OctopInf: Workload-Aware Inference Serving for Edge Video Analytics

Thanh-Tung Nguyen, Lucas Liebe, Nhat-Quang Tau, Yuheng Wu, Jinghan Cheng, Dongman Lee
School of Computing, KAIST, Republic of Korea
{tungnt, lucasliebe, quangntau1223, yuhengwu, chengjh, dlee}@kaist.ac.kr (*Correspondence*: {tungnt, dlee}@kaist.ac.kr)

***Abstract*—Edge Video Analytics (EVA) has become a major application of pervasive computing, enabling real-time visual processing. EVA pipelines, composed of deep neural networks (DNNs), typically demand efficient inference serving under stringent latency requirements, which is challenging due to the dynamic Edge environments (e.g., workload variability and network instability). Moreover, EVA pipelines face significant resource contention due to resource (e.g., GPU) constraints at the Edge. In this paper, we introduce OctopInf, a novel resource-efficient and workload-aware inference serving system designed for real-time EVA. OctopInf tackles the unique challenges of dynamic edge environments through fine-grained resource allocation, adaptive batching, and workload balancing between edge devices and servers. Furthermore, we propose a spatiotemporal scheduling algorithm that optimizes the co-location of inference tasks on GPUs, improving performance and ensuring service-level objectives (SLOs) compliance. Extensive evaluations on a real-world testbed demonstrate the effectiveness of our approach. It achieves an effective throughput increase of up to $10\times$ compared to the baselines and shows better robustness in challenging scenarios. OctopInf can be used for any DNN-based EVA inference task with minimal adaptation and is available at https://github.com/tungngreen/PipelineScheduler.**

I. Introduction

Recently, *Edge Video Analytics (EVA)* has emerged as a major area of pervasive computing [1], offering real-time visual sensing and processing. The pervasive nature of EVA systems enables seamless integration into diverse tasks such as surveillance [2], [3], [4], [5], health care [6], and activity recognition [7], [8]. These systems can perform continuous, on-site video stream analysis with reduced dependency on remote cloud infrastructures. In practice, VA services are typically organized as cascading pipelines of deep neural networks (DNNs) [9]. For instance, the pipeline of [Object Detect $\rightarrow$ Vehicle Classify, Plate Detect] can be employed for traffic monitoring (Figure 1). The task of executing the pipeline is defined as *inference serving* and is subjected to stringent latency demands (e.g., 200 ms), specified by service-level objectives (SLOs). It has been studied in works such as [10], [11], [12], [13], [14], [15] with the common goal of efficiently allocating and scheduling resources to meet SLOs.

Advancements in embedded computing now enable DNN models to run on both Edge servers and nearby embedded devices (e.g., Jetson Orin Nano) deployed next to data sources such as CCTV cameras [16]. Despite their limited capabilities, Edge devices can execute part of the EVA pipeline, enabling server-device cooperation to share

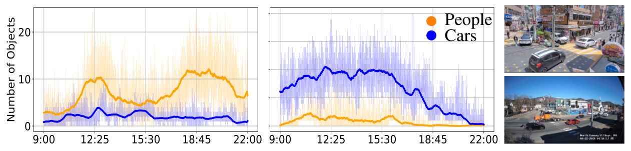


Fig. 1: Real-world footage at 2 locations shows varying number of objects leading to variations in workload for pipeline models (e.g., car classification).

workloads [9], [14], [15] increasing efficiency and flexibility. However, implementing this deployment scenario to reap its full potential involves two key considerations. (1) *Highly dynamic environments*. At the Edge, significant variations in video content over time lead to changes in workloads for models within an EVA pipeline (Figure 1), and constant changes in network conditions (i.e., bandwidth and latency) cause fluctuations in compute time budgets for the pipeline, complicating the workload distribution task. (2) *Resource contention*. At the Edge, multiple DNN models run concurrently, or *co-located*, on the same processing unit (e.g., GPUs), leading to unpredictable performance degradations in terms of latency (defined as *co-location interference* [17]). While the adverse effects can be lessened by grouping models with low workloads, the workload dynamicity mentioned above makes this impractical.

Several solutions have been proposed. Jellyfish [13] leverages *dynamic batching* to increase throughput and uses multiple DNN versions to adapt to network latency variations. During network instability, it reduces data resolution by opting for less accurate model versions, and adjusts batch sizes to meet throughput demands. However, its centralized architecture, which transfers raw videos to the server and assumes ample GPU resources to avoid co-location interference, is unsuitable for resource-constrained Edge environments. Furthermore, Jellyfish can only adjust batch sizes for different versions of the same model, not systematically schedule the whole pipeline. Distributed architectures like Distream [14] and Rim [18] instead utilize Edge devices and balance workloads between devices and servers. Distream introduces a stochastic method to determine the "*split point*," adaptively dividing EVA pipelines between local and server-based workloads. However, to reduce the optimization space, it uses a static batch size for models, which fails to account for dynamic workloads and network conditions. This is because while batching improves throughput, it increases end-to-end latency for queries in the batch, risking SLO violations [12]. Batch sizes must therefore be dynamically adjusted to real-time

TABLE I: Comparisons to the state-of-the-art EVA inference systems

System	Workload Distribution	Dynamic Batching	Spatiotemporal GPU Scheduling	Horizontal Scaling
Jellyfish [13]	*centralized*	*single tasks*	✗	✗
Distream [14]	✓	✗	✗	✗
Rim [18]	✓	✗	✗	✗
OCTOPINF	✓	*pipeline*	✓	✓

workloads and conditions. Rim [18] argues that Edge models rarely benefit from batching due to lower workloads compared to the cloud. It selects the *split point* by maximizing concurrent model execution to improve hardware utilization. However, this approach does not hold under dynamic Edge conditions. For example, during rush-hour traffic, a car-type classifier may experience high query volumes that benefit significantly from batched inference. Otherwise, without batching, throughput will suffer. Furthermore, all distributed architectures place multiple inference models on the same GPU without addressing the performance degradations caused by *co-location interference* [17]. In short, these works can only solve a small part of the problem set, making their solutions incomplete.

Goal and Insight. We aim to propose a system that tackles the complete problem of Edge inference serving, addressing limitations of the SOTAs, improving throughput, and reducing latency.. Our key insight is that (1) systematically adjusting model placements and dynamically configuring batch sizes can eliminate computation and communication bottlenecks, enhancing robustness to environmental dynamicity. Additionally, (2) temporally multiplexing model executions mitigates *co-location interference*, improving resource efficiency while ensuring compliance with SLO and throughput demands.

Technical Challenges. Inference serving at the Edge is a highly complex Integer Linear Program (section II) and poses two non-trivial challenges.

