Model-driven Cluster Resource Management for AI Workloads in Edge Clouds

QIANLIN LIANG, University of Massachusetts Amherst, USA
WALID A. HANAFY, University of Massachusetts Amherst, USA
AHMED ALI-ELDIN, University of Massachusetts Amherst, USA
PRASHANT SHENOY, University of Massachusetts Amherst, USA

Since emerging edge applications such as Internet of Things (IoT) analytics and augmented reality have tight latency constraints, hardware AI accelerators have been recently proposed to speed up deep neural network (DNN) inference run by these applications. Resource-constrained edge servers and accelerators tend to be multiplexed across multiple IoT applications, introducing the potential for performance interference between latency-sensitive workloads. In this paper, we design analytic models to capture the performance of DNN inference workloads on shared edge accelerators, such as GPU and edgeTPU, under different multiplexing and concurrency behaviors. After validating our models using extensive experiments, we use them to design various cluster resource management algorithms to intelligently manage multiple applications on edge accelerators while respecting their latency constraints. We implement a prototype of our system in Kubernetes and show that our system can host 2.3X more DNN applications in heterogeneous multi-tenant edge clusters with no latency violations when compared to traditional knapsack hosting algorithms.

CCS Concepts: • **Computing methodologies** → **Model verification and validation**.

Additional Key Words and Phrases: cloud computing, edge computing, resource management, analytics modeling, ML inference

ACM Reference Format:
Qianlin Liang, Walid A. Hanafy, Ahmed Ali-Eldin, and Prashant Shenoy. 2023. Model-driven Cluster Resource Management for AI Workloads in Edge Clouds. 1, 1 (January 2023), **??** pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Recent technological advances have resulted in the emergence of new applications such as mobile Augmented Reality (mobile AR) [8] and Internet of Things (IoT) analytics [40]. A common characteristic of these applications is that their data needs to be processed with tight latency constraints. Consequently, edge computing, where edge resources can process this data close to the point of generation and at low latencies, has emerged as a popular approach for meeting the needs of these emerging applications [43].

It is increasingly common for such IoT applications to use AI inference as part of their data processing tasks. In contrast to *model training*, which involves training a machine learning (ML) model in the cloud, typically using large GPUs, *model inference* involves executing a previously

Authors' addresses: Qianlin Liang, qliang@cs.umass.edu, University of Massachusetts Amherst, Amherst, MA, USA, 01002; Walid A. Hanafy, whanafy@cs.umass.edu, University of Massachusetts Amherst, Amherst, MA, USA, 01002; Ahmed Ali-Eldin, ahmeda@cs.umass.edu, University of Massachusetts Amherst, Amherst, MA, USA, 01002; Prashant Shenoy, shenoy@cs.umass.edu, University of Massachusetts Amherst, Amherst, MA, USA, 01002.

XXXX-XXXX/2023/1-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

trained model for inference (i.e., predictions) over new data. For instance, video or audio data generated by IoT devices such as AR headsets, smart cameras, or smart speakers can be sent to a trained machine learning model for inference tasks such as object or speech recognition. The model, which is often a deep neural network (DNN), runs on an edge server to provide low-latency inference processing to the application.

Edge clouds, which extend cloud computing to the edge, are an increasingly popular approach for running low-latency inference for IoT applications. Edge clouds consist of small edge clusters that are deployed at a number of edge locations, where each edge cluster hosts multiple IoT applications. To efficiently run deep learning inference in constrained edge environments, such edge clouds have begun to use accelerators that are capable of executing DNN models using specialized hardware. Examples of edge accelerators include Google's EdgeTPU [48], Nvidia Jetson line of embedded GPUs [35], and Intel's Movidius Vision Processing Units (VPUs) [5]. When equipped with such hardware accelerators, edge cloud servers can significantly improve DNN inference tasks' latency—similar to how cloud servers use larger GPUs to speed up DNN training. As AI inference for IoT data processing gains popularity, the use of such accelerators to optimize DNN inference is likely to become commonplace in edge cloud environments.

Executing AI inference over IoT data in edge clouds raises new challenges. Similar to traditional cloud platforms, edge clouds will also be multi-tenant in nature, which means that each edge cloud server will run multiple tenant applications. These applications share the hardware resources of edge servers, including accelerators [13, 16, 26]. While conventional resources such as CPU and even server GPUs [25] support virtualization features to enable them to be multiplexed across applications, edge accelerators lack such hardware features. Consequently, multiplexing a DNN accelerator across multiple tenant applications can result in performance interference due to the lack of isolation mechanisms such as virtualization, which can degrade the response times seen by latency-sensitive IoT applications. This motivates the need for developing new cluster resources management techniques for efficiently multiplexing shared edge cloud resources across latency-sensitive applications.

To address the aforementioned challenges, in this paper, we present *Ibis*, a model-driven cluster resource management system for edge clouds. Ibis is designed to multiplex cluster resources, such as DNN accelerators, across multiple edge applications while limiting the performance interference between co-located tenants. Ibis uses a principled resource management approach based on analytic queueing models of hardware accelerators, such as edgeTPUs and edge GPUs, that are extensively experimentally validated on real edge clusters. These models are incorporated into Ibis' cluster resource manager and used to manage the online placement and dynamic migration of edge applications. In designing, implementing, and evaluating Ibis, our paper makes the following research contributions.

First, we develop analytical models based on queueing theory to estimate the response times seen by co-located DNN-based IoT applications running on shared edge servers with accelerators. We develop models to capture a range of multiplexing behaviors such as FCFS, processor sharing, batching, and multi-core parallelism that are seen when sharing edge GPUs and TPUs. We experimentally validate our models using two dozen different DNN models, drawn from popular model families such as AlexNet, ResNet, EfficientNet, Yolo, Inception, SSG, VGG, and DenseNet, and a variety of IoT workloads. Our extensive validation demonstrates the abilities of our queueing models to capture performance interference and accurately predict response times of co-located applications. [1]

[1]Note that the term "model" in this paper refers to both a deep neural network (DNN) inference model as well as an analytic queueing model; the two are distinct and we disambiguate them in the text as DNN inference and analytic/queueing model.

Second, we present an edge cluster resource manager that uses our analytic models for resource management tasks such as online placement and dynamic migration. Unlike traditional placement problems, which can be viewed as an online knapsack, we formulate a new problem called online knapsack with latency constraints for DNN application placement onto shared accelerators results. We present greedy heuristics that use our analytic models for latency-aware placement and migration in heterogeneous clusters.

Third, we implement a prototype of Ibis on a Kubernetes-based edge cluster and conduct detailed experiments to demonstrate the efficacy of our model-driven cluster resource management approach. Our results show that Ibis can host up to 2.3× the number of DNN models in heterogeneous edge clusters when compared to traditional knapsack hosting algorithms, and can dynamically mitigate hotspots using edge migration.

2 BACKGROUND

In this section, we provide background on edge computing and edge accelerators.

2.1 Edge Computing and AI Inference for IoT

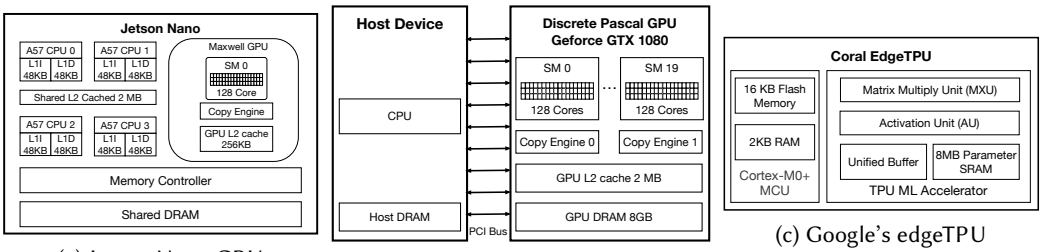
Edge clouds are a form of edge computing that involves deploying computing and storage resources at the edge of the network to provide low latency access to users [42]. However, edge clusters are smaller, and hence more resource-constrained than traditional cloud data centers. Analogous to cloud platforms that run multi-tenant applications in a server cluster, each edge cluster and edge server, is multiplexed across multiple applications—to maximize the utilization of scarce edge resources. Since many edge applications are inherently latency-sensitive, it is important to limit performance interference between co-located tenant applications. This can be achieved through the use of resource isolation mechanisms (e.g., virtualization), where available, and by carefully limiting the utilization and sharing of each server across tenant applications.

Our work focuses on emerging edge IoT applications, such as mobile AR, visual analytics over live videos from smart cameras, and voice assistants (e.g., Alexa, Siri) that run on smart speakers. Since these applications interact with users, it is necessary to process IoT data at low latencies to provide high user responsiveness, which imposes latency constraints on edge processing. A common characteristic of many IoT applications is that they employ machine learning models, often in the form of a Deep Neural Network (DNN), to process their data at the edge. Recent advances in computer vision technology have yielded a number of sophisticated and highly effective DNNs for common image processing tasks such as classification, object detection, and segmentation [29]. Advances in the field have allowed practitioners to provide a library of pre-trained DNNs such as ResNet [24], Inception [47], MobileNet [41], and Yolo [39], among others. An edge cloud developer can simply use one of these pre-trained DNN models within their application for performing inference tasks such as object detection or recognition over images. Alternatively, the developer can train a custom DNN for their application using easy-to-use ML frameworks such as TensorFlow [1], PyTorch [37], and Caffe [28] and datasets such as ImageNet [12] and CIFAR [30].

2.2 Edge Inference Accelerators

The growing popularity of DNN inference at the edge has led to the design of special-purpose accelerator hardware such as Nvidia's Jetson GPUs [35] and Google's edgeTPUs [48]. Edge cloud servers are beginning to employ such hardware for efficient inference execution. We provide a brief overview of these devices here.

2.2.1 Edge GPUs. Today's GPUs come in three main flavors, each targeting a different application workload. Server GPUs are high-end GPUs that are designed to accelerate parallel scientific



(a) Jetson Nano GPU (b) GeForce GTX 1080 GPU (c) Google's edgeTPU

Fig. 1. Architectural depiction of edge GPU, discrete GPU, and edgeTPU.

computations or speedup the training of machine learning models. Discrete GPUs are designed for gaming as well as scientific desktop applications (e.g., CAD) and have also been recently used for edge processing [33, 55] due to their interesting multiplexing capabilities. Finally, embedded GPUs are designed for edge (or on-device) applications. They have a low power footprint and are less capable than server or discrete GPUs, but well-suited for edge processing workloads. For example, Nvidia's Jetson family [35] of embedded GPUs are designed for running AI inference at the edge.

Figure 1a depicts the architecture of a Jetson Nano edge GPU—the smallest embedded GPU in the Jetson family. All GPUs in the Jetson family are *integrated* GPUs that integrate the CPU, GPU, and memory onto a single System-on-Chip (SoC). The figure shows that the device has 4 ARM-based CPU cores and a GPU comprising one Streaming Multiprocessor (SM) with 128 CUDA (GPU) cores. Notably, the device has 4 GBs of RAM that is shared between the CPU and the GPU. Like any Nvidia GPU, the Jetson Nano runs programs written in CUDA [14].

