Rim: Offloading Inference to the Edge

Yitao Hu
University of Southern California
yitaoh@usc.edu

Weiwu Pang
University of Southern California
weiwupan@usc.edu

Xiaochen Liu
University of Southern California
liu851@usc.edu

Rajrup Ghosh
University of Southern California
rajrupgh@usc.edu

Bongjun Ko
IBM Research
bongjun_ko@us.ibm.com

Wei-Han Lee
IBM Research
wei-han.lee1@ibm.com

Ramesh Govindan
University of Southern California
ramesh@usc.edu

Abstract

Video cameras are among the most ubiquitous sensors in the Internet-of-Things. Video and audio applications, such as cross-camera activity detection, avatar extraction or language translation will, in the future, offload processing to an edge cluster of GPUs. Rim is a management system for such clusters that satisfies throughput and latency requirements of these applications, while enabling high cluster utilization. It uses coarse-grained knowledge of application structure to profile throughput of applications on resources, then uses these profiles to place applications on cluster nodes to achieve these goals. It dynamically adapts placement to load and failures. Experiments show that on maximal workloads on a testbed, Rim can satisfy requirements of all applications, but competing approaches designed for low-latency GPU execution cannot.

CCS Concepts

• **General and reference** → **Performance**; • **Computing methodologies** → **Neural networks**; **Computer vision**; **Natural language processing**; • **Software and its engineering** → **Scheduling**.

Keywords

edge computing, GPU scheduling, serving system, deep learning

1 Introduction

Today, with the ubiquity of camera-enabled mobile devices, applications increasingly *process* images in near real-time, either on device, or in the cloud. Adding filters, identifying landmarks or people, or even simply re-sizing images are examples of such processing. To support this, recent research [19, 55] has explored predictable latency image processing on a cloud cluster, using deep learning models (*DL models*) executed on CPUs and GPUs.

Near real-time processing of *video and audio streams* is the natural next step in the evolution of media-processing (§2), especially for IoT. Already, video cameras are among the most widely deployed outdoor IoT sensors. In indoor settings, virtual home assistants like Google Home, Google's Nest hub and Amazon's Echo are experiencing significant market penetration and have microphones and, more recently, video calling capabilities.

This will drive the development of several novel applications. Complex activity detection [42] seeks to detect activities occurring across multiple non-overlapping cameras. Avatar extraction [31, 45] enables avatar-based video conferencing, where avatars represent participants. Language translation helps participants converse in different languages.

These applications require predictable *low latency* and *throughput* (in frames per second, or fps). Video-based applications can generate significant volumes of traffic. Moreover, many such applications often use multiple DL models connected together in a DAG (directed acyclic graph). Recognizing this, recent research in IoT systems has explored the execution of pared-down DL models on mobile devices [17, 30, 66]. However, the resource requirements of DL models continue to outstrip the computational capacity of mobile devices, so that, as the complexity of applications increases, mobile devices may not be able to satisfy application requirements. To meet these requirements, it will be necessary to offload computation to an *edge cluster* of GPUs located topologically close to user device (*e.g.,* at a cell tower, or cable head-end). This *edge computing* ensures low latency to user devices, and avoids having to transmit video across the wide-area network to remote cloud data centers.

Industry has recognized edge computing's potential. The global edge computing market size was valued at $3.5 billion in 2019 [3], and is anticipated to reach $43.4 billion by 2027 [4]. Telecommunication companies like Verizon and AT&T are starting to deploy edge computing infrastructure alongside their cellular networks, in order to provide low latency and reduce bandwidth [62]. Nvidia has released a GPU-based edge computing platform called EGX [8].

Motivated by the confluence of these trends, this paper explores the design and implementation of Rim, an edge GPU cluster management system for media-processing applications. Rim strives to achieve high cluster GPU utilization, while satisfying both throughput and latency objectives of each application session.

Contributions. Rim makes three contributions (§3).

IoTDI '21, May 18–21, 2021, Charlottesvle, VA, USA

ACM ISBN 978-1-4503-8354-7/21/05.
https://doi.org/10.1145/3450268.3453521

The first is the design of an abstraction (§3.2), called an *mDAG*, that exposes the DAG structure of applications, where a DAG vertex represents either CPU execution or GPU DL model invocation, and DAG edges represent data flow. mDAGs form a unit of execution and resource allocation. Each application session is bound to one mDAG, and Rim translates application throughput and latency requirements to resources allocated to an mDAG (or sub-graph thereof).

The second is a suite of novel techniques that manage *placement* of mDAGs on the cluster (§3.3). Placement relies on throughput *profiles* of the entire mDAG and each vertex. Rim uses profiles to place mDAGs either on a single node in the cluster, or splits mDAG execution across multiple nodes. The former ensures lower latency, but the latter helps increase utilization in the face of fragmentation. Having determined a placement, Rim automatically loads DL models and generates steering configurations to route media streams from the client through cluster nodes. To ensure high GPU utilization, Rim uses spatial multiplexing; prior work has *batched* inputs and temporally multiplexed models, which works less well at the edge because of lower statistical multiplexing.

The third is dynamic *quality adaptation* in which Rim dynamically switches to lighter-weight mDAG implementations that trade-off a little accuracy for significant reductions in resource usage. Motivated by quality adaptation techniques for video delivery [11, 43], this helps Rim serve more clients than would otherwise be possible.

Evaluation Results. Using an implementation of Rim on a cluster of 14 GPUs (§4.1), we show that Rim is able to satisfy the throughput and latency requirements while maintaining GPU utilization of about 52%, about $2\times$ of the reported utilization of GPUs in cloud clusters [2, 35, 37]. Unlike Rim which is able to handle 100% of its offered load, recent work such as Nexus [55] and Clipper [19] can only sustain 59.6% to 89.4% of the offered load (§4.2). An extensive ablation study (§4.3) shows that switching to lighter-weight mDAGs can help Rim accommodate 78.9% more sessions, that other alternatives to profile-based placement fail to achieve performance objectives, and that spatial multiplexing improves utilization by 8% to 99% over an approach that does not use it.

2 Background and Motivation

The impending confluence of three trends motivates Rim: novel near real-time *media-processing applications* using video and audio generated by IoT devices, sophisticated *deep learning* techniques for processing these streams, and *edge computing* services that make it possible to satisfy application performance objectives.

Deep learning. Armed with large training data sets and powerful GPU hardware, deep neural network-based learning has transformed computer vision, speech recognition, and machine translation. Much prior systems work on deep learning has focused on *training* DL models. Less work [19, 32, 55] has explored model *inference* (the use of the model to perform a task), the focus of our paper.

Deep learning has achieved fast and highly accurate object detection on *images* using deep learning models (henceforth, *DL models*) like SSD [41] and Yolo [52]. Popular speech to text DL models like Jasper [36], Wav2letter [16], and Deepspeech2 [13] also have high accuracy [48].

Novel Media-Processing Applications. These DL models are *building blocks* for future media-processing applications. Today, although

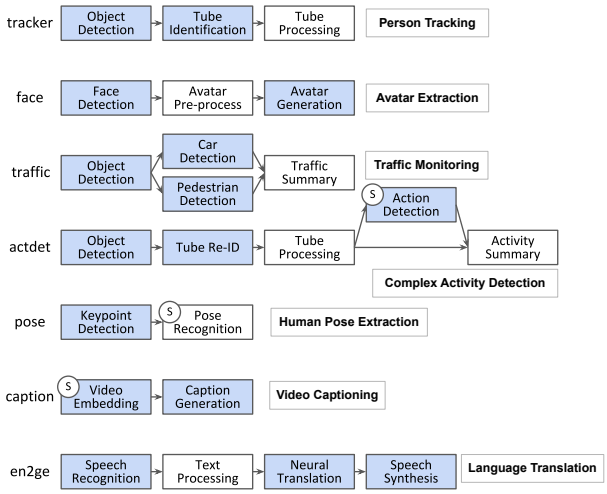


Figure 1: Examples of media-processing applications. The circled S denotes a *stateful* component.

video forms the largest proportion of Internet traffic, much of this video is streamed to, and consumed by, user devices. User devices are starting to generate video [63], but few applications *process* video. In this paper, motivated by the success of DL, we explore an emerging class of applications in which video and audio are (a) generated by user devices and (b) processed in near real-time. We call these *media processing applications* (Fig. 1).

Consider *complex activity detection* [42], in which multiple (potentially non-overlapping) surveillance cameras generate streams of video, and the task is to determine, in near real-time, *complex activities* across these cameras. An example of a complex activity is: "A person walking while talking on the phone in one camera, and the same person talks to another person at a different camera a short while later". More generally, the task is to recognize participants and objects, their spatial and temporal relationships, as well as the activities they perform, across multiple cameras. This application's processing pipeline uses DL models for object detection, re-identification (the task of determining whether two images from different cameras belong to the same person), and activity detection. These DL models need a GPU. In addition, the pipeline uses fast *CPU-based* trackers to track moving objects in the video, and intermediate stages of the pipeline accumulate *state* (*e.g.,* a sequence of object detections across successive frames).

