SEEB-GPU: Early-Exit Aware Scheduling and Batching for Edge GPU Inference

Srinivasan Subramaniyan
subramaniyan.4@osu.edu
The Ohio State University
Columbus, Ohio, USA

Rudra Joshi
joshi.485@osu.edu
The Ohio State University
Columbus, Ohio, USA

Xiaorui Wang
wang.3596@osu.edu
The Ohio State University
Columbus, Ohio, USA

Marco Brocanelli
brocanelli.1@osu.edu
The Ohio State University
Columbus, Ohio, USA

Abstract

The deployment of deep neural networks (DNNs) on edge devices is becoming increasingly common in latency-sensitive applications such as autonomous driving, real-time video analytics, and augmented reality. However, modern DNNs are rapidly growing in complexity, and edge GPUs often lack the computational resources available in cloud counterparts. This leads to increased inference latency and challenges in meeting strict Service Level Agreements (SLAs).

In this paper, we introduce SEEB-GPU, a portable and efficient inference framework that addresses such challenges through a combination of application-level and system-level optimizations. At the system level, SEEB-GPU assigns dedicated Thread Processing Cluster (TPC) masks to individual inference tasks, preventing resource contention by spatially isolating GPU Streaming Multiprocessors (SMs). At the application level, it employs a deadline-aware heuristic that dynamically adjusts batch sizes and confidence thresholds, utilizing early-exit mechanisms in DNNs to significantly reduce inference latency while minimally reducing accuracy. Experimental results demonstrate that SEEB-GPU reduces inference latency by up to 15×, while maintaining competitive accuracy compared to existing edge inference methods. To our knowledge, SEEB-GPU is the first framework to jointly integrate batching, early-exits, and fine-grained GPU partitioning, thereby meeting SLAs and improving throughput on edge GPUs.

ACM ISBN 979-8-4007-2238-7/25/12
https://doi.org/10.1145/3769102.3772715

CCS Concepts

• **Computing methodologies** → **Machine learning**; • **Computer systems organization** → **Embedded and cyber-physical systems**.

Keywords

Edge Inference, GPU Resource Management, Early-Exit DNNs

ACM Reference Format:
Srinivasan Subramaniyan, Rudra Joshi, Xiaorui Wang, and Marco Brocanelli. 2025. SEEB-GPU: Early-Exit Aware Scheduling and Batching for Edge GPU Inference. In *The Tenth ACM/IEEE Symposium on Edge Computing (SEC '25), December 3–6, 2025, Arlington, VA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3769102.3772715

1 Introduction

Deep neural networks (DNNs) play an essential role in numerous real-world applications, including driving assistance systems [16, 20], traffic monitoring [1, 27], facial recognition [13], mobile augmented reality, adaptive video streaming, and vehicle re-identification. Due to rising concerns about network overhead, data privacy, and strict latency requirements, developers are increasingly moving inference tasks from cloud platforms to edge servers embedded with small GPUs.

However, contemporary DNN models are becoming increasingly complex due to the rapid growth in the number of model parameters. Edge GPUs provide far less computational power and DRAM capacity than cloud GPUs [12]. For example, NVIDIA's A100 GPU [35], a high-end cloud accelerator, delivers up to 19.5 TFLOPS of peak FP32 performance, while Jetson's TX2 [34], a commonly used edge device, offers only 0.5-0.7 TFLOPS in FP32 (or 1.3 TFLOPS in FP16). This represents a compute gap of over 30×. As a result, effective optimizations are essential to ensure that edge-based inference tasks meet strict latency and throughput requirements.

Existing research primarily focuses on application-level optimizations to enhance model efficiency and performance. These approaches include model pruning, which removes redundant weights to reduce model size and computation [54], low-bit quantization, which reduces the precision of weights and activations to accelerate inference and lower memory usage [7], layer sharing, where parameters or computations are reused across layers to minimize overhead, layer skipping, which dynamically bypasses certain layers based on input features to save computation [26], and early-exits, which enable intermediate predictions when *confidence* is sufficient, avoiding the need to process the entire model [14]. These optimizations largely overlook GPU-specific resource management.

GPU scheduling mechanisms strongly affect inference latency and throughput, but pose several challenges [8, 12]. For instance, temporal sharing mechanisms can lead to GPU underutilization, as tasks are executed sequentially [17]. In contrast, *spatial sharing*, particularly through NVIDIA's Multi-Process Service (MPS), allows for some level of concurrent task execution. However, MPS introduces significant overhead, requires process termination to dynamically adjust resource allocation, and lacks fine-grained control over the precise allocation of GPU Streaming Multiprocessors (SMs). Solutions such as GSLICE [12] attempt to adapt batching strategies and utilize MPS proportionally, but these still result in potential oversubscription without fine-grained SM control. To the best of our knowledge, no existing solution simultaneously exploits application-level and GPU system-level optimizations to address inference latency and response time constraints comprehensively for edge inference.

To address these challenges, we propose SEEB-GPU, a solution designed to ensure Service Level Agreement (SLA) compliance for diverse edge inference scenarios. SEEB-GPU takes a unique approach to GPU resource allocation by providing precise control over Thread Processing Clusters (*TPCs*) and directly managing the allocation of Streaming Multiprocessors (SMs). This level of control significantly reduces resource contention and enables better adherence to soft real-time latency requirements. The solution aggregates inference requests at runtime and proportionally allocates *TPCs* to each task according to its computational demand. At the model level, we implement an advanced heuristic based on the Earliest Deadline First (EDF) scheduling algorithm. This dynamic approach determines suitable *batch sizes* and adjusts model *confidence thresholds* to permit early-exits. By combining these techniques, SEEB-GPU minimizes latency, maximizes inference accuracy, and significantly reduces SLA violations. Notably, the proposed method is portable across both NVIDIA and AMD GPUs, with implementation details and portability considerations thoroughly discussed in Section 3. While batching, early-exits, and GPU *spatial sharing* are individually established techniques, their integration within SEEB-GPU is non-trivial. Prior work typically explores these methods in isolation, which can lead to inefficiencies such as contention or deadline misses in real-time, multi-tenant edge scenarios. The novelty of SEEB-GPU lies in its unified framework that jointly manages batching, exit decisions, and fine-grained GPU partitioning to deliver SLA compliance and high throughput. To our knowledge, no previous work has demonstrated such integration on resource-constrained GPUs.

Overall, this paper makes the following contributions:

- We propose a novel pre-batching approach that aggregates inference requests to allocate GPU resources, explicitly controlling *TPCs* to reduce contention.
- We design a heuristic algorithm based on EDF scheduling that dynamically adjusts *batch sizes* and *confidence levels* to leverage early model exits and meet the soft deadlines.
- We demonstrate our approach through extensive evaluations, showing significant latency reduction, improved accuracy, and better SLA compliance compared to existing edge inference methods.

The remainder of this paper is organized as follows. Section 2 provides the necessary background for this work and discusses the motivation behind our proposed solution. Section 3 outlines the design of our proposed solution, while Section 4 describes the experimental setup used in our evaluations. Hardware evaluations are discussed in Section 5, and Section 6 reviews the related work in the field. Finally, Section 7 presents the conclusions of our paper.

2 Background and Motivation

In this section, we review key background concepts relevant to our work. We begin with GPU sharing techniques and the use of early-exits in machine learning models. We then examine several motivational results for choosing SEEB-GPU.

2.1 Background

2.1.1 GPU Sharing. It allows a single GPU to be used by multiple applications or users simultaneously. This is achieved through two main techniques: *spatial sharing* and *temporal sharing*. These methods enable different tasks or processes to access the GPU's computational resources either concurrently or sequentially, depending on the workload and system configuration. GPU sharing enhances resource utilization and system efficiency, making it essential in environments where multiple users or applications need access to GPU resources [44, 47, 48]. In this paper, we focus only on *spatial sharing* since it allows multiple applications to access

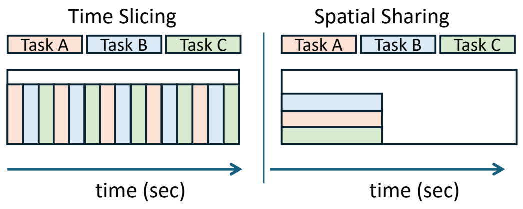


Figure 1: Example of temporal and spatial sharing on GPUs.

the GPU simultaneously. *Spatial Sharing* divides the GPU's computational resources between multiple tasks, allowing them to run concurrently on separate SMs. In NVIDIA GPUs, a SM is the fundamental hardware unit executing parallel threads. Each SM contains a set of cores, registers, schedulers, and caches that manage and execute thousands of lightweight GPU threads simultaneously. By using *spatial sharing*, we optimize the parallel processing capabilities of the GPU, enabling the simultaneous execution of multiple workloads without interference while improving the overall computational efficiency. Figure 1 shows an example of *spatial sharing*, where latency is reduced compared to temporal sharing, where tasks are time-sliced.

There are two ways to enable *spatial sharing* in NVIDIA GPUs [17]. First, if each of the inference tasks is launched as a separate process, we can enable *spatial sharing* through the MPS (Multi-Process Service) from NVIDIA. The MPS server schedules kernels from different processes to the GPU, allowing them to run concurrently and even share the same SMs. However, when multiple inference workloads with varying latency and throughput requirements share GPU resources, ensuring fair allocation and predictable performance becomes critical. This is where Service Level Agreement (SLA) mechanisms are needed. MPS supports coarse-grained SLA through the environment variable `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE`, which limits the maximum fraction of GPU thread contexts a process can use. This helps prevent any single process from monopolizing compute resources and enables controlled concurrency across multiple workloads.