Each CPU process is associated with a CUDA *context* that is responsible for offloading compute-intensive functions, referred to as *kernels*, to the GPU. Kernels are submitted as a *stream* and executed sequentially. Nvidia GPUs support two basic types of concurrency, namely multi-processing (MP) and multi-threading (MT) [4, 53]. In multiprocessing, the GPU is time-shared between processes—processes (CUDA contexts) take turns to execute on the GPU for a time slice. On the other hand, in multi-threading, multiple threads, each associated with a separate CUDA context, can execute kernels concurrently via thread-level parallelism.

Researchers have noted that multi-threaded execution introduces significant synchronization overheads on GPUs, with increased blocking and non-determinism for latency-sensitive tasks [4, 53]. Since such non-deterministic blocking behavior is problematic for latency-sensitive edge applications, we focus our work on process-level concurrency, which has been shown to provide more predictable behavior [4, 53]. In our case, this implies a separate process executes each DNN model, and different processes can issue concurrent inference requests to execute using time-sharing on GPU cores.

2.2.2 Discrete GPUs. Discrete GPUs support additional multiplexing capabilities that are useful in multi-tenant edge settings [33, 55]. Discrete GPUs, such as GeForce 1080, are higher-end GPUs with their own on-GPU memory that is separate from CPU's RAM (see Fig 1b). As shown, the 1080 GPU has 20 SMs, 2560 GPU cores, and 8GM RAM, which is significantly greater than Jetson GPUs. Discrete GPUs support both multi-processing and multi-threading like their embedded counterparts. In addition, they also support Nvidia's Multi-Process Service (MPS) that enables true parallelism across concurrent GPU requests from independent processes [36]. When MPS is enabled in a GPU, all kernel requests are forwarded to MPS for scheduling. The MPS system partitions the GPU cores and memory across CUDA contexts and schedules kernels for execution on each

partition in parallel. In our case, this implies that DNN inference of two DNN models can execute in parallel increasing the utilization of the GPU cores (while in embedded GPUs, they execute using time-sharing but not in parallel).

2.2.3 EdgeTPU. EdgeTPU is an ASIC designed by Google for high performance DNN inference using very low power. In contrast to GPUs, which are optimized for performing floating-point operations, EdgeTPU uses 8-bit integers for computation and requires the DNN models to be quantized to 8-bit [48]. Employing quantization greatly reduces the hardware footprint and energy consumption of the EdgeTPU.

Figure 1c shows the architecture of EdgeTPU [48]. In contrast to CUDA cores, *Matrix Multiply Unit* (MXU) is the heart of EdgeTPU. It employs a systolic array architecture, which reuses inputs many times without storing them back to a register. By reducing access to registers, MXU is optimized for power and area efficiency for performing matrix multiplications and allows EdgeTPU to perform 4 trillion fixed-point operations per second (4 TOPS) using only 2 watts of power. EdgeTPU is designed specifically for DNN inference and is less general than GPUs. Its design is strictly deterministic and has a much smaller control logic than GPU. Thus, it can only run one task at a time in a non-preemptive FCFS manner. Unlike GPUs, time-sharing and multiprocessing are not supported.

3 ANALYTIC MODELS FOR INFERENCE WORKLOADS

The goal of our work is to design a cluster resource manager for edge clouds that can efficiently multiplex edge server resources, and specifically accelerator resources, across multiple applications. Our system employs a *model-driven resource management* approach, where we first design analytic models of edge workloads and then use these models to design practical cluster resource management algorithms. In this section, we design analytical models based on elementary queueing theory and use extensive experimentation to show that queueing models can (i) capture a range of multiplexing behavior seen in real-world accelerators such as edge GPUs and edgeTPUs, and (ii) accurately estimate the response times of a broad range of real-world DNN models, and (iii) can capture the impact of interference from co-located applications on application response times. For readability, Table 1 summarizes the notation and common equations used in our models.

3.1 Network of Queues Model

Queueing theory has been used to analytically model the behavior of web applications [15, 45, 49], server farms[21, 23] and cloud computing [3]. Here we use it to analytically model concurrent DNN applications running on edge servers with accelerators such as GPU and TPU.

Symbol	***Description***	***Notes***
λ	Arrival Rate	$\lambda = \sum \lambda_i$
μ	Service Rate	$\mu = 1/S$
S	Service time	$S = \sum \lambda_i S_i / \sum \lambda_i$
c	number of cores	
ρ	Utilization	$\rho = \lambda / c\mu$
$E[w]$	Expected Waiting Time	
$E[R]$	Expected Response Time	$E[w] + S$
$P(M_i)$	Probability of a request for DNN i	$P(M_i) = \lambda_i / \lambda$
e_i	Execution Time for DNN i	
o_i	Context Switch overhead for DNN i	

Table 1. Used Notations

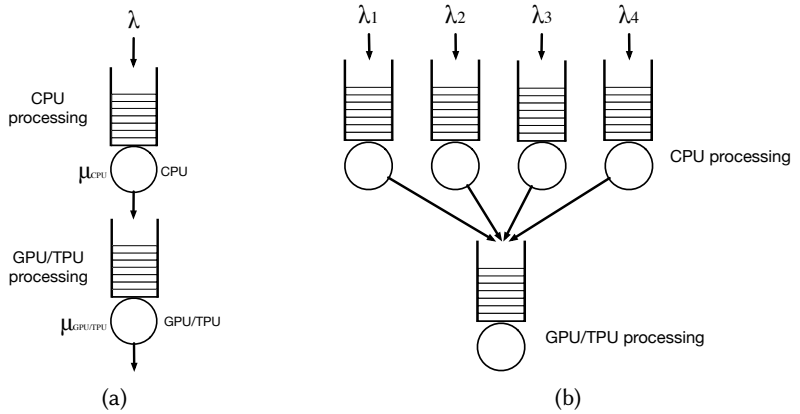


Fig. 2. (a) Tandem queue model for CPU and GPU/TPU processing of a request. (b) Network of queues model showing one CPU queue per application and a single GPU/TPU queue for all applications.

To do so, we assume each edge server has at least one accelerator and runs k concurrent DNN applications, $k \geq 1$. We assume that application i receives DNN inference requests at rate λ_i from an IoT device and specifies a mean response time R_i that should be provided by the edge cloud. We assume that each inference request undergoes a combination of CPU and GPU/TPU processing by the application.

We model this CPU and GPU/TPU processing using a network of queues model, with separate queues to capture the CPU and GPU/TPU processing of a request (see fig 2). In the simplest case where a single application runs on the edge server (i.e., $k = 1$), this model reduces to a tandem queue shown in fig 2a. In the general case where k applications are co-located on an edge server, we model each application's CPU processing as a separate queue. In contrast, the GPU/TPU processing of all applications is modeled as a single queue that is fed by the k CPU queues. Fig 2b shows the network of queueing model. We make this design choice since CPU processing of the k co-located applications is *isolated* from each other (since applications run inside containers or virtual machines and the CPU is a virtualized resource with isolation). Edge accelerators, in contrast, lack hardware support for virtualization. As a result, edge GPU and TPU processing is not isolated, and GPU/TPU requests from all k applications will be multiplexed onto the accelerator without isolation. Hence, we model GPU/TPU processing as a single queue that services the aggregate GPU/TPU workload $\lambda = \sum_{i=1}^{k} \lambda_i$ of all k applications.

By using a shared queue, our analytic model can capture the performance interference between applications on the accelerator and its impact on per-application response time. We need to design different queueing models for edgeTPUs and GPUs since they differ in three main aspects: (i) Multiplexing abilities: GPUs natively allow for multiple multiplexing modes such as process sharing and FCFS, while TPUs only allow FCFS. (ii) Memory Management: TPUs allow one model only to reside on the device (even if multiple models can fit in memory). However, GPUs allow multiple models as long as they fit in memory. (iii) Context switch overhead: GPUs have negligible context switching overhead while TPUs have switching overhead proportional to the model size. Given these characteristics, we next derive closed form equations of response times for TPU, GPU, and CPU processing for each queue in the network.

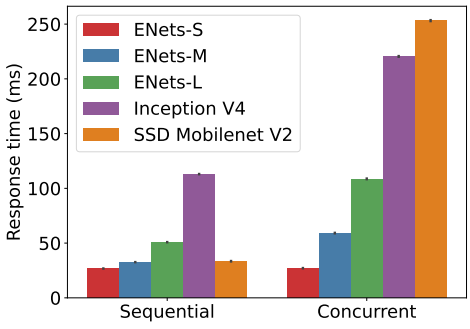


Fig. 3. Effect if Concurency on EdgeTPU

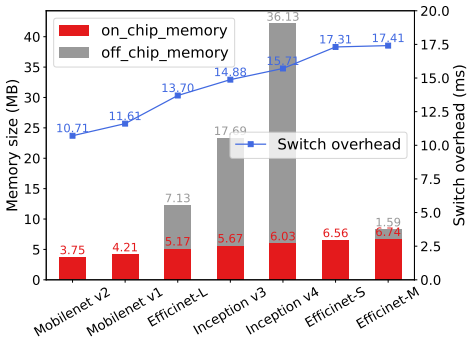


Fig. 4. EdgeTPU context switch overhead for various models.

3.2 Modeling TPU Inference Processing

We first model an edgeTPU accelerator that executes inference requests from k co-located applications. We assume that all k DNN models are loaded onto the TPU. When application i receives an inference request, it invokes the i^{th} DNN model for executing that request. Incoming TPU requests from all k applications are queued up in a single shared queue and processed in FCFS order. TPU request processing is sequential and non-preemptive in nature. Once TPU begins processing a request, the processing cannot be preempted. Upon completion, the DNN model corresponding to the next queued request is loaded from host RAM into device memory, resulting in context switching overhead before DNN inference can begin; no context switching overhead is involved if the next request invokes the same model as the previous one.

To demonstrate that request multiplexing on an edgeTPU is FCFS and non-preemptive, we experimentally ran five different DNN models on an edgeTPU node. We first ran each DNN model by itself in isolation and then with all DNNs executing concurrently. Fig 3 depicts the TPU execution (service) time in each case. As can be seen, in the presence of concurrent arrivals, the response time of requests beyond the first arrival includes the service time of previous arrivals, indicating FCFS and non-preemptive service. In addition, there is a non-negligible context switching overhead when loading a new DNN model to the edgeTPU, making it important to model. We experimentally quantify this overhead in Figure 4 in §3.2.1.

Therefore, when modeling edgeTPUs, the analytic model needs to consider three important characteristics. 1) The device processes concurrent requests in an FCFS manner. 2) Context switching overhead to a different model is not negligible. 3) The mean service time across all requests seen by the TPU will be the weighted sum of the service times of requests of co-located DNN models. Hence, we model the queue for a TPU as an $M/G/1/FCFS$ queueing system, where arrivals are Poisson with a rate λ, the service times have a general distribution, and requests are scheduled in an FCFS manner.

