle the problems, we divide time into intervals of equal length t. Let $T = nt$ and $n > 1$. We adjust the workload assignments of the instances belonging to the same primitive every t seconds and create or invoke instance every T seconds.

To model the state (busy or idle) of instances, we use the information gathered by *S-Balancer*, which are the task queue length len_m, task arrival rate in_m, and task processing rate out_m of each instance m. An instance can be considered overloaded when the tasks in the queue and the tasks arrived per second exceeds the processing rate. Thus, we can define the imbalance index of instance m as

$$ib_m = \frac{in_m + len_m/t}{out_m}. \tag{9}$$

When the ib_m is greater than a threshold β, the instance m can be considered overloaded.

Algorithm 3: t-workload balancing

Input: Q_g for each primitive g
$Q \leftarrow \emptyset$;
for each instance I_u^g **do**
  calculate ib_u^g using Eq. (9);
  insert ib_u^g into Q_g;
for each Q_g **do**
  if Q_g.top $\geq \beta$ **then**
    insert Q_g.top into Q;
while $Q \neq \emptyset$ **do**
  $\hat{ib_u^g} \leftarrow Q$.top;
  let $\hat{u}, \hat{g}$ be the device and the pipeline primitive of $\hat{ib_u^g}$;
  if exists relatively idle $I_{u'}^{\hat{g}}$ **then**
    spare part of $I_{\hat{u}}^{\hat{g}}$'s task to $I_{u'}^{\hat{g}}$;
    return;
  Q.pop;

4.2.1. t-workload balancing

S-Balancer maintains an instance status ordered list for each primitive, which records each instance's imbalance index (i.e., Eq. (9)). In the beginning of each time interval t, S-Balancer calculates each node instance's imbalance index and updates the instance lists. Then, S-Balancer puts the largest imbalance index of each logical node into a priority queue. If the largest imbalance index in the queue is greater than a threshold β, S-Balancer then tries to adjust the workload assignments of the instances.

Let I_u^g denote an instance of logical node g deployed on device u, and the imbalance index is ib_u^g. Alg. 3 shows the algorithm. As the instances on devices are interdependent, S-Balancer tries to select instances with a higher imbalance index in each time slice t for adjustment. As long as the adjustment is successful, S-Balancer quits the procedure. In other words, at most only one primitive's instances can be adjusted every t seconds in order to reduce the mutual influences between different primitives.

4.2.2. T-workload balancing

Similarly, S-Balancer adjusts the number of instances of at most one primitive every T seconds, and at most one more instance would be added. Each newly-deployed instance has a lease term of T, which means the algorithm should check the newly-deployed instances every T seconds and remove the instance when its workload is below a threshold.

To filter out workload spikes, we adopt a delayed reaction method. Specifically, we maintain two queues. Instances that are detected with a high workload are added to the tier 1 queue but not immediately deployed. Instances that are detected with high workloads twice in two adjacent time intervals are moved to the tier 2 queue. In each interval, *S-Balancer* attempts to deploy a new instance for the highest instance in the tier 2 queue. Alg. 4 summarizes our idea. We maintain two sorted queue: Q_1 and Q_2. Every T seconds, we calculate each instance's imbalance index and move them into or out of the two queues and try to deploy a new instance for the most imbalanced primitive.

The removal of instances is the same as Alg. 4. We also maintain two tiers of queues and remove the idlest instance. We utilize the resources in a greedy way, which means in every T seconds, we perform the instance creation first and remove the idlest instance if the creation is infeasible.