Fig. 1 lists other media processing applications and their constituent components. We describe these later, but highlight three important features of these applications that inform our design: (a) media processing may include audio as well (*e.g.,* language translation); (b) applications use a combination of CPU-based components in addition to DL models (in Fig. 1, the latter have a darker background); (c) some components accumulate state across multiple frames before invoking the next component.

More important, media-processing applications have two important performance requirements: *throughput* (*e.g.,* for video, expressed by the frame rate), and *end-to-end latency*. These requirements determine *usability* (*e.g.,* conferencing may be unusable at

low frame rate or high latency) or even *correctness* (*e.g.,* activity detection might miss some activities at low frame rates).

Where to execute media-processing applications? DL models in Fig. 1 are heavyweight and require GPU acceleration to be able to satisfy throughput and latency requirements of media-processing applications. Where should these applications execute?

On the device. These applications process streams generated by mobile devices. Mobile devices will likely soon incorporate moderately powerful GPUs [46, 47]. Novel techniques like model compression [28, 33] and communication compression [30] make DL inference on the device a promising option. Unfortunately, even with these development, model inferences on the device may not be sufficient to support many of our applications. For example, our measurements show that it takes more than 4 seconds to run language translation on a 2-sec audio segment on the Jetson AGX Xavier (one of the most powerful mobile GPUs available today) [47].

On the cloud. Media-processing could potentially be *offloaded to the cloud*, since cloud providers have recently added support for GPUs. However, the latency from the user to the cloud may be too high for some applications (*e.g.,* avatar extraction for live teleconferencing, Fig. 1). The average response time from popular cloud providers [12, 25, 44] ranges from 66 ms to 75 ms [57], which accounts for one third of the latency budget for real-time streaming applications [7, 9]. For others, the bandwidth cost of streaming video to the cloud at large scale can be prohibitive; it is for this reason that, today, video streaming uses front-ends of large CDNs nearest to users, thereby minimizing this cost.

At the edge. Motivated by media-processing applications, and by the development of low latency high-bandwidth wireless standards like 5G, ISPs are starting to deploy *edge computing* clusters (containing a few racks of servers) topologically close to users (*e.g.,* at cell towers, cable head-ends) [62]. These edge clusters have more powerful compute capabilities than devices because they can deploy server-class hardware; *e.g.,* on a server-class GPU, translating a 2-second audio clip needs only 0.5 s. Moreover, edge clusters are closer to devices than the cloud, and can respond in less than half the time [57]. Therefore, edge computing represents a sweet spot in the space of architectural choices for media-processing applications.

Goal and Requirements. In this paper, we explore the design and implementation of a programming system and an associated runtime, called ***Rim***, for DL-based media-processing on an edge cluster containing multiple GPUs. Rim must support the performance requirements (throughput and latency) of *concurrent* media-processing sessions, while maintaining high cluster utilization.

Design principles. Rim uses the following design principles to satisfy these requirements.

Expose application structure and requirements. The media-processing applications we use employ a pipeline (or, more precisely, a directed acyclic graph or DAG [51, 55], Fig. 1) of computing *modules*, where each module represents either a DL model or computation on a CPU. For example, complex activity detection [42] uses an object detection DL model to extract tubes (sequences of object bounding boxes), a re-identification model to identify tubes belonging to the same person, CPU processing to determine spatial and temporal relationships between tubes, and an DL model for activity detection [60]. Similarly, traffic monitoring [55] uses an object detector, followed by a vehicle classifier or a human recognition model. Exposing this coarse-grained application structure to Rim is important to support performance requirements while maintaining high utilization.

Moreover, unlike the cloud, edge clusters have limited elasticity, so, to prevent applications from overloading the cluster, each *client* (typically, an end-device that requests edge cluster processing) must explicitly specify its desired frame rate and latency requirements.

Exploit accuracy/performance trade-offs. Rim should be able to exploit application-specified accuracy/performance tradeoffs; it can support more clients each at a slightly lower fidelity. This leverages a line of deep learning research that has explored resource/accuracy tradeoffs for DL models, using, for example, *model compression* [15, 28, 33] techniques that reduce the resource footprint of a DL model, while only minimally impacting model accuracy. If Rim is explicitly aware of alternative *model instances* for a given model, it can reduce resource usage for one application to accommodate others, thereby increasing utilization and throughput.

This requires the application developer to determine if the accuracy degradation from leaner models is acceptable: we argue that developers will need to benchmark their end-to-end application accuracy anyway, and they have an incentive to explore leaner models if Rim can support more clients for their applications.

Capitalize on the predictability and efficiency of DL model execution. Prior work has observed that DL model execution is predictable [19, 32, 55], and leveraged this predictability to either ensure high utilization [55], or achieve fairness [32]. Rim can use similar techniques to estimate resource allocation. Rim must also pack DL models efficiently to ensure high GPU utilization; recent work [55] batches inputs and temporally multiplexes DL models, while other work [65] employs spatial multiplexing for training.

3 Rim Design

In this section, we begin with an overview of Rim, followed by a detailed description of its components.

3.1 System Overview

Basic abstractions and Rim workflow. Rim expresses processing of video and audio streams using an abstraction we call a *media DAG*, or mDAG (Fig. 1). In an mDAG, a vertex (called a *module*) represents either a CPU computation, or an invocation of a DL model on a GPU. An edge in the graph represents a data dependency. Similar data-flow programming models exist for packet processing [38], massively parallel processing [34], and scientific computations [54].

Clients of Rim initiate *sessions* by invoking the Rim *master* (Fig. 2). A session requests allocation of resources to execute an mDAG on the Rim cluster on an input stream, with a specified *frame rate* and *end-to-end latency*. If the master admits the session, clients send video frames or audio segments to *workers*, who collectively execute various mDAG modules, while respecting data dependencies, then return the results to a client. Rim allocates each worker one or more CPU cores, and exactly one GPU.

Architecture. Rim re-uses two architectural principles commonly seen in cluster schedulers [21, 24]. The first is a *master-worker design*, in which a centralized master makes placement decisions,

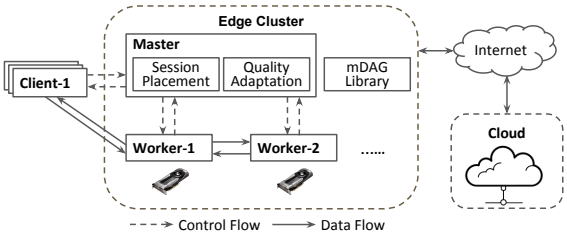


Figure 2: Rim overview.

and workers at each node manage local scheduling to achieve end to end scheduling objectives (in our case, performance objectives). The other is *control-data separation*, in which clients contact the master for control decisions, while data flows directly through workers to avoid the master becoming a bottleneck.

Design principles and challenges. This design applies the principles identified in §2 as follows. By using an mDAG (§3.2), Rim explicitly exposes application structure, which helps it satisfy performance objectives and meet utilization goals. Using this, together with client-specified performance objectives, Rim can better manage cluster. In Rim, each mDAG has one or more mDAG *instances*, where each instance represents a different point in the performance/accuracy trade-off space (or a different *quality*). The Rim runtime can dynamically adapt session quality based on resource availability (§3.3). Finally, Rim leverages the predictability of DL execution by *profiling* mDAG instances, and uses these profiles to make initial *placement* decisions that *pack DL models efficiently* (§3.3).

3.2 Sessions and mDAGs

The Session API. A client running on a user device instantiates a session using `setup_session(mDAG, perfobj)`, where `mDAG` is a unique name for the mDAG, and `perfobj` specifies the performance objectives of the client. Currently, Rim supports two performance objectives: a desired ***frame rate*** and a target ***end-to-end latency***. `setup_session` returns a session `handle`. The client can tear down the session using `teardown_session(handle)`. Clients send data to Rim in application data units that are either individual video frames, or segments of audio of a fixed duration (for convenience, we refer to both of these as *frames*) using `send(handle,frame)`, and can receive results from Rim using `receive(handle)`. Explicitly exposing a session abstraction[1] helps Rim achieve the performance requirements of a given media stream, and manage overload. A client library implements this API.

mDAGs and the mDAG library. Rim represents media-processing using a data-flow graph called an mDAG. Each distinct media-processing application has its own mDAG, identified by a unique name. For instance, the complex activity detection [42], or *actdet*, mDAG can detect complex activities using video streamed from multiple surveillance cameras, while the *en2ge* mDAG can translate English audio into German audio (Fig. 1).

mDAGs reside in an mDAG library and session instantiation uses the mDAG's unique name. A future version of Rim might permit user-defined mDAGs; today, putting together an mDAG requires

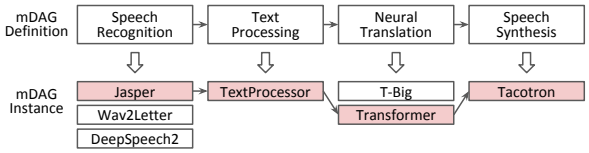


Figure 3: mDAGs and mDAG instances for *en2ge*. Each module in the mDAG definition can be implemented by one or multiple module instances. During runtime, Rim can dynamically switch between instances.

significant understanding of DL models and their performance and accuracy properties, so, at least in the near term, we expect specialized mDAG developers to specify mDAG definitions.