If k applications run on the edge server, where $k \geq 1$, and each sees an arrival λ_i, then $\lambda = \sum_i \lambda_i$ denotes the aggregate request rate at the TPU. Let S_i denote the expected service time, e_i denote the execution time and o_i be the switch overhead for workload i. Then we have

$$S_i = P(M_i)e_i + (1 - P(M_i))(e_i + o_i), \tag{1}$$

where $P(M_i) = \lambda_i / \sum_k \lambda_k$ is the probability that a randomly chosen request in the system runs DNN model i. Then the mean service time of the aggregated workload is the weighted sum of service

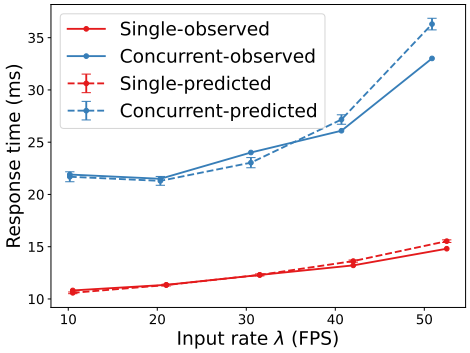


Fig. 5. TPU response time of Efficientnet-S

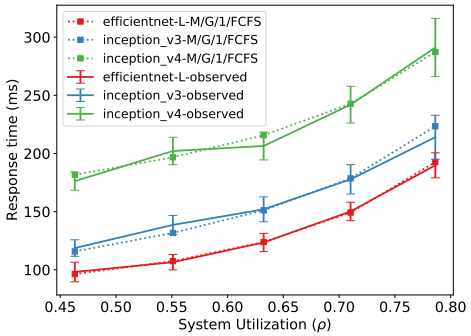


Fig. 6. TPU response times for concurrent DNN models.

time, weighted by arrival rates:

$$S = \tilde{\sum_i} P(M_i)S_i \tag{2}$$

For an $M/G/1/FCFS$ queueing system, the mean queueing delay seen by requests is given by the well-known Pollaczek-Khintchine (P-K) formula [17, 22]:

$$E[w] = \frac{\rho + \lambda\mu \text{Var}[S]}{2(\mu - \lambda)} \tag{3}$$

where $\mu = 1/S$ is the TPU service rate, $\rho = \lambda/\mu$ is the utilization, and $\text{Var}[S]$ denotes the variance of service time S. In the special case where there is a single tenant on the TPU, the DNN execution times can be modeled as a deterministic process, which reduces the $M/G/1/FCFS$ model to a $M/D/1/FCFS$ model. In this case, the waiting time for a $M/D/1/FCFS$ system is well-known and is given by:

$$E[w] = \frac{\rho}{1-\rho} \cdot \frac{1}{2\mu} \tag{4}$$

In either case, the mean response time of a particular model i can be approximated as

$$E[R_i] = E[w] + S_i \tag{5}$$

3.2.1 Experimental Validation of TPU Models. We conduct an experimental validation of our analytic TPU model to demonstrate that it can accurately predict TPU response time for a broad range of real-world DNN models running concurrently on the edgeTPU. To do so, we consider IoT applications that use different DNN models for image classification and object detection. We use two dozen DNN models from many of the most popular model families, namely, AlexNet, ResNet, EfficientNet, Yolo, Inception, SSG, VGG, and DenseNet. The characteristics of the used DNN models, along with their memory footprint and inference time are summarized in Table 2 in Section 4.4. The model sizes range from 3.5 million parameters to 144 million with a memory footprint of 22MB to 617MB. Collectively, these models range from small to large, both in their memory footprint and execution cost.

Our first experiment shows the context switch overhead when starting a new request execution. We loaded five DNN models on an edge server and sent a sequence of requests invoking these models in random order. Figure 4 shows the context switch overhead as well as on-chip and off-chip memory consumption of various models. Since the TPU stores all models in host server RAM and loads them to on-chip device memory on-demand, the context switch overhead is strongly correlated with model size (due to the overhead of copying the model from host RAM to TPU

memory). The context switch overhead ranges from 10 to 17ms, which is not negligible when compared to the DNN execution times that are shown in Figures 5 and 6. This experiment confirms the need to incorporate the context switch overhead when modeling request service times in Eq 1.

Next, we evaluate our TPU queueing model in the presence and absence of other co-located applications to demonstrate the impact of performance interference. We run an application that uses the Efficientnet-S model and vary its request rate. Initially we run the application by itself on the edge server and measure the response time under different request rates. We then run the application along with a second application running Mobilenet-V1 that sees a constant workload. We measure the response times in the presence of this co-located application. Fig 5 depicts the observed response times and those predicted by our analytic models when running the Efficientnet-S by itself and with the background MobilenetV2 application. As can be seen, the response time seen by Efficientnet-S is higher when it is running with a second application due to the performance interference from the background load. Further, our response times of our analytic model match closely with the observed response time, indicating that our model is able to capture the performance interference when sharing the TPU across multiple applications.

To further validate our TPU model, we run various mixes of the DNN models on a TPU cluster node with varying request arrival rates. Fig 6 shows the response times seen by three co-located applications—Efficientnet-L, Inception-V3 and Inception-V4—under different levels of system utilization. In all cases, the analytic model predictions closely match the observed response times over a range of utilization values. Finally, we repeat the experiment with other mixes of DNN models and observed similar prediction accuracies (graphs omitted due to space constraints).

Overall, our validation experiments show that our analytic models can (1) accurately capture the context switch overheads and multiplexing behavior of the TPU, (2) can capture performance interference from co-located applications, and (3) can accurately predict TPU response time for real-world DNN models and workloads.

3.3 Modeling GPU Inference Processing

We next model an edge GPU accelerator that executes inference requests from k co-located applications. Like in the TPU case, we assume that all k DNN models are loaded onto the GPU, and applications issues requests to these models upon receiving a request. Like before, all issued requests arrive at a shared queue and are processed by the edge GPU. Unlike the TPU that processes queued requests in an FCFS manner, edge GPUs, specifically those from Nvidia, have more sophisticated multiplexing capabilities. In particular, GPUs can process concurrent inference requests issued by different processes via *preemptive time-sharing* [36]. That is, if n concurrent requests are issued by n independent applications, the GPU will serve the requests using round-robin time-sharing, where each request receives a time slice before being preempted [36]. Unlike TPUs, which incur significant context switch overhead from memory coping, GPU's context switches are very efficient so long as models fit in GPU memory. Since GPUs have several GBs of on-device memory, we assume they can hold several models in RAM and are context switch overheads are negligible.

To demonstrate time-sharing behavior of an edge GPU, we ran several DNN models on a Jetson Nano GPU, first in isolation and then with all of them executing concurrently. Fig 7 shows that when requests to various DNN models arrive concurrently, their completion times reflect time-sharing behavior. For example, requests to Yolo4 and Resnet DNNs, shown in red and blue, complete at nearly the same time, which can not happen if they were processed sequentially in an FCFS manner. This experiment confirms the time-sharing behavior of the GPU.

From a queueing perspective, the time-sharing capability of the GPU lends itself to a process-sharing (PS) queueing discipline. However, there are some important system issues to consider. First, despite the GPU's time-sharing capabilities, multiple requests issued by the *same* process (i.e.,

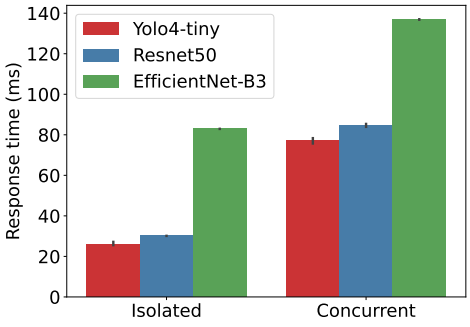


Fig. 7. Effect of Concurrency on Jetson Nano

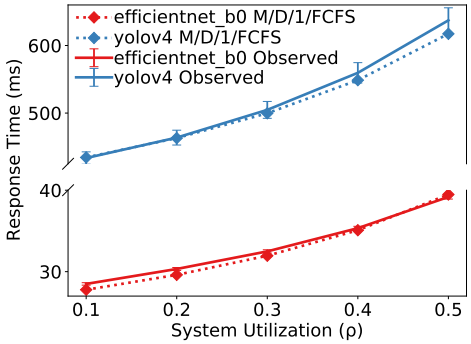


Fig. 8. GPU response times for small and large DNN models.

application) are serviced in FIFO order. This is because the GPU associates a single CUDA context to each OS process, and all requests from a CUDA context go into a FIFO queue on the device and are serviced sequentially (FCFS fashion). Time-sharing is possible only when concurrent requests are issued by *separate* processes (i.e., separate applications)) from distinct CUDA contexts.

Since we model the workload from all applications using a single queue, the resulting behavior will resemble some combination of an FCFS and PS queueing system. If multiple requests arrive concurrently from the same edge application, they will see FCFS processing. Conversely, if different applications issue concurrent requests to the GPU, these requests will see concurrent time-sharing processing (i.e., PS behavior).

Put another way, if an incoming request sees an idle system, it experiences FCFS processing. If the GPU is busy processing a request and another request arrives at the *same* application, it will be queued, also yielding FCFS behavior. In contrast, if the GPU is busy and a new request arrives at a *different* application, all requests receive service via process-sharing (i.e., time-sharing).

We model this GPU behavior using a combination of $M/G/1/FCFS$ and $M/G/1/PS$ system, which serve as the upper and lower bounds of what requests actually experience in the system. The waiting time and response time for $M/G/1/FCFS$ are given by the P-K formula and are the same as Equations 3 and 5. The response time of a $M/G/1/PS$ queueing system has the following closed form solution [22]

$$E[R] = \frac{1}{\mu - \lambda} \tag{6}$$

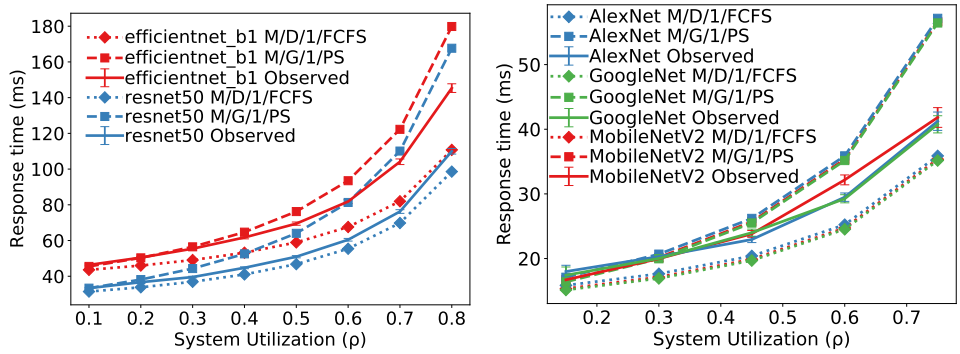
where $\lambda = \sum_i \lambda_i$ and $S = \frac{\sum_i \lambda_i S_i}{\sum_i \lambda_i}$ and $\mu = 1/S$. Note that $M/G/1/PS$ has a well-known insensitively property, where the behavior is independent of job size distribution, which yield the same response time solution of $M/M/1/FCFS$ [22]. The mean queueing delay is

$$E[w] = E[R] - E[S] = \frac{1}{\mu - \lambda} - \frac{1}{\mu} \tag{7}$$

The application-specific response time can be approximated as:

$$E[R_i] = E[w] + S_i \tag{8}$$

3.3.1 Experimental Validation of GPU Model. We now experimentally validate the above GPU model to show that it can accurately predict GPU response time and capture the multiplexing behavior of the GPU under different conditions. We also show that the real-world edge GPU behavior is bounded between our FCFS and PS models depending on arrival patterns. To do so,