Algorithm 4: T-workload balancing

Input: Q_1 and Q_2
for each instance I_u^g **do**
  calculate ib_u^g using Eq. (9);
  if $ib_u^g \geq \beta$ **then**
    if instance $I_u^g \in Q_2$ **then**
      continue;
    if instance $I_u^g \in Q_1$ **then**
      Q_1.remove(I_u^g); Q_2.insert(I_u^g);
    else
      Q_1.insert(I_u^g);
  else
    if $ib_u^g \in Q_1$ **then**
      Q_1.remove(ib_u^g);
    if $ib_u^g \in Q_2$ **then**
      Q_2.remove(ib_u^g); Q_1.insert(ib_u^g);
while $Q_2 \neq \emptyset$ **do**
  $\hat{ib_u^g} \leftarrow Q_2$.front;
  let $\hat{u}, \hat{g}$ be the device and the primitive of $\hat{ib_u^g}$;
  $deviceSet \leftarrow$ devices with enough memory;
  if $deviceSet \neq \emptyset$ **then**
    $u' \leftarrow$ device with the most available resource in $deviceSet$;
    create an instance $I_{u'}^{\hat{g}}$;
    return;
  Q_2.pop;

5. Performance evaluation

5.1. Evaluation settings

We implement SplitStream using Python. Message queues based on ZeroMQ (2022) are used to communicate between different modules. The instances of primitives are implemented using Python threads.

Pipelines. We evaluate the performance of SplitStream using two video analytics pipelines, shown in Fig. 1 of this paper and Fig. 1 in Zeng et al. (2020). We use the core codes of Yolo-v5s (YoloV5, 2022) as the object detector and prune out unneeded functional modules. The gender classifier and behavior classifier on the person branch are implemented using Resnet18 (He et al., 2015), while the vehicle color, type, and maker classifiers as well as explosive classifier are implemented using Resnet34 (He et al., 2015).

Metrics. We use "retrieval rate" (Xu et al., 2021) and "resource utilization" in performance comparison. In video query tasks, a query result is a target in a frame, with related information obtained from pipeline processing. As we know, users usually want to obtain as analytics results as fast as possible, thus, we use retrieval rate to indicate the analytics speed of a pipeline. In our evaluation, we set the percentage marks of results obtained as 50%, 75%, 95%, and 100%, in which 100% means all results are collected. For resource utilization, we only consider the average normalized utilization of GPU accelerators on edge devices.

Baselines. *Central* deploys the full query pipeline on the most powerful device, i.e., an Nvidia Xavier AGX in our experiment, which is commonly used in many existing systems such as NoScope (Kang et al., 2017) and VideoStorm (Zhang et al., 2017). *Distream* adaptively balances the workloads across smart cameras and partition the workloads between cameras and the edge cluster uses partition points of pipeline DAGs. Furthermore, we also want to know the contribution of each component of SplitStream, thus, we use *SplitStream-mi* to represent

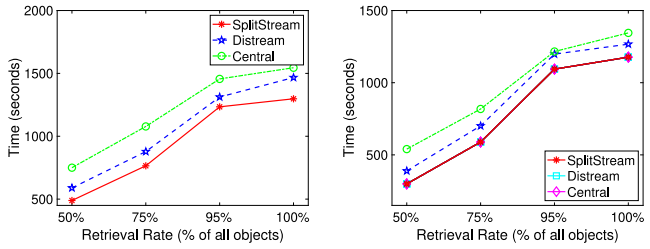

Fig. 6. Retrieval rate comparison.

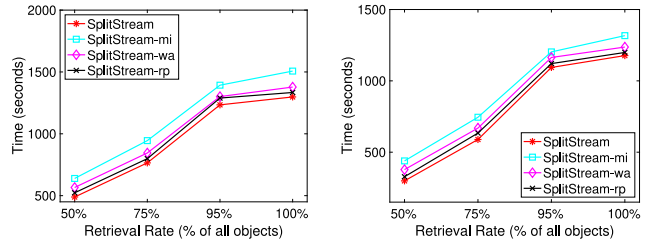

Fig. 7. Contribution of each component of SplitStream on retrieval rate.

SplitStream without the multi-instance feature, hence each primitive of the query pipeline can only have one instance; we use *SplitStream-wa* to represent workload-agnostic SplitStream, i.e., it does not perform t-workload and T-workload balancing; we use *SplitStream-rp* to represent SplitStream without pre-profiling.