A node (or module) in an mDAG represents either a computation on the CPU, or a DL model invocation on a GPU. For instance, the *en2ge* mDAG (Fig. 3) consists of 3 GPU modules and 1 CPU module. The GPU modules perform speech recognition, neural translation and speech synthesis respectively. The CPU module performs audio decoding and assembly of translation results. A link between two modules in the mDAG represents a data dependency between them; for instance, the output of speech recognition is fed into a neural translation module that converts English text to German text. In *actdet*, the GPU modules invoke object detection, re-identification and activity detection DL module, while the CPU modules apply rules to determine spatio-temporal relationships between people and objects in the video.

Modules in Rim can be *stateful* unlike prior work [55]. For example, in *actdet*, one of the DL models takes as input on a sequence of bounding boxes, called *tubes*, derived from a fixed sequence of frames. A CPU module assembles a tube before invoking that DL model by maintaining successive bounding boxes detected by an object detector. In §4, we evaluate Rim using several mDAGs including three stateful ones (Fig. 1).

mDAGs separate DL model invocation from CPU processing because, while CPU scheduling techniques are mature enough to be able to multiplex computations on a CPU to achieve high utilization, increasing GPU utilization still requires leveraging application structure. Exposing the DL model invocations and dependencies to Rim's runtime allows it to ensure high GPU utilization, as we show later. This is particularly important given the high relative cost of GPUs.

mDAG instances. The DL community has invested significant effort in model compression and acceleration techniques (*e.g.,* [28, 33]) such as parameter pruning, low rank factorization, and knowledge distillation [15]. These techniques can reduce memory footprint and GPU resource requirements while marginally impacting accuracy. In practice, applications may be able to tolerate these accuracy drops, so Rim allows mDAG definitions to specify multiple instances for each GPU module.[2] For example, the *speech recognition* module in Fig. 3 can use three different instances: Jasper [36], Wav2letter [16], and Deepspeech2 [13], where Jasper has the lowest error rate [48] but the highest GPU resource consumption (which leads to the lowest throughput).

When each module has multiple instances, an mDAG can have many *mDAG instances*, where an mDAG instance contains one

[1] Recent work on inference, Nexus [55], uses a slightly different notion of a session. In Nexus, a session is an internal (*i.e.,* not client visible) construct that tracks processing of requests for a given model. We elaborate on the importance of this distinction in §4.

[2] Accuracy/resource trade-offs are also possible for CPU modules. We have left this to future work.

instance chosen from each module (shown in Fig. 3). As we describe later, Rim (a) *ranks* (§3.3) mDAG instances by resource usage which correlates with accuracy, (b) dynamically *adapts* (§3.3) which mDAG instance a client session uses based on resource availability. Internet video streaming uses similar quality adaptation [11, 43].

Many mDAGs can share a module instance. For example, both *actdet* and another mDAG for *traffic monitoring* [55] can use the object detector Yolo [52].

3.3 Placement and Quality Adaptation

Overview. To initiate a session, a client invokes the `setup_session()` method via RPC on the Rim Master. The master performs two functions: initial mDAG placement, and mDAG adaptation. The task of placement is to determine which workers should execute the session's mDAG instance to satisfy the session's performance objectives. The master may choose to place a session on an existing mDAG instance (*e.g.,* belonging to another session), or assign the session to an under-utilized worker (Rim assigns each worker one or more CPU cores, and exactly one GPU).

Placement relies on off-line *profiling*. Rim performs two kinds of profiling: *per-mDAG profiling*, and *per-module profiling*. The former allows it to place an mDAG on a single worker to reduce latency. To improve utilization, Rim resorts to cross-worker placement, for which it uses per-module profiling. Rim ranks mDAG instances using mDAG profiles; this helps it adapt quality to load variations. Once it determines a placement, Rim's master automatically instructs the assigned workers to load and warm-up DL models for session execution, then generates steering configuration to route frames from clients directly to those assigned workers (§3.4).

To batch or spatially multiplex? Rim aims to achieve high GPU utilization. There are two general ways to do this. One is *batching* input data [32, 55], but, for batching to be effective, batch sizes have to be large (on the order of 10s of frames). Rim has fewer opportunities than other systems to leverage batching, because: (a) batching frames within the same session can increase end-to-end delay significantly, (b) batching frames from different sessions works only if there are 10s of concurrent sessions that invoke the same DL model. Because edge clusters are likely to see lower statistical multiplexing than cloud clusters, Rim does not rely on batching. Moreover, as we show in §4, some models in our mDAGs can utilize a significant fraction of the GPU even with a single frame input (*i.e.,* without batching). For these reasons, Rim *uses spatial multiplexing*, in which the GPU concurrently executes kernels from multiple DL models (to maximize GPU core usage).

Profiling. Rim performs two kinds of off-line profiling[3].

Per-mDAG instance profiling. It obtains an mDAG instance's resource footprint on a given worker by (off-line) profiling the *maximum frame rate* the worker can sustain for that instance. Fig. 4 illustrates the profiling procedure for two different instances of two mDAGs. For each instance, Rim experimentally determines the highest frame rate beyond which the worker is unable to keep up with the inputs; this determines the instance's maximum frame rate. In Fig. 4, the higher-quality *traffic* mDAG saturates at 16 fps, the lower quality one can sustain up to 24 fps. In addition to profiling the maximum frame rate for each mDAG instance on each worker, Rim also profiles the latency incurred by each frame across the mDAG. It uses these profiles in *single-worker* placement, described below.

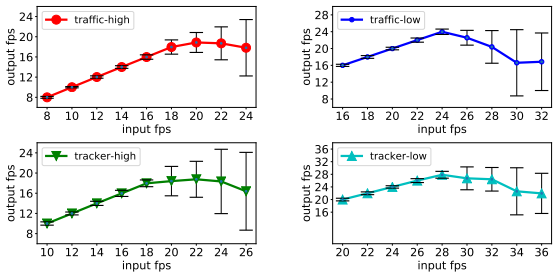


Figure 4: Spatial profiling for four mDAG instances.

Per-module profiling. To enable *cross-worker placement*, Rim also profiles individual modules (both CPU and GPU) on each type of worker. Specifically, for each module, it determines the maximum frame rate that the module can sustain on that worker. Rim profiles the CPU module as well, since the CPU module can be a bottleneck for some mDAGs (like avatar extraction). Existing systems [19, 55] either ignore profiling the CPU module or have only considered relatively lightweight CPU computation, an important reason for their poor performance relative to Rim (§4).

mDAG instance ranking. Rim uses profiles to rank mDAG instances. Instance ranking allows Rim to exploit performance/accuracy trade-offs (§2). In ranking mDAG instances, it assumes that instances with a higher overall GPU resource usage will have comparable or higher accuracy. This simplifies the task of instance ranking: while each DL model developer often documents the model's resource footprint and accuracy, and an mDAG's resource usage can be easily determined by profiling (discussed below), Rim cannot derive its end-to-end accuracy from the accuracy of the individual DL models (because, for instance, mDAG's can use simpler CPU-based processing that can improve accuracy).

Rim only generates a *total order* when ranking instances. Consider a two-module mDAG M_1, M_2, where each module M_i has two instances m_{i1} and m_{i2}, and m_{i1} has a larger resource requirement than m_{i2}. In theory, there are four different instances of this mDAG: $m_{11}m_{21}$, $m_{11}m_{22}$, $m_{12}m_{21}$ and $m_{12}m_{22}$. Rim selects $m_{11}m_{21}$ and $m_{12}m_{22}$, but *only one of* $m_{11}m_{22}$ and $m_{12}m_{21}$[4], since, even though it may be possible to order these two instances by resource footprint, it is unclear which one dominates in accuracy.

Rim assumes that, as long as it satisfies a session's performance objectives, it can use any one of the mDAG instances. This relies on mDAG developer intuitions, in much the same way as video quality adaptation today relies on guidelines developed for acceptable qualities for different devices [1]. An mDAG instance with a lower maximum frame rate requires more resources and is likely to be more accurate, so Rim *ranks it as having higher quality*.