(a) EfficientNet B1 & ResNet 50 (b) MobileNet, AlexNet, & GoogleNet

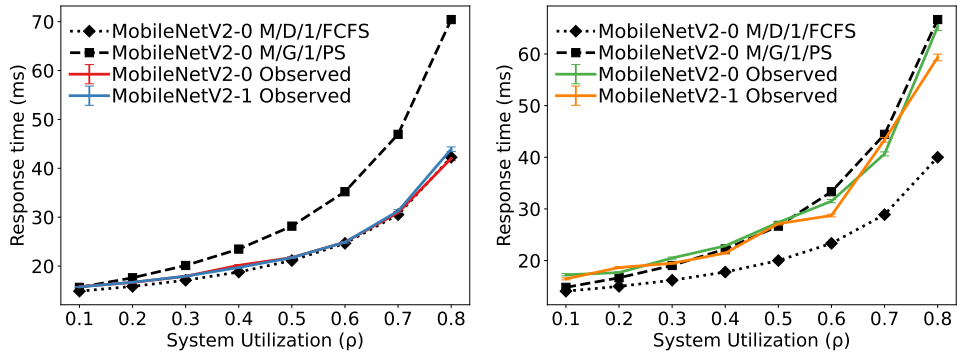
Fig. 9. GPU behaviour with multiple applications per node.

we first run a single DNN model on the GPU and subject it to various arrival rates. We choose an isolated EfficientNet_b0 (a small classification DNN) for this experiment and then repeat it with an isolated YoloV4 (a large object detection DNN) in isolation. In either case, since only a single CUDA context is running in the application, the request processing will be FCFS. Figure 8 shows the GPU response times at different utilization levels compared to the queueing models predictions. As shown, the observed GPU response times closely match the values predicted by our FCFS GPU model. This validates the FCFS behavior of the device under concurrent requests from the same application and also the ability of our model to capture this FCFS behavior.

Next, we validate our queueing models when multiple models run on a single node. Here, the observed response time should be lower and upper bounded by the pure FCFS and PS queueing models. Figure 9 describes the response time when the GPU is multiplexing multiple models, where Figure 9a shows the predicted response times according to the queueing models along with the observed response times when running two concurrent models on the cluster, namely, ResNet 50 and EfficientNet_b1. In addition, Figure 9b depicts a node with three concurrent DNN models (MobileNetV2, AlexNet and GoogleNet). In all cases, we see that the observed response time curve for each model lies *between* the PS and FCFS model curves. When a request arrives to an idle system or to the same application that served the previous request, it experiences FCFS behavior. Concurrent requests to different DNN models see time-sharing behavior. To further demonstrate the accuracy of the queueing models and behavior mentioned earlier, we run an experiment where we use a node with two identical MobileNetV2 models seeing the same arrival rates and force the models to follow a certain scheduling paradigm by issuing requests in a coordinated fashion.

In Figure, 10a we force FCFS scheduling by generating arrivals with no execution overlap. This guarantees that the model sees no time-sharing behaviour yielding exact matching with the FCFS curve. We then, as shown in Figure 10b, repeat the run forcing PS behavior using perfectly synchronized arrivals, hence causing time-shared processing. The results of the experiment are shown in Figure 10. As shown, the curve shifts down and closely matche the PS model curve. This validates our assumption that the Nano GPU exhibits a mix of both behaviors.

Together, these experiments show that our models can capture the time-sharing behavior of the GPU and also predict GPU response times in the presence of concurrent applications.



(a) Enforcing FCFS behavior (b) Enforcing PS behavior

Fig. 10. Enforcing PS and FCFS processing behaviour of two applications.

3.4 Modeling Parallel Inference Processing

While time-sharing is the default multiplexing behavior of a GPU, GPUs also support two types of parallelism, which we model in this section. First, GPUs support batch processing, where a batch of requests is issued to the GPU and data parallelism is used to process the batch in parallel. Second, discrete GPUs support request parallelism, where GPU cores are used to process multiple requests in parallel. We present models to capture both data parallelism and request parallelism multiplexing behaviors.

3.4.1 Modeling Batch Inference. Consider a GPU device that receives a batch of b requests for inference. Batch inference processing exploits data parallelism to distribute the processing of multiple requests in a batch across device cores. Consequently, processing the b requests on a single batch is faster than processing them individually (as b separate requests). From a modeling perspective, batch inference can be viewed as an increase in the GPU processing capability due to the speedup seen by batch inference—based on data parallelism. Consequently, we can model batch inference processing by viewing the batch of b requests as a single logical request that sees a faster service rate μ_b than the service rate μ seen by individual requests. Similar to [44], which reported this behavior as well, we model this faster service rate by estimating the service time of the logical request comprising the batch as follows:

$$S_b = k_1 + \frac{k_2}{b} \text{ where } b \geq 1 \tag{9}$$

where, k_1, k_2 are DNN model and device dependent constants, b denotes the batch sizes. The constants k_1, k_2 are estimated empirically for each model. This model captures typical device behavior where the latency decreases initially with increasing batch size, followed by asymptotically diminishing improvements in the latency. Consequently, we model service time of a batch $S_b \propto 1/b$. The service rate μ_b is then $1/S_b$. The adjusted service time and service rate can then be used in the $M/G/1/FCFS$ system from §3.2.

3.4.2 Experimental Validation of Batched Inference. We conduct experiments to validate data parallelism with batched requests. We consider three models GoogleNet, InceptionV3, and DenseNet. We execute each DNN model individually on the Geforce-GTX-1080 GPU and empirically profile each model. To do so, we run each DNN with batch sizes of $b = 1, 2, 4, 8$, and find the parameters

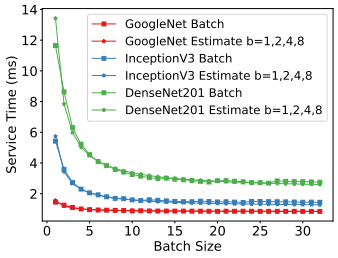


Fig. 11. GPU response times for batched request execution.

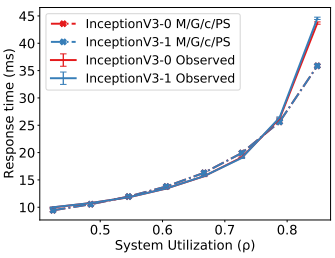


Fig. 12. Response time under MPS for two DNNs with c=1.65.

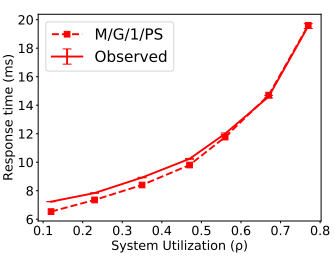


Fig. 13. CPU response times at different utilization levels.

for k_1 and k_2 of equation 9. Next we run each model on the GPU and vary the batch size from $b = 1$ to $b = 32$. Figure 11 compares the model predicted service time from equation 9, and the observed service time. Again, the model predictions closely match the observed response times over a range of batch sizes. The figure also shows that service time reductions show diminishing returns with the increase in batch size and the most significant gains are seen at small batch sizes. This is in line with our analytic model, which assumes asymptotically diminishing latency improvements with increasing batch size.

3.4.3 Modeling Parallel Inference using MPS. In contrast to batching, which provided data parallelism, Nvidia's multi-process service (MPS) provides request parallelism when executing concurrent requests from different DNN models in parallel on different GPU cores. MPS also supports memory partitioning and isolation of GPU resources [36]

The behavior of MPS can be modeled as a $M/G/c/PS$ system, with the GPU providing c servers that can execute c DNN models in parallel. Assuming an aggregate arrival rate λ across all GPU containers, a mean service time S and service rate μ, the response time yielded by a $M/G/c/PS$ system is given as

$$E[R] = \frac{c}{\lambda}.\frac{\rho}{1-\rho} = \frac{c}{c\mu - \lambda} \tag{10}$$

In practice, the degree of multiprocessing depends on the (i) the actual number of cores on the device, and (ii) the model size in terms of its processing and memory needs. In most cases, with MPS enabled, the GPU acts as a multi-processor with a small value of c (e.g., c is often 2 or 3 for desktop-class GPUs, indicating a limited degree of parallelism).

3.4.4 Experimental Validation of MPS-based Parallel Inference. We conduct experiments to validate the above model that captures request parallelism due to Nvidia's MPS scheduler. In this case, we ran two InceptionV3 models on the GeForce 1080 GPU with the MPS daemon running with each having a varying request arrival rate. Figure 12 compares the observed response time to those predicted by the queueing model for $c = 1.65$ (empirically measured speed up). As shown, our model has a good match with the observed values, which shows that our analytic model is able to accurately capture parallel request processing on GPUs.

3.5 Modeling CPU processing

Having analytically modeled GPU and TPU processing under a range of multiplexing behaviors such as FCFS, time-sharing, and data/request parallelism. We turn to CPU processing incurred by each edge request. Unlike GPU and TPU accelerators, where we use a single queue to model the aggregate workload of all k applications, the CPU model assumes a separate queue for each application. This is because each application is assumed to run in a separate container or a VM

on the edge server and is allocated a dedicated amount of CPU capacity (e.g., a fraction of a CPU core or multiple cores). We assume that the operating system uses standard CPU time-sharing for processing requests within each container or VM.

Under this scenario, the CPU processing of each application can be modeled as a separate $M/G/c/PS$ queueing system. The $M/G/c/PS$ queueing system is well studied in the literature and has a closed form equation for average response time, which is given by [22]:

$$E[R] = \frac{c}{\lambda} \cdot \frac{\rho}{1-\rho} = \frac{c}{c\mu - \lambda}, \text{ since } \rho = \lambda/c\mu \tag{11}$$

3.5.1 CPU Model Validation. To experimentally validate our CPU model, we allocate a fixed amount of CPU and memory to each application container and ran a multi-threaded process to accept incoming image requests and perform CPU processing on this request prior to issuing it to the GPU or TPU We varied the arrival rate and experimentally measured the response time for various degrees of CPU utilization. Figure 13 shows that the observed CPU response time increases with utilization and closely matches the prediction of our model, showing that the model accurately captures the CPU processing of the application. Results for multi-core containers (c ¡ 1) are similar and omitted due to space constraints. Our results show that our model can capture CPU processing of DNN inference requests and that use of separate queueing models for each applications yields accurate response time predictions.