Devices and Datasets. We evaluate the performance in two settings: *Small-S* and *Setting-L*. The first one consists of 1 Nvidia Jetson Nano 4 GB equipped with a 128-core Maxwell GPU, 1 Nvidia AGX Xavier with limited 8 GB memory, and a CPU server equipped with 2 Intel E5 2640v4 CPUs. Two video clips shown in Fig. 2 are used for evaluation. The results on Setting-S are averaged over these two video clips. The second one is large-scale simulation, in which we simulate an edge environment with 4 Nvidia Jetson Nano 4 GB, 4 Nvidia AGX Xavier, and 2 CPU server equipped with 2 Intel E5 2640v4 CPUs. We use publicly available live video streams from six traffic cameras deployed at different traffic intersections and roads in Jackson Hole, WY (JacksonHole, 2019). The results on Setting-L are averaged over these six videos.

5.2. Retrieval rate

Fig. 6 shows the comparison on retrieval rate of Central, Distream, and SplitStream in two experiment settings. We see that, SplitStream outperforms Distream and Central significantly in either setting. For example, in Setting-S, the time for Distream and Central to retrieve all objects is 113% and 119%, respectively, of that of Splitstream. The main reason is that, SplitStream may split a video analytics primitive into multiple instances and assign different portions of tasks to maximize the throughput, while Distream uses partition points on a DAG. Furthermore, SplitStream uses a two-level workload balance mechanism to handle video content dynamics, while Distream only dynamically updates the partition points. Note that, since there are more resources in Setting-L than Setting-S, the retrieval time is relatively small in Setting-L.

Fig. 7 shows the contribution of each component of SplitStream in terms of retrieval rate in two settings. In general, SplitStream achieves the best performance, while SplitStream-mi has the worst. This is because SplitStream-mi disable the multi-instance feature, thus it is meaningless to do workload balancing. Besides, without the workload balancing mechanism, SplitStream-mi suffers from huge performance degradation. SplitStream-wa deploys the pipeline with multi-instance enabled but without workload balancing, which means it heavily depends on the initial deployment schema. We know, S-Scheduler takes DAG configurations and device resources into account and generates an initial deployment. Note that the policy is fixed and relies on the constants we set up at the beginning, which is hard to estimate without pre-profiling.

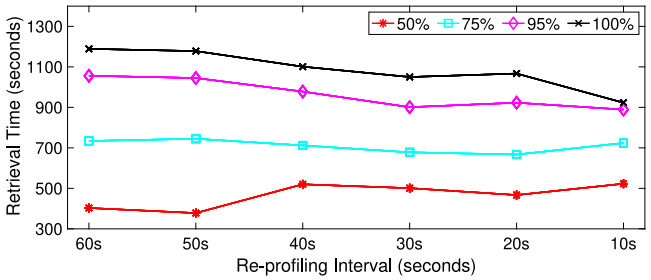

Fig. 8. The effect of re-profiling interval under Setting-S. Increasing re-profiling frequency improves the performance.

The comparison between SplitStream-rp and SplitStream in Fig. 7 shows the effect of pre-profiling. Pre-profiling catches the overall workload of each primitive and assigns the nodes that tend to have a high workload more resources. Our strategy of workload balance would not attempt to add or remove the instances of the initial deployment scheme, thus the pre-reserved resources (which means more instances for the potential high primitives) help improve the throughput.

We are also interested in evaluating the effect of the re-profiling interval. Fig. 8 shows the results under Setting-S; the results under Setting-L is similar and thus is omitted. We can see that, increasing re-profiling frequency indeed reduces the retrieval time.

5.3. Resource utilization

Table 3 shows the evaluations results in terms of resource utilization. The utilization is achieved by averaging the normalized resource utilization of all devices. A higher utilization means making full use of heterogeneous resources.

The CPU/GPU utilization roughly shows the utilization of computing resources on all devices, including the GPU/accelerators on the cameras and the CPUs on the storage servers. Central and SplitStream-mi have low utilization because both of them do not create multiple instances for each primitive. Without workload balancing, the total amount of CPU or GPU computing resources used is not much different. SplitStream-wa outperforms Central and Distream; it splits the task stream of a single primitive into many instances on different devices, which means there is more than one device sharing its tasks, and the computing resources of CPU or GPU on these devices are used, resulting in much higher utilization. In both settings, SplitStream achieves the best resource utilization. Since there are more resources in Setting-L than Setting-S, the resource utilization under Setting-L is smaller than that under Setting-S.