1) How can workloads be efficiently distributed while determining optimal batch sizes for pipeline models? Network and content dynamics significantly affect request rates, which vary widely among models. During high workload periods, increasing a model's batch size boosts throughput, raising request rates for downstream neighbors. Yet, batch selection for multiple models must occur simultaneously, creating a vast optimization space. Additionally, while larger batches enhance throughput, they also add latency, requiring a careful balance based on workload dynamics and latency constraints. Moreover, executing the entire pipeline on the server is inefficient, as transmitting raw data over unstable networks adds overhead and delay. Collaborative execution between servers and devices, where only essential information is transmitted, mitigates these issues but adds complexity. *Together, these factors make a highly complex challenge.*

2) How can models be multiplexed to avoid interference while satisfying SLOs and maintaining resource efficiency? Multiplexing model executions on shared GPUs demands a balance between maximizing parallelism and minimizing interference, which requires consideration of spatial aspects such as GPU memory and computation capability. Scheduling is further complicated by temporal dependencies, as upstream models must process data before the downstreams. Additionally, Current software stacks (e.g., NVIDIA CUDA, Intel OpenVINO) support only kernel-level scheduling, lacking model-level coordination. *This limitation increases the challenge while creating opportunities for custom strategies to improve resource usage and system performance.*

Proposed Approach. In this paper, we propose OCTOPINF, a workload-aware real-time inference serving system for EVA, designed to tackle the mentioned challenges with 3 main components. **First**, the **Cross-device Workload Distributor** (**CWD**), pronounced *seaweed*, employs a *workload-aware* greedy algorithm to optimize resource allocation through **dynamic batching** and balanced workload distribution between Edge devices and the server. CWD leverages environmental factors like workload burstiness to guide batch size exploration and uses IO ratios to filter out inefficient placement options, minimizing the optimization space for higher throughput and reduced latency. **Second**, the **Co-location Inference Spatiotemporal Scheduler** (**CORAL**) coordinates the execution of co-located models. To enable CORAL, we introduce the *inference stream* abstraction, simplifying resource allocation and model execution sequencing. Using a temporally best-fit algorithm with spatial constraints, CORAL ensures resource usage remains within hardware limits, reducing interference and enhancing efficiency while meeting SLO requirements. Additionally, OCTOPINF incorporates a **Horizontal Auto Scaler** to adapt to abrupt workload changes, ensuring scalability and robustness in dynamic settings. Comparisons to SOTAs on various categories are summarized in Table I.

OCTOPINF is designed to integrate seamlessly with various inference platforms (e.g., NVIDIA-TensorRT, Intel-OpenVINO, ONNX) using native programming APIs. To validate its effectiveness, we implemented OCTOPINF on a real-world testbed featuring industry-standard Edge devices and EVA workloads. Our **contributions** are as follows:

- We provide a comprehensive formulation and analysis of the EVA inference optimization problem, addressing the unique challenges posed by dynamic Edge environments and resource constraints at run-time (section II).
- We propose CWD, an efficient **Cross-device Workload Distributor**, and a workload-aware greedy algorithm to adjust batch sizes dynamically and balances workloads, leveraging factors such as workload burstiness and IO ratio, to minimize latency and maximize throughput (section III).
- We introduce *inference stream*, a novel GPU resource abstraction and present CORAL, a **Co-location Inference Spatiotemporal Scheduler**, which uses *stream* to simplify resource allocation and enable precise execution sequencing, reducing contention and enhancing throughput (section III).
- We deploy OCTOPINF on a real-world testbed with actual Edge video data and show up to 10× *effective throughput* improvement over state-of-the-arts (SOTAs) (section IV).

II. PROBLEM FORMULATION

We follow a common scenario, extensively considered by both academia [9], [14], [15] and the industry [19]. The

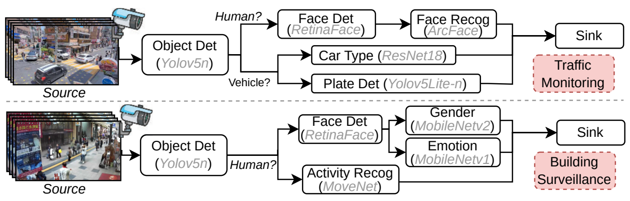


Fig. 2: Examples of common EVA pipelines.

EVA pipeline consists of multiple DNN models (Figure 2), organized as a directed acyclic graph (DAG). A cluster of **Edge devices** are connected to real-time data sources and the pipeline workload can be partially offloaded to an **Edge server** collaboratively [4], [20]. The **main objective** of OCTOPINF is to maximize the ***effective throughput***, defined as *the number of inference results that meet SLOs every second* while also minimizing GPU memory usage and end-to-end latency.

Optimization Problem. The serving of each model m in our scenario demands a configuration of batch size, device selection, GPU selection, and execution time $[bz_{m,g}, d, g, t]$. The SOTAs [13], [14], [18] do not provide temporal scheduling ($t \in [-\infty, +\infty]$), leading to severe co-location interference and unpredictable performance [17]. Instead, OCTOPINF performs temporal scheduling to reduce interference and guarantee that models perform as expected. Each configuration results in a batch inference latency of $\mathcal{L}_{m|bz_{m,g},d,g,t}$. Since all batched queries finish at the same time [12], the latency of each is averaged across the batch as $\mathcal{L}_m^{\text{infer}} = \frac{\mathcal{L}_{m|bz_{m,g},d,g,t}}{bz_{m,g}}$.

Furthermore, there is a latency in transferring the output of the previous model, m_{prev}, to m, which we define as $\mathcal{L}_m^{\text{io}} = \frac{\text{size}(In_m)}{BW}$, where I_m is the input of m and BW is the bandwidth of the transfer medium between two devices. For two models located on the same device, the bandwidth only depends on the device's hardware, which can be considered a large constant ϵ, and the latency can be negligible. However, for two models located on two different devices, the bandwidth is subjected to the network conditions. Thus, the average latency for each query of m is $\mathcal{L}_m = \mathcal{L}_m^{\text{infer}} + \mathcal{L}_m^{\text{io}}$. Thus, we can have:

$$\mathcal{L}_m = \frac{\mathcal{L}_{m|bz_{m,g},d,g,t}}{bz_{m,g}} + \begin{cases} \frac{\text{size}(In_m)}{\epsilon_{d_m}}, & d_m = d_{m_{prev}} \\ \frac{\text{size}(In_m)}{BW_{d_m,d_{m_{prev}}}}, & \text{otherwise} \end{cases} \tag{1}$$

The pipeline's average latency is defined as $\mathcal{L}_p = \sum^{m\in p} \mathcal{L}_m$. The **effective throughput**s of p and the whole system are calculated as $\mathcal{G}_p = 1/\mathcal{L}_p$ and $\mathcal{G} = \sum^{p\in \mathbf{P}} \mathcal{G}_p$, respectively. Our problem can be presented as a integer linear program (ILP):

$$\max_{bz_{m,g},d,g,t} \sum^{p\in\mathbf{P}} \sum^{t\in T} \frac{1}{\sum^{m\in p} \mathcal{L}_m} \tag{2}$$

which is subject to the following spatiotemporal constraints:

a) The SLO of all requests is met even in this worst-case scenario as the first request in each batch waits the longest for the batch to fill, experiencing the highest latency.

$$\sum_{m\in p} \mathcal{L}_m . bz_{m,g} = \sum_{m\in p} \mathcal{L}_m^{\text{worst}} \leq SLO_p, \forall p \in P \tag{3}$$

b) The total memory consumption of all models on g, including persistent weights W_m and temporary intermediate inference outputs I_m [21], does not exceed g's capacity. This ensures no model crashes during runtime.

$$\sum_{m\in g} W_m + I_m \leq M_g, \ \forall d \in D \tag{4}$$

c) The current GPU utilization rate U_g – the sum of individual models' GPU utilization rates U_{m_g} – does not exceed GPU's maximum compute capability. This ensures all models yield expected performances.

$$\sum_{m\in g} U_{m,g} = U_g \leq U_g^{\max}, \forall g \tag{5}$$

Since solving the ILP problem is NP-hard, finding an optimal solution in real-time, especially under resource constraints at the Edge, is impractical. Instead, we break it into two sub-problems: **cross-device workload distribution and co-location inference spatiotemporal scheduling**, whose solutions will be detailed in the next section. For convenience, the notations are summarized in Table II.

TABLE II: Summary of frequently used notations.