3.6 Estimating End-to-End Response Time

Our previous sections presented analytic models for the CPU and GPU/TPU processing stages for each inference request. To estimate the end-to-end response time, we can use the network of queues model from figure 2. The mean end-to-end is the sum of the mean response time of each stage in the network. For application i the end-to-end response time $R_i^{total} = R_i^{CPU} + R_i^{GPU/TPU}$, where R_i^{CPU} and $R_i^{GPU/TPU}$ can be computed using the above analytic models.

4 MODEL-DRIVEN CLUSTER RESOURCE MANAGEMENT

In this section, we show how the predictive capabilities of our analytic models can be employed for cluster resource management tasks such as online DNN placement and dynamic migration. We also discuss the implementation of our algorithms into our Ibis prototype that is based on the kubernetes cluster manager.

4.1 Latency Aware Online Knapsack Placement

A key resource management task performed by cluster managers in cloud computing platforms is online placement.[2] In this case, new applications arrive into the cluster and must be placed onto a server with adequate unused capacity to run that application. The placement problem in cloud computing has been well studied for over a decade [50] and is typically viewed as a multi-dimensional knapsack problem [6]. In this case, each server in the cluster is a knapsack and the various dimensions of the knapsack represent the resource capacities of the server (e.g., capacities of the CPU, memory, network). An application request specifies the amount of each resource it needs, and the knapsack problem involves selecting an edge server with sufficient unused resources.

In the case of edge clusters with accelerators, the traditional knapsack placement approach is not applicable. This is because traditional knapsack placement in cloud computing has assumed that resource requirements are additive and an application can run on a server if it "fits" on that server.

[2]Offline placement assumes that all applications arrive at once and must be placed together onto an empty cluster, while online placement assumes applications arrive sequentially and must be incrementally placed without knowledge of future arrivals.

Algorithm 1: Latency-aware Online Knapsack Placement

Input: A dictionary `dnnConfig` contains DNN profile, task type, input rate λ and latency constrain τ; A list of available nodes, `nodeList`.

Output: A `selectedNode` to place the incoming DNN such that all applications running on the node will not violate their latency SLOs after placement

```
feasibleNodes ← []
memNeed ← computeMemNeed(dnnConfig)
for node in nodeList do
    resTime ← computeResponseTime(dnnConfig, node)   /* Use queueing models
     */
    sysUtil ← computeSysUtilization(dnnConfig, node)
    memOk ← memNeed Ÿ node.freeMem            /* Check knapsack resources */
    sloOk ← resTime Ÿ τ
    utilOk ← sysUtil Ÿ maxRho
    if memOk && sloOk && utilOk then
        feasibleNodes.append(node)
    end
end
if dnnConfig.type == "AIaaS" then
    groupNodes ← findNodeWithSameApp(dnnConfig, feasibleNodes)
    feasibleNodes ← groupNodes if !groupNodes.empty()
end
selectedNode ← findNodeWithLeastUtil(feasibleNodes) return selectedNode
```

This assumption is reasonable since all applications runs in VMs or containers and are isolated from each other using virtualization.

As explained in §3, accelerators do not provide support for virtualization or isolation and co-located DNN applications on a GPU or TPU see performance interference. Hence, it is no longer sufficient for placement techniques to check if the DNN model will "fit" on a GPU or TPU memory. The placement technique should additionally ensure that the increased workload and performance interference from placing a new application will not cause response time requirements to be violated. We refer to this new problem as *online placement with latency constraint*, where the knapsack (i.e., edge server) has both resource capacity constraints as well as a response time (latency) property. While used resource capacity increases in an additive manner with each placed application, response time increases non-linearly. Thus, traditional packing algorithms that are based on linear packing assumptions do not hold in our setting.

Our placement approach is based on our analytic queueing models to address the latency constraint. We assume that a newly arriving DNN application specifies its resource (e.g., CPU, memory) needs as well as a latency constraint (R_i^{th}), for an expected request rate λ_i.

Our placement technique first determines a list of all feasible servers in the cluster that can house the new application. A server is feasible if (1) it has sufficient free resources such as CPU, CPU Memory, GPU memory to house the DNN application and (2) the end-to-end response time seen by the new as well as existing applications are below their specified thresholds. That is, for each application on that server $R_i^{cpu} + R_i^{GPU/TPU} \leq R_i^{threshold}$, R_i^{cpu} and $R_i^{GPU/TPU}$ are computed using our analytic models.

If no feasible server exits, the application placement request is rejected. Otherwise, a greedy heuristic is used to pick a specific server from the list of feasible candidate servers. Currently, our system supports two greedy heuristics. (1) highest utilization (aka worst fit) that is designed to achieve a tight packing on the smallest number of servers and (2) lowest utilization, which chooses the least loaded server to house the new application. Algorithm 1 lists the pseudo code for our online knapsack placement with latency constraints.

4.1.1 Heterogeneous and Grouped Placement. We next present two enhancements to our baseline placement algorithm, namely heterogeneous and grouped placement. The heterogeneous placement algorithm assumes an edge cluster with heterogeneous servers, where each server is equipped with a GPU, TPU, or both. The goal of the placement algorithm is to choose a suitable server and the best accelerator type for a newly arriving application. To do so, the placement algorithm computes the CPU response time of the application on each feasible server as well as the GPU and TPU response times, depending on the accelerators on each feasible server. The placement algorithm greedily chooses the GPU server as well as the TPU server with the least latency from this feasible set.

The server with the lower of the two latencies is then chosen to house the application, which also determines the best accelerator for the application. As we show in §4.4.2, no one accelerator is optimal for all DNN models, and this enhancement enables the best accelerator to be chosen, based on the performance offered by each type of accelerator and the load on servers.

Our second enhancement use a *grouped* placement technique that we refer to as *AI-as-a-Service* (AIaaS) placement. In the AIaaS model, the edge cloud provider offers a choice of several pre-trained DNN models as an edge service. An application can simply choose one of these DNN models and avoid having to supply its own DNN for inference tasks. The advantage of the AIaaS model is that the cluster manager can opportunistically group applications that choose the same DNN model on the same server — in doing so, it can load a single copy of the DNN for all applications that have chosen it, instead of loading one DNN model per application. This can potentially increase the cluster capacity due to the reduced memory requirements. In the grouped placement, if the DNN model is already executing on the edge cluster (by being chosen by one or more prior applications), then we need to determine if the newly arriving application can be grouped with the existing ones. To do so, we aggregate the request rate λ_i of all grouped applications and use the analytic queueing model to determine the GPU and TPU response time of the shared container. If the latency threshold of the entire group can be met for the aggregate workload, the new application is co-located with the current group. If the application cannot be placed with the current group, it is placed onto a new server to start a new grouping.

4.2 Model-driven Dynamic Migration

Application workloads in edge clouds tend to be dynamic and will fluctuate over time. While our analytic model ensures that response times meet latency objectives for a specified request rate λ_i, applications will nevertheless experience latency violations if the workload fluctuates dynamically and the observed request rate $\hat{\lambda_i}$ exceeds the specified workload. To handle dynamic workloads, Ibis uses a *police-and-migrate* strategy. Each application container includes a token bucket regulator to police the incoming workload—the token bucket is configured with a rate λ_i and configurable burst b_i. Hence, if the observed request rate exceeds the specified rate, requests get queued by the token bucket regulator. Doing so avoids overloading the underlying accelerator and isolates other co-located tenants from experiencing performance degradation due to the overloaded tenant. A sustained hotspot is mitigated by dynamically migrating the overloaded tenants to a new less-loaded server. To do so, Ibis monitors the mean end-to-end response time for each application over a moving time window. It also tracks the observed request rate $\hat{\lambda_i}$ for each application. If latency

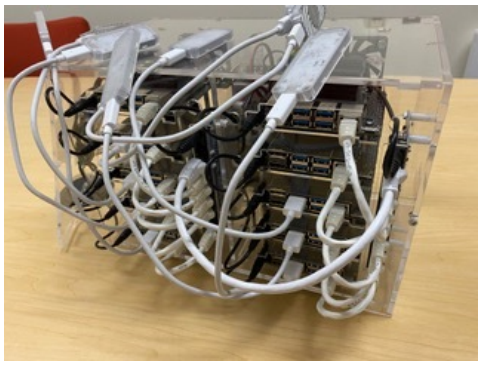


Fig. 14. 10-node Jetson Nano cluster

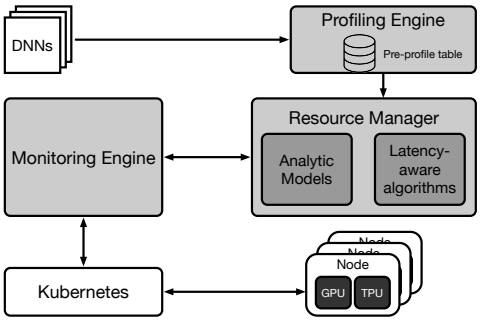


Fig. 15. Ibis implementation overview

violations or request drops are observed on any node, the application whose observed request rate $\hat{\lambda}_i$ exceeds the specified rate λ_i is flagged for migration. The analytic queueing models are used to determine a new node and accelerator for this application using a new higher request rate estimate. The application is then migrated to the new node using container or VM migration.

4.3 Ibis Implementation

We have implemented a prototype of our system, named Ibis, using Kubernetes on a custom-built edge cluster of ten nodes depicted in Figure 14, each comprising a Jetson Nano GPU, 4 GB of RAM, and quad-core ARM processor. We use a similar ten node cluster equipped with USB edgeTPUs for our TPU experiment. Our cluster also contains a Dell PC with a 3.2 GHz i7-8700 processor, 16 GB RAM, and Geforce-GTX-1080 GPU. All machines are connected via a gigabit ethernet switch and run Linux Ubuntu 18.04. Further, each Jetson Nano runs CUDA 10.2, cuDNN 8, and TensorRT 7.1.3 while the PC runs CUDA 11, cuDNN 7.6.5, and TensorRT 7.1.3. All machines are virtualized using Docker 19 and managed by Kubernetes 11; gRPC 1.31 is used for inter-communication.

Figure 15 depicts Ibis system overview. Our prototype currently runs DNN models in TensorRT engine format for GPU and quantized `tflite` format for EdgeTPU. For AIaaS models, Ibis first retrieves its resource requirements (e.g., memory, service time) using a pre-profile table. On the other hand, in the case of User-Supplier models (provided in ONNX format), we compile and profile it on an idle node and save the result back to the profile table for future use. Our application containers are deployed as Kubernetes `Deployment`, and we use Kubernetes `Service` to route network traffic. Ibis collects node resource status and chooses a node and device using our analytic models and scheduling strategies. Finally, Ibis deploys the incoming workload to the target node and device via Kubernetes interface. Source code for our prototype is available on Github *(URL blinded)*.

4.4 Experimental Evaluation of Model-driven Resource Management Algorithms

In this section, we experimentally evaluate our cluster resource management techniques on an edge cluster using realistic workloads. To do so, we deploy Ibis on our 10 node edge cluster and use the trace workloads discussed below for our experiments.