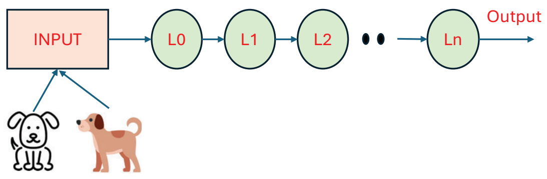
Second, if all inference tasks are launched from the same process, then *spatial sharing* is enabled, as only a single context is present in the GPU. To achieve coarse-grained SLA on the number of SMs assigned to a task in the GPU, we use the `libsmctrl` package, which applies *TPC masks* to map tasks to specific *TPCs* [4]. Since the Pascal architecture, each *TPC* contains 2 SMs. Thus, controlling *TPCs* effectively enables coarse-grained control over SM allocation. Assigning tasks to different multiprocessors reduces contention and decreases the inference latency. In addition, with `libsmctrl` we load the model only once at the beginning and reassign SMs at runtime. It is worth noting that once a task has been assigned to SMs via CUDA streams using *TPC* masking, it cannot be preempted during execution [22]. Reassignment is possible only after the task completes, at which point the SMs can be reassigned with negligible overhead. In AMD GPUs, *spatial sharing* is present by default [8].

We follow the second approach, where we launch all the tasks from a single process on the edge server (receiving all requests from different users and applications) to *spatially* share the GPU. The reason is two-fold: (1) MPS does not dynamically adjust a process's GPU resources during execution. For example, if a process is launched with 80% MPS threads using `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` and if we want to adjust the resources assigned to the process, we must terminate and restart the process. (2) After termination, restarting requires data copying; as a result, dynamically adjusting MPS introduces 5-15 seconds of overhead [5]. Therefore, edge applications that require low overhead need a *spatial sharing* mechanism capable of directly assigning SMs to tasks.

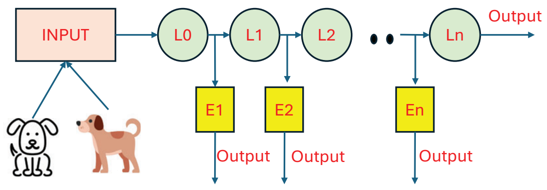
2.1.2 *Model-Specific Optimization.*

Designing and deploying DNNs on resource-constrained mobile and IoT devices requires a careful balance between inference latency, memory efficiency, and model accuracy. Recent research has focused on model-specific optimization techniques to address these challenges. These include (1) model pruning, which reduces the network size by removing less important weights to improve computational efficiency [54]; (2) low-bit quantization, which compresses the model by reducing the precision of weights and activations [7]; (3) layer sharing, which reuses parameters across layers to minimize redundancy and save memory; (4) layer skipping, which dynamically bypasses certain layers based on input characteristics to reduce computation [26]. While these static optimization techniques significantly reduce model complexity and inference cost, they typically lack flexibility during runtime. To enable adaptive computation, dynamic inference methods such as model slicing [6], model switching [51], and early-exits [39] have been proposed.

In this work, we adopt early-exits as our primary control knob due to their adaptability and deployment simplicity. Unlike model slicing [6] or model switching [51], which require maintaining and selecting among multiple model variants and result in higher memory usage and orchestration overhead, early-exit networks operate within a single architecture by attaching intermediate classifiers at various depths. These classifiers enable dynamic inference by allowing early termination when the model's prediction *confidence*, typically the maximum softmax probability, exceeds a threshold (θ). This mechanism reduces latency for easier inputs while maintaining accuracy for harder ones. Early-exits provide



(a) Conventional DNN: always produces the results from the final output layer



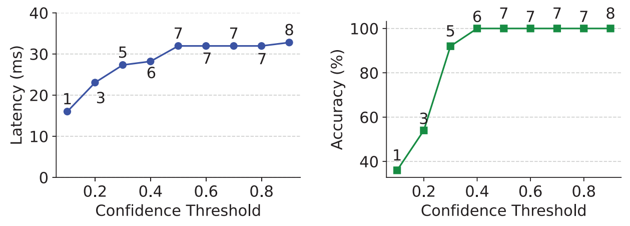
(b) Early-exit DNN: produces results via intermediate exit branches

Figure 2: Comparison of conventional and early-exit DNN architectures.

a broad latency-accuracy tradeoff controlled by θ, allowing seamless runtime adjustment without retraining or modifying the base network. Since the *confidence threshold* can be adjusted at runtime, it offers a broad adaptation range; we therefore choose it as another control knob.

Figure 2a illustrates the inference process of a conventional DNN, where each input is processed sequentially from the input layer to the output layer. Early-exit DNNs offer a solution by allowing some inputs to exit the model sooner, which speeds up the inference process while maintaining accuracy. These networks incorporate multiple side branches into the traditional DNN architecture. The inference can be halted at a side branch if it meets predefined *confidence* criteria. Figure 2b shows the inference process of early-exit DNNs, in which each side branch evaluates the prediction *confidence* for each input.

2.1.3 Batching. Batching is a widely studied approach that improves the throughput of DNN inference. Instead of processing input data individually, this method feeds the data to the DNN in batches. This approach is particularly effective for GPU-based DNN accelerators, as it takes advantage of the GPU's parallel architecture to significantly enhance throughput. Several state-of-the-art works implement batching for inference tasks to increase performance [52], [32], [2]. Batching improves overall inference throughput and amortizes per-request overhead, but it introduces trade-offs in latency. Let N be the number of requests in a batch, t_o the fixed GPU overhead (e.g., kernel launch), and t_c the compute time for a single input. Without batching, each request takes $L_{\text{no\_batch}} = t_o + t_c$, and total time for N requests is $T_{\text{no\_batch}} = N \cdot (t_o + t_c)$. With batching, the total execution



(a) Latency **(b) Accuracy**

Figure 3: (a) Latency increases with deeper exits. (b) Accuracy improves with increased latency. The numbers in the plot indicate the exits taken.

time becomes $T_{\text{batch}} = t_o + t_b(N)$, where $t_b(N) < N \cdot t_c$ is the compute time for the entire batch, and the amortized latency is $L_{\text{amortized}} = \frac{T_{\text{batch}}}{N}$. However, batching can increase end-to-end latency for individual requests due to queuing delays while waiting to fill a batch. Thus, while batching improves throughput and efficiency, it must be carefully tuned (e.g., with a max *batch size* or timeout) to avoid violating latency requirements in real-time systems.

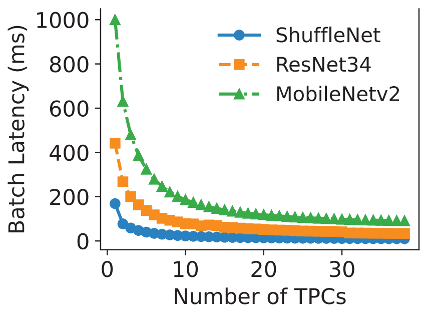
2.2 Motivation

In this subsection, we discuss the necessity of *spatial sharing* and how early-exits, *TPC* count, and batching have an impact on latency and accuracy of inference tasks.

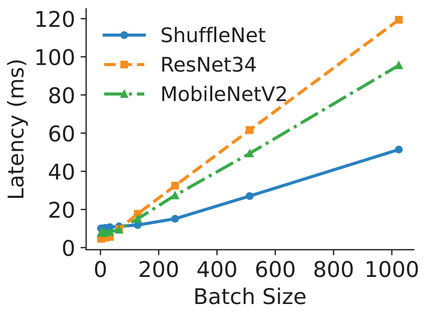
2.2.1 Impact of Confidence on Latency and Accuracy. As mentioned in Section 2.1.2, traditional DNNs require the execution of all layers to produce a result. In contrast, early-exit DNNs allow inference to stop at intermediate layers when a high-confidence prediction is reached. In this section, we show the impact of adjusting *confidence threshold* (θ) on latency and accuracy for ResNet34. We vary the threshold from 0.1 to 0.9 and observe two key trends: 1) As the *confidence threshold* increases, the latency also increases, as Figure 3a shows. 2) As the *confidence threshold* increases, the accuracy improves, as Figure 3b indicates. The model achieves its highest accuracy after certain thresholds, but its latency continues to rise slightly. For example, after threshold 0.3 for ResNet34, the model achieves over 90% accuracy while maintaining nearly half the latency compared to the final exit.

Observation 1: *Selecting an appropriate confidence threshold is crucial for effectively balancing latency and accuracy in early-exit DNNs. A well-chosen threshold determines at which exit the inference stops, enabling faster predictions without significantly compromising accuracy.*

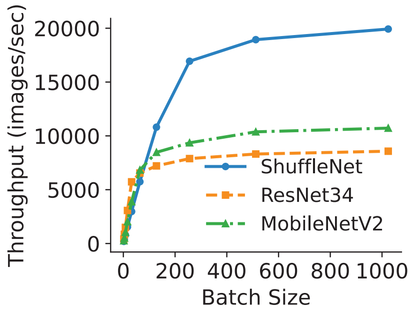
2.2.2 Impact of TPC assignment on latency. Here, we demonstrate the impact of *TPC* assignment on latency and explain



(a) TPC



(b) Latency



(c) Throughput

Figure 4: As the number of *TPCs* (a) assigned to a task increases, the latency decreases. As the *batch size* increases, (b) latency and (c) throughput also increase.

the necessity of a fine-grained *TPC* assignment, as mentioned in Section 2.1.1. To illustrate this, we select three DNN models: *ResNet34*, *MobileNetV2*, and *ShuffleNet*. We fix the number of images at 500 for all models, and each model is tasked with processing images of size 128 × 128. To limit the number of *TPCs* available, we use the `libsmctrl_set_mask` function and record the inference latencies. As Figure 4a shows, a higher number of *TPCs* assigned to a task leads to a reduction in the inference latency since workload can be better parallelized across different SMs. Nonetheless, we also observe that an increasingly larger number of *TPCs* assigned to each model inference does not lead to much performance improvement beyond a certain point. For example, the latency of ShuffleNet, ResNet34, and MobileNetV2 do not decrease much beyond 3, 8, and 15 *TPCs* assigned, respectively.