Notation	Description
p, P	pipeline p and list of all pipelines including p
m, m_{prev}	model m and the upstream model of m
d, D	host device d and list of all host devices including d
d_m	device that hosts an instance of model m
g, G_d	GPU g and list of available GPUs on device d
M_g	Total memory available on GPU g
$bz_{m,g}$, $BZ_{m,g}$	optimal and list of available batch sizes of m on g
In_m, Out_m	input and output of m (e.g., image - bounding box)
$\mathcal{L}_{m\|bz_{m,g},d,g,t}$	Inference latency of m under $[bz_{m,g}, d, g, t]$
$\mathcal{L}_m^{\text{io}}$, $\mathcal{L}_m^{\text{infer}}$, $\mathcal{L}_m$	Average IO, inference, and total latency of m
W_{m}, I_{m}, W_{g}, I_{g}	weight/intermediate memory of model m and all models on g
$U_{m,g}$, U_g	Utilization rates of m on GPU g and the current rate of g

III. OCTOPINF'S DESIGN

A. Overall Architecture

OCTOPINF's architecture (Figure 3) comprises 3 main components: a **Controller** that oversees system-wide scheduling and resource allocation; a **Device Agent** responsible for running containerized inference models on each device; and a **Knowledge Base** (KB) to store system-wide metrics. The operation cycle of OCTOPINF contains 5 steps:

① Each round starts with pipeline generation upon user requests. The Controller collects network/workload statistics and model/device profiles necessary for scheduling from **KB**.

② Then the **Controller** runs CWD (Figure 4) to select batch sizes, host devices, and container instance counts for pipeline models to maximize throughput while meeting SLOs.

③ The **Controller** runs CORAL to calculate timings and perform spatiotemporal scheduling for all scheduled models from step 2 to minimize the interference among co-located models.

④ Once finished, the scheduling results are communicated to the **Agent** on each device. It then deploys the models in containers, equipped with an inference engine (e.g., TensorRT) and managed by a container engine like Docker. The **Agent** also monitors the containers' metrics throughout the run time.

⑤ Container and device operation metrics including workloads, resource usage, and network bandwidth are monitored, and pushed to **KB** for monitoring and scheduling purposes.

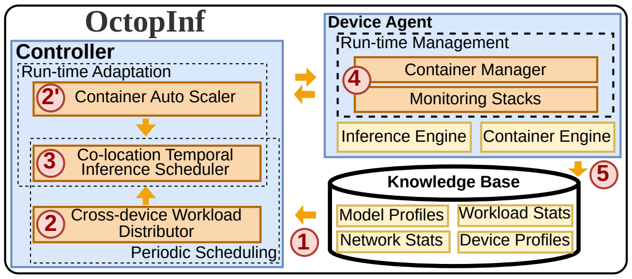


Fig. 3: The overall architecture of OctopInf

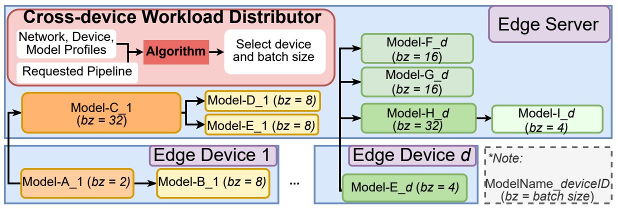


Fig. 4: Cwd searches for the near-optimal model deployments (location and batch size), considering requirements, network profiles, and device resources.

During run time, when containers reach throughput limits, (2') the **Auto Scaler** is triggered to scale up the containers by cloning to increase capacity. These container *instances* are then removed when demand drops.

B. Cross-Device Workload Distributor – Cwd

Selecting the optimal batch sizes and workload distribution can eliminate bottlenecks within the pipeline in response to workload changes and network variations (Figure 4). However, due to the large number of configurations, finding an optimal solution is prohibitively expensive. To efficiently navigate the large search space, we propose a *workload-aware* greedy algorithm based on the naive greedy min-max cost optimization algorithm proposed in [12], which treats all models in a pipeline similarly. By incorporating observations and insights into workload characteristics such as burstiness and IO ratio, we can choose better starting points and quickly remove unfruitful configurations from consideration.

Observation 1: Inference workloads are inherently bursty, with fluctuating intensity and varying from model to model depending on content dynamics. For example, an object detector suddenly detects a crowd, generating many human boxes, which in turn creates a surge of traffic to a downstream face detector. These surges can become even more pronounced during rush hour. Also, this burstiness can propagate to different downstream models, affecting overall performance.
⇒ *Insight 1: It is beneficial to prioritize larger batches for models with bursty workloads, which increases potential throughput. Additionally, since bursty workloads fill the batch quickly, each request has a shorter waiting time, which reduces the risk of SLO violations.*

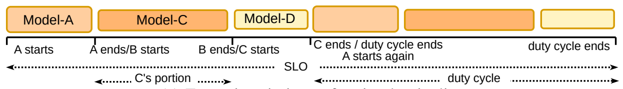
Observation 2: Higher network traffic between Edge devices and the server raises SLO-violation risks. Optimized batch selections enable adaptation to workload variations, removing computation bottlenecks. Still, unstable networks can be a bottleneck in the pipeline, slowing the whole pipeline.
⇒ *Insight 2: Placements should be chosen to minimize network traffic bottlenecks, improving system stability. For instance, during rush hour, an object detector capturing a high number of bounding boxes should not placed at the Edge being the split point because this generates higher network overhead than actually sending the raw frame to the server.*

Observation 3: More split points also increase the risk of SLO-violations. Multi splits offer flexibility in model placement (e.g., *edge→server→edge→server*), but each additional split increases network traffic, raising the risk.
⇒ *Insight 3: Number of pipeline splits should be minimized.*

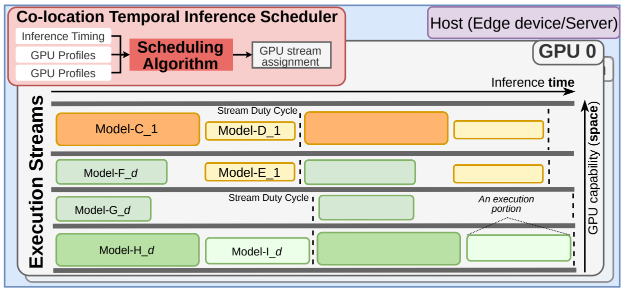
Algorithm 1: Cross-device Workload Distributor

```
1  Function cwd():
2      for Pipeline $p \in unscheduledPipelines$ do
3          for Model $m \in p$ do
4              $d_m =$ server ; $g$ = server_gpu; $bz_{m,g} = 1$
5              numInstances$_m = 1$; AddInstances($m$) // minimum cfg
6          $p_s =$ Sort $m \in p$ descendingly on burstiness // Insight 1
7          repeat
8              tmp_cfg$_p$ = NULL
9              for Model $m \in p_s$ do
10                 $bz_{m,g} *= 2$; ReduceInstances($m$) // min=1
11                 if EstLat($p$) > SLO$_p/2$ then
12                     $bz_{m,g} = bz_{m,g}/2$; AddInstances($m$)
13                 else
14                     if EstThrpt($p$) > max then
15                         cfg$_m$ = $[d_m, g, bz_{m,g}, 0]$ // set new cfg
16                         Update max; Add cfg$_m \rightarrow$ cfg$_p$
17         until tmp_cfg$_p$ = NULL;
18         ToEdge($p[0]$)
19         Add $p$ to $scheduledPipelines$
20     return $scheduledPipelines$
21 Function ToEdge($m$) // DFS-style traverse :
22     cfg$_p$ = Find a new configuration only for $m$.
23     if cfg$_p$ = NULL then
24         return
25     for $m_{next}$ sorted ascendingly on burstiness do
26         ToEdge($m_{next}$) // Insight 1
27     if Overhead($In_m$) $\cdot \alpha <$ Overhead($Out_m$) && downstreams are not on Edge then
           // Insights 2 and 3
28         Revert cfg$_m$
```

The operation of Cwd's algorithm (Algorithm 1) incorporating the above insights is detailed as follows. Cwd begins scheduling workloads for each pipeline p by initializing minimal configurations for all models on the server and adding active instances to match incoming request rates (lines 3-5). Then, the models are sorted in descending order based on their burstiness, measured by the coefficient of variation of inter-request arrival times (line 6). Next, Cwd greedily explores the configuration search space, increasing the batch size of each model m starting with the burstiest (lines 9-10). This follows ***Insight 1*** to maximize the throughput-enhancing capability of higher batch sizes. Though, due to the nature of batched inference, each request experiences a higher latency, the burstiness helps minimize it by reducing the waiting time to fill the batch. Next, thanks to the higher throughput, the number of instances of m can be reduced to conserve resources (line 12). However, if the new *temporary* configuration violates the pipeline's SLO, it is dropped. Otherwise, if the configuration improves throughput without violating the end-to-end SLO



(a) Execution timings of a simple pipeline.