The memory utilization results is basically consistent with the CPU/GPU utilization results. The memory usage is almost entirely determined by the number of instances deployed. Central and SplitStream-mi have the almost same memory utilization because they have the same number of instances. Compared with SplitStream-wa and SplitStream-rp, they only use no more than half of all the available memories and leave a lot of memory idle. SplitStream uses a two-level workload balance mechanism to handle video content dynamics and thus may create more instances of video analytics primitives, while Distream only dynamically updates the partition points, therefore, SplitStream achieves the highest resource utilization.

Table 3
Resource utilization results.

	Setting	Central	Distream	SplitStream	SplitStream-mi	SplitStream-wa	SplitStream-rp
GPU/CPU	Setting-S	0.41	0.78	0.96	0.53	0.82	0.87
	Setting-L	0.23	0.52	0.75	0.36	0.65	0.70
Memory	Setting-S	0.56	0.73	0.94	0.62	0.78	0.82
	Setting-L	0.30	0.43	0.72	0.39	0.59	0.62

6. Related work

Related studies can be roughly grouped into 3 types.

The first type of existing studies assumes that the camera is not equipped with accelerators like GPUs and cannot run DNN tasks but can only run simple tasks such as tracking algorithms, and the actual DNN inferences are processed in the cloud server. Because all of the main work is on the cloud, their main concern is to save network costs. AWStream (Zhang et al., 2018) adjusts the video encode quality, resolution, and frame rate to adapt to the dynamic change of network bandwidth, trading off between accuracy and bandwidth. Reducto Li et al. (2020) only sends frames that have relatively large inter-frame difference to the cloud server. Chameleon (Jiang et al., 2018) uniformly samples frames in a video stream, adjusts the frame resolution, and decides an appropriate DNN model to achieve bandwidth-accuracy tradeoff.

The second type assumes that the camera has some capability of running light-weight DNNs, thus using the camera to run cheap DNN models to reduce inference cost and save bandwidth. However, the light-weight DNNs on the camera are usually used to filter out frames/regions with low confidence. NoScope (Kang et al., 2017) uses specialized DNN models to reduce inference overheads for throughput gain with small accuracy loss. Focus (Hsieh et al., 2018) uses the camera to run a cheap DNN model to generate the object index, enabling fast retrospective queries on the cloud. O^3 (Hanyao et al., 2021) combines camera-side lightweight tracking with edge-side computation-intensive detection to improve object detection performance.

The third type also assumes the camera has accelerators like GPUs and can run cheap DNNs, but they split the video query pipeline across the camera and the cloud servers. VideoEdge (Hung et al., 2018) partitions the pipeline on cameras and multi-level clusters according to their respective computational power. ATTEN (Yan et al., 2022) proposes attention-aware on-device object detection. Distream (Zeng et al., 2020) is the closest work to SplitStream. Although both Distream and SplitStream are built upon a distributed architecture at the edge and consider the workload balance across cameras and video content dynamics, they have three main differences. First, SplitStream targets more general video query pipelines that may be unable to be fully deployed on a single camera, while Distream uses a fine-tuned and pruned pipeline that can be fully deployed on each camera. Second, SplitStream splits some DAG nodes into multiple instances and assigns different portions of tasks to maximize the throughput, while Distream uses partition points on DAG to realize that. For workload balance, SplitStream uses a two-level workload balance mechanism to handle video content dynamics, while Distream only dynamically updates the partition points. In addition, SplitStream is built up for cheap and fast queries, so it does not need powerful edge clusters if the edge devices can hold a full query pipeline.

7. Conclusion and future work

In this paper, we present the motivation, design, and evaluation of SplitStream. SplitStream intelligently deploys these primitives on different edge devices and adaptively adjusts the deployment when necessary. Evaluations show that, SplitStream reduces the result retrieval time by up to 19% and improves the resource utilization by up to 234%.