The previous experiment considered only a single task. When multiple tasks are executing [53], it is crucial to assign non-overlapping *TPCs* to each instance to avoid latency increases. To emphasize the necessity of fine-grained assignment of *TPCs*, we measure the latency of a simple convolution kernel using CUDA, repeating each experiment 30 times to report average latency values. First, we launch a single instance of the kernel and measure its inference latency. For this run, we manually configure the kernel to execute exclusively on physical *TPCs* 10 to 20. This setup yields a baseline latency of approximately 0.068 ms, representing a non-contentious execution.

Next, we evaluate a concurrent execution scenario by launching two tasks in parallel. We begin by determining the total number of available *TPCs* in the GPU using the `libsmctrl_get_tpc_info_cuda` and `libsmctrl_get_gpc_info` APIs. We then allocate the *TPCs* between the two streams: one stream uses *TPCs* 10–20, while the other uses *TPCs* 21–31. As indicated in Table 1, the latency slightly increases to 0.077 ms. This suggests that even though the compute units are divided between the two streams, the memory subsystem remains shared, leading to some contention. Finally, we test a configuration where both kernels are mapped to the same subset of *TPCs* (10 to 20), resulting in competition for compute resources. In this scenario, the latency significantly rises to 0.128 ms, demonstrating the impact of contention when multiple workloads compete for the same physical resources.

Table 1: Average kernel latency (30 runs) for *TPC* partitioning strategies.

Configuration	TPC Assignment	Latency (ms)
Single kernel instance	10–20	0.068
Two instances, non-overlapping	10–20 & 21–31	0.077
Two instances, overlapping	10–20	0.128

Observation 2: *It is important to properly assign TPC partition resources to minimize contention and reduce performance degradation.*

2.2.3 Impact of batching on latency. As mentioned in Section 2.1.3, batching occurs during model execution, where multiple inputs are processed together in a single forward pass. In this experiment, we investigate the impact of *batch sizes* on inference latency and throughput. For this experiment, we fix the number of *TPCs* at 42 for all three models: *ResNet34*, *MobileNetV2*, and *ShuffleNet*. We test each model using *batch sizes* of 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024. As Figure 4b shows, increasing the *batch size* from 2 to 8 leads to modest latency gains due to better utilization. However, the latency increases significantly as the *batch size* grows beyond this point. This is because larger batches increase the processing load per inference step, making it more demanding for the GPU to handle efficiently. Although batching increases latency, the system throughput also increases, as shown in Figure 4c as discussed in Section 2.1.3.

Observation 3: *While batching improves GPU throughput, it must be judiciously configured to avoid violating latency constraints.*

Batching increases system throughput by processing multiple tasks simultaneously, but also introduces additional latency for individual requests. Consequently, throughput maximization and latency minimization are inherently conflicting objectives. To quantify this trade-off, GSLICE [12] introduced a metric called GPU utilization efficacy, defined as the ratio of throughput to latency. However, this metric does not account for *online-accuracy*, which becomes particularly important in adaptive inference systems such as SEEB-GPU that employs early-exits. Early-exits can significantly reduce latency, but often at the cost of slightly reduced *online-accuracy*. If only latency and throughput are considered, such models could appear overly efficient despite degraded output quality. To capture both system-level performance and model quality, we propose *GPU efficacy (GE)* by including the *online inference accuracy* as follows

$$GE = \frac{\text{Throughput}}{\text{Latency}} \times \text{Accuracy} \tag{1}$$

Since SEEB-GPU employs early-exits, this metric ensures that it does not gain an unfair advantage in latency and throughput through low accuracy. We report the GPU efficacy of SEEB-GPU in our hardware experiments (Section 5).

Based on the above three observations, it is important to dynamically adjust the appropriate *batch sizes*, *confidence thresholds*, and *TPC* assignments to ensure that the SLAs are met while maximizing accuracy. *To the best of our knowledge, this is the first work to focus on this challenging joint problem.*

3 System Design

Our primary focus in designing SEEB-GPU as an edge model-serving platform is to enable flexible and adaptive execution of inference requests for various applications. We describe SEEB-GPU's key design components and provide an overview. Next, we outline the knobs in our design and go through the architecture of SEEB-GPU. Lastly, we conclude by discussing complexity analysis.

3.1 Overall Architecture

Figure 5 illustrates the overall architecture of SEEB-GPU. We consider a typical edge server scenario with multiple inference requests from various applications. Each application may submit an inference request for tasks such as image classification, object detection, or semantic segmentation. These requests can contain multiple images to be processed and may originate from diverse sources such as smartphones, edge drones, or CCTV cameras. For example, detection requests from a drone and a CCTV camera can be aggregated

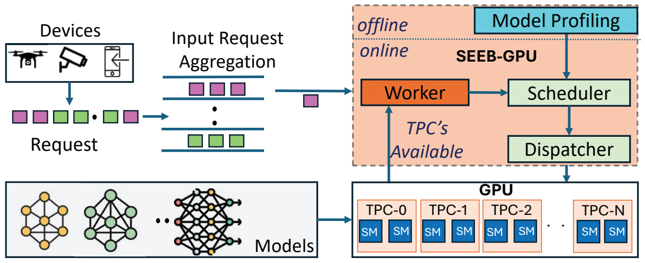


Figure 5: SEEB-GPU Architecture Overview. SEEB-GPU manages edge inference through three key components: Worker, Scheduler, and Dispatcher. The worker module assigns the *TPCs* with the libsmctrl library. The scheduler selects *batch sizes* and *confidence thresholds* to balance latency and accuracy under SLA constraints, while the dispatcher offloads the tasks to the GPU.

earlier in the system and forwarded to the *worker* for execution. The aggregation of inference requests is employed in frameworks like Triton [33] and TensorRT [25]. We make no particular assumptions on the arrival rate of requests, and design SEEB-GPU to activate upon availability of *TPCs*.

The main design contribution of our work is to ensure that the inference tasks meet their SLA requirements and incur negligible inference latency even when a burst of requests may arrive at any time, which is one of the main concerns for edge servers. The *worker* module, described in Algorithm 1 and shown in Figure 5, upon availability of GPU resources fetches the aggregated input requests and assigns each one a certain number of *TPCs*. At this point, it triggers the execution of the *scheduler*, which implements a deadline-aware policy inspired by EDF, tailored for sporadic inference workloads. For each group of requests, it leverages a latency model trained offline by the *model profiling* module to find the best configuration of batch size and confidence threshold for the assigned *TPCs*. Specifically, if a predicted configuration risks violating deadlines, the scheduler iteratively adjusts the *batch size* or *confidence* to find a feasible configuration. Among feasible options, it selects the one that has the lowest cost value, which aims at minimizing latency and maximizing confidence. Then, the *dispatcher* receives the inference request from the *scheduler* and dispatches it through the *driver* module. In the subsequent subsections, we describe the knobs for our design and the details of each component.

3.2 Control Knobs

In this section, we describe the control knobs used in our solution and justify their selection. Specifically, we leverage three control knobs: *TPC masks*, *confidence thresholds*, and *batch size*.

We use *TPC masks* to spatially partition the GPU among concurrent tasks, as introduced in Section 2.1.1. All tasks are launched from a single process running on the edge server (e.g., listening to requests from mobile devices) and the 64-bit *TPC* mask is configured to assign non-overlapping sets of *TPCs* to each aggregated request. As discussed in Section 2.2.2, failing to isolate *TPCs* can lead to contention and increased inference latency. By carefully managing the *TPC* mask, we ensure that different tasks run independently across separate SMs, reducing interference and maintaining predictable performance.

We adopt the *confidence threshold* (θ) as the second control knob, exploiting the dynamic nature of early-exit DNNs (Section 2.2.1). This threshold determines whether each inference should terminate at an intermediate exit or proceed to deeper layers. A lower θ allows the model to exit earlier with faster response times but may reduce accuracy; conversely, a higher θ enforces stricter *confidence*, improving prediction accuracy at the cost of increased latency. Because θ can be tuned at runtime without modifying the network architecture, it provides a powerful mechanism for balancing latency and accuracy based on runtime conditions.

Our third control knob is *batch size*, which directly affects throughput and responsiveness. As shown in Section 2.2.3, increasing the *batch size* improves GPU utilization and throughput but can also increase latency, potentially violating application SLAs (Service Level Agreements). We use *batch size* alongside *confidence threshold* to ensure that the system meets its latency requirements while maximizing performance. Further details on this integration are presented in Section 3.3.

Algorithm 1: *TPC* allocation and mask generation

```
Input: Upon availability of TPCs: Set $\mathcal{B}$ of
       aggregated inference requests $\mathcal{B}_j$ for each
       application $j$, each one with image count $I_j$
       sorted from earliest to latest arrival time; total
       number of TPCs available $A$; index of the first
       available TPC $idx$
1  $T \leftarrow \sum_{j=1}^{|\mathcal{B}|} I_j$;
2  used $\leftarrow 0$;
3  $tpc\_masks_{|\mathcal{B}|} \leftarrow [0]$;
4  $tpcs_{|\mathcal{B}|} \leftarrow [0,0]$;
5  for $j = 1$ to $|\mathcal{B}|$ do
6      $T_j \leftarrow \max\left(1, \left\lfloor \frac{I_j}{T} A \right\rfloor\right)$;
7      $T_j \leftarrow \min(T_j,\ A - used)$;
8      $t^s \leftarrow idx$;
9      $t^e \leftarrow idx + T_j - 1$;
10     $idx \leftarrow idx + T_j$;
11     $used \leftarrow used + T_j$;
12     $tpcs_j \leftarrow [t^s,\ t^e]$;
13     $tpc\_mask \leftarrow 0$;
14     for $i = 0$ to $A - 1$ do
15         if $i < t^s \vee i > t^e$ then
16             $tpc\_mask \leftarrow tpc\_mask \vee (1 \ll i)$;
17     $tpc\_masks_j \leftarrow tpc\_mask$;
18     if $used \geq A$ then
19         break;
20 Scheduler ($\mathcal{B}$, $tpc\_masks$, $tpcs$);
```

3.3 Design of SEEB-GPU

In this subsection, we describe the detailed design of each component of `SEEB-GPU`.