(b) Overview of co-location spatiotemporal scheduling.

Fig. 5: Co-location Inference Spatiotemporal Scheduler (CORAL)

of the pipeline, it is adopted as the new configuration for m (lines 14-16). This exploration process continues until no better configuration is found for p (line 17).

Next, to distribute the workloads, we propose a DFS-style function, `ToEdge()` which incrementally distributes the models to the edge device where the data source is located. This follows ***Insights 2 and 3*** to choose a minimal number of split points, each of which reduces the network traffic and overhead. `ToEdge()` begins with the first model (line 18) and applies a similar process as above to explore new configurations, though the search is constrained to avoid costly explorations (line 22). If a suitable configuration is found, it continues traversing downstream, temporarily treating each model m as a split point (lines 25-26). In particular, m is first assumed to fit *Insight 2*, significantly reducing network traffic. On the return, `ToEdge()` tests this assumption by evaluating m's input-output size ratio which represents trade-offs between receiving inputs from the network and transmitting the output. Particularly, if m's output does not exceed that of its input data multiplied by a factor α, there are network benefits from placing m at the Edge. However, if the condition does not hold (line 27, upper) and no downstream models serve as better split points (i.e., they are still on the server) (line 27, lower), `ToEdge()` reverts m's configuration and returns it to the server (line 28). Additionally, `ToEdge()` leverages *Insight 1* by attempting to move less bursty models to the edge first, as their outputs are less likely to cause network bottlenecks.

C. *Co-location Inference Spatiotemporal Scheduler* – CORAL

After completing Algorithm 1, OCTOPINF retrieves the list of pipelines and their model instances, $scheduledPipelines$. OCTOPINF then invokes CORAL to enable multi-model GPU sharing, aiming to maximize utilization while ensuring models operate without resource starvation. To achieve this, CORAL addresses challenges in navigating the dual dimensions of **time** and **space** (i.e., GPU hardware capacity at any moment). We introduce the ***inference stream*** (or simply **stream** abstraction to enable spatiotemporal GPU sharing and propose a best-fit scheduling algorithm leveraging streams.

1) Inference Stream: A GPU's inference capacity is divided into multiple concurrently executed streams, as shown in Figure 5. Each stream represents a temporal sequence of model executions. Each model execution occupies a *portion*, whose length and width represent the execution time and required computation capability, respectively.

Algorithm 2: Co-location Inference Spatiotemporal Scheduler

```
1  Function Main():
2      $instanceNumber = 0$
3      while Has unscheduled instances do
4          for $p \in scheduledPipelines$ do
5              for $m \in p$ do
6                  if $m$ has $instanceNumber$ then
7                      CORAL($m$, $instanceNumber$)
8          $instanceNumber++$
9  Function CORAL($m$, $instanceNumber$):
10     $r = m[instanceNumber]$
11     for portion $pt$ in $FreePortions$ do
12         $s = pt_{\text{stream}}$; $g = s_{\text{gpu}}$;
13         $w_g = W_g + W_m$ // total weight memory
14         $i_g = I_g - I_s + \max(I_s, I_m)$ // total intermediate memory
15         $u_g = \sum_{s' \in g} \max_{r' \in s}(U'_r, U_r)$ // total GPU utilization
16         if $(pt_{\text{start}} \leq m_{\text{start}}$ & $pt_{\text{end}} \geq m_{\text{end}})$ & // fully available
17             $(w_g + i_g \leq M_g$ & $u_g \leq U_g^{\max})$ & // resource sufficiency
18             $(\text{duty\_cycle}_r \geq \text{duty\_cycle}_s)$ then
19             if $\text{duty\_cycle}_s = 0$ then
20                 $\text{duty\_cycle}_s = \text{duty\_cycle}_r$ // update stream's
21             $\text{stream}_r = s$; $\text{duty\_cycle}_r = \text{duty\_cycle}_s$ // stats
22             $W_g = w_g$; $I_g = i_g$; $U_g = u_g$ // update resource usage
23             selected_pt, freed_pts = DividePortion($pt$)
24             UpdateList(freed_pts, $FreePortions$)
25             return selected_pt
26     return not found
```

When considering a pipeline, the end-to-end latency can be broken into the latencies of its models (Equation 2), each of which occupies a portion of streams. The length of each portion can be calculated based on their selected batch sizes and batch inference profiles. The execution sequence is arranged according to p's DAG following a natural order ensuring downstream models immediately process data generated by their upstream. For example, scheduling Model-D before C would waste D's portion, as its input has not yet been generated by C (Figure 5a). Each stream also has a **duty cycle**, which lasts for half of p's SLO. After D's portion ends, GPU access is cycled back to A, which processes the new requests that arrive during the executions of C and D.

2) Spatiotemporal Scheduling Algorithm: The operation of CORAL's scheduling algorithm is shown in Algorithm 2. After CWD finishes, CORAL retrieves the list of pipelines with scheduled workloads, $scheduledPipelines$. It loops through these and calls `CORAL()` for unscheduled instances (lines 3-7). In each iteration, only one instance per model is scheduled to ensure fairness, so all pipelines have at least one active instance when priorities are equal. Complex scenarios with different priority policies can easily be integrated into CORAL.

CORAL's objective is to schedule as many instances on available GPU streams as possible, efficiently stacking execution portions one after another to minimize gaps, which waste resources. For each container instance r of model m, CORAL searches for the best-fitting portion pt among the list of free portions (line 11). For each, CORAL estimates the potential resource consumptions of GPU g if r is scheduled on pt (lines 12-15). Then CORAL evaluates pt on three conditions:

(1) pt **fully contains** r**'s portion with minimal empty space**, which ensures stream s is fully available during the execution of r (line 16).

(2) GPU g **has sufficient memory and compute capacity for** r **(line 17)** to satisfy Equation 4 and 5. This ensures no model crashes due to insufficient memory and there is enough computation to avoid co-location interference.

(3) Pipeline p**'s duty cycle must be longer than that of stream** s **(line 18)**. This ensures that having r on does not prolong other models' duty cycles and causes them to violate their SLOs, satisfying Equation 3.

Once the best-fit portion is found, r is assigned to stream s, the duty cycle of s is adjusted, and g's operational stats are updated (lines 19-22). If r cannot fully use the portion, the original pt portion is divided and the leftover is added back to the $FreePortions$ list for other instances (lines 23-24), ensuring optimal GPU resource utilization.

D. Run-time Horizontal AutoScaler

During runtime, workload fluctuations in models can occur due to content dynamics. While OctopInf can generate near-optimal schedules, frequently running Cwd incurs higher scheduling overhead, especially given the large search space. To address this, OctopInf complement periodical scheduling with a quick response strategy for surges and dips. When OctopInf detects significant workload surges for model m, it triggers the *AutoScaler* (Figure 3 create additional instances of m and schedules them temporally as described earlier. Conversely, unnecessary instances are removed, and their assigned portions are reclaimed.