In future, we may attempt to combine SplitStream with CEVAS (Zhang et al., 2021) and VideoEdge (Hung et al., 2018) to build a complete framework across edge devices and the cloud. By applying a message agency, the worker instances on each device can be accessed transparently, which may reduce the workload balancing overheads and enhance the scalability. Moreover, it will be easier to handle bandwidth constraints in the wireless environment and the backbone network between the cloud and the edge by adjusting video configuration knobs and applying pipeline modules with different costs (cheap but inaccurate models or expensive but accurate models) to make bandwidth-accuracy tradeoffs in the network-sensitive environment. Besides, we may incorporate intelligent frame filters (Li et al., 2020; Yuan et al., 2023) into SplitStream to reduce unnecessary analytics workloads.

CRediT authorship contribution statement

Yu Liang: Conceptualization, Formal analysis, Funding acquisition, Investigation, Writing – original draft, Writing – review & editing. **Sheng Zhang:** Software, Supervision. **Jie Wu:** Resources.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

We thank the editor and anonymous reviewers. This work was supported in part by NSFC, China (62202233), Double Innovation Plan of Jiangsu Province (JSSCBS20220409).

References

Bolle, Ruud M., Yeo, B.-L., Yeung, Minerva M., 1998. Video query: Research directions. IBM J. Res. Dev. 42 (2), 233–252.

2022. Elastic GPU service. https://www.alibabacloud.com/zh/product/gpu/. [Online; accessed 20-August-2022].

Elgamal, Tarek, Shi, Shu, Gupta, Varun, Jana, Rittwik, Nahrstedt, Klara, 2020. Sieve: Semantically encoded video analytics on edge and cloud. In: 2020 IEEE 40th International Conference on Distributed Computing Systems. ICDCS, IEEE, pp. 1383–1388.

Fortinet, 2022. Understanding IP surveillance camera bandwidth. https://www.fortinet.com/content/dam/fortinet/assets/white-papers/wp-ip-surveillance-camera.pdf. [Online; accessed 20-August-2022].

Google, 2022. Edge TPU: Google's purpose-built ASIC designed to run inference at the edge. https://cloud.google.com/edge-tpu. [Online; accessed 20-August-2022].

Hanyao, Mengxi, Jin, Yibo, Qian, Zhuzhong, Zhang, Sheng, Lu, Sanglu, 2021. Edge-assisted online on-device object detection for real-time video analytics. In: IEEE Conference on Computer Communications. INFOCOM, IEEE, pp. 1–10.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, Sun, Jian, 2015. Deep residual learning for image recognition.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, Sun, Jian, 2016. Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. CVPR, pp. 770–778.

Hsieh, Kevin, Ananthanarayanan, Ganesh, Bodik, Peter, Venkataraman, Shivaram, Bahl, Paramvir, Philipose, Matthai, Gibbons, Phillip B, Mutlu, Onur, 2018. Focus: Querying large video datasets with low latency and low cost. In: 13th USENIX Symposium on Operating Systems Design and Implementation. OSDI, pp. 269–286.

Hung, Chien-Chun, Ananthanarayanan, Ganesh, Bodik, Peter, Golubchik, Leana, Yu, Minlan, Bahl, Paramvir, Philipose, Matthai, 2018. Videoedge: Processing camera streams using hierarchical clusters. In: 2018 IEEE/ACM Symposium on Edge Computing. SEC, IEEE, pp. 115–131.

Intel, 2022. Intel movidius vision processing units (VPUs). https://www.intel.com/content/www/us/en/products/details/processors/movidius-vpu.html. [Online; accessed 20-August-2022].

2018. International trends in video surveillance. https://cms.uitp.org/wp/wp-content/uploads/2020/06/18-07Statistics-Brief-Videosurveillance-web.pdf. [Online; accessed 21-December-2023].

2019. JacksonHole. https://www.seejh.com/live. [Online; accessed 22-December-2023].

Jang, Si Young, Kostadinov, Boyan, Lee, Dongman, 2021. Microservice-based edge device architecture for video analytics. In: 2021 IEEE/ACM Symposium on Edge Computing. SEC, IEEE Computer Society, pp. 165–177.