The Placement Algorithm. Rim also uses profiling for session placement. When the client invokes `session_setup()`, it specifies the mDAG name, and two performance objectives: the *target frame rate*, and the *target end-to-end latency*. Rim checks which mDAG instances can meet the target end-to-end latency, using two pieces of

[3] Rim profiles each mDAG once, and not per session, so profiling does not affect client perceived latency.

[4] Currently, Rim randomly selects one of these two for simplicity.

information: round-trip time estimates from the client to master, and the profiled latency[5] (discussed above) for that instance. It filters out mDAG instances that cannot meet the latency objective. Of the remaining mDAG instances, Rim *tries to place the highest-ranked mDAG instance* on a worker on the cluster.

Frame-rate proportionality. Rim's mDAG instance placement algorithm uses an experimentally derived observation, *frame rate proportionality*. Consider an mDAG instance m_i, whose profiled maximum frame rate on worker j is F. Suppose that a client wishes to run the mDAG at a frame rate $f < F$, and Rim allocates worker j to m_i. Then, frame-rate proportionality suggests that m_i uses up $\frac{f}{F}$ of the worker's resources, leaving $1 - \frac{f}{F}$ free for other sessions (we call this the *residual capacity*).

This is a lower bound on the residual capacity of the worker. To understand why, observe that, for media-processing applications, processing complexity is linearly proportional to the frame rate. Consider two mDAG instances A and B, whose maximum frame rates on the worker are 10 and 20 respectively. If the GPU (or CPU) is a bottleneck for both these instances, it makes intuitive sense that A and B can concurrently execute on the worker at half their rates (5 and 10 fps respectively). In other words, when A executes at 5 fps, the worker's residual capacity is 50%. However, if the GPU is the bottleneck for A and the CPU for B, then both chains can run on the worker concurrently, which is why our residual capacity estimate is a lower bound.

As an aside, for all the mDAGs we evaluate in this paper, the most memory intensive mDAG instance, the highest ranked *en2ge* instance, has a peak memory utilization of 2.34 GB, well below the total memory available in modern GPUs (*e.g.,* the Nvidia 1080Ti has 11 GB). Thus, for Rim, *GPU memory is not a bottleneck* (as, for instance, it can be in systems that use batching [19, 32]).

Single-worker placement. Rim's placement algorithm uses this observation to place an mDAG instance on a single worker whenever possible. Suppose that Rim has a cluster with N workers, and each worker is currently executing one or more mDAG instances. Let the residual capacity of the i-th worker be c_i. Now, suppose a client invokes `setup_session()` for an mDAG whose highest ranked instance (after filtering instances that don't satisfy the latency objective) has a maximum frame rate of F, and the client's performance objective specifies a frame rate of f. Rim uses a *best-fit* strategy to minimize fragmentation: it allocates the session to that worker whose $c_i > \frac{f}{F}$ and $c_i - \frac{f}{F}$ is least.

Cross-worker mDAG placement. Even with best-fit frame-rate proportional placement, single worker placement can strand resources. In this case, Rim accommodates new sessions by placing their mDAG modules on multiple workers. For example, if an mDAG instance has two modules x and y, and the cluster has two workers A and B neither of which have enough residual capacity to accommodate the entire mDAG instance, Rim tries to allocate x to one worker (say A) and y to another worker (say B).

To be able to do this, Rim also profiles the maximum frame rate for each module, and makes the same frame-rate proportionality assumption for each module in deciding the best-fit worker for the module. While profiling modules, Rim is careful to include *serialization overhead* necessary for communicating the output of one module to another worker. Moreover, when two successive modules in an mDAG have very high serialization overhead, Rim *pins* them together to ensure that those two modules are co-located on the same worker.

[5]Prior work in inference serving systems such as Clipper [19] and Nexus [55], has used profiling of DNN models to make scheduling decisions. Profiling has also shown to accurately predict execution times for such models [32].

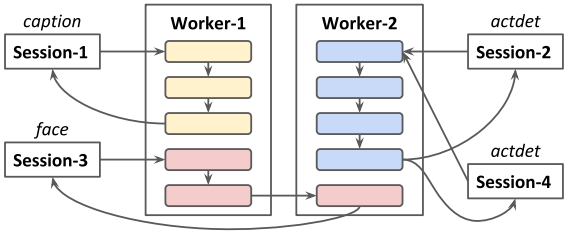


Figure 5: Example of *single-worker* placement and *cross-worker* placement, where three sessions are placed on a single worker, and one session is placed across two workers.

Quality Adaptation. When a client requests a new session, Rim may not have enough capacity to place the highest-ranked mDAG instance in the cluster. Its runtime then invokes *quality adaptation*, a technique that frees up resources by changing the mDAG instance used for a given session.

In order to accommodate the new session, Rim *demotes* an existing session: uses a lower-ranked (or lower "quality") mDAG instance for that session in order to free up resources. When a session terminates, Rim attempts to *promote* one or more existing sessions, *i.e.,* uses a higher-ranked mDAG instance for that session. In our current implementation, Rim uses a simple promotion (demotion) policy: promote (respectively demote) the session demoted (respectively promoted) the furthest in the past. We have left to future work to explore other policies such as finding the session which would free up the most resources if demoted, or use the least additional resources if promoted. Thus, using promotion and demotion, Rim attempts to satisfy as many clients as possible, while giving client sessions the highest quality when possible.

3.4 Other Details

Generating steering configurations. When a client invokes `setup_session()`, Rim, after determining the mDAG placement, *automatically generates steering configurations* which instruct: (a) the client how to steer frames to the worker hosting the first module in the mDAG; (b) the client how to proportionally split frames when session has parallel mDAG instances; (c) each module which module (on which worker) to send its output to. These steering configurations enable control/data separation in Rim (§3.1).

Model loading and warm-up. The placement algorithm can decide to place a session's mDAG instance on a worker where another session is already using the same instance. In that case, the new session simply reuses models from the existing instance. If the worker does not have the mDAG instance loaded, Rim instructs the worker to load the DL models corresponding to the mDAG instance. Rim also warms-up the model [22] by testing it with dummy data to avoid

delays resulting from kernel just-in-time compilation [26]. Without this, the first few frames of the client incur significant latency.

Handling stateful modules. By design, Rim dispatches requests for a given session to the same worker, and its runtime allocates memory to maintain state across requests for the same session, to ensure correctness of stateful modules.

High-rate sessions. Depending on the resources in a cluster, an mDAG's maximum frame rate might be lower than a client session's target frame rate. In this case, Rim can process the session's traffic using two mDAGs running concurrently. To determine the frame rate splits between these mDAGs, Rim searches for the split that best fits the residual capacities on the workers. Rim cannot split mDAGs with stateful modules; instead, it can attempt to migrate sessions (see below) to free up resources, which we leave to future work.

Failure and session migration. When a worker fails, Rim must migrate its sessions to other workers.[6] To do this, Rim's master must first find a placement for each session (potentially by demoting the session if necessary), load and warm-up the DL models in the mDAG instance, then generate a steering configuration for the new placement and send it to the client. Of these steps, model loading and warm-up can be expensive (on the order of seconds, §4).

Admission control and cloud offload. Rim can reject client sessions because the cluster does not have enough residual capacity to satisfy the target frame rate using the highest-ranked mDAG instance. It strives hard to support admitted sessions. It reserves a small amount of capacity at each worker to handle failures. However, when a worker failure occurs, the cluster may not have enough capacity to satisfy all the sessions. Rim finds the session with the loosest latency target, and if the target is loose enough to accommodate a round-trip to the cloud, Rim offloads the mDAG instance to the cloud. When no such mDAG exists, Rim has no option but to evict the session (for now, it uses a random drop policy).

4 Rim Evaluation

We compare Rim against two other model inferencing systems, Nexus [55] and Clipper [19], then perform an ablation study that illustrates the impact of Rim's design decisions.

4.1 Methodology

Implementation. We have implemented all the features of Rim described in §3. Each worker runs in a separate container, and uses TF-Serving [59] to spatially multiplex DL models on the worker's dedicated GPU. Rim is 10,478 lines of Python code, and includes the client library, the master, and worker implementations. The mDAG library is an additional 13,892 lines of Python.

Testbed. We deployed Rim on a cluster containing 8 servers with a total of 14 GPUs. We *deliberately designed the testbed to be heterogeneous*: this illustrates Rim's ability to place and adapt mDAGs across heterogeneous clusters. The testbed contains a range of Nvidia GPU models: one Titan, one Titan X, one Titan Xp, two 1080s, three 1080 Tis, two 2080s and four 2080 Tis. It also includes different Intel CPUs from the Xeon E5 to the Core i9.