4.4.1 Trace Workloads. The trace workload for each IoT application is constructed using our 21 DNN models and a sequence of images from the ImageNet dataset [12]. Table 2 shows the characteristics of the DNN models used in our experiments. We adopt models for both image

Models	Scale	Input Shape	Parameters	Static Size (MB)*	Runtime Footprint (MB)*	FLOPs (GF)	Inference Time (ms)*
Classification Models							
AlexNet	S	[N, 3, 224, 224]	62M	138	992	0.7	14.18
GoogleNet	S	[N, 3, 224, 224]	6 M	39	893	2	13.37
InceptionV3	M	[N, 3, 224, 224]	24 M	81	836	6	30.56
MobileNetV2	S	[N, 3, 224, 224]	3.5 M	22	1130	0.6	13.02
ResNet18	S	[N, 3, 224, 224]	12 M	69	930	1.8	10.83
ResNet34	S	[N, 3, 224, 224]	21.2 M	155	1044	3.6	19.51
ResNet50	M	[N, 3, 224, 224]	26 M	106	965	3.8	29.2
ResNet101	L	[N, 3, 224, 224]	44.5 M	247	1135	7.6	50.32
EfficientNet-b0	S	[N, 3, 224, 224]	5.3 M	30	1168	0.4	26.03
EfficientNet-b1	M	[N, 3, 240, 240]	7.8 M	42	1184	0.7	41.32
EfficientNet-b2	M	[N, 3, 260, 260]	9.2 M	49	1196	1	49.58
EfficientNet-b3	L	[N, 3, 300, 300]	12 M	77	1229	1.8	81.67
EfficientNet-b4	L	[N, 3, 380, 380]	19 M	124	1042	4.2	166.15
EfficientNet-b5	L	[N, 3, 456, 456]	30 M	180	1320	9.9	337.44
DenseNet121	M	[N, 3, 224, 224]	7.2 M	50	910	3	30.14
DenseNet201	L	[N, 3, 224, 224]	20 M	103	964	4	89.11
VGG16	L	[N, 3, 224, 224]	138 M	407	1275	16	86.36
VGG19	L	[N, 3, 224, 224]	144 M	463	1333	20	99.19
Object Detection Models							
YoloV3	L	[N, 3, 416, 416]	62 M	617	1501	65.88	190.24
YOLO-tinyV4	M	[N, 3, 416, 416]	6.06 M	75	938	6.91	23.79
YoloV4	L	[N, 3, 608, 608]	64.43 M	445	1329	128.46	407.91

Table 2. DNN characteristics for GPU (Values are based on Jetson Nano with FP16, batch size of 1).

classification and object detection tasks. Each model contains a set of parameters whose size directly affects its memory footprint. All parameters are pre-trained. Classification models are pre-trained on ImageNet dataset and detection models are pre-trained on COCO [32] dataset. In addition to model parameters, TensorRT allocates extra memory for CUDA context and runtime. The Runtime Footprint column shows the total memory used by each model, note that the large difference between runtime memory and static size is due to CUDA runtime requirements, while actual inputs and intermediate results are overheads are minimal. The input of the models is a batch of images in NCHW format, where N is the batch size, C is the number of channels, H is the height, and W is the width of the image. During inference, a certain number of floating point operations (FLOPs) are performed. We categorize the models into small (S), medium (M), and large (L) based on their memory footprint and FLOPs per input sample.

Further, we use the public Azure trace [10] to construct realistic mixes of containerized applications of varying sizes. Azure Public Dataset provides data on the characteristic of the production virtual machine workloads of large cloud providers. It describes the characteristics of Microsoft Azure's VM workload, including distributions of the VMs' lifetime, deployment size, and resource consumption. The trace is created using data from the 2019 Azure VM workload containing information about 2.6M VMs and 1.9B utilization readings. To create realistic workloads, we analyze the distribution of VM sizes (CPU utilization and memory footprint) in the Azure trace and use them to generate the utilization and model size of our DNN-based applications. We categorize VMs as small (Ÿ2GB), medium (2-8GB), and large (¡ 8GB) and find that proportions of small, medium, and large applications are 47%, 33%, and 20% respectively. The mix of small, medium, and large DNN models housed on the edge cluster is chosen in the same proportion. Finally, models in the same category are chosen evenly.

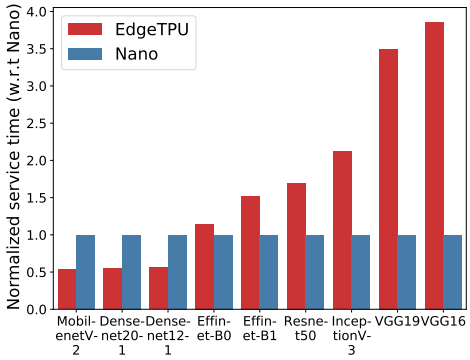


Fig. 16. Normalized service time for various DNN models.

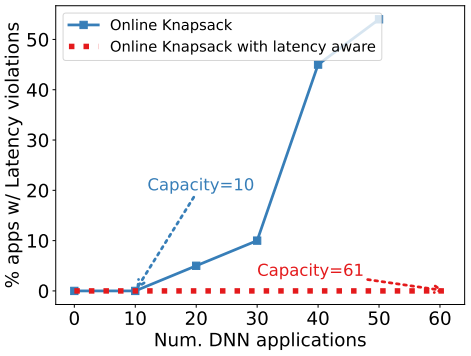


Fig. 17. Latency violations under different co-location methods

We use this Azure trace to generate an arrival trace of DNN applications that require placement on our cluster. Each arriving application is chosen from the above DNN types and is associated with a response time constraint R_i, and worst-case request rate λ.

4.4.2 DNN profiling. We start with profiling of various DNNs to measure their service times in isolation on a Nano GPU and an edgeTPU. Figure 16 shows the normalized execution time of various models. Interestingly, the figure shows that smaller DNN models run faster on the TPU than the GPU, while larger ones run faster on the GPU than the TPU. This is because the edgeTPU is an ASIC designed specifically for DNN inference and has higher operations per second than more general-purpose GPU devices. But it has a small (i.e., 8 MB) device or on-chip memory. To run models with a memory footprint larger than this limit, it employs host, or off-chip memory, to store the extra model parameters. This incurs a context switch overhead since accessing off-chip memory is much slower than accessing on-chip memory, and hence, larger DNNs have worse performance than smaller DNNs on EdgeTPU. On the other hand, GPU does not have this problem as it can store all DNN runtime and parameters in its larger device memory; the reduction in context switch overhead offsets the somewhat higher execution times, yielding lower service times for larger models.

Our results show that neither the TPU nor GPU is optimal for all DNN models, and the optimal choice will depend on the characteristics of each model. This result also motivates the need for our heterogeneous placement technique discussed in Sec 4.1.1.

4.4.3 E icacy of our placement algorithm. We conduct an experiment to show the efficacy of our online knapsack placement with latency constraints. We do so by comparing our approach with a traditional online knapsack placement approach that is used for cloud application placement. As noted earlier, traditional knapsack placement is latency oblivious and performs placement using additive resource packing. We construct application arrival traces with an increasing number of applications and place them on our edge cluster using the two placement methods and measure whether there are any response time violations when subjecting applications to the specified request rates. In the following experiments, DNN inference requests are dispatched to the container from an idle local node on the network. Figure 17 show our results. As shown, the latency-oblivious traditional knapsack sees latency violations even with a small number of co-located applications, and the number of applications seeing violations increases with more application arrivals. When we try to load more than 50 applications with a latency-oblivious policy, the system crashes. In

contrast, our latency-aware policy is able to place 61 applications onto the cluster, with no latency violations for any application.

Next, we run a simulation experiment where we vary the number of applications that need to be placed from 10 to 70, and construct 1000 random arrival traces for each data point. We try to place applications from each trace on a ten node cluster of discrete GPUs using three knapsack aware methods i) latency (Proposed model), ii) utilization [38], and iii) online knapsack. Then we determine whether each placement is successful (i.e., whether the applications fit on the cluster and if there are any latency violations using our queuing equations). Fig 18 shows the fraction of workloads that can be successfully placed on the cluster using these three methods. As the number of applications increases, the probability of successful placement for a random arrival pattern decreases for all methods. However, the success rate decreases much faster for the latency-oblivious traditional knapsack due to the higher rate of response time violations, followed by a moderate decrease in the number of applications. Finally, our method can encounters a higher probability of successful placement. For a 90% or higher cutoff success rate, the traditional method can place at most 30 applications, utilization method is able to host 40 applications, while our method can place nearly 70 applications, yielding a 2.3X increase in cluster capacity.

4.4.4 E icacy of our heterogeneous placement. We next show the efficacy of our heterogeneous placement enhancement. We conduct an experiment where we construct an arrival sequence where each application randomly makes a static choice of a GPU or TPU accelerator, and we use our latency-aware online knapsack to make placement decisions. We compare this static placement policy with our heterogeneous placement approach, where the choice of the accelerator is made dynamically by our placement algorithm. Figure 19 shows the number of applications placed onto the cluster in each case. As can be seen, when a static choice is made, some applications can make a bad choice and use more resources than they need (e.g., as see in fig 16 VGG16 has 4× more service time on TPU than on GPU). In the case of heterogeneous placement, the best accelerator is chosen for each application, and resources are effectively used to pack more applications. The heterogeneous policy yield a cluster capacity improvement of 10%. This experiment shows that Ibis is able to maximize resource sharing while maintaining response time guarantees.

Moreover, we notice that the efficacy of heterogeneous placement also depends on the model type. As shown in figure 16, the service time saving of heterogeneous placement varies from 15% to 75%. If the workloads consist of models with strong hardware preferences (e.g., VGG16), the cluster capacity improvement can be much larger. On the other hand, if most of the models do not have hardware preferences (e.g., Effinet-B0), the improvement would be minor. In addition, device memory capacity also has a great impact on the efficacy of heterogeneous placement. If one type of accelerators run out of memory, we have to place all incoming models to the other type of accelerators, which may yield worse performance. In our experiment, on GPUs, distinct models are loaded within different CUDA contexts, which introduces a huge memory overhead. As a result, the GPUs run out of memory quickly, and all incoming models have to be placed on the TPUs, which offset the benefits of our heterogeneous placement policy.