IV. Evaluation

A. Experiment Setups

1) Testbed & System Implementation: We evaluate OctopInf on a real-world testbed comprising a server with 4 NVIDIA RTX 3090 GPUs and 9 heterogeneous Jetson Edge devices (1 AGX, 5 Xavier NXs, and 3 Orin Nanos). The Edge server and devices run Ubuntu 20.04 with CUDA 11.4.

OctopInf is implemented in over 25K lines of C++ at https://github.com/tungngreen/PipelineScheduler. It adopts a container-based architecture, where each model runs within a container for seamless scaling and adaptation in dynamic environments. Containers communicate via gRPC for efficient inter-service communication. The **Controller** are run at the server. On each device that handles workloads (including the server), a **Device Agent** is run as a separate process to manage and monitor containers via the Docker API and NVIDIA driver APIs, reporting statistics to the PostgreSQL **Knowledge Base** (KB). During scheduling, the **Controller** queries the **KB**, performs scheduling and allocation, and communicates decisions to the **Agents**, which enforces them on their containers.

2) Experiment Pipelines: Our experiments are conducted using 2 pipelines depicted in Figure 2. The pipeline end-to-end SLOs are set at 200ms for *traffic* and 300ms for *surveillance*. These values remain the same for all experiments except in section IV-C4, where we restrict the SLOs even further.

3) Data: We collected nine 13-hour videos from real-world public online streams (e.g., www.earthcam.com) at 15 fps and 1280x720 resolution. These streams, including 6 *traffic* and 3 building *surveillance*, contain human and vehicle targets and were chosen to represent a wide range of target distributions and content dynamics. For the main experiments, we extracted 30-min segments from three different times of the day to capture varying content dynamics. For full replicability, the code for data collection is also documented in the source code.

4) Baselines: For fair comparisons, we implement three state-of-the-art Edge inference, namely **Distream** [14], **Jellyfish** [13], and **Rim** [18], on top of the same source code and also make slight adjustments to show their best performances:

- All baselines do not provide any GPU scheduling. Thus, we implement a best-fit algorithm to spread models evenly based on resource consumption across GPUs.
- We adjust the static batch sizes for Distream and Rim to 4 at the edge, 8 at the server, and 2 for Object Det. This configuration shows the best performance.
- We implement lazy dropping of late requests to Distream and Rim to give them a higher effective throughput.
- Jellyfish does not consider pipelines but places multiple YOLOv5 versions at the Server. We match the number of downstream model instances to that of YOLOv5 versions with a static batch of 8 similar to Distream and Rim.

5) Experiment settings: Each collected video is assigned to an edge device, streamed like a real data source (e.g., cctv camera). To emulate the real-world network conditions, we extract network traces from an Irish 5G data set for all Edge devices [22]. The time between scheduling periods is set to 6 minutes for all algorithms. For fairness, the averaged results of 3 runs are reported for all experiments.

B. Evaluation Metrics:

The following three key metrics are used to evaluate our system and baselines:

- **Effective throughput:** The main metric for OctopInf is end-to-end effective throughput – ***the number of objects arriving on time at the sink every second***. We compare effective throughput to total throughput, considering the difference as wasted computations as requests arrive later than the specified service-level objectives (SLOs). This wasted computation causes further delays and consumes additional energy.
- **End-to-end latency**: A slim latency distribution indicates reliable performance and ability to satisfy stringent demands.
- **Total memory allocation:** A smaller memory footprint indicates better resource utilization.

C. Evaluation Results

1) ***Can OctopInf outperform the baselines on real-world dynamic workloads and network conditions?*** Figure 6a and b show that OctopInf significantly outperforms the baselines with higher effective throughput and shorter latency. While Distream and Rim achieve throughputs close to OctopInf, around 20% and 30% of their requests, respectively,

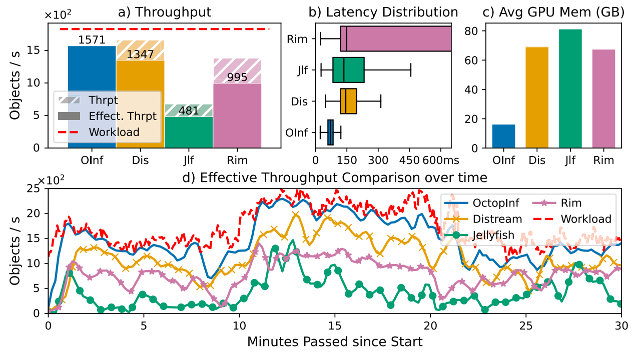


Fig. 6: Overall performance comparisons under environmental dynamics

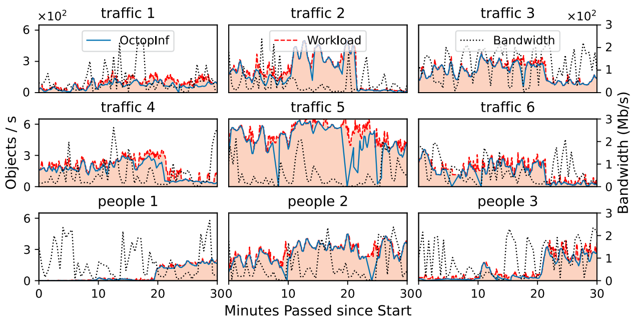


Fig. 7: Detailed workload, bandwidth and throughput patterns for OctopInf.

violate their SLOs, rendering them wasted computation. In terms of effective throughput, OctopInf outperforms them by 16% and 57% respectively. Also, Figure 6d shows that OctopInf finely adapts and matches the constantly changing workload throughout the entirety of the experiment. These can be achieved thanks to OctopInf's effective scheduling. First, OctopInf, with fine-grained batch size selection, can distribute the workload more efficiently and avoid communication and computation bottlenecks, which slow down the processing of the whole pipeline and lead to SLO violation. Second, with GPU spatiotemporal scheduling to allow orderly GPU access avoiding co-location interference, OctopInf's requests are processed on time according to calculation instead of being delayed due to resource contention. Jellyfish shows the worst throughput and 2nd worst latency due to its centralized architecture. Particularly, Jellyfish relies resolution reduction to lower network latency. Yet these resized frames still constantly have to be transferred to the server requiring a significant bandwidth, which is unstable in real-world conditions. Moreover, Rim shows the worst latency, because it attempts to move as many models to the Edge as possible without accounting for the workload dynamics and network conditions. More models on edge devices amplify the effects of co-location interference due to reduced parallel capabilities.

As illustrated in Figure 6c, OctopInf utilizes significantly less memory compared to the baselines, primarily thanks to temporal sharing of the GPU. When a model is idle, the GPU only needs to allocate memory for its relatively small weight.In contrast, running a model demands substantial memory for I/O buffers and intermediate layers [21]. This issue is exacerbated

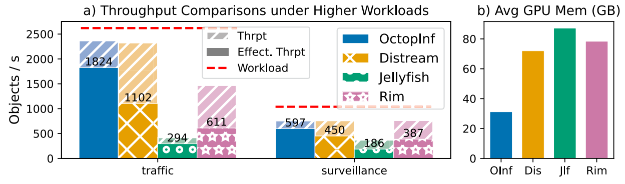


Fig. 8: Performance comparisons under higher workloads.

when many models are deployed simultaneously on the server (e.g., 30 for Jellyfish) without proper scheduling. Notably, Rim demonstrates the second lowest GPU memory usage as it seeks to accommodate as many models as possible at the Edge.