3.3.1 Worker. Algorithm 1 shows the pseudo-code of the worker module. It is triggered when contiguous *TPCs* become available at each application's inference task and its input is the set $\mathcal{B}$ of aggregated image sets $\mathcal{B}_j$, one per application j. Each image set $\mathcal{B}_j$ includes I_j images sorted by arrival time, from earliest to latest.

The worker first calculates the total number of images $T = \sum_j I_j$ (Line 1). The worker then iterates over each application's image set $\mathcal{B}_j$ Line 5 and computes a *TPC* count proportional to the size of the image set relative to T (Line 6). The allocation is capped to ensure it does not exceed the total available *TPCs* (Line 7). For each application, the assigned *TPC* range $[t^s, t^e]$ is computed (Lines 8–9), and the index tracker idx and the counters are updated (Lines 10–11).

For example, if the total available *TPCs* ($A = 42$) and the aggregated set $\mathcal{B} = \{B_1, B_2\}$ where $\mathcal{B}_1$ contains 10 images and $\mathcal{B}_2$ contains 20 images. The total workload is $T = 30$, resulting in allocations of $\frac{10}{30} \times 42 = 14$ *TPCs* for $\mathcal{B}_1$ and 28 *TPCs* for $\mathcal{B}_2$. The *TPC* index assignments would thus be as follows: $\mathcal{B}_1$ is assigned *TPCs* 0 to 13 and $\mathcal{B}_2$ is assigned 14 to 41.

After assigning *TPC* ranges, the worker generates a corresponding 64-bit mask (Lines 13–17) for each application to enforce spatial partitioning. The mask disables all *TPCs* outside the assigned range by setting their corresponding bits to 1, leaving the allocated *TPC* bits to 0 (enabled). For example, if an application is assigned *TPCs* [0–2], the resulting mask is `0xFFFFFFF8`, disabling all but the first three *TPCs*. Similarly, for a range [14–41], the computed mask disables *TPCs* outside that interval, producing a value such as `0xFFFFC03FFF`. These masks are applied using the `libsmctrl_set_stream_mask` function, which sets the active mask for all kernel

Algorithm 2: Scheduler

```
Input: Aggregated inference request ($\mathcal{B}$), tpc_masks, tpcs
1  for $j = 1$ to $|\mathcal{B}|$ do
2      $k \leftarrow \lfloor \log_2(I_j) \rfloor$;
3      $S \leftarrow \{2^i \mid i = k, k-1, \ldots, 0\}$;
4      $C^* \leftarrow \infty$;
5      $B^* \leftarrow \emptyset$;
6      $\theta^* \leftarrow \emptyset$;
7      Initialize_Weights $(\lambda_1, \lambda_2)$;
8      $\alpha, \beta_1, \beta_2, \beta_3 \leftarrow$ get_model_parameters$(j)$;
9      $d_{min} \leftarrow$ minimum deadline for batch $B_j$;
10     $\tau \leftarrow$ safety slack on deadline $d_{min}$;
11     for $s \in S$ do
12         for $\theta \in \Theta$ do
13             $L \leftarrow \alpha + \beta_1 \cdot s + \beta_2 \cdot \theta + \frac{\beta_3}{|tpcs[j]|}$;
14             if $L \leq d_{\min} - \tau$ then
15                 $C \leftarrow \lambda_1 \cdot L - \lambda_2 \cdot \theta$;
16                 if $C < C^*$ then
17                     $C^* \leftarrow C$;
18                     $B^* \leftarrow s$;
19                     $\theta^* \leftarrow \theta$;
20     if $B^* \neq \emptyset$ then
21         dispatcher $\leftarrow \langle \mathcal{B}_j,\ B^*,\ \theta^*, tpc\_masks[j] \rangle$;
22     else
23         dispatcher $\leftarrow \langle \mathcal{B}_j,\ B^*,\ \theta_{\min}, tpc\_masks[j] \rangle$;
```

launches on a given CUDA stream. This allows each application to execute on an isolated set of *TPCs*, even when kernels are issued from a single host process. Although `libsmctrl` is currently available only for NVIDIA GPUs (CUDA 8.0 through 12.1), similar compute unit (CU) masking primitives are supported by AMD GPUs [8], enabling our design to generalize across vendors. *TPC* assignments are recomputed every 100 ms. The choice of 100 ms is motivated by a combination of trace-driven analysis and empirical evaluation; shorter intervals introduced excessive recomputation overhead without substantial performance benefits. In comparison, longer intervals led to delayed responsiveness to workload fluctuations. Upon completion, the *worker* module triggers the *scheduler* routine described in Algorithm 2.

3.3.2 *Scheduler.* As discussed in Section 3.2, the three control knobs used in our system are the *confidence threshold* (θ), the *batch size* (B), and the *TPC mask*. Here the *TPC mask* is already used in the *worker* module. In the *scheduler* module, we utilize two control knobs at the application level: the *confidence threshold* (θ) and the *batch size* (B). The scheduler is invoked once the worker generates per-application *TPC* masks. Algorithm 2 shows its pseudocode.

The scheduling process begins by iterating over each application's image group $\mathcal{B}_j \in \mathcal{B}$ (Line 1). For each group, the scheduler first computes $k = \lfloor \log_2(I_j) \rfloor$ and constructs the candidate *batch size* set $S = \{2^i \mid i = k, k-1, \ldots, 0\}$ (Line 3). This ensures that *batch sizes* are powers of two, which aligns well with the organization of GPU hardware. Since SMs are grouped into *TPCs*, GPUs perform more efficiently when workloads are sized in powers of two [23].

For each candidate *batch size*, the scheduler evaluates combinations of *batch size* and *confidence threshold* θ to minimize a cost function. Before iterating over these candidates, it initializes the cost $C^* \leftarrow \infty$, the best *batch size* $B^* \leftarrow \emptyset$, and the best threshold $\theta^* \leftarrow \emptyset$ (Lines 4–6). It then retrieves model-specific parameters $(\alpha, \beta_1, \beta_2, \beta_3)$ and weight coefficients (λ_1, λ_2) (Lines 7-8). The model-specific parameters are offline-trained coefficients obtained from latency profiling across *batch sizes*, *thresholds*, and *TPC* masks, as described in Sections 2.2.1, 2.2.2, and 2.2.3. For each application, the scheduler then computes the tightest deadline $d_{\min}$ across all requests in $\mathcal{B}_j$. That is, if $\mathcal{B}_j = \{r_1, r_2, \ldots, r_n\}$ denotes the set of inference requests for application j, each with its own deadline d_i, then $d_{\min} = \min\{d_i \mid r_i \in \mathcal{B}_j\}$ represents the earliest deadline among them (Line 9). The safety slack is defined (Line 10).

The *scheduler* iterates over all combinations of *batch size* $s \in S$ (Line 11) and $\theta \in \Theta$ (Line 12). For each combination, it estimates the inference latency using the profiling-based model: $L = \alpha + \beta_1 \cdot s + \beta_2 \cdot \theta + \frac{\beta_3}{|tpcs[j]|}$ (Line 13). The *scheduler* checks whether the estimated latency L satisfies the deadline constraint $L \leq d_{\min} - \tau$ (Line 14). By subtracting τ from the minimum deadline $d_{\min}$, the scheduler enforces a stricter condition $L \leq d_{\min} - \tau$, which accounts for possible inaccuracies in the offline latency model. This slack guarantees that selected *batch sizes* and *confidence thresholds* remain safe under real-time execution conditions, despite relying on static latency predictions.

Among all valid candidates, the scheduler selects the configuration that minimizes the cost function $C = \lambda_1 \cdot L - \lambda_2 \cdot \theta$ (Line 15), which balances latency and accuracy preferences. If a better configuration is found, it updates the current best (Lines 17–19). At the end of the search, if a valid configuration exists, the scheduler dispatches the group using $(B^*, \theta^*, tpc\_masks[j])$ (Line 21). Otherwise, it falls back to the minimum threshold $\theta_{\min}$ to ensure no SLA violations (Line 23). The *TPC* allocation $tpc\_masks[j]$ is provided by Algorithm 1.

3.3.3 *Dispatcher.* The dispatcher is responsible for the launch of inference tasks and managing their execution. When a task is scheduled, the dispatcher allocates the required *TPCs* and runs the inference in a separate thread. It monitors the task's execution status and, upon successful completion, releases the allocated *TPCs*. This ensures that GPU resources are efficiently recycled and promptly made available for future tasks. By operating concurrently, the dispatcher enables parallel inference execution while maintaining fine-grained control over GPU resource usage.

3.4 Complexity Analysis

The overhead in SEEB-GPU arises from two main components. First, the generation and assignment of *TPC* masks involves iterating over the set of available *TPCs*, with a complexity of $O(A)$, where A is the number of available *TPCs*. Since A is typically much smaller than the total number of inference requests across applications ($A \ll N$), this step remains lightweight. Second, the latency-aware scheduler explores all power-of-two *batch sizes* ($\log I_j$) and *confidence thresholds* ($|\Theta|$) for each application group $\mathcal{B}_j$, resulting in a per-group scheduling complexity of $O(|\Theta| \cdot \log I_j)$. Here, I_j denotes the number of images for application j, and $|\Theta|$ is typically small.

Overall, the computational overhead introduced by SEEB-GPU is minimal. Since both mask generation and scheduling are triggered on demand, only when *TPCs* become available rather than at fixed intervals, the system avoids unnecessary computation. All core components operate in constant or logarithmic time, making SEEB-GPU efficient and scalable for real-time, multi-tenant GPU scheduling.

3.5 Discussion

Portability. An important feature of our solution is that it is easily portable to other vendor GPUs. We can port our solution to AMD GPUs using `hipExtStreamCreateWithCUMask` [37][42] and perform CU masking for CU control, equivalent to *TPC* control. For *TPC* assignment, we specify the number of *TPCs*, while we can directly assign the number of CUs here when assigning with a mask.