[6] In theory, Rim could also migrate sessions during promotion and demotion in order to better pack mDAG instances across the cluster; we have left this to future work.

tracker*	face*	traffic*	actdet	pose	caption	en2ge
2	1	2	2	2	2	3

Table 1: Number of mDAG instances for quality adaptation used in our experiments. These mDAGs are described in Fig. 1. The starred mDAGs are used in our comparison experiments (which do not employ quality adaptation). Our ablation study uses all mDAGs.

Comparison Alternatives. ***Clipper*** [19] provides a uniform interface for model serving, but supports a variety of frameworks (Tensorflow, Caffe *etc.*) in the backend. Each of Clipper's models is encapsulated in a Docker container, and a Clipper cluster can host multiple instances of a DL model. Moreover, multiple model containers can access a GPU. Clipper batches inputs adaptively to increase model utilization, but does not attempt to do performance-aware model placement. It profiles model latency for different batch sizes to support *latency SLOs.*

Nexus [55] also serves image requests using a cluster of GPUs while satisfying a latency SLO. Like Clipper, it profiles model latency for different batch sizes, then dynamically batches inputs and schedules them on GPUs to ensure that it meets latency objectives for each request (in our experiments, a request is a frame). For each DL model, Nexus hosts multiple instances of the model, but, unlike Clipper, manages model placement and dynamically adapts the number of instances based on the observed workload and the latency SLO in order to minimize GPU usage.

Comparison methodology. For our comparisons, we disable several features of Rim, because these alternatives do not support these features: multiple mDAG instances (we only use the highest-ranked instances), admission control, and all quality adaptation (we evaluate these in our ablation study, §4.3). Moreover, neither of these systems provides Rim's session abstraction, so for them we send each frame as an independent request.

Metrics. For all three systems, we focus on three metrics: (1) The average *finish rate* of each session in a cluster, which is defined as the ratio between the output frame rate and the target frame rate of the session. The ideal finish rate is 1 (or 100%), but if a system is either overloaded or has design flaws, it may not be able to sustain a high finish rate. (2) The average *SLO-compliance* of each session, which is the percentage of frames whose *end-to-end latency* satisfied the *latency SLO*. (3) The *average gpu utilization* across the cluster, using the Nvidia system management interface [6]. The first two metrics guarantee correctness and usability for media-processing applications (§2), while the third measures efficacy of resource management.

Workloads. Tbl. 1 lists the mDAGs used in our experiments. The *actdet* mDAG performs complex activity detection [42] using Yolo [52] or SSD [41] for object detection, and [64] for re-identification. We use a trimmed version of this mDAG, *tracker*, for our comparison experiments; *actdet* includes a *stateful* module (§3.2) for activity recognition (which maintains state across several frames), but Nexus and Clipper do not support such modules. The *face* mDAG extracts facial keypoints used for avatar generation from each frame, using a face detector model and [23] for extracting keypoints. The *traffic* mDAG is taken from [55] and uses SSD [41] to detect objects, and two other models to recognize pedestrian and vehicle makes. The *pose* mDAG from [5], which recognizes human

	tracker	face	traffic
W-1	8(6)	10(6)	10(6)
W-2	6(6)	8(6)	12(6) & 2(9)
W-3	10(10) & 3(5)	4(6)	8(5)

Table 2: Number of sessions of each mDAG in each workload. Each entry shows the number of sessions, and the target frame rate for each session. Thus, an entry 8(6) means the corresponding workload had 8 sessions each with a 6 fps requirement. Some workloads contain mDAGs with different frame rate (*e.g., tracker* in W-3).

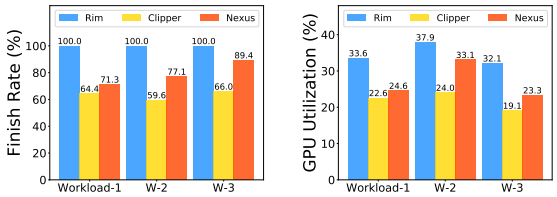


Figure 6: The average *finish rate* and GPU utilization for Rim, Clipper and Nexus under three *maximal* workloads.

poses in video by analyzing features extracted by Openpose [14] in individual frames. The *caption* mDAG from [61] uses S2VT to generate captions describing events in video by feeding the output of the fully-connected layer from VGG16 [56] or AlexNet [39] to S2VT's LSTM unit. The *en2ge* mDAG is described in §3.2.

Tbl. 1 lists the number of instances of each mDAG for quality adaptation used in §4.3; our comparison (§4.2) uses only the highest ranked instance. Moreover, our comparisons only use three of these mDAGs (*tracker*, *face* and *traffic*). We do not use *pose*, *caption* and *actdet* since Nexus and Clipper do not support stateful chains, and *eng2e* because Nexus' current implementation does not support audio streams. Each experiment's workload contains multiple sessions of each mDAG, as described below.

4.2 Comparison

Methodology. Can Nexus and Clipper plausibly support media-processing applications even though they were not explicitly designed for it? To address this question, we first generate three *maximal* workloads in Rim (labeled W-1, W-2 and W-3, Tbl. 2): a maximal workload in Rim is a collection of sessions, such that Rim rejects a new session request. In W-1, the total resource requirements are roughly equally divided between the 3 mDAGs. The GPU-intensive *traffic* dominates (requires most resources in) W-2, while *tracker* dominates W-3. We then run these sessions on Nexus and Clipper and evaluate the frame rate they achieve for each session.

Rim carefully places mDAG instances on the cluster. Clipper and Nexus perform resource allocation decisions at the granularity of individual DL models and they allocate different *containers* (in Clipper) or *backends* (in Nexus) to different models. Nexus handles model placement and provisioning as well as steering requests to model replicas, so we simply profile all our DL models in Nexus, and allow Nexus to manage model placement. Clipper, on the other hand, does not reason about model placement, so for Clipper we: (a) allocate as many model instances as Rim does, (b) ensure that all model containers for an mDAG are co-located with the client/front-end.

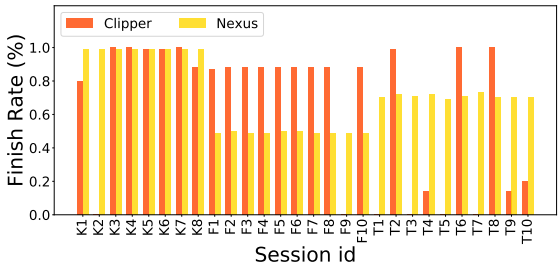


Figure 7: For $W-1$, the observed *finish rate* for each session for Clipper and Nexus. *K* denotes *tracker* sessions, *F* denotes *face* and *T* denotes *traffic*.

However, because Clipper does not do frame-rate aware placement, we randomly assign mDAG instances to GPUs.

Target frame-rate. Clipper has 34%-40% and Nexus has 10%-28% lower aggregate finish rate than Rim (Fig. 6) under all three *maximal* workloads. Under the first workload (W-1 in Tbl. 2), both Clipper and Nexus have more than 28% lower finish rate. To understand why, Fig. 7 shows the achieved *finish rate* for each alternative, for each session, under W-1.

Clipper. Clipper is nearly able to match the frame-rate requirement for 8 of the 28 sessions (*e.g.,* 5 of the *tracker* sessions), but it fails in two ways. First, because Clipper does not do resource-aware placement, our random placement places models of two relatively heavyweight mDAGs (*e.g., face* and *traffic*) on the same GPU (*e.g.,* it co-located *F5* and *T9* on the same GPU). The resulting contention reduces the frame rate. Second, some sessions exceed the GPU memory usage (*e.g., F9* and *T7*); model containers independently allocate and hoard memory resources, so memory often becomes a bottleneck. By contrast, Rim's worker carefully de-allocates memory after use, so Rim never encounters memory limits.

In part, Clipper performs as well as it does because we have been generous to Clipper in two ways. Clipper does not profile CPU modules, but our random placement gives it sufficient resources to handle CPU-intensive mDAGs, like *face* (by contrast, Nexus is significantly impacted by CPU-intensive mDAGs, see below). Moreover, Clipper does not have techniques to automatically split the latency SLO across different modules (since it does not support DAG structured applications); we manually assign generous per-module latency SLOs that result in higher finish rates. Without these, we expect Clipper to perform worse than it does.

Nexus. Nexus is also unable to satisfy the target frame rate for any *face* or *traffic* sessions, for three different reasons. First, Nexus is focused on GPU-intensive applications, and only profiles GPU modules. However, many practical mDAGs involve CPU-intensive computations (*e.g., face*) which should be profiled to assess resource needs and inform scheduling and placement. In our experiments, Nexus co-located all *face* sessions' CPU modules on the same machine, so all *face* sessions had a finish rate (*F1* to *F10* in Fig. 7) lower than 50%. To confirm this hypothesis, we manually doubled the number of machines that serve *face* mDAG's CPU module, and observed an increase in *face* finish rates from 49.43% to 76.76%. Because Rim profiles all modules and is aware of the resource requirements of CPU modules, its placement allocated five machines to run *face* sessions, and achieved a perfect finish rate.