2) How effectively can **OctopInf** *handle real-world workload dynamics and network conditions?* We assess OctopInf using individual data sources under LTE bandwidth traces [22]. Figure 7 shows that OctopInf effectively adapts to fluctuating workloads, maintaining throughput aligned with bandwidth changes. Yet, there are instances of network disconnection (e.g., *traffic 5* at the 19th and 25th minutes) throughput unavoidably falls to 0. Otherwise, during low bandwidth periods, OctopInf remains resilient, closely matching workload demands (e.g., *people 2* at the 20th minute and *people 3* at the 28th minute).

This performance is achieved through the intelligent design of Cwd when considering the network instability. From an analytics perspective, users prioritize identifying objects and their attributes, making throughput (*objects/s*) a key metric. However, content dynamics, such as changes in object shapes and sizes, can lead to variations in network traffic for the same throughput. To address this, OctopInf considers the input-output size ratio, which assesses the trade-off between receiving inputs over the network versus transmitting outputs. If yes, placing the model at the edge will certainly reduce network traffic from the edge to the server. By incorporating this practical view, OctopInf mitigates network bottlenecks, ultimately improving overall throughput.

3) Can **OctopInf** *scale to higher system-wide workloads?* We introduce an additional data source to each device to simulate scenarios where multiple cameras (e.g., different angles) are connected to the same device, effectively doubling the frame rate and system-wide workload to evalutate OctopInf's resilience. Figure 8 shows the baselines have significantly lower throughput ratios compared to Figure 6, with even Distream only achieving a 50% ratio. This is because while the workload doubles, the relative burstiness ratio among more actually quadruples. It causes Jellyfish to encounter severe network bottlenecks, being able to complete only 14% of the requests. Distream and Rim, which do not have dynamic batching, still fare better with workload distribution. Yet, due to fixed batch sizes, their coarse-grained distribution overloads edge devices leading to poor performances and does not account for network traffic, though less severely compared to Jellyfish. This proves the validity of our design, which uses workload dynamics (e.g., burstiness) to navigate dynamic batching for workload allocation. Additionally, higher workload means more hardware usage (Figure 8b), leading to

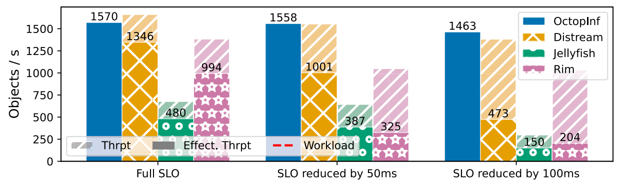


Fig. 9: Throughput comparisons under stricter SLO demands.

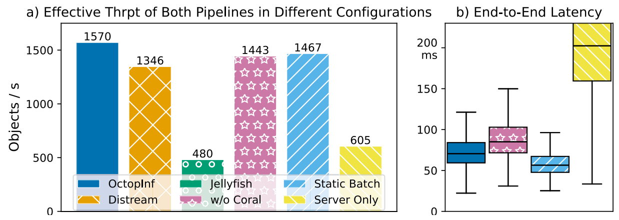


Fig. 10: Ablation Study of OCTOPINF in comparison to Jellyfish and Distream.

more severe co-location interference. This further showcases the positive effects of CORAL's spatiotemporal scheduling.

4) ***Can*** **OCTOPINF** ***flexibly adapt to stricter SLO demands?*** In this experiment, we gradually reduce the pipeline SLOs by 50-100ms from their original value of 200ms and 300ms (*traffic* and *surveillance* pipelines respectively) to test how well the systems adapt to stricter demands. Figure 9 demonstrates that OCTOPINF maintains its performance and continues to outperform the baselines, which experience significant performance degradation. Compared to full SLOs, the baselines' *effective throughputs* drop 3-5$\times$ (Rim) because despite their attempts to adjust scheduling and allocation to the new requirements, *co-location interference* has a greater impact at tight latency targets. Moreover, beside Jellyfish which still suffers heavily from network bottlenecks, Distream and Rim have less options to reduce the latency due to their fixed batches (similar to assembling clunky latency chunks). On the other hand, OCTOPINF can effectively adjust its batch sizes and balance between latency and throughput (similar to having access to a much wider variety of chunks). With SLOs reduced by 100ms, OCTOPINF achieves staggering 7 and 10$\times$ throughput improvements compared to Rim and Jellyfish.

5) ***How does each component contribute to*** **OCTOPINF*'s*** ***overall performance?*** To quantify the impacts of different components, we conduct an ablation study by *turning off* specific features individually, leading to 3 settings: *w/o Coral* (no spatiotemporal scheduling), *Static Batch* (fixed batches with spatiotemporal scheduling), and *Server Only* (dynamic batching for server-only deployments with spatiotemporal scheduling).

As shown in Figure 10, the best performance is achieved when all components of OCTOPINF are active. The performance decreases about 10% for *w/o Coral*. Here, many models simultaneously submit their inference workloads to CUDA, which treats each workload as a set of computation kernels instead of a single unit that requires to be completed on time. To ensure fairness, CUDA alternatively schedules hardware for kernels of different models, leading to higher latency for all models (Figure 10b). OCTOPINF schedules and reserves

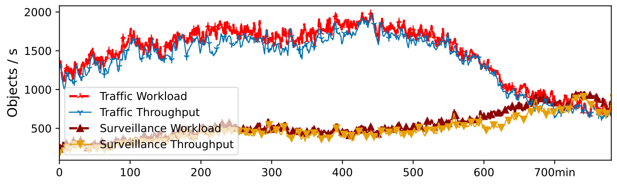


Fig. 11: Effective throughput of *traffic* and *surveillance* plotted over 13h.

an exclusive *portion* for each model to ensure that it has enough resource to complete on time. A smaller performance drop occurs with *Static Batch*, OCTOPINF still maintains a high throughput by dynamically distributing the workloads following CWD's *Insights 2 and 3*. In this case, the present of spatiotemporal scheduling also helps mitigate the issue of co-location interference. Interestingly, this setting achieves less throughput but has lower latency Figure 10. This proves that the latency-throughput balancing ability from dynamic batching is essential. It is also worth noting that both *w/o Coral* and *Static Batch* still outperform the baselines. The enormous performance drop occurs with *Server Only*. Similar to Jellyfish, it suffers from severe network bottlenecks and thus high latency, though thanks to spatiotemporal scheduling it still outperforms Jellyfish. This shows the effectiveness and necessity of OCTOPINF's contributions to solving the challenges of EVA inference serving.

6) ***Is*** **OCTOPINF** ***reliable for long-term operations?*** In this experiment, we test the long-term reliability of OCTOPINF using the complete dataset, which includes nine 13-hour video segments. Each video records continuously from 9 AM to 10 PM, capturing the full range of content dynamics at the Edge.

OCTOPINF maintains high *effective throughput* for both pipelines throughout the entire runtime, following patterns that align with human circadian rhythms as depicted in Figure 11. The *traffic* pipeline starts with lower throughput, peaks around the 450th minute (3:30 PM), and tapers off by the 600th minute (8 PM). During this period, the results show minimal throughput reduction compared to the workload. This demonstrates that OCTOPINF is stable, adaptable to real-world environments over extended periods, and capable of maintaining optimal performance with minimal resource usage.

V. DISCUSSION

1) ***Scalability****:* We discuss how OCTOPINF can work at scale by reducing the complexity and being able to seamlessly integrate major inference platforms and model architectures.

Complexity Reduction. Let D, G, M, and BZ represent the numbers of devices, GPUs, models, and available batch sizes for each model, respectively. The inference serving problem's optimization space has a complexity of $\mathcal{O}(D \cdot (BZ \cdot G)^M)$, rendering the optimal solution computationally infeasible. OCTOPINF simplifies this by decomposing it into two ***manageable*** subproblems: *cross-device workload distribution* and *co-location inference spatiotemporal scheduling*. For workload distribution, a *workload-aware* greedy algorithm approximates the solution with $\mathcal{O}(D \cdot M \cdot BZ)$. For spatiotemporal scheduling, the *stream* abstraction reduces complexity to $\mathcal{O}(M \cdot PT)$, where PT, capped by the number of model instances, denotes the maximum number of free portions. This approach yields an overall complexity of $\mathcal{O}(D \cdot M \cdot BZ + M \cdot PT)$. While only near-optimal, OCTOPINF surpasses SOTA methods and, critically, operates in real-time – vital for EVA systems.