Periodicity. Our current scheduling scheme is designed for sporadic tasks, but it can be adapted to accommodate periodic workloads that have either implicit or explicit deadlines. For periodic tasks, we know the arrival times (or periods) in advance, which enables us to pre-allocate specific *TPCs* and ensure timely access to GPU resources.

Co-Scheduling. At present, our scheduling process assumes that tasks arriving at the edge server are assigned solely to the GPU, and we focus on scheduling those tasks accordingly. Our next step is to expand this approach to include CPU+GPU scheduling, where we will schedule tasks for both the CPU and GPU, effectively assigning tasks to each based on their requirements.

Power Consumption. Power optimization is not included in the current work. Since SEEB-GPU introduces only a minimal runtime overhead (as noted in Section 5), its energy consumption should be similar to that of the baseline models. Future work could explore further savings by implementing dynamic voltage and frequency scaling (DVFS) [30].

Scalability. We believe that SEEB-GPU has the potential to scale beyond a single edge device and support clusters of various sizes. However, fluctuations in network bandwidth may lead to increased communication latency. To address this issue and enhance the scalability of SEEB-GPU, we plan to design a bandwidth adaptation policy.

4 Experimental Setup

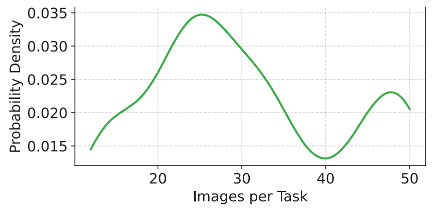
Hardware. We test SEEB-GPU on an RTX 3090 GPU, which uses the Ampere architecture, has 82 Streaming Multiprocessors (SMs) which are equivalent to 41 *TPCs* since SMs are arranged in groups of two per *TPC* [4]. We set the operating frequency of the GPU to 1395 MHz. As mentioned in Section 2.1, we use the `libsmctrl` library to implement spatial sharing. SEEB-GPU integrates with PyTorch by dynamically loading the `libsmctrl` shared library via `ctypes`. At runtime, this allows direct control of GPU SM masks and scheduling behavior from Python without modifying the PyTorch runtime.

Datasets and Early-Exit Training. We individually train all DNNs used in our evaluation on their corresponding datasets, summarized in Table 2. These datasets include Adience [15], Animals [9], GTSRB [24], and FER [19]. To train the models, we fine-tune ImageNet-pretrained backbones using standard pipelines. The images are resized to 224×224, and we apply Adam or SGD optimization with stepwise learning rate schedules. In all models, we attach multiple *early-exit classifiers* that allow inference to terminate at intermediate depths. During training, predictions from all exits (including the final classifier) are supervised, and we optimize the average (or weighted average) of their cross-entropy losses. For example, in the *MobileNetV2* model on GTSRB, we insert exits after selected inverted residual blocks at layers $\{1, 2, 3, 5, 7, 9, 11, 13, 15, 17\}$. Each exit consists of a small convolutional layer, global average pooling, and two fully connected layers to produce logits. The training of early-exit models is analogous to this work, and further details on the training of early-exit models can be found in [22, 45].

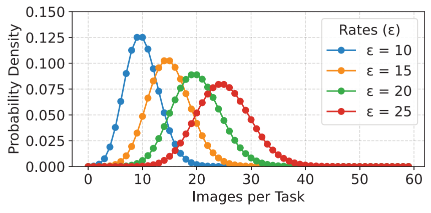
Baselines. We compare SEEB-GPU with several state-of-the-art solutions summarized below. 1) **Batch** is part of the Triton Inference Server [41], which combines request data into batches of a fixed, empirically chosen size. We set the batch size as small as possible to minimize job latency. 2) **DeepRT** uses time-slicing and adaptive batching to manage soft real-time inference requests. It introduces the DisBatcher

Table 2: Details of the DNN workloads used for evaluation.

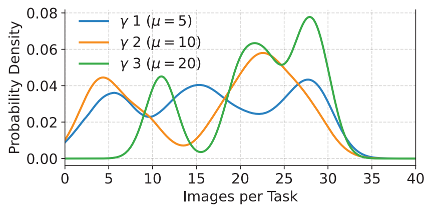
Application	Task Type	Dataset	DNN Model
General vision	Animal classification	Animals [9]	ResNet50
Age-aware robotics	Age classification	Adience [15]	ShuffleNet
Autonomous driving	Traffic sign classification	GTSRB [24]	MobileNetV2
Human-Computer Interaction	Emotion recognition	FER [19]	ResNet34



(a) PAI trace [49]



(b) Poisson



(c) Gamma

Figure 6: Images per-task distribution for traces used in our evaluation. The real-world PAI trace [49] provides empirical workload patterns. The *poisson trace* emulates random independent task arrivals and *gamma trace* captures bursty and variable inter-arrival times.

mechanism, which packs as many requests as possible into a batch and applies the Earliest Deadline First (EDF) policy for prioritization [50]. 3) **GSLICE** is a spatial-temporal scheduling method for NVIDIA GPUs. It leverages Multi-Process Services (MPS) to partition CUDA cores adaptively, prioritizes models with high request rates, and applies a greedy rescheduling strategy to maintain efficient partition sizes [12]. 4) **BranchyNet** is a dynamic early-exit framework that places auxiliary classifiers at intermediate layers of DNNs. During inference, a sample exits once a branch reaches a confidence threshold, reducing average latency while preserving accuracy [45].

Traces. We use traces to simulate request arrivals in a repeatable, controllable way. Traces enable us to vary load, burstiness, and variance without relying on live traffic, allowing us to stress-test SEEB-GPU and compare policies under identical conditions. We evaluate one real (*PAI*) and two synthetic (*Poisson and Gamma*) arrival patterns. 1) **PAI trace:** The PAI trace from Alibaba [49] captures a mix of ML training and inference tasks on a cluster with over 6,500 GPUs during July–August 2020. Each task records a unique ID, start time, duration, requested GPU memory, requested GPU utilization, and status. We preprocess the trace by (i) excluding failed tasks, (ii) removing tasks with zero GPU memory or GPU utilization, and (iii) filtering out tasks with execution times greater than 20 seconds, as these typically correspond to training workloads [5, 43]. For the remaining tasks, we apply the same pre-processing steps as those used in the tweet trace in DeepRT [50] to build a 30-second burst workload. The probability distribution of the PAI trace is shown in Figure 6a. 2) **Poisson trace:** We generate the trace using a Poisson distribution with different arrival rates ($\epsilon = 10, 15, 20, 25$) to simulate the arrival of inference tasks over time, as shown in Figure 6b. Each trace records the arrival time, application type, deadline, and number of images for each task. Smaller ϵ values produce infrequent, well-separated bursts, while larger ϵ values generate rapid bursts within shorter intervals. 3) **Gamma trace:** This trace models task arrivals based on a gamma distribution for inter-arrival times. For each configuration, we define the mean and variance of the inter-arrival time, which determine how tasks are spaced. We use three configurations, as shown in Figure 6c, adjusting the shape and scale parameters to control burstiness and timing. Each trace records the arrival time, deadline, number of images per task, and application type.

Metrics. We report the following metrics: *queuing time*, *inference latency*, *response time*, *online accuracy*, *throughput*, *SLA violations*, and *GPU efficacy*. *Queuing time* is the interval from request release until it is dispatched to the GPU. *Inference latency* denotes the GPU execution time of a task and does not include model loading or initialization time. *Response time* is the end-to-end time from release to completion, i.e., *queuing time* + *inference latency*. *Online accuracy* is the fraction of requests correctly classified during the run. *Throughput* is the number of requests completed per second. An SLA violation occurs when the response time of a request exceeds its deadline. We set each deadline to 120% of its expected execution time, based on offline profiling. Each

Table 3: Comparison of latency, throughput, accuracy, GPU efficacy, and SLA violations for the PAI trace.

Metric	Batch	BranchyNet	DeepRT	GSLICE	SEEB-GPU
Latency (s)	0.152 ± 0.014	0.084 ± 0.001	0.738 ± 0.288	0.343 ± 0.008	**0.047 ± 0.002**
Throughput (tasks/s)	1.061 ± 0.000	1.097 ± 0.000	1.096 ± 0.002	1.097 ± 0.000	**1.094 ± 0.000**
Accuracy (%)	**96.260 ± 0.000**	76.622 ± 0.000	**96.285 ± 0.000**	**96.285 ± 0**	83.528 ± 0.476
GPU Efficacy	6.737 ± 0.585	10.002 ± 0.179	1.643 ± 0.819	2.945 ± 0.048	**19.577 ± 0.543**
SLA Violation (%)	4.444 ± 1.925	**0.000 ± 0.000**	**0.000 ± 0.000**	**0.000 ± 0.000**	**0.000 ± 0.000**

experiment is run three times, and the reported results represent the mean of the three runs. Standard deviation error bars are included in all bar graphs to visualize the variance in each run.

5 Results

In this section, we first compare SEEB-GPU using a real-world inference trace in Section 5.1. We then evaluate its performance on two synthetic traces, described in Section 5.2. In Section 5.3, we present a sensitivity analysis by varying the weights used in Algorithm 2. Section 5.4 presents an ablation study to examine the impact of different control knobs on overall performance. Finally, Section 5.5 provides a detailed analysis of the overhead.

5.1 Evaluation of Real-world Traces

In this experiment, we evaluate SEEB-GPU using real-world traces. Since there are no publicly available traces for inference tasks on edge servers, we utilize an inference trace from Alibaba Cloud to test SEEB-GPU. The details of trace preprocessing are already mentioned in Section 4. Table 3 reports latency, throughput, accuracy, GPU efficacy, and SLA violations of all solutions for the PAI trace. SEEB-GPU achieves the lowest latency of 0.047 s, making it approximately 3.2× faster than Batch, 1.8× faster than BranchyNet, 7.3× faster than GSLICE, and over 15× faster than DeepRT. SEEB-GPU maintains zero SLA violations, demonstrating its ability to sustain rapid response times while meeting service-level agreement (SLA) guarantees. The throughput remains comparable to that of other approaches, indicating that SEEB-GPU efficiently processes tasks without introducing bottlenecks. The accuracy for SEEB-GPU is lower than that of the full-depth baselines (Batch, DeepRT, and GSLICE) because it employs early-exits to meet tight deadlines; however, it remains substantially higher than BranchyNet, which also uses early-exits but lacks adaptive control.