Second, in the experiment where we doubled the number of machines serving *face*, the increase in *finish rate* for *face* sessions adversely impacted the *finish rate* for other sessions. This *interference* arises because Nexus' temporal sharing (where it time-slices the GPU across multiple DL models) with batching is not effective for our workloads (and for edge clusters in general). When batch sizes are large, as in a cloud setting, temporal sharing can achieve good throughput. But for an edge workload, with lower statistical multiplexing, it is hard to form a large batch without increasing latency (§3.3). The resulting small batch sizes increase the relative overhead of temporal sharing, so Nexus can support fewer GPU computations than Rim for the same amount of resources. Thus, in Nexus, the rest of sessions have to compete against *face* sessions on a over-subscribed GPU, resulting in interference.

Third, Nexus uses a latency splitting algorithm [55] to derive each GPU module's latency SLO, but this algorithm assumes homogeneous GPUs in a cluster. But our cluster contains a range of GPUs (from Nvidia 1080 to Nvidia 2080 Ti). We deliberately designed our testbed this way to mimic hardware heterogeneity resulting from incremental upgrades [10]. In our experiment, we used as input to the latency splitting algorithm the average latency profile[7] for a given batch size across all GPUs. Because this doesn't accurately reflect the latency on some GPUs, Nexus could not consistently satisfy the latency SLO for some chains on some GPUs. To validate this, we manually increased the latency SLO of each module for Nexus and observed an increase in *traffic finish rate* from 70.79% to 88.04%.

GPU Utilization. Furthermore, both Clipper and Nexus have lower GPU utilization under all three *maximal* workloads as shown in Fig. 6. There are two reasons for this. First, because they sustain lower frame rates, they fundamentally do less work. More important, both systems rely on *batching* to increase utilization. In our experiments, most models achieve batch sizes in Nexus of less than a handful of frames under all three workloads (Fig. 8). In our workload, frame rates are on the order of 6-10 fps, so inter-frame arrival times are on the order of 100-166 ms. If a model's latency SLO is 500 ms, it can probably afford to batch just 3-5 frames before invoking the DL model. To achieve high utilization, some models require a high degree of batching; for instance, Inception [58] requires a batch size of 32. Media-processing workloads at the edge are unlikely to achieve such high degrees of batching, which is why Rim uses spatial multiplexing instead (§4.4).

4.3 Ablation Study

Quality Adaptation. To evaluate the benefit of quality adaptation, we compare Rim against: (1) Rim-high, which always uses the highest-ranked mDAG and does not perform admission control and (2) Rim-low, which always uses the lowest-ranked mDAG.

Workload. This experiment uses the complete testbed (§4.1) with 14 GPUs. We evaluated all three alternatives (Rim, Rim-high and Rim-low) by adding a new session every 10 seconds until we reach the full capacity of the edge cluster (when Rim rejects new sessions). Then we maintain the workload for 30 seconds, and remove a session every 10 seconds in the reverse order in which they have been added. Each session randomly chooses an mDAG from Tbl. 1.

[7]We use the average latency profile *only* for latency splitting. Nexus still uses the per-GPU profile for its scheduling decisions.

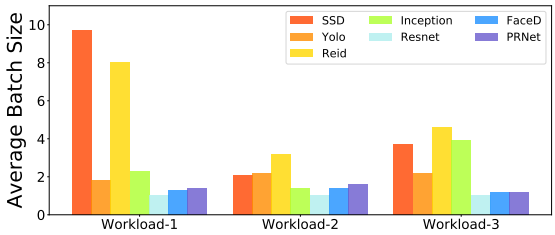


Figure 8: The average batch size for Nexus.

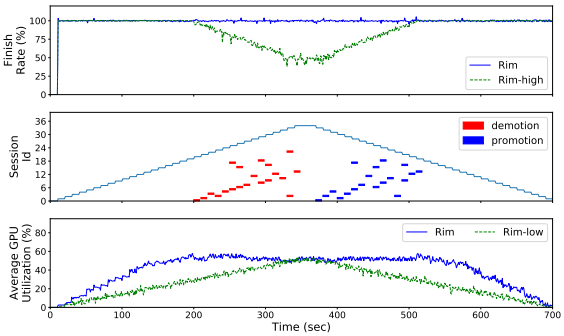


Figure 9: The *finish rate* and *GPU utilization* for Rim for quality adaptation.

Results: Finish rate. Fig. 9 shows how Rim dynamically adapts to workload changes during a 700 s window. The top panel shows the *finish rate* for Rim and Rim-high (Rim-low's finish rate is always as high as Rim's so we omit it). The middle panel shows when each session is started, as well as when quality adaptation decisions are made by Rim. The bottom panel shows the average GPU utilization.

Until the workload saturates the cluster ($t < 200$), both Rim and Rim-high have comparable *finish rate*. At 200 s, a total of 19 sessions run on the cluster. When the 20th session starts, Rim detects that it is running at the full capacity, and starts demoting mDAGs to release resources for new session. In the current implementation, Rim demotes the session that hasn't been demoted for the longest time (§3.3), if demoting it can release enough resources for the new session. For example, at $t = 200$, Rim demotes $session_1$ to accommodate $session_{20}$ (Fig. 9, middle panel). At $t = 220$, instead of demoting $session_3$ which doesn't release enough resources, Rim demotes $session_4$ to accommodate $session_{22}$. One interesting case is $session_3$, an *en2ge* session, which has 3 instances (Tbl. 1). Rim demotes this session *twice* (once at $t = 230$ and again at $t = 330$). Even as it demotes sessions, Rim is able to achieve a nearly perfect finish rate. On the other hand, Rim-high's finish rate drops beyond the 19th session because the cluster is at capacity. This illustrates the importance of profiling and admission control in Rim. At around $t = 340$, all mDAGs have been demoted to their lowest ranked instance, and Rim rejects subsequent sessions.

Beyond $t = 370$, Rim starts removing sessions; when a session is removed, an ongoing session is promoted if possible. In our current

implementation, Rim promotes the session that hasn't been promoted for the longest time (§3.3), if the released resources from the recently removed session is enough for its promotion. For example, at t = 370, $session_{34}$ has ended, therefore, Rim promotes $session_1$ to utilize the released resources. At t = 380, $session_{33}$ has ended, but Rim demotes $session_3$ instead of $session_2$ because the released resources are not enough to promote $session_2$. In this way, Rim tries to support each session at its highest ranked mDAG instance when possible.

Results: Utilization. Rim achieves an average GPU utilization of 52.36% across the 14 GPUs at t = 200 (Fig. 9, bottom), while Rim-low only achieves 28.73%.

Results: Number of sessions. Rim is able to maintain high *finish rate* for 34 concurrent sessions in our edge cluster, while Rim-high can only maintain high *finish rate* for 19 concurrent sessions. Besides, Rim achieves much higher GPU utilization than Rim-low by promoting sessions when resources are available. Overall, Rim supports 78.9% more sessions than Rim-high, and utilizes resources more efficiently than Rim-low.

Results: SLO-compliance. We now quantify the SLO-compliance (the fraction of frames that satisfied the chain's SLO) in this experiment (figure omitted for brevity). Four mDAGs have a perfect SLO-compliance, while three mDAGs have an average SLO-compliance of over 97%. Moreover, for those frames that violated the latency SLO from these three mDAGs, their actual latencies were only up to 11.3%, 14.9% and 4.8% beyond the latency SLO for mDAG *actdet*, *caption* and *en2ge* respectively. To stress test our system, for these three chains, we assigned much tighter SLO requirements (close to their end-to-end latencies) than for other chains. Our current placement algorithm is aggressive, placing a chain on servers whose latency may be close to the chain SLO; small latency variations for a few chains trigger these violations. A more conservative placement would leave more headroom, at the risk of lower utilization; we left it to future work to explore this tradeoff.

Placement. In this experiment, we compare Rim's placement against: (1) Rim-RR, which places a new session on one of the GPUs in a round robin fashion; (2) Rim-memory, which places a new session on the worker with largest remaining GPU memory available; and (3) Rim-utilization, which places the new session on the GPU with lowest GPU utilization. We compare these on the full testbed, on the maximal workload that we used in Fig. 9.

Results. Fig. 10a shows the *finish rate* for Rim's maximum frame rate based placement and the three other placement algorithms. Rim achieves nearly perfect *finish rate* (frames that violate *latency SLO* are counted as *unfinished*), but Rim-RR achieves poor *finish rate* because it ignores GPU and workload heterogeneity. Rim-memory does better, but because memory is not the bottleneck in Rim (§3.3), it cannot match Rim. Rim-utilization comes closest, but the GPU is not the only bottleneck for streaming applications. For example, *face* has a relatively low GPU utilization, but a high CPU utilization on average. Rim-utilization places three *face* sessions on the same worker, which overloads the CPU, leading to lower *finish rate*.