4.4.5 E icacy of AIaaS Placement. Next, we conduct an experiment to compare the degree of sharing achieved using AI-as-a-Service applications, user-trained models, and a mix of both. We construct a trace of application arrivals and use our algorithms to place these applications using all three approaches on the Jetson Nano GPU cluster. As shown in Fig 20 , AI-as-a-Service placement achieves 2.1× cluster capacity improvement compare to user-trained models. This is because when loading a model on GPU, a significant amount of memory is allocated by CUDA context and runtime. As shown in table 2, model parameters only consume 15% of total memory footprint. Since user-trained models are run in different containers, a new CUDA context is created when a

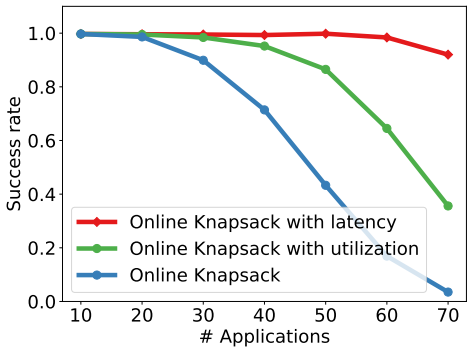


Fig. 18. Latency aware vs traditional knapsack

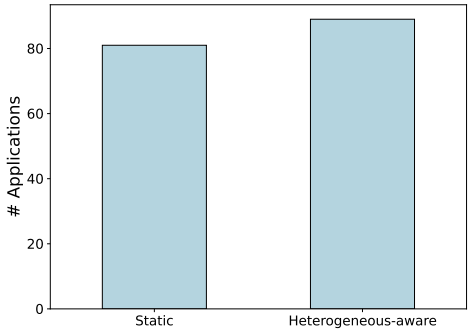


Fig. 19. Benefits of heterogeneous placement.

model is placed to the cluster. As a result, a Jetson Nano node runs out of memory after loading 2-3 applications. No additional applications can be loaded even though the GPU is underutilized. In this case, the cluster capacity is bounded by memory. On the other hand, AIaaS applications are run in a shared container and thus share the CUDA context. Since sharing amortizes the memory overhead of CUDA context, AIaaS is able to achieve a greater degree of resource sharing. Finally, the mixed workload placement achieves co-location performance that lies between user-trained and AIaaS placement.

Moreover, the effectiveness of AIaaS placement is dependent on device memory capacity and runtime memory dynamics. If the memory capacity is high, the improvement of AIaaS will be limited. However, when the memory capacity is limited, which is generally the case for edge servers, the capability of amortizing memory overhead allows AIaaS to achieve better resource sharing and cluster utilization.

4.4.6 Evaluation of hotspot migration with token bucket. Finally, we conduct an experiment to demonstrate the efficacy of hotspot mitigation with token bucket. We start the experiment with the system moderately loaded with two DNN models — EfficientNet-S and MobileNetV2. We then increase the request rate of EfficientNet-S beyond the placement time estimate. As shown in figure 21, the workload increase causes the response time of EfficientNet-S to exceed the latency threshold but does not affect the MobileNetV2 because of the token bucket. Our system monitors the response time seen by customers and triggers a migration when violations are observed. The overloaded application is migrated to a new node with sufficient idle capacity, causing the response time of the overloaded application to fall below the latency violation threshold. This experiment shows the ability of our techniques to mitigate hotspots dynamically.

4.5 Limitations

Our current Ibis prototype has four main limitations:

Mean Vs Tail Latencies. Ibis uses queueing models to predict the system behavior under certain load conditions. However, our queueing models focus on mean latency and do not consider tail latencies. Extending our models to support tail latencies is part of future work.

GPU Memory Overheads. In GPU deployments, Ibis deploys a model per container/process resulting in a CUDA context per process. However, as mentioned in section 4.4.1, this adds a high memory overhead per container. Although one alternative is to share the context between DNN models using multiple threading. However, as mentioned in [4, 53], this can add higher unpredictability due to interference.

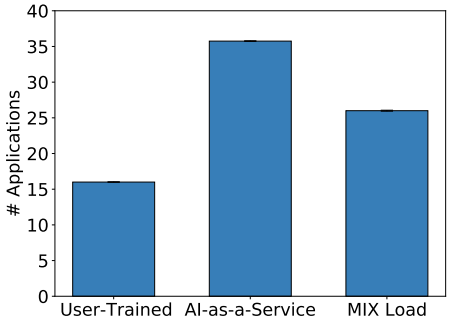


Fig. 20. Max application capacity for AIaaS and user-trained models

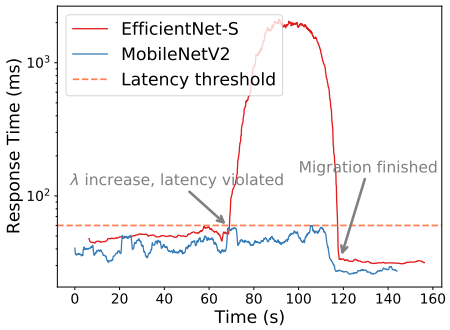


Fig. 21. Hotspot mitigation with token bucket.

TPU Limitations. TPUs have two main limitations: its low memory and the execution model. We expect that larger edgeTPUs will appear in the near future, resolving the first issue. However, the execution model presents a fundamental source of overhead. TPUs load only a single model at a time (even if memory can accommodate multiple), which adds a context switch overhead. One solution to this problem is to co-compile the models into a single model graph which allows the TPU to decrease the mandatory context switch overhead across models. However, this approach comes at a higher cost since compiling the models beforehand is infeasible, and online compilation will add overheads.

Security Model. Currently, edgeTPUs and edge GPUs don't provide any isolation/virtualization mechanisms. Our threat model assumes a trusted cloud manager while users are not trusted, similar to most cloud services. To solve this problem, Ibis takes a DNN model in a standard format such as onnx and builds a run-time container without any user-provided code.

5 DISCUSSION

Geographical Placement. Ibis assumes all nodes reside in one edge cluster and does not explicitly consider multiple edge sites with different geographical locations. However, Ibis can be extended to support the multiple edge sites. For example, we can adaptively change the response time SLOs based on network latency estimations of different edge sites once this is done. Ibis will then ensure the SLOs are met. Moreover, Ibis separates feasibility checking and scheduling, which allows it to incorporate the ever-growing number of scheduling policies.

Network. Ibis does not explicitly consider the network latency between CPU and GPU/TPU containers. In our experiment, the containers of one application are placed in the same node. The low network latency, in this case, had a negligible impact on our results. However, in the case when containers are in different nodes, the network latency can affect our model accuracy. We can remove this limitation by enforcing the placement of the containers in the same node or adjusting the SLOs to tolerate the network latency.

Security. Security is important for all shared systems. Ibis addresses the security problem by separating user code and DNN models. Users are allowed to run arbitrary code in the CPU container with isolated resources, but they have no access to the accelerators. The DNN model is loaded in GPU/TPU container, which has access to the accelerator. Users can only execute the DNN model through provided APIs.

6 RELATED WORK

DNN Inference systems. Building SLO aware inference systems have been widely discussed [11, 18, 44, 46, 54]. Zhang et al. [54] proposed a scalable system for DNN inference workloads, where different classes of resources are provisioned to allow efficient and effective scaling. Similarly, Nexus [44] provides a solution for deploying multi-level AI workload utilizing batching to decrease the processing time. Clipper [11] proposes a general-purpose low-latency prediction serving system by introducing caching, batching, and adaptive model selection techniques. Soifer et al. [46] provide insights on the inference infrastructure at Microsoft, which duplicates requests with cross cancellation tokens to ensure predictable response times. Clockwork [18] proposes a model serving system that fulfills aggressive tail-latency SLOs by restricting the choices available to lower system layers. In our work, we provide a proactive estimation tool for predicting the performance of DNNs and provide a latency and heterogeneity aware scheduling mechanism for edge deployments.

Edge inference. Although most DNN inference systems focus on cloud deployments. The promise of edge computing is to provide lower latency for real-time applications by eliminating the latency of offloading to remote clouds [2, 40, 42, 55]. Hardware accelerators such as GPUs and TPUs provide a powerful add-on for edge infrastructure [7, 34, 56]. Many researchers have studied and characterized the performance of various edge accelerators [19, 20, 31, 51]. [19] characterizes several commercial edge devices on popular frameworks using well-known CNNs. [31] further characterizes edge accelerators using typical device-cloud-edge computing paradigm, such as split processing. Moreover, [20] highlights the correlation between DNN models and edge accelerators. Finally, [51] demonstrates an FPGA-accelerated and general-purpose distributed stream processing system for Edge stream processing.

Multi-tenancy on accelerators. Sharing of GPUs across applications has been studied for cloud servers [13, 16, 26]. Olympian [26] and GPUShare [16] focus on sharing a single GPU across multiple users, while GSLICE [13] focuses cluster-level sharing. PERMA [9] proposes a preemption mechanism that facilitates neural network accelerators to become preemptible and share between multiple DNN tasks. Also, [18] highlights the effect of multiplexing Inference requests, while proposing using FCFS scheduling to provide latency guarantees. In contrast to these efforts, we focus on analytic models, using queueing theory, to enable GPU or TPU multiplexing while providing response time guarantees.

Model-aware placement. Model (queueing models) have been used extensively to monitor and predict the performance of traditional processing units (i.e., CPU) [3, 15, 21, 23, 45, 49]. There has been recent work on capturing the performance of accelerators using queueing models. The work in [27] presents a queueing model to capture the effect of GPU batching of a single application. Zhang et al. [54] proposes using queueing models to meet SLO requirement of machine learning inference serving on public cloud. They assume deterministic inference time and only one inference job running on a node. Also, [52] shows how queuing models can be used to schedule DNN models on cloud CPU clusters. In contrast, we cover multiple accelerator architectures at the edge, which in resource constrained environments.

7 CONCLUSION

In this paper, we presented analytic models to estimate the latency behavior of DNN inference workloads on shared edge accelerators, such as GPU and edgeTPU, under different multiplexing and concurrency behaviors. We then used these models to design resource management algorithms to intelligently co-locate multiple applications onto edge accelerators while respecting their latency constraints. Our results showed that our models can accurately predict the latency behavior of

DNN applications on shared nodes and accelerators, while our algorithms improve resource sharing by up to 2.3X when providing response time guarantees.

Our future work will extend our models and implementation in three main directions: i) Our current queueing model captures response time on CPUs, TPUs, and GPUs. We plan to extend the queueing models to include other ASICs such as FPGA, DSP units, etc. Further, since such hardware support different runt-time configurations (e.g., Power modes and DVFS), our models can be extended to provision resources in an energy efficient way, which is crucial for energy-constrained edge deployments. ii) Multi-tier Deployments: In our current implementation, we assume that both the pre-processing and inference phases are deployed on the same physical node. Multi-tier deployments are beneficial in cases where the edge is connected to the cloud or heterogeneous nodes. In these cases, our tandem queue can be extended to include multiple phases such as transmission from client to edge or across nodes. iii) Multi-Edge Deployments: Although we focused on single edge deployment, our queueing models can be used in multi-edge deployments, where tandem queues can be dynamic. However, this will require adding new scheduling policies based on management goals such as fault-tolerance and energy aware placement.

ACKNOWLEDGEMENTS

We thank the TAAS reviewers for their valuable comments, which improved the quality of this paper. This research is supported by NSF grants 2211302, 2211888, 2213636, 2105494, US Army contract W911NF-17-2-0196, Adobe and VMware.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

[2] Yuan Ai, Mugen Peng, and Kecheng Zhang. 2018. Edge computing technologies for Internet of Things: a primer. *Digital Communications and Networks* 4, 2 (2018), 77 – 86.

[3] Pradeep Ambati, Noman Bashir, David W. Irwin, and Prashant J. Shenoy. 2020. Waiting game: optimally provisioning fixed resources for cloud-enabled schedulers. In *SC*.