Additionally, SEEB-GPU demonstrates a superior balance between responsiveness and reliability. The high GPU efficacy highlights SEEB-GPU's ability to maximize the useful computational output per unit time. Specifically, SEEB-GPU achieves a GPU efficacy of 19.58, which is approximately 3× higher than Batch, 6× higher than GSLICE, and almost 12× higher than DeepRT. By dynamically adjusting confidence thresholds and batch sizes, SEEB-GPU minimizes latency while sustaining high accuracy and throughput. Static schedulers, such as DeepRT or GSLICE, fail to adapt to workload variability, resulting in either longer delays or wasted GPU cycles.

Table 4: SLA violation rates for the *poisson trace*.

Technique	SLA Violation (%)
Batch	50.88
BranchyNet	0.00
DeepRT	45.64
GSLICE	60.23
SEEB-GPU	0.00

5.2 Synthetic Trace Evaluation

In this section, we conduct experiments using two synthetic traces, specifically *poisson trace* and *gamma trace*, as mentioned in Section 4.

5.2.1 Poisson Trace. Figure 7 illustrates the comparative performance of all baselines under the *poisson trace*. SEEB-GPU consistently achieves the lowest inference latency, as shown in Figure 7a. BranchyNet exhibits low inference latency, as it employs *early-exits*. For *Batch*, the optimal batch size is configured at runtime, ensuring that the latency remains low. As summarized in Table 4, SEEB-GPU attains the smallest SLA violation rate of 0.0%, outperforming all other baselines. This improvement comes from the adaptive confidence control mechanism of SEEB-GPU, which dynamically tunes the confidence threshold to ensure that requests meet their respective deadlines. For the *poisson Trace*, Batch, DeepRT and GSLICE achieve a higher accuracy compared to SEEB-GPU as shown in Figure 7b, as it does not take *early-exits*. SEEB-GPU demonstrates the highest GPU efficacy among all methods as shown in Figure 7c.

5.2.2 Gamma Trace. The *gamma trace* models workloads with bursty and non-uniform inter-arrival times, reflecting realistic variations in request arrivals. Figure 8a shows that SEEB-GPU achieves the lowest latency across all *gamma* configurations, indicating a strong adaptability to dynamic input rates. Since SEEB-GPU uses both early-exits and batching, we observe decreased latency with non-overlapping resource partitioning. All solutions obtain 0% SLA violations

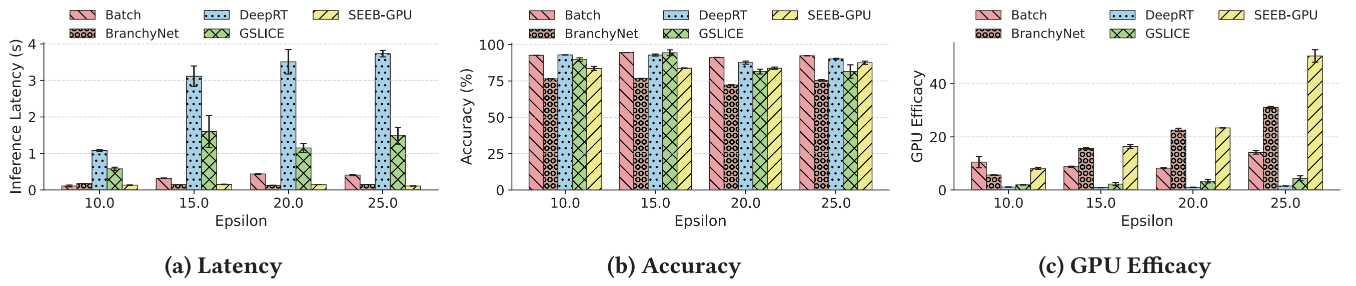


Figure 7: Evaluation of the poisson trace for different ϵ values. (a) Our proposed SEEB-GPU achieves the lowest latency. (b) Variation in accuracy for all the solutions. (c) SEEB-GPU shows the highest GPU efficacy, indicating efficient GPU usage.

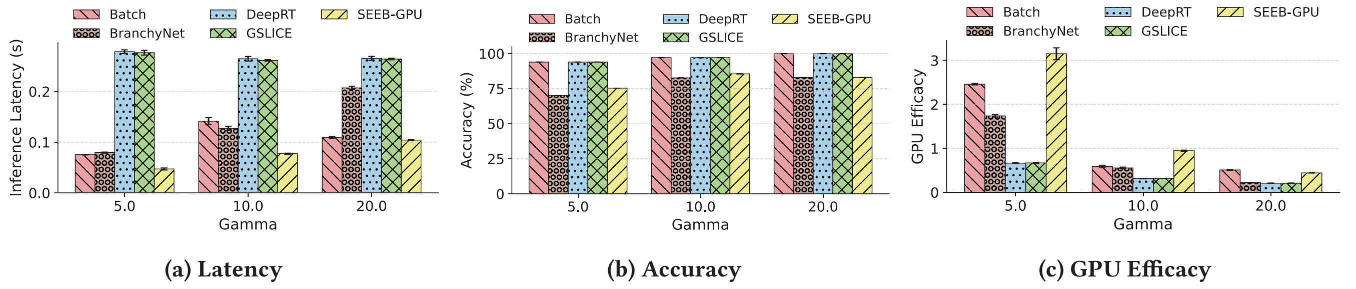


Figure 8: Evaluation for the gamma trace for different γ values. (a) Our proposed SEEB-GPU achieves the lowest latency. (b) Variation in accuracy for all the solutions. (c) SEEB-GPU shows the highest GPU Efficacy, indicating efficient GPU usage.

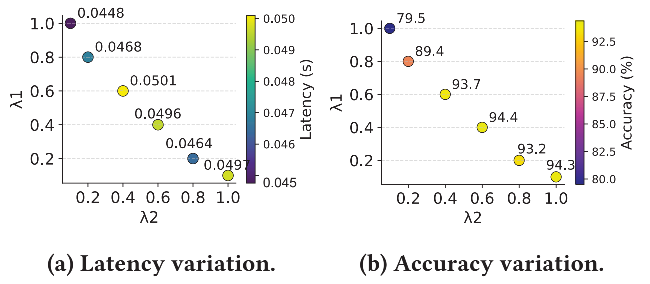


Figure 9: Sensitivity analysis of SEEB-GPU under different weighting factors λ_1 and λ_2 in Algorithm 2.

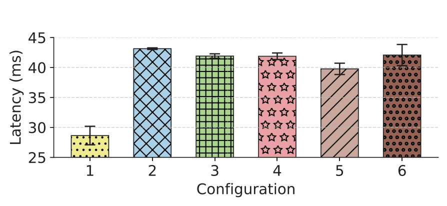
for the gamma trace. For the *gamma Trace* Batch, DeepRT and GSLICE achieve higher accuracy compared to SEEB-GPU as shown in Figure 8b, as it does not take early-exits. For the *gamma Trace*, SEEB-GPU achieves the highest GPU efficacy at all points as shown in Figure 8c, indicating that it makes the most efficient use of the GPU by maintaining high throughput and low latency simultaneously.

5.3 Sensitivity Analysis

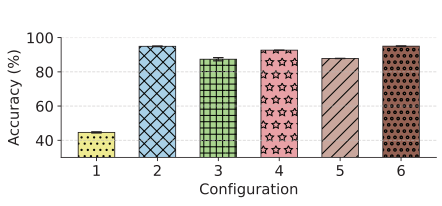
In this experiment, we perform a sensitivity analysis to study how the weighting factors λ_1 and λ_2 in the SEEB-GPU cost function affect the overall performance. The experiments were conducted using the real-world PAI trace to capture realistic request arrival patterns and workload variability. Algorithm 2 uses these parameters to dynamically balance latency and accuracy during scheduling decisions. When λ_1 increases, the algorithm places more emphasis on minimizing latency. As a result, SEEB-GPU selects earlier exits more frequently, which reduces the average latency to approximately 0.045 s. As shown in Figure 9a, increasing λ_1 prioritizes latency reduction, leading to smaller inference delays. In contrast, increasing λ_2 emphasizes accuracy, as seen in Figure 9b, improving prediction precision while slightly increasing latency. Conversely, when λ_2 increases, SEEB-GPU prioritizes accuracy by enabling deeper model exits, achieving up to 94% accuracy but with a small increase in latency (approximately 0.050 s). This analysis provides a clear trade-off between latency and accuracy, demonstrating that SEEB-GPU can adaptively balance both by tuning λ_1 and λ_2.

5.4 Ablation Study

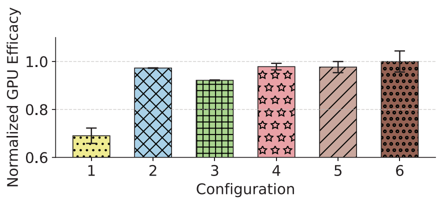
In this experiment, we vary two *application-level* control knobs, *confidence threshold* and *batch size*, and one system-level knob *TPC-masks* to evaluate their individual and combined effects on inference latency, accuracy, and GPU efficacy.



(a) Latency



(b) Accuracy



(c) Normalized GPU Efficacy

Figure 10: Evaluation of the ablation Study for the different configurations mentioned in Table 5. SEEB-GPU (*configuration 6)* chooses the *confidence threshold* and *batch sizes,* thereby reducing the latency without compromising accuracy.

Table 5: Experimental configurations for the Ablation Study.

Config	Confidence Threshold	Batch Size	TPC Assignment
1	0.3 (Fixed)	Tuned	Disabled
2	0.9 (Fixed)	Tuned	Disabled
3	Tuned	2 (Fixed)	Disabled
4	Tuned	128 (Fixed)	Disabled
5	Tuned	Tuned	Disabled
6	Tuned	Tuned	Enabled

Table 5 summarizes the six configurations. All experiments were conducted on the real-world PAI trace described in Section 4, and the workloads listed in Table 2 are used for this experiment.