The need for cross-worker placement. We compare Rim with two other alternatives: (1) Rim-NC, which does not include cross-worker placement, but places the mDAG on the worker with highest remaining resources available when overloaded; and (2) Rim-NCdrop, which also does not include cross-worker placement, but rejects the session instead. The workload for this experiment consists of twenty-nine sessions on the entire testbed. This workload is maximal for Rim (no additional sessions can be accommodated), but is a *different* maximal workload than the ones used in the previous experiments, and is designed to maximize the likelihood of resource fragmentation to coerce Rim to invoke cross-worker placement.

Results. Rim-NCdrop can only accommodate twenty-four sessions out of twenty-nine, since the remaining five sessions cannot find a GPU for single-worker placement, while Rim is able to accommodate all twenty-nine sessions by placing those five sessions across ten GPUs. Rim-NC can accommodate all twenty-nine sessions, but has a low average *finish rate* of 79.3%; Rim's *finish rate* is 99.14% for this workload.

Spatial multiplexing. We compare Rim against Rim-temporal in which each DL model has exclusive access to the GPU during its execution (*i.e.,* temporal multiplexing). Let *spatial gain* be the ratio between the highest maximum frame rate with spatial multiplexing over that with temporal multiplexing.

Results. Fig. 10b shows the spatial gain for different mDAG instances from Tbl. 1. Spatial multiplexing increases the supported frame rate across these mDAG instances by 8% to 99%. In this experiment, the gains come from spatially multiplexing models of the same mDAG on the GPU. Even *actdet*, an mDAG with a heavyweight *ACAM* [60] module (Fig. 10c) gains 36% to 40%.

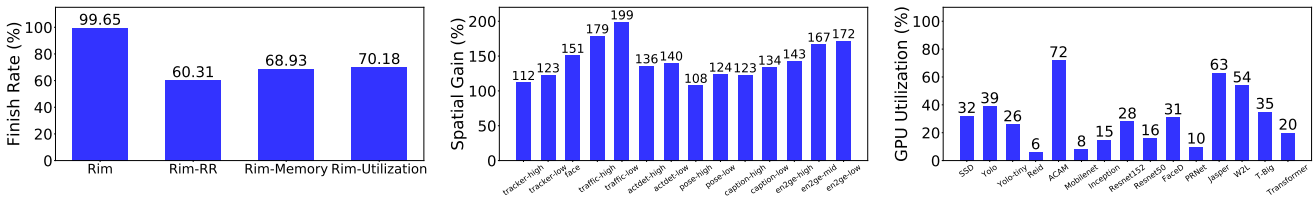
4.4 Justifications

No batching. Other image-based inference systems like Nexus and Clipper batch aggressively. For media-processing, DL models are often based on deeper and more complex neural network architectures than those used for images, so some of the DL models in our mDAG can incur high gpu utilization even on a single frame. In Fig. 10c, Jasper [36] utilizes 63% of a 1080 Ti, and the activity detection model ACAM [60] utilizes 72%, so Rim does not use batching.

Admission control. Fig. 4 shows that if we over-commit resources, instead of using admission control, by running mDAG at higher frame rates than the maximum the GPU can sustain, the actual number of frames processed drops with increasing offered load and the standard deviation also increases dramatically, leading to poor predictability. For this reason, Rim's placement algorithm with frame-rate proportionality does not over-commit resources.

Model loading and warmup. During initial placement and migration, Rim waits for DL models to load and warm up (§3.3). These steps take between 1.37 to 4.50 sec for our mDAGs (Tbl. 1), of which 0.48 to 2.82 sec is the model loading latency.

Serialization overhead. Rim co-locates two modules that exchange high-volume data with one another; in the absence of this, data serialization overhead can significantly impact latency. To demonstrate this, we ran two instances of *actdet*. One instance ran on a single worker. In the other, *actdet* was split across two workers where the



(a) (b) (c)

Figure 10: (a) The *finish rate* for various placements. (b) Spatial gain for different mDAG instances. (c) GPU utilization for DL models.

Feature	Edge vs. Cloud	Support DAGs	Frame-rate requirement	Quality adaption	Resource demands	Stateful models
Rim	edge	yes	yes	yes	CPU and GPU	yes
Nexus [55]	cloud	yes	no	no	GPU	no
Clipper [19]	cloud	no	no	no	GPU	no
Triton [49]	cloud	yes	no	no	GPU	yes
OoO [35]	cloud	no	no	no	GPU	no
INFaaS [53]	cloud	no	no	yes	CPU and GPU	no
InferLine [18]	cloud	yes	no	no	CPU and GPU	no

Table 3: Comparison of Inference Systems

two modules that exchanged high-volume data were placed on different workers. The former instance had an *end-to-end latency* of 0.85 sec, but the latter incurred a latency of 7.60 sec.

GPU utilization. In our comparison experiments (§4.2), Rim achieved an average GPU utilization from 32.1% to 37.9% under *maximal* workloads (Fig. 6), while in the ablation study, Rim achieved an average GPU utilization of 52.36% (Fig. 9). This difference comes from the difference in the corresponding workload; the ablation study uses several GPU-intensive mDAGs that the comparison experiments do not. For example, *en2ge* mDAG achieves an average GPU utilization of 73.55% at its maximum frame rate, and the *caption* mDAG 92.61%. To put these numbers in context, AWS reports an average GPU utilization between 10% to 30% [2, 35], and Google reports an average utilization of 28% for its Tensor Processing Unit (TPU) [37]. Rim's average GPU utilization is almost 2× of these values.

5 Related Work

Cluster Management for Inference. Much prior work has considered DL model inference in cloud clusters; Tbl. 3 summarizes the differences between Rim and this body of work. Rim explores a unique part of the design space, focusing on supporting frame rate requirements for DAG-structured media-processing applications at the edge with quality adaptation.

Of these, we have discussed Nexus [55] and Clipper [19] in §4 and explained how they differ from Rim. Nvidia Triton [49] uses CUDA streams to spatially multiplex models together on the same GPU, but it requires the system operator to manually specify the degree of parallelism, unlike Rim which determines this using profiling. OoO [35] uses a combination of temporal and spatial multiplexing to increase GPU utilization by merging small kernels into super-kernels, and reordering them to satisfy the latency constraints, but these merging techniques can only be applied to models using the same architecture; in contrast, Rim is able to share the GPU across heterogeneous models.

INFaaS [53] abstracts resource management and model selection for image inference. While it supports quality adaptation and profiles both CPU and GPU components, it is not designed to support frame-rate requirements of complex DAG-structured media-processing applications. InferLine [18] schedules machine learning pipelines to satisfy the end-to-end latency constraints. For a given pipeline, it selects the hardware accelerator and batch size using the offline profiling. Unlike Rim, it does not target frame-requirements, and uses temporal multiplexing which is likely to perform less well in an edge setting (§4.2).

TensorFlow-Serving [59] can group individual requests into batches to increase throughput, deploy multiple versions of the same model without need changes to client code, and minimize inference overhead. Rim uses TensorFlow-Serving on each worker, but adds profiling, cross-cluster placement and adaptation. GRNN [27] accelerates RNN execution by minimizing synchronization overhead and balancing on-chip resource usage. Rim supports RNNs as well as other neural network architectures, as well as data dependencies between models. Other work has explored specific optimizations to reduce resource usage that can be useful for inference in general, and Rim in particular: PRETZEL [40] explores operator and parameter sharing in model inference, and Focus [29] explores a cascade classifier for processing video.

Cluster Management for Training. Less relevant to Rim, Gandiva [65] uses spatial GPU multiplexing for training; Optimus [50] uses an online resource-performance model to estimate the training speed, given the amount of allocated resources, then dynamically allocate resources to each training job to minimize overall job completion time; GeePS [20] uses GPU-specific optimizations like background GPU/CPU data movement and data-parallel execution to achieve good efficiency and scalability.

Pushing DL Models to the Edge. DL models are commonly deployed on powerful computers equipped with GPUs. Recent work has explored been pushing DL model execution closer to the input source. FastAcc [30] uses auto-encoder to compress the data volume for communication, so as to run DL models on mobile GPUs.

Besides, novel DL model architectures [28, 33] further reduce the model size. Rim can take advantage of these developments, by partitioning the mDAG across devices and the edge cluster. Prior work on complex activity detection [42] has already demonstrated the benefits of this approach; we have left it to future work to extend Rim's mDAG orchestration to permit on-device execution.