[4] Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. 2017. GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*. 104–115. https://doi.org/10.1109/RTSS.2017.00017

[5] Brendan Barry, Cormac Brick, Fergal Connor, David Donohoe, David Moloney, Richard Richmond, Martin O'Riordan, and Vasile Toma. 2015. Always-on Vision Processing Unit for Mobile Applications. *IEEE Micro* 35, 2 (2015), 56–66. https://doi.org/10.1109/MM.2015.10

[6] Chandra Chekuri and Sanjeev Khanna. 2005. A Polynomial Time Approximation Scheme for the Multiple Knapsack Problem. *SIAM J. Comput.* 35, 3 (2005), 713–728. https://doi.org/10.1137/S0097539700382820

[7] J. Chen and X. Ran. 2019. Deep Learning With Edge Computing: A Review. *Proc. IEEE* 107, 8 (2019), 1655–1674.

[8] Kaifei Chen, Tong Li, Hyung-Sin Kim, David E. Culler, and Randy H. Katz. 2018. MARVEL: Enabling Mobile Augmented Reality with Low Energy and Low Latency. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems* (Shenzhen, China) *(SenSys '18)*. Association for Computing Machinery, New York, NY, USA, 292–304. https://doi.org/10.1145/3274783.3274834

[9] Yujeong Choi and Minsoo Rhu. 2020. PREMA: A Predictive Multi-Task Scheduling Algorithm For Preemptible Neural Processing Units. *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2020), 220–233.

[10] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. 153–167. https://doi.org/10.1145/3132747.3132772

[11] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *NSDI*.

[12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[13] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. 2020. GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. 492–506. https://doi.org/10.1145/3419111.3421284

[14] M. Fatica. 2008. CUDA toolkit and libraries. In *IEEE Hot Chips (HCS)*. 1–22.

[15] Anshul Gandhi and Amoghvarsha Suresh. 2019. Leveraging Queueing Theory and OS Profiling to Reduce Application Latency. In *Proceedings of the 20th International Middleware Conference Tutorials* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/3366625.3368853

[16] Anshuman Goswami, Jeffrey Young, Karsten Schwan, Naila Farooqui, Ada Gavrilovska, Matthew Wolf, and Greg Eisenhauer. 2016. GPUShare: Fair-Sharing Middleware for GPU Clouds. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1769–1776. https://doi.org/10.1109/IPDPSW.2016.94

[17] Donald Gross, John F. Shortle, James M. Thompson, and Carl M. Harris. 2008. *Fundamentals of Queueing Theory* (4th ed.). Wiley-Interscience, USA.

[18] Arpan Gujarati, Reza Karimi, Safya Alzayat, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *OSDI*.

[19] Ramyad Hadidi, Jiashen Cao, Yilun Xie, Bahar Asgari, Tushar Krishna, and Hyesoon Kim. 2019. Characterizing the Deployment of Deep Neural Networks on Commercial Edge Devices. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. 35–48. https://doi.org/10.1109/IISWC47752.2019.9041955

[20] Walid A. Hanafy, Tergel Molom-Ochir, and Rohan Shenoy. 2021. Design Considerations for Energy-Efficient Inference on Edge Devices. In *Proceedings of the Twelfth ACM International Conference on Future Energy Systems* (Virtual Event, Italy) *(e-Energy '21)*. Association for Computing Machinery, New York, NY, USA, 302–308. https://doi.org/10.1145/3447555.3465326

[21] Mor Harchol-Balter. 2008. Scheduling for Server Farms: Approaches and Open Problems. In *Proceedings of the SPEC International Workshop on Performance Evaluation: Metrics, Models and Benchmarks* (Darmstadt, Germany) *(SIPEW '08)*. Springer-Verlag, Berlin, Heidelberg, 1–3. https://doi.org/10.1007/978-3-540-69814-2_1

[22] Mor Harchol-Balter. 2013. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action* (1st ed.). Cambridge University Press, USA.

[23] Mor Harchol-Balter, Takayuki Osogami, Alan Scheller-Wolf, and Adam Wierman. 2005. Multi-Server Queueing Systems with Multiple Priority Classes. *Queueing Systems* 51 (2005), 331–360.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[25] Cheol-Ho Hong, Ivor Spence, and Dimitrios S. Nikolopoulos. 2017. GPU Virtualization and Scheduling Methods: A Comprehensive Survey. *ACM Comput. Surv.* 50, 3, Article 35 (June 2017), 37 pages. https://doi.org/10.1145/3068281

[26] Yitao Hu, Swati Rallapalli, Bongjun Ko, and Ramesh Govindan. 2018. Olympian: Scheduling GPU Usage in a Deep Neural Network Model Serving System. In *Proceedings of the 19th International Middleware Conference* (Rennes, France) *(Middleware '18)*. 53–65. https://doi.org/10.1145/3274808.3274813

[27] Yoshiaki Inoue. 2021. Queueing analysis of GPU-based inference servers with dynamic batching: A closed-form characterization. *Performance Evaluation* 147 (2021), 102183. https://doi.org/10.1016/j.peva.2020.102183

[28] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia* (Orlando, Florida, USA) *(MM '14)*. Association for Computing Machinery, New York, NY, USA, 675–678. https://doi.org/10.1145/2647868.2654889

[29] Xiaoyue Jiang, Abdenour Hadid, Yanwei Pang, Eric Granger, and Xiaoyi Feng (Eds.). 2019. *Deep Learning in Object Detection and Recognition*. Springer Singapore. https://doi.org/10.1007/978-981-10-5152-4

[30] Alex Krizhevsky. 2012. Learning Multiple Layers of Features from Tiny Images. *University of Toronto* (may 2012).

[31] Qianlin Liang, Prashant J. Shenoy, and David E. Irwin. 2020. AI on the Edge: Characterizing AI-based IoT Applications Using Specialized Edge Architectures. In *International Symposium on Workload Characterization,*. IEEE, 145–156.

[32] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. 2014. Microsoft COCO: Common Objects in Context. In *ECCV*.

[33] Yuhua Lin and Haiying Shen. 2016. CloudFog: Leveraging fog to extend cloud gaming for thin-client MMOG with high quality of service. *Transactions on Parallel and Distributed Systems* 28, 2 (2016), 431–445.

[34] Vittorio Mazzia, Aleem Khaliq, Francesco Salvetti, and Marcello Chiaberge. 2020. Real-Time Apple Detection System Using Embedded Systems With Hardware Accelerators: An Edge AI Application. *IEEE Access* 8 (2020), 9102–9114. https://doi.org/10.1109/ACCESS.2020.2964608

[35] Nvidia. 2020. *NVIDIA Jetson Modules.* Retrieved October 19, 2020 from https://developer.nvidia.com/embedded/jetson-modules

[36] Nvidida. 2020. *Multi Process Service.* https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32.

[38] Michael Pawlish, Aparna S. Varde, and Stefan A. Robila. 2012. Analyzing utilization rates in data centers for optimizing energy management. In *2012 International Green Computing Conference (IGCC)*. 1–6. https://doi.org/10.1109/IGCC.2012.6322248

[39] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, Vol. 2016-Decem. 779–788.

[40] Mohammad Salehe, Zhiming Hu, Seyed Hossein Mortazavi, Iqbal Mohomed, and Tim Capes. 2019. VideoPipe: Building Video Stream Processing Pipelines at the Edge. In *Proceedings of the 20th International Middleware Conference Industrial Track* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 43–49. https://doi.org/10.1145/3366626.3368131

[41] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[42] Mahadev Satyanarayanan. 2017. The Emergence of Edge Computing. *Computer* 50, 1 (2017), 30–39. https://doi.org/10.1109/MC.2017.9

[43] Mahadev Satyanarayanan, Paramvir Bahl, Ramon Caceres, and Nigel Davies. 2009. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing* 8, 4 (2009), 14–23. https://doi.org/10.1109/MPRV.2009.82

[44] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. 322–337. https://doi.org/10.1145/3341301.3359658

[45] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. 2011. CloudScale: Elastic Resource Scaling for Multi-Tenant Cloud Systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (Cascais, Portugal) *(SOCC '11)*. Article 5, 14 pages. https://doi.org/10.1145/2038916.2038921

[46] Jonathan Soifer, Jason Li, Mingqin Li, Jeffrey Zhu, Yingnan Li, Yuxiong He, Elton Zheng, Adi Oltean, Maya Mosyak, Chris Barnes, Thomas Liu, and Junhua Wang. 2019. Deep Learning Inference Service at Microsoft. In *2019 USENIX Conference on Operational Machine Learning (OpML 19)*. Santa Clara, CA, 15–17.

[47] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2818–2826. https://doi.org/10.1109/CVPR.2016.308

[48] Olivier Temam, Harshit Khaitan, Ravi Narayanaswami, and Dong Hyuk Woo. 2019. Neural network accelerator with parameters resident on chip. Patent# US20190050717A1.

[49] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. 2005. An Analytical Model for Multi-Tier Internet Services and Its Applications. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (Banff, Alberta, Canada) *(SIGMETRICS '05)*. 291–302. https://doi.org/10.1145/1064212.1064252

[50] Bhuvan Urgaonkar, Arnold L. Rosenberg, and Prashant J. Shenoy. 2007. Application Placement on a Cluster of Servers. *Int. J. Found. Comput. Sci.* 18 (2007), 1023–1041.

[51] Song Wu, Die Hu, Shadi Ibrahim, Hai Jin, Jiang Xiao, Fei Chen, and Haikun Liu. 2019. When FPGA-Accelerator Meets Stream Data Processing in the Edge. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 1818–1829. https://doi.org/10.1109/ICDCS.2019.00180

[52] Feng Yan, Yuxiong He, Olatunji Ruwase, and Evgenia Smirni. 2018. Efficient Deep Neural Network Serving: Fast and Furious. *IEEE Transactions on Network and Service Management* 15, 1 (2018), 112–126. https://doi.org/10.1109/TNSM.2018.2808352

[53] Ming Yang, Nathan Otterness, Tanya Amert, Joshua Bakita, James H. Anderson, and F. Donelson Smith. 2018. Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems. In *30th Euromicro Conference on Real-Time Systems, ECRTS 2018, July 3-6, 2018, Barcelona, Spain*, Vol. 106. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 20:1–20:21. https://doi.org/10.4230/LIPIcs.ECRTS.2018.20

[54] Chengliang Zhang, Minchen Yu, wei wang, and Feng Yan. 2020. Enabling Cost-Effective, SLO-Aware Machine Learning Inference Serving on Public Cloud. *IEEE Transactions on Cloud Computing* (2020), 1–1. https://doi.org/10.1109/TCC.20

20.3006751

[55] Wuyang Zhang, Jiachen Chen, Yanyong Zhang, and Dipankar Raychaudhuri. 2017. Towards Efficient Edge Cloud Augmentation for Virtual Reality MMOGs. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing* (San Jose, California) *(SEC '17)*. Article 8, 14 pages. https://doi.org/10.1145/3132211.3134463

[56] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang. 2019. Edge Intelligence: Paving the Last Mile of Artificial Intelligence With Edge Computing. *Proc. IEEE* 107, 8 (2019), 1738–1762.