In *Configuration 1*, the *confidence threshold* is fixed at 0.3, while the *batch size* is selected dynamically using Algorithm 2, which determines the largest permissible value without violating SLA constraints. *Configuration 2* increases the *confidence threshold* to 0.9, with the *batch size* again selected by Algorithm 2. In *Configuration 3*, the *batch size* is fixed at 2 to enable minimal parallelism, while the *confidence threshold* is tuned by Algorithm 2 to maintain SLA compliance. *Configuration 4* sets the *batch size* to 128 and uses a dynamically tuned *confidence threshold. Configuration 5* jointly tunes both application-level control knobs. For Configurations 1–5, *TPC* assignment is disabled, and all tasks are executed in individual streams with access to all SMs. Finally, *Configuration 6* (SEEB-GPU) enables *TPC* assignment using Algorithm 1 and jointly tunes both the *confidence threshold* and *batch size* using Algorithm 2.

Figure 10a–10c show the latency, accuracy, and GPU efficacy for all ablation configurations. In *Configuration 1*, latency is the smallest because the confidence threshold is fixed at 0.3, causing SEEB-GPU to take the earliest exit for most tasks. However, this aggressive early-exit policy results in low accuracy (around 40%), as shown in Figure 10b. In *Configuration 2*, the confidence threshold is increased to 0.9, forcing tasks to pass through deeper exits, which improves accuracy to above 90% but increases latency to about 43 ms. *Configuration 3* uses the smallest batch size 2, so tasks are dispatched almost immediately without waiting for additional requests, resulting in moderate latency and stable accuracy. In contrast, *Configuration 4* uses the largest batch size 128, which increases waiting time and thus leads to the larger latency compared to *Configuration 3. Configuration 5* jointly tunes both confidence and batch size, achieving a balance between latency and accuracy without *TPC* assignment. Finally, *Configuration 6* (SEEB-GPU) enables the *TPC* partitioning and adaptive tuning of both parameters, achieving the highest GPU efficacy, as shown in Figure 10c. These results confirm that joint optimization with *TPC* assignment allows SEEB-GPU to operate efficiently while maintaining SLA compliance.

5.5 Overhead Analysis

This section examines the overhead associated with all scheduling solutions. For SEEB-GPU, the overhead primarily arises from computing the *confidence threshold* and selecting the appropriate *batch size*. Specifically, we measure the overhead for all solutions as the time required to compute their respective scheduling logic. In the overhead analysis, we focus on the execution time of the algorithms because all baselines have the same 100 ms time to collect requests, at which point the algorithm makes a decision, which is what we report. For GSLICE, we do not include GPU wait time in the overhead measurement. This is because the wait time arises from resource unavailability (i.e., lack of free GPU capacity), not from the scheduling or algorithmic logic itself. To measure the scheduling overhead, we record the execution time of the scheduling algorithm for each experiment and calculate the average runtime across all runs. In Figure 11, SEEB-GPU shows a slightly higher scheduling overhead due to its extra optimization logic. However, this overhead is only about 1 ms, which is negligible compared to the average inference latency of 45 ms observed across all baselines. Therefore, the

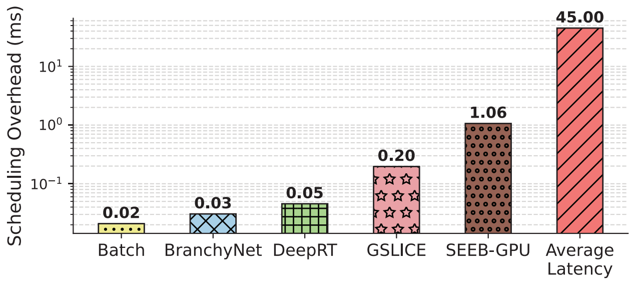


Figure 11: Scheduling overhead for all the solutions.

runtime overhead of SEEB-GPU is relatively small compared to the total execution time.

6 Related Work

Extensive research in edge computing focuses on improving latency, scheduling efficiency, and throughput. We review the most relevant prior work, divided into two main categories: model-level optimization and deep learning inference at the edge.

Model-level Optimization. Several works propose methods to reduce inference latency through model-level optimization [46][3]. Techniques such as weight pruning [54], channel pruning [29], quantization [38], compression [40], [31] and layer decomposition [18], [11] help reduce the complexity of DNNs. All these solutions aim to reduce redundant weights in convolutional neural networks (CNNs) since over-parameterization leads to unnecessary correlations. Although these methods can speed up machine learning inference, they alter the CNN model structure offline, which prevents them from dynamically balancing latency and accuracy at runtime. Additionally, these approaches primarily focus on model-specific optimization and overlook system-level improvements.

Deep Learning Inference on GPUs. A variety of industrial and academic efforts develop systems for DNN inference on both cloud and edge GPUs. TensorRT [25] and Triton [33] are general-purpose inference servers used in production, but they do not provide SLA guarantees. Clipper [10] targets throughput and latency using a modular architecture. Swayam [21] and GPUColo [5] use controllers to allocate resources dynamically in environments where inference and training share GPUs, but they do not perform model-specific optimization. DeepRT [50] introduces adaptive batching for edge GPUs. Pantheon [22] proposes fine-grained preemption for early-exit DNNs. Dělen [28] applies early exits to reduce latency on GPUs.

These solutions focus primarily on model optimization, SLA control, or GPU resource control. SEEB-GPU addresses all three areas. Our solution combines model-specific optimization with proportional GPU resource assignment, ensuring fewer SLA violations as shown in Table 6.

Table 6: Comparison with Prior Work

Service Framework	Model Optimization[†]	GPU Resource Control	SLA Control
TF-Serving [36]	✓	✗	✗
Triton [33]	✓	✓	✗
Clipper [10]	✓	✗	✓
GPUColo [5]	✗	✓	✓
DeepRT [50]	✓	✗	✓
Dělen [28]	✓	✗	✗
BCEdge [53]	✓	✗	✓
GSLICE [12]	✗	✓	✓
BranchyNet [45]	✓	✗	✗
SEEB-GPU (Ours)	✓	✓	✓

[†]Optimization includes any of the following: pruning, quantization, early exits, and batching techniques.

7 Conclusion and Future Work

SEEB-GPU provides a comprehensive and efficient solution for deep learning inference on edge GPUs, addressing model optimization, SLA control, and GPU resource management simultaneously. It introduces a pre-batching mechanism that groups inference requests, allowing for proportional allocation of GPU resources. Additionally, SEEB-GPU utilizes fine-grained *TPC* control to minimize contention among processes. The system employs *confidence*-based early-exits, dynamically adjusted through an EDF scheduling heuristic, to reduce latency while maintaining high accuracy. SEEB-GPU operates effectively with minimal overhead and remains portable across different GPU vendors. Extensive evaluations using various workload traces demonstrate that SEEB-GPU consistently reduces inference latency, improves SLA compliance, and maintains or enhances accuracy compared to existing edge inference solutions. Overall, SEEB-GPU enables scalable, SLA-aware, and latency-sensitive deep learning inference at the edge, making it well-suited for real-time applications in resource-constrained environments. Future work includes testing SEEB-GPU on more edge GPUs, including Nvidia Jetson Orin and AMD GPUs, and including power-aware scheduling on SEEB-GPU.

Acknowledgments

This work was supported, in part, by the U.S. NSF under Grants CNS-2149533 and CNS-2421244. We would also like to thank the anonymous reviewers for their valuable comments.

References

[1] Mahmoud Abbasi, Amin Shahraki, and Amir Taherkordi. 2021. Deep learning for network traffic monitoring and analysis (NTMA): A survey.

Computer communications 170 (2021), 19–41.

[2] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*. IEEE Press, Atlanta, Georgia, Article 69, 15 pages.

[3] MR Ashuthosh, Santosh Krishna, Vishvas Sudarshan, Srinivasan Subramaniyan, and Madhura Purnaprajna. 2022. MAPPARAT: A Resource Constrained FPGA-Based Accelerator for Sparse-Dense Matrix Multiplication. In *2022 35th International Conference on VLSI Design and 2022 21st International Conference on Embedded Systems (VLSID)*. IEEE, Virtual Conference, 102–107.

[4] Joshua Bakita and James H Anderson. 2023. Hardware compute partitioning on NVIDIA GPUs. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, San Antonio, Texas, 54–66.

[5] Guoyu Chen, Srinivasan Subramaniyan, and Xiaorui Wang. 2024. Latency-Guaranteed Co-Location of Inference and Training for Reducing Data Center Expenses. In *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, Jersey City, New Jersey, USA, 473–484.

[6] Guoyu Chen and Xiaorui Wang. 2022. Performance optimization of machine learning inference under latency and server power constraints. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, Bologna, Italy, 325–335.

[7] Yoni Choukroun, Eli Kravchik, Fan Yang, and Pavel Kisilev. 2019. Low-bit quantization of neural networks for efficient inference. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. IEEE, Seoul,Korea, 3009–3018.

[8] Marcus Chow, Ali Jahanshahi, and Daniel Wong. 2023. Krisp: Enabling kernel-wise right-sizing for spatial partitioned gpu inference servers. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Montreal, QC, Canada, 624–637.

[9] Alessio Corrado. 2019. Animals-10 dataset. https://www.kaggle.com/datasets/alessiocorrado99/animals10. Contact: alessiocorrado99@gmail.com. Accessed June 19, 2025.

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

[11] Emily Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. 2014. Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation. *Advances in neural information processing systems* 27 (2014).

[12] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. 2020. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. Association for Computing Machinery, virtual event, 492–506.

[13] Changxing Ding and Dacheng Tao. 2015. Robust face recognition via multimodal deep face representation. *IEEE transactions on Multimedia* 17, 11 (2015), 2049–2058.