Inference Platform Heterogeneity. Scalability at the Edge requires generalizability across diverse hardware and software platforms. OCTOPINF achieves this through two key concepts: *batched inference* and *inference stream*. The benefits of *batched inference* are supported by major platforms, including GPUs with TensorRT [23], CPUs with OpenVINO [24], and TPUs with TensorFlow [25]. OCTOPINF's *inference stream* abstraction leverages low-level APIs for asynchronous and sequential execution, such as TensorRT streams, OpenVINO streams, and ONNX threads. This design ensures OCTOPINF can scale efficiently and adapt to dynamic Edge environments.

Model Heterogeneity. While designed for EVA, OCTOPINF can extend to other domains like time-series analysis with RNNs. Batched inference improves RNN throughput, as shown in [26], while their recurrent operations allow precise inference time estimation for scheduling. These features enable OCTOPINF to scale its workload distribution and spatiotemporal scheduling to diverse applications.

*2) **Co-location Interference among Streams**:* OCTOPINF mitigates co-location interference by ensuring the total resource consumption of models is within GPU capability enabling models' peak performances. However, these peaks are typically brief in terms of memory consumption and utilization, and not all models within the same streams require the same amount of memory and achieve similar utilization due to factors like model architecture and implementations. Aligning these peaks more precisely could allow for more streams and better GPU utilization, but this requires a detailed analysis of workload patterns, which are highly diverse.

*3) **Future Work**:* We highlight the following future work to improve OCTOPINF. **First,** we plan to expand the testbed and integrate other inference platforms. Our first hardware target is Raspberry Pi devices using software frameworks ONNX and Tensorflow for their high compatibility. **Second,** we aim to leverage fine-grained performance data as time series for a prediction model that estimates co-located model performance. To accommodate the heterogeneity of model architectures, we plan to leverage techniques like Meta Learning and Test-time Adaptation (TTA) for quick adaptation to new architectures. **Third,** we plan to enhance OCTOPINF with fine-grained, second-scale local adaptation. Each model container will use Reinforcement Learning to learn workload patterns and make runtime decisions, such as adjusting batch sizes or request priorities, improving responsiveness to workload spikes.

VI. RELATED WORK

*1) **Edge Video Analytics*** can be categorized into:

Centralized Architecture: dictates that video data is collected from multiple sources and processed at a central location. To reduce overhead, approaches like VideoStorm [27], [28], [29], [30], [31] exploit the resource-quality tradeoff by adjusting parameters such as encoding quality, frame rate, and resolution. Another strategy [32], [33] uses models of varying accuracy, reserving high-precision models for complex tasks while employing cheaper ones for simpler tasks. Jellyfish [13] combines both methods and provides a dynamic programming algorithm to handle network variability.

Distributed Architecture: supports both device-server and device-device workload distributions. While better suited to dynamic environments, it creates a larger search space, including the challenge of finding optimal *split points* for workload partitioning. DRLTO [34] uses a fixed split and device-based analysis to optimize offloaded frames. Rim [18] maximizes workload at one device to increase resource utilization while approaches like Distream [14], [20], [35] use stochastic methods to explore the search space. EdgeVision [36] applies reinforcement learning to learn optimal configurations.

A common drawback of these systems is not fully utilizing an effective tool, *dynamic batching*, to flexibly handle dynamic environments, as it expands the complex search space. Additionally, they fail to address the challenges posed by GPU execution of models, such as *co-location interference*.

*2) **Inference Serving beyond the Edge*** serves a inference requests of various tasks at the Cloud [10], [11], [12], [37], [38], [39]. These systems consider large GPU clusters where one can take several GPUs or a whole cluster to serve a high concentration of requests. Although they do not address challenges at the Edge, such as resource constraints and dynamic environments (e.g. network), techniques such as *dynamic batching* and GPU scheduling have inspired OCTOPINF.

VII. CONCLUSION

In this paper, we presented OCTOPINF, a workload-aware inference serving system designed for Edge Video Analytics. It uses dynamic inference batching combined with adaptive cross-device workload distribution to manage fluctuating workloads and introduces a novel co-location inference spatiotemporal scheduling algorithm to mitigate resource contention. Our experiments demonstrate up to $10\times$ improvement in *effective throughput* compared to state-of-the-art baselines across various experiments, with a stable end-to-end latency distribution even under challenging conditions. While our evaluation focused on Video Analytics, the proposed system can be extended to optimize any machine learning workload, with testing and optimization in those contexts left as future work. With its low-level optimizations, OCTOPINF is poised to be a key component of future Edge AI applications.

ACKNOWLEDGEMENT

We want to thank our shepherd Dr. Paul Castro and anonymous reviewers for their insightful feedback. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (under grant No. RS-2019-II191126, Self-learning based Autonomic IoT Edge Computing), the KAIST Convergence Research Institute Operation Program, and KAIST Deep Mobility Consortium.

References

[1] P. V. Bahl, R. Caceres, N. Davies, and R. Want, "Pervasive Computing at the Edge," *IEEE Pervasive Computing*, vol. 19, no. 4, pp. 8–9, 2020.

[2] H. B. Pasandi and T. Nadeem, "CONVINCE: Collaborative Cross-Camera Video Analytics at the Edge," in *2020 IEEE Int. Conf. on Pervasive Computing and Communications Workshops, PerCom Workshops 2020*. IEEE, 2020, pp. 1–5.

[3] K. Hayashi, A. Hiromori, H. Yamaguchi, M. Suzuki, and T. Kitahara, "Synthesizing Town-scale Vehicle Mobility from Traffic Surveillance Cameras: A Case Study," in *2022 IEEE Int. Conf. on Pervasive Computing and Communications Workshops*. IEEE, 2022, pp. 593–598.

[4] T.-T. Nguyen, S. Y. Jang, B. Kostadinov, and D. Lee, "PreActo: Efficient Cross-Camera Object Tracking System in Video Analytics Edge Computing," in *2023 IEEE Int. Conf. on Pervasive Computing and Communications*. IEEE, 2023, pp. 101–110.

[5] K. Anjum, T. Chowdhury, S. Mandava, B. Piccoli, and D. Pompili, "Leveraging On-Board UAV Motion Estimation for Lightweight Macroscopic Crowd Identification," in *2024 IEEE Int. Conf. on Pervasive Computing and Communications (PerCom)*, 2024, pp. 11–17.

[6] S. K. Sahu, A. L. Ruscelli, G. Cecchetti, M. Gharbaoui, and P. Castoldi, "A perspective of telemedicine videostreaming systems for emergency care," in *2023 IEEE Int. Conf. on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, 2023, pp. 122–127.

[7] T. Kumrai, J. Korpela, T. Maekawa, Y. Yu, and R. Kanai, "Human Activity Recognition with Deep Reinforcement Learning using the Camera of a Mobile Robot," in *2020 IEEE Int. Conf. on Pervasive Computing and Communications (PerCom)*. IEEE, 2020, pp. 1–10.

[8] N. Bicocchi, M. Lasagni, and F. Zambonelli, "Bridging vision and commonsense for multimodal situation recognition in pervasive systems," in *2012 IEEE Int. Conf. on Pervasive Computing and Communications*. IEEE, 2012, pp. 48–56.