Jiang, Junchen, Ananthanarayanan, Ganesh, Bodik, Peter, Sen, Siddhartha, Stoica, Ion, 2018. Chameleon: scalable adaptation of video analytics. In: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication. SIGCOMM, pp. 253–266.

Kang, Daniel, Emmons, John, Abuzaid, Firas, Bailis, Peter, Zaharia, Matei, 2017. Noscope: optimizing neural network queries over video at scale. arXiv preprint arXiv:1703.02529.

Kleinrouweler, Jan Willem, Dimitrovski, Toni, Braam, Sjors, Hindriks, Rick, Van Den Berg, Hans, D'Acunto, Lucia, Niamut, Omar, 2021. Dynamic edge offloading for real-time video processing pipelines. In: 2021 IEEE 23rd International Workshop on Multimedia Signal Processing. MMSP, IEEE, 1–1.

Li, Yuanqi, Padmanabhan, Arthi, Zhao, Pengzhan, Wang, Yufei, Xu, Guoqing Harry, Netravali, Ravi, 2020. Reducto: On-camera filtering for resource-efficient real-time video analytics. In: Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication. SIGCOMM, pp. 359–376.

Liu, Wei, Anguelov, Dragomir, Erhan, Dumitru, Szegedy, Christian, Reed, Scott, Fu, Cheng-Yang, Berg, Alexander C, 2016. SSD: Single shot multibox detector. In: European Conference on Computer Vision. ECCV, Springer, pp. 21–37.

Luo, Quyuan, Hu, Shihong, Li, Changle, Li, Guanghui, Shi, Weisong, 2021. Resource scheduling in edge computing: A survey. IEEE Commun. Surv. Tutor. 23 (4), 2131–2165.

Mach, Pavel, Becvar, Zdenek, 2017. Mobile edge computing: A survey on architecture and computation offloading. IEEE Commun. Surv. Tutor. 19 (3), 1628–1656.

Mao, Yuyi, You, Changsheng, Zhang, Jun, Huang, Kaibin, Letaief, Khaled B, 2017. A survey on mobile edge computing: The communication perspective. IEEE Commun. Surv. Tutor. 19 (4), 2322–2358.

Meel, Vidushi, 2021. AI hardware: Overview of edge machine learning inference in 2022. https://viso.ai/edge-ai/ai-hardware-accelerators-overview/. [Online; accessed 20-August-2022].

Nvidia, 2022. Advanced AI embedded systems. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/. [Online; accessed 20-August-2022].

Oh, Sangmin, Hoogs, Anthony, Perera, Amitha, Cuntoor, Naresh, Chen, Chia-Chih, Lee, Jong Taek, Mukherjee, Saurajit, Aggarwal, JK, Lee, Hyungtae, Davis, Larry, et al., 2011. A large-scale benchmark dataset for event recognition in surveillance video. In: CVPR 2011. IEEE, pp. 3153–3160.

Pasteris, Stephen, Wang, Shiqiang, Herbster, Mark, He, Ting, 2019. Service placement with provable guarantees in heterogeneous edge computing systems. In: IEEE Conference on Computer Communications. INFOCOM, IEEE, pp. 514–522.

Preeti, C.M., Sri, T. Santhi, 2021. A deep learning survey on content detection from images powered by computer vision framework. In: 2021 Third International Conference on Inventive Research in Computing Applications. ICIRCA, IEEE, pp. 935–939.

2022. Public bandwidth. https://www.alibabacloud.com/help/en/elastic-compute-service/latest/public-bandwidth. [Online; accessed 20-August-2022].

Redmon, Joseph, Divvala, Santosh, Girshick, Ross, Farhadi, Ali, 2016. You only look once: Unified, real-time object detection. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. CVPR, pp. 779–788.

Ren, Shaoqing, He, Kaiming, Girshick, Ross, Sun, Jian, 2015. Faster R-CNN: Towards real-time object detection with region proposal networks. In: Advances in Neural Information Processing Systems. NeuIPS, Vol. 28.