6 Conclusions

Rim supports deep-learning based processing of audio and video streams on edge clusters. Applications for processing these streams often employ multiple DL models, and have an application structure well represented as a DAG. They also have target frame-rate and latency requirements, often for usability reasons. Given a client session's performance objectives, Rim uses an mDAG's performance profiles to derive placements that ensure that the session's frame-rate and latency objectives are met. It also leverages performance/accuracy tradeoffs to increase utilization and admit more sessions than otherwise possible. Experiments show that competing approaches designed to satisfy latency SLOs are not able to satisfy session target frame rates while Rim can. Future work includes exploring more applications in Rim, understanding Rim performance across multi-rack clusters, and exploring admission control techniques that permit over committing resources in order to increase utilization even higher.

Acknowledgments

This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Apple: Apple's HTTP Live Streaming. https://developer.apple.com/streaming/.

[2] AWS re:Invent 2018 Keynote. https://www.youtube.com/watch?v=ZOIkOnW640A&ab_channel=AmazonWebServices.

[3] Edge Computing Market Size, Share and Trends Analysis. https://www.grandviewresearch.com/industry-analysis/edge-computing-market.

[4] Edge Computing Market Worth $43.4 Billion By 2027. https://www.grandviewresearch.com/press-release/global-edge-computing-market.

[5] Multi-person Real-time Action Recognition Based-on Human Skeleton. https://github.com/felixchenfy/Realtime-Action-Recognition.

[6] NVIDIA System Management Interface. https://developer.nvidia.com/nvidia-system-management-interface.

[7] The Low Latency Live Streaming Landscape in 2019. https://mux.com/blog/the-low-latency-live-streaming-landscape-in-2019/.

[8] The NVIDIA EGX Platform for Edge Computing. https://www.nvidia.com/en-us/data-center/products/egx-edge-computing/.

[9] Video Conferencing Network Requirements. https://www.videonations.co.uk/resources/video-conferencing-news/video-conferencing-network-requirements/.

[10] Faraz Ahmad, Srimat T Chakradhar, Anand Raghunathan, and TN Vijaykumar. Tarazu: Optimizing mapreduce on heterogeneous clusters. *ACM SIGARCH Computer Architecture News*, 40(1):61–74, 2012.

[11] Zahaib Akhtar, Yun S. Nam, Ramesh Govindan, Sanjay Rao, Jessica Chen, Ethan Katz-Bassett, Bruno M. Ribeiro, Jibin Zhan, and Hui Zhang. Oboe: Auto-Tuning Video ABR Algorithms to Network Conditions. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, 2018.

[12] https://aws.amazon.com/, 2020.

[13] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*, pages 173–182, 2016.

[14] Z. Cao, G. Hidalgo Martinez, T. Simon, S. Wei, and Y. A. Sheikh. Openpose: Realtime multi-person 2d pose estimation using part affinity fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019.

[15] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *CoRR*, abs/1710.09282, 2017.

[16] Ronan Collobert, Christian Puhrsch, and Gabriel Synnaeve. Wav2letter: an end-to-end convnet-based speech recognition system. *arXiv preprint arXiv:1609.03193*, 2016.

[17] Vittorio Cozzolino, Jörg Ott, Aaron Yi Ding, and Richard Mortier. Ecco: Edge-cloud chaining and orchestration framework for road context assessment. In *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 223–230. IEEE, 2020.

[18] Daniel Crankshaw, Gur-Eyal Sela, Corey Zumar, Xiangxi Mo, Joseph E Gonzalez, Ion Stoica, and Alexey Tumanov. Inferline: Ml inference pipeline composition framework. *arXiv preprint arXiv:1812.01776*, 2018.

[19] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, March 2017. USENIX Association.

[20] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh EuroSys Conference*, page 4. ACM, 2016.

[21] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[22] TensorFlow Documentation. Savedmodel warmup. https://www.tensorflow.org/tfx/serving/saved_model_warmup.

[23] Yao Feng, Fan Wu, Xiaohu Shao, Yanfeng Wang, and Xi Zhou. Joint 3d face reconstruction and dense alignment with position map regression network. *CoRR*, abs/1803.07835, 2018.

[24] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[25] https://cloud.google.com/, 2020.

[26] Mark Harris. CUDA Pro Tip: Understand Fat Binaries and JIT Caching. https://devblogs.nvidia.com/cuda-pro-tip-understand-fat-binaries-jit-caching/, 2013.

[27] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. Grnn: Low-latency and scalable rnn inference on gpus. In *Proceedings of the Fourteenth EuroSys Conference*, page 41. ACM, 2019.

[28] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[29] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 269–286, Carlsbad, CA, October 2018. USENIX Association.

[30] Diyi Hu and Bhaskar Krishnamachari. Fast and accurate streaming cnn inference via communication compression on the edge. In *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 157–163. IEEE, 2020.

[31] Liwen Hu, Shunsuke Saito, Lingyu Wei, Koki Nagano, Jaewoo Seo, Jens Fursund, Iman Sadeghi, Carrie Sun, Yen-Chun Chen, and Hao Li. Avatar digitization from a single image for real-time rendering. *ACM Trans. Graph.*, 36(6):195:1–195:14, November 2017.

[32] Yitao Hu, Swati Rallapalli, Bongjun Ko, and Ramesh Govindan. Olympian: Scheduling gpu usage in a deep neural network model serving system. In *Proceedings of the 19th International Middleware Conference*, pages 53–65. ACM, 2018.

[33] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016.

[34] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.

[35] Paras Jain, Xiangxi Mo, Ajay Jain, Alexey Tumanov, Joseph E Gonzalez, and Ion Stoica. The ooo vliw jit compiler for gpu inference. *arXiv preprint arXiv:1901.10008*, 2019.

[36] https://nvidia.github.io/OpenSeq2Seq/html/speech-recognition/jasper.html, 2019.

[37] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017.

[38] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, August 2000.

[39] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[40] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: Opening the black box

of machine learning prediction serving systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 611–626, Carlsbad, CA, October 2018. USENIX Association.

[41] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[42] X. Liu, P. Ghosh, O. Ulutan, K. Chan, B. S.Manjunath, and R. Govindan. Caesar: Cross-Camera Complex Activity Detection. In *Proc. ACM Sensys*, 2019.

[43] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication*, SIGCOMM, 2017.

[44] https://azure.microsoft.com/en-us/, 2020.

[45] Koki Nagano, Jaewoo Seo, Jun Xing, Lingyu Wei, Zimo Li, Shunsuke Saito, Aviral Agarwal, Jens Fursund, and Hao Li. pagan: Real-time avatars using dynamic textures. *ACM Trans. Graph.*, 37(6):258:1–258:12, December 2018.

[46] https://developer.nvidia.com/embedded/jetson-tx2, 2019.

[47] https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/, 2020.

[48] https://nvidia.github.io/OpenSeq2Seq/html/speech-recognition.html#models, 2019.

[49] https://developer.nvidia.com/tensorrt, 2019.

[50] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, page 3. ACM, 2018.

[51] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. Odessa: Enabling interactive perception applications on mobile devices. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 43–56, New York, NY, USA, 2011. ACM.

[52] Joseph Redmon and Ali Farhadi. Yolov3: an incremental improvement. *arXiv*, 2018.

[53] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. Infaas: Managed & model-less inference serving. *arXiv preprint arXiv:1905.13348*, 2019.

[54] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. Numpywren: Serverless linear algebra. *CoRR*, abs/1810.09679, 2018.

[55] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 322–337, New York, NY, USA, 2019. Association for Computing Machinery.

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

[57] https://www.stackpath.com/products/edge-computing/, 2020.

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

[59] https://github.com/tensorflow/serving, 2019.

[60] Oytun Ulutan, Swati Rallapalli, Carlos Torres, Mudhakar Srivatsa, and BS Manjunath. Actor conditioned attention maps for video action detection. *arXiv preprint arXiv:1812.11631*, 2018.

[61] Subhashini Venugopalan, Marcus Rohrbach, Jeffrey Donahue, Raymond Mooney, Trevor Darrell, and Kate Saenko. Sequence to sequence - video to text. In *Proceedings of the IEEE international conference on computer vision*, pages 4534–4542, 2015.

[62] https://enterprise.verizon.com/business/learn/edge-computing/5G-and-edge-computing/, 2020.

[63] Bolun Wang, Xinyi Zhang, Gang Wang, Haitao Zheng, and Ben Y. Zhao. Anatomy of a personalized livestreaming system. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, pages 485–498, New York, NY, USA, 2016. ACM.

[64] Nicolai Wojke and Alex Bewley. Deep Cosine Metric Learning for Person Re-identification. *CoRR*, abs/1812.00442, 2018.

[65] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, October 2018. USENIX Association.

[66] Daniel Zhang, Tahmid Rashid, Xukun Li, Nathan Vance, and Dong Wang. Heteroedge: Taming the heterogeneity of edge computing system in social sensing. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, pages 37–48, 2019.