[9] S. Y. Jang, B. Kostadinov, and D. Lee, "Microservice-based Edge Device Architecture for Video Analytics," in *6th ACM/IEEE Symp. on Edge Computing, SEC 2021*. ACM, 2021, pp. 165–177.

[10] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram, "Nexus: A gpu cluster engine for accelerating dnn-based video analysis," in *Proc. of the 27th ACM Symp. on Operating Systems Principles*, 2019, pp. 322–337.

[11] S. Choi, S. Lee, Y. Kim, J. Park, Y. Kwon, and J. Huh, "Serving heterogeneous machine learning models on Multi-GPU servers with Spatio-Temporal sharing," in *2022 USENIX ATC*, 2022, pp. 199–216.

[12] D. Crankshaw, G.-E. Sela, X. Mo, C. Zumar, I. Stoica, J. Gonzalez, and A. Tumanov, "Inferline: latency-aware provisioning and scaling for prediction serving pipelines," in *Proc. of the 11th ACM Symp. on Cloud Computing*, 2020, pp. 477–491.

[13] V. Nigade, P. Bauszat, H. Bal, and L. Wang, "Jellyfish: Timely inference serving for dynamic edge networks," in *2022 IEEE Real-Time Systems Symp. (RTSS)*. IEEE, 2022, pp. 277–290.

[14] X. Zeng, B. Fang, H. Shen, and M. Zhang, "Distream: scaling live video analytics with workload-adaptive distributed edge intelligence," in *Proc. of the 18th SenSys Conf.*, 2020, pp. 409–421.

[15] X. Hou, Y. Guan, and T. Han, "Dystri: A Dynamic Inference based Distributed DNN Service Framework on Edge," in *ACM Int. Conf. Proceeding Series*. ACM, 2023, pp. 625–634.

[16] M. Mendula, P. Bellavista, M. Levorato, and S. L. de Guevara Contreras, "Furcifer: a Context Adaptive Middleware for Real-world Object Detection Exploiting Local, Edge, and Split Computing in the Cloud Continuum," in *2024 IEEE Int. Conf. on Pervasive Computing and Communications (PerCom)*, 2024, pp. 47–56.

[17] J. Wu, L. Wang, Q. Pei, X. Cui, F. Liu, and T. Yang, "HiTDL: High-Throughput Deep Learning Inference at the Hybrid Mobile Edge," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4499–4514, 2022.

[18] Y. Hu, W. Pang, X. Liu, R. Ghosh, B. Ko, W.-H. Lee, and R. Govindan, "Rim: Offloading inference to the edge," in *Proc. of the Int. Conf. on Internet-of-Things Design and Implementation*, 2021, pp. 80–92.

[19] "Microsoft rocket for live video analytics," pp. 1–6, 2020. [Online]. Available: https://www.microsoft.com/en-us/research/project/live-video-analytics/

[20] C.-C. Hung, G. Ananthanarayanan, P. Bodik, L. Golubchik, M. Yu, P. Bahl, and M. Philipose, "Videoedge: Processing camera streams using hierarchical clusters," in *2018 IEEE/ACM Symp. on Edge Computing (SEC)*, 2018, pp. 115–131.

[21] B. Cox, J. Galjaard, A. Ghiassi, R. Birke, and L. Y. Chen, "Masa: Responsive Multi-DNN Inference on the Edge," in *2021 IEEE Int. Conf. on Pervasive Computing and Communications*, 2021, pp. 1–10.

[22] D. Raca, D. Leahy, C. J. Sreenan, and J. J. Quinlan, "Beyond throughput, the next generation: A 5g dataset with channel and context metrics," in *Procs. of the 11th ACM multimedia systems Conf.*, 2020, pp. 303–308.

[23] Y. Zhou and K. Yang, "Exploring TensorRT to Improve Real-Time Inference for Deep Learning," in *Proc. of HPCC*. IEEE, 2022, pp. 2011–2018.

[24] A. Demidovskij, A. Tugaryov, A. Suvorov, Y. Tarkan, M. Fatekhov, I. Salnikov, A. Kashchikhin, V. Golubenko, G. Dedyukhina, A. Alborova, R. Palmer, M. Fedorov, and Y. Gorbachev, "OpenVINO Deep Learning Workbench: A Platform for Model Optimization, Analysis and Deployment," in *2020 IEEE 32nd Int. Conf. on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2020, pp. 661–668.

[25] Y. Kochura, Y. Gordienko, V. Taran, N. Gordienko, A. Rokovyi, O. Alienin, and S. Stirenko, "Batch Size Influence on Performance of Graphic and Tensor Processing Units During Training and Inference Phases," in *Proc. of ICCSEEA*, 2020, pp. 658–668.

[26] F. Silfa, J. M. Arnau, and A. González, "E-BATCH: Energy-Efficient and High-Throughput RNN Batching," *ACM Transactions on Architecture and Code Optimization*, vol. 19, no. 1, 2022.

[27] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman, "Live video analytics at scale with approximation and Delay-Tolerance," in *14th USENIX Symp. on Networked Systems Design and Implementation*, 2017, pp. 377–392.

[28] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica, "Chameleon: scalable adaptation of video analytics," in *Proc. of the 2018 Conf. of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 253–266.

[29] T. Tan and G. Cao, "FastVA: Deep Learning Video Analytics Through Edge Processing and NPU in Mobile," in *IEEE Conf. on Computer Communications*. IEEE, 2020, pp. 1947–1956.

[30] B. Zhang, X. Jin, S. Ratnasamy, J. Wawrzynek, and E. A. Lee, "AWStream: adaptive wide-area streaming analytics," in *Proc. of the 2018 Conf. of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 236–252.

[31] Y. Li, A. Padmanabhan, P. Zhao, Y. Wang, G. H. Xu, and R. Netravali, "Reducto: On-Camera Filtering for Resource-Efficient Real-Time Video Analytics," in *Proc. of SIGCOMM*. ACM, 2020, pp. 359–376.

[32] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia, "NoScope: optimizing neural network queries over video at scale," *Proc. of the VLDB Endowment*, vol. 10, no. 11, pp. 1586–1597, 2017.

[33] T. Zhang, A. Chowdhery, P. V. Bahl, K. Jamieson, and S. Banerjee, "The Design and Implementation of a Wireless Video Surveillance System," in *Proc. of the 21st Annual Int. Conf. on Mobile Computing and Networking*. ACM, 2015, pp. 426–438.

[34] J. Wang, J. Hu, G. Min, W. Zhan, A. Y. Zomaya, and N. Georgalas, "Dependent task offloading for edge computing based on deep reinforcement learning," *IEEE Transactions on Computers*, vol. 71, no. 10, pp. 2449–2461, 2022.

[35] Y. Liang, S. Zhang, and J. Wu, "Splitstream: Distributed and workload-adaptive video analytics at the edge," *Journal of Network and Computer Applications*, vol. 225, p. 103866, 2024.

[36] G. Gao, Y. Dong, R. Wang *et al.*, "Edgevision: Towards collaborative video analytics on distributed edges for performance maximization," *IEEE Transactions on Multimedia*, 2024.

[37] C. Tan, Z. Li, J. Zhang, Y. Cao, S. Qi, Z. Liu, Y. Zhu, and C. Guo, "Serving DNN Models with Multi-Instance GPUs: A Case of the Reconfigurable Machine Scheduling Problem," *arXiv*, 2021.

[38] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving DNNs like clockwork: Performance predictability from the bottom up," in *14th USENIX Symp. on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 443–462.

[39] H. Zhang, Y. Tang, A. Khandelwal, and I. Stoica, "SHEPHERD: Serving DNNs in the wild," in *20th USENIX Symp. on Networked Systems Design and Implementation*, 2023, pp. 787–808.