Shi, Weisong, Cao, Jie, Zhang, Quan, Li, Youhuizi, Xu, Lanyu, 2016. Edge computing: Vision and challenges. IEEE Internet Things J. 3 (5), 637–646.

Simonyan, Karen, Zisserman, Andrew, 2014. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

Singh, Brijraj, Jain, Yash, Das, Mayukh, Naidu, Praveen Doreswamy, 2021. A framework for asymmetrical DNN modularization for optimal loading. In: 2021 International Joint Conference on Neural Networks. IJCNN, IEEE, pp. 1–8.

Softtek, 2020. Edge AI: The future of artificial intelligence. https://softtek.eu/en/tech-magazine-en/artificial-intelligence-en/edge-ai-el-futuro-de-la-inteligencia-artificial/. [Online; accessed 20-August-2022].

Szegedy, Christian, Liu, Wei, Jia, Yangqing, Sermanet, Pierre, Reed, Scott, Anguelov, Dragomir, Erhan, Dumitru, Vanhoucke, Vincent, Rabinovich, Andrew, 2015. Going deeper with convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. CVPR, pp. 1–9.

Teutsch, Steven M, Lee, LM, Teutsch, SM, Thacker, SB, St Louise, ME, 2010. Considerations in planning a surveillance system. In: Principles and Practice of Public Health Surveillance. Vol. 18, Oxford University Press New York, NY, USA, (10.1093).

Voulodimos, Athanasios, Doulamis, Nikolaos, Doulamis, Anastasios, Protopapadakis, Eftychios, 2018. Deep learning for computer vision: A brief review. Comput. Intell. Neurosci. 2018.

Wang, Shibo, Yang, Shusen, Zhao, Cong, 2020. Surveiledge: Real-time video query based on collaborative cloud-edge deep learning. In: IEEE Conference on Computer Communications. INFOCOM, IEEE, pp. 2519–2528.

Xu, Mengwei, Xu, Tiantu, Liu, Yunxin, Lin, Felix Xiaozhu, 2021. Video analytics with zero-streaming cameras. In: 2021 USENIX Annual Technical Conference. ATC, pp. 459–472.

Yan, Yuting, Jin, Yibo, Zhang, Sheng, Cheng, Fangwen, Qian, Zhuzhong, Lu, Sanglu, 2022. Focus! provisioning attention-aware detection for real-time on-device video analytics. In: 19th Annual IEEE International Conference on Sensing, Communication, and Networking. SECON, IEEE, pp. 271–279.

Yeh, Yang-Ming, Hu, Jennifer Shueh-Inn, Lin, Yen-Yu, Lu, Yi-Chang, 2019. Compressing DNN parameters for model loading time reduction. In: 2019 IEEE International Conference on Consumer Electronics-Asia. ICCE-Asia, IEEE, pp. 78–79.

2022. YoloV5. https://github.com/ultralytics/yolov5/. [Online; accessed 20-August-2022].

Yuan, Mu, Zhang, Lan, You, Xuanke, Li, Xiang-Yang, 2023. PacketGame: Multi-stream packet gating for concurrent video inference at scale. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication. SIGCOMM, pp. 724–737.

Zeng, Xiao, Fang, Biyi, Shen, Haichen, Zhang, Mi, 2020. Distream: scaling live video analytics with workload-adaptive distributed edge intelligence. In: Proceedings of the 18th Conference on Embedded Networked Sensor Systems. SenSys, pp. 409–421.

2022. Zeromq. https://zeromq.org/. [Online; accessed 20-August-2022].

Zhang, Haoyu, Ananthanarayanan, Ganesh, Bodik, Peter, Philipose, Matthai, Bahl, Paramvir, Freedman, Michael J, 2017. Live video analytics at scale with approximation and delay-tolerance. In: 14th USENIX Symposium on Networked Systems Design and Implementation. NSDI, pp. 377–392.

Zhang, Ben, Jin, Xin, Ratnasamy, Sylvia, Wawrzynek, John, Lee, Edward A, 2018. Awstream: Adaptive wide-area streaming analytics. In: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication. SIGCOMM, pp. 236–252.