[14] Maryam Ebrahimi, Alexandre da Silva Veith, Moshe Gabel, and Eyal de Lara. 2022. Combining DNN partitioning and early exit. In *Proceedings of the 5th International Workshop on Edge Systems, Analytics and Networking*. Association for Computing Machinery, Rennes, France, 25–30.

[15] Eran Eidinger, Roee Enbar, and Tal Hassner. 2014. Age and gender estimation of unfiltered faces. *IEEE Transactions on information forensics and security* 9, 12 (2014), 2170–2179.

[16] Hironobu Fujiyoshi, Tsubasa Hirakawa, and Takayoshi Yamashita. 2019. Deep learning-based image recognition for autonomous driving. *IATSS research* 43, 4 (2019), 244–252.

[17] Guin Gilman and Robert J Walls. 2022. Characterizing concurrency mechanisms for NVIDIA GPUs under deep learning workloads. *ACM SIGMETRICS Performance Evaluation Review* 49, 3 (2022), 32–34.

[18] Ross Girshick. 2015. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*. IEEE, Santiago, Chile, 1440–1448.

[19] Ian J Goodfellow, Dumitru Erhan, Pierre Luc Carrier, Aaron Courville, Mehdi Mirza, Ben Hamner, Will Cukierski, Yichuan Tang, David Thaler, Dong-Hyun Lee, et al. 2013. Challenges in representation learning: A report on three machine learning contests. In *Neural information processing: 20th international conference, ICONIP 2013, daegu, korea, november 3-7, 2013. Proceedings, Part III 20*. Springer, Springer, Berlin, Heidelberg, 117–124.

[20] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. 2020. A survey of deep learning techniques for autonomous driving. *Journal of field robotics* 37, 3 (2020), 362–386.

[21] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S McKinley, and Björn B Brandenburg. 2017. Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX middleware conference*. Association for Computing Machinery, Las Vegas, NV, USA, 109–120.

[22] Lixiang Han, Zimu Zhou, and Zhenjiang Li. 2024. Pantheon: Preemptible Multi-DNN Inference on Mobile Edge GPUs. In *Proceedings of the 22nd Annual International Conference on Mobile Systems, Applications and Services (MOBISYS '24)*. Association for Computing Machinery, Minato-ku, Tokyo, Japan, 465–478. doi:10.1145/3643832.3661878

[23] Jianwei Hao, Piyush Subedi, Lakshmish Ramaswamy, and In Kee Kim. 2023. Reaching for the sky: Maximizing deep learning inference throughput on edge devices with ai multi-tenancy. *ACM Transactions on Internet Technology* 23, 1 (2023), 1–33.

[24] Sebastian Houben, Johannes Stallkamp, Jan Salmen, Marc Schlipsing, and Christian Igel. 2013. Detection of traffic signs in real-world images: The German Traffic Sign Detection Benchmark. In *The 2013 international joint conference on neural networks (IJCNN)*. IEEE, Dallas, TX, USA, 1–8.

[25] EunJin Jeong, Jangryul Kim, and Soonhoi Ha. 2022. Tensorrt-based framework and optimization methodology for deep learning inference on jetson boards. *ACM Transactions on Embedded Computing Systems (TECS)* 21, 5 (2022), 1–26.

[26] Yu-Gang Jiang, Changmao Cheng, Hangyu Lin, and Yanwei Fu. 2020. Learning layer-skippable inference network. *IEEE Transactions on Image Processing* 29 (2020), 8747–8759.

[27] Donghwoon Kwon, Hyunjoo Kim, Jinoh Kim, Sang C Suh, Ikkyun Kim, and Kuinam J Kim. 2019. A survey of deep learning-based network anomaly detection. *Cluster Computing* 22, Suppl 1 (2019), 949–961.

[28] Qianlin Liang, Walid A. Hanafy, Noman Bashir, Ahmed Ali-Eldin, David Irwin, and Prashant Shenoy. 2023. Dělen: Enabling Flexible and Adaptive Model-serving for Multi-tenant Edge AI. In *Proceedings of the 8th ACM/IEEE Conference on Internet of Things Design and Implementation* (San Antonio, TX, USA). Association for Computing Machinery, New York, NY, USA, 209–221. doi:10.1145/3576842.3582375

[29] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. 2019. Rethinking the Value of Network Pruning. arXiv:1810.05270 [cs.LG] https://arxiv.org/abs/1810.05270

[30] Yuan Ma, Srinivasan Subramaniyan, and Xiaorui Wang. 2025. Power Capping of GPU Servers for Machine Learning Inference Optimization. In *ICPP*. Association for Computing Machinery, San Diego, CA, USA.

[31] Rahul Mishra, Hari Prabhat Gupta, and Tanima Dutta. 2020. A Survey on Deep Neural Network Compression: Challenges, Overview, and Solutions. arXiv:2010.03954 [cs.LG] https://arxiv.org/abs/2010.03954

[32] Seyed Morteza Nabavinejad, Sherief Reda, and Masoumeh Ebrahimi. 2022. Coordinated batching and DVFS for DNN inference on GPU

accelerators. *IEEE transactions on parallel and distributed systems* 33, 10 (2022), 2496–2508.

[33] NVIDIA. 2024. *Triton Inference Server*. https://github.com/triton-inference-server/server

[34] NVIDIA. 2025. NVIDIA Jetson AGX Orin Technical Brief. https://www.nvidia.com/content/dam/en-zz/Solutions/gtcf21/jetson-orin/nvidia-jetson-agx-orin-technical-brief.pdf. Accessed: June 20, 2025.

[35] NVIDIA Corporation. 2020. NVIDIA A100 Tensor Core GPU Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf. Accessed: 2025-06-20.

[36] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. arXiv:1712.06139 [cs.DC] https://arxiv.org/abs/1712.06139

[37] Nathan Otterness and James H Anderson. 2021. Exploring AMD GPU scheduling details by experimenting with "worst practices". In *Proceedings of the 29th International Conference on Real-Time Networks and Systems*. ACM, Nantes, France, 24–34.

[38] Antonio Polino, Razvan Pascanu, and Dan Alistarh. 2018. Model compression via distillation and quantization. arXiv:1802.05668 [cs.NE] https://arxiv.org/abs/1802.05668

[39] Haseena Rahmath P, Vishal Srivastava, Kuldeep Chaurasia, Roberto G Pacheco, and Rodrigo S Couto. 2024. Early-exit deep neural network-a comprehensive survey. *Comput. Surveys* 57, 3 (2024), 1–37.

[40] Remya Ramakrishnan, Aditya KV Dev, AS Darshik, Renuka Chinchwadkar, and Madhura Purnaprajna. 2021. Demystifying Compression Techniques in CNNs: CPU, GPU and FPGA cross-platform analysis. In *2021 34th International Conference on VLSI Design and 2021 20th International Conference on Embedded Systems (VLSID)*. IEEE, Virtual Event, 240–245.

[41] NVIDIA Triton Inference Server. 2021. Triton inference server.

[42] Srinivasan Subramaniyan and Xiaorui Wang. 2023. OptiCPD: optimization for the canonical polyadic decomposition algorithm on GPUs. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, St. Petersburg, FL, USA, 403–412.

[43] Srinivasan Subramaniyan and Xiaorui Wang. 2025. Exploiting ML Task Correlation in the Minimization of Capital Expense for GPU Data Centers. In *Proceedings of the IEEE International Performance, Computing, and Communications Conference (IPCCC)*. IEEE, Austin Tx, USA.

[44] Srinivasan Subramaniyan and Xiaorui Wang. 2025. FC-GPU: Feedback Control GPU Scheduling for Real-time Embedded Systems. *ACM Transactions on Embedded Computing Systems* 24, 5s (2025), 1–25.

[45] Surat Teerapittayanon and Bradley McDanel. 2016. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd international conference on pattern recognition (ICPR)*. IEEE, IEEE, Cancún, México, 2464–2469.

[46] K Vanishree, Anu George, Srivatsav Gunisetty, Srinivasan Subramanian, Shravan Kashyap, and Madhura Purnaprajna. 2020. CoIn: Accelerated CNN co-inference through data partitioning on heterogeneous devices. In *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*. IEEE, India, 90–95.

[47] Yidi Wang, Mohsen Karimi, and Hyoseung Kim. 2022. Towards energy-efficient real-time scheduling of heterogeneous multi-gpu systems. In *2022 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, Houston, USA, 409–421.

[48] Yidi Wang, Mohsen Karimi, Yecheng Xiang, and Hyoseung Kim. 2021. Balancing energy efficiency and real-time performance in GPU scheduling. In *2021 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, virtual event, 110–122.

[49] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. {MLaaS} in the wild: Workload analysis and scheduling in {Large-Scale} heterogeneous {GPU} clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, USA, 945–960.

[50] Zhe Yang, Klara Nahrstedt, Hongpeng Guo, and Qian Zhou. 2021. Deeprt: A soft real time scheduler for computer vision applications on the edge. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, San Jose, CA, USA, 271–284.

[51] Jeff Zhang, Sameh Elnikety, Shuayb Zarar, Atul Gupta, and Siddharth Garg. 2020. Model-Switching: Dealing with Fluctuating Workloads in Machine-Learning-as-a-Service Systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, virtual event.

[52] Xinyuan Zhang, Jiang Liu, Zehui Xiong, Yudong Huang, Gaochang Xie, and Ran Zhang. 2024. Edge intelligence optimization for large language model inference with batching and quantization. In *2024 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, Dubai, United Arab Emirates, 1–6.

[53] Ziyang Zhang, Yang Zhao, Huan Li, and Jie Liu. 2024. BCEdge: SLO-Aware DNN Inference Services With Adaptive Batch-Concurrent Scheduling on Edge Devices. *IEEE Transactions on Network and Service Management* 21, 4 (2024), 4131–4145.

[54] Michael Zhu and Suyog Gupta. 2017. To prune, or not to prune: exploring the efficacy of pruning for model compression. arXiv:1710.01878 [stat.ML] https://arxiv.org/abs/1710.01878