Zhang, Miao, Wang, Fangxin, Zhu, Yifei, Liu, Jiangchuan, Wang, Zhi, 2021. Towards cloud-edge collaborative online video analytics with fine-grained serverless pipelines. In: Proceedings of the 12th ACM Multimedia Systems Conference. pp. 80–93.

Zhao, Han, Cui, Weihao, Chen, Quan, Leng, Jingwen, Yu, Kai, Zeng, Deze, Li, Chao, Guo, Minyi, 2020. Coda: Improving resource utilization by slimming and co-locating dnn and cpu jobs. In: 2020 IEEE 40th International Conference on Distributed Computing Systems. ICDCS, IEEE, pp. 853–863.

Zhao, Bo, Feng, Jiashi, Wu, Xiao, Yan, Shuicheng, 2017. A survey on deep learning-based fine-grained object classification and semantic segmentation. Int. J. Autom. Comput. 14 (2), 119–135.

Zhou, Jiahang, Yao, Yuanzhe, Yang, Rong, 2022. Deep learning for single-object tracking: A survey. In: 2022 IEEE 2nd International Conference on Software Engineering and Artificial Intelligence. SEAI, IEEE, pp. 12–19.

Yu Liang is a lecturer at Nanjing Normal University. She received her MS and Ph.D. degrees from Nanjing University in 2011 and 2021, respectively. She was a senior software engineer in Trend Micro China Development Center between 2011 and 2017. Her research interests include resource allocation in cloud and edge computing. Her publications include those which appeared in TMC, TPDS, TON, Computer Networks, Computer Communications, IEEE ICDCS, IEEE MSN, and IEEE Globecom.

Sheng Zhang is an associate professor with Nanjing University, and a member of the State Key Lab. for Novel Software Technology. He received the BS and Ph.D. degrees from Nanjing University. His current research interests include edge computing and edge intelligence. He regularly publishes in scholarly journals and conference proceedings, such as *JSAC*, *TMC*, *TON*, *TPDS*, *TC*, *MobiHoc*, *ICDCS*, and *INFOCOM*. He is the recipient of CCFSys Best Paper Award (2023), IEEE ICPADS Outstanding Paper Runner-Up Award (2021), IEEE ICCCN Best Paper Award (2020), IEEE MASS Best Paper Runner-Up Award (2012), ACM Nanjing Rising Star (2020), ACM China Doctoral Dissertation Nomination Award (2015). He is a senior member of IEEE and CCF, and a member of ACM.

Jie Wu (F'09) is the Director of the Center for Networked Computing and Laura H. Carnell professor at Temple University. He also serves as the Director of International

Affairs at the College of Science and Technology. He served as Chair of Department of Computer and Information Sciences from the summer of 2009 to the summer of 2016, and Associate Vice Provost for International Affairs from the fall of 2015 to the summer of 2017. Prior to joining Temple University, he was a program director at the National Science Foundation and was a distinguished professor at Florida Atlantic University. His current research interests include mobile computing and wireless networks, routing protocols, cloud and green computing, network trust and security, and social network applications. Dr. Wu regularly publishes in scholarly journals, conference proceedings, and books. He serves on several editorial boards, including IEEE Transactions on Mobile Computing, IEEE Transactions on Service Computing, the Journal of Parallel and Distributed Computing, and the Journal of Computer Science and Technology. Dr. Wu was general co-chair for IEEE MASS 2006, IEEE IPDPS 2008, IEEE ICDCS 2013, ACM MobiHoc 2014, ICPP 2016, and IEEE CNS 2016, as well as program co-chair for IEEE INFOCOM 2011 and CCF CNCC 2013. He was an IEEE Computer Society Distinguished Visitor, ACM Distinguished Speaker, and chair for the IEEE Technical Committee on Distributed Processing (TCDP). Dr. Wu is a CCF Distinguished Speaker and a Fellow of the IEEE. He is the recipient of the 2011 China Computer Federation (CCF) Overseas Outstanding Achievement Award.