SPLIT: QoS-Aware DNN Inference on Shared GPU via Evenly-Sized Model Splitting

Diaohan Luo*
Tian Yu*
luodiaohan21@otcaix.iscas.ac.cn
yutian20@otcaix.iscas.ac.cn
University of Chinese Academy of Sciences
Beijing, China

Yuewen Wu†
Heng Wu†‡
wuyuewen@otcaix.iscas.ac.cn
wuheng@iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China

Tao Wang
wangtao08@otcaix.iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China

Wenbo Zhang§
zhangwenbo@otcaix.iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China

ABSTRACT

Improving QoS by simultaneously reducing the latency violation rate and jitter in the presence of multiple deep learning inference (DLI) tasks sharing a single edge computing processor remains a challenge. However, existing DLI systems at the edge, designed to maximize throughput, face performance challenges when confronted with requests with varying QoS.

In this paper, we present SPLIT, a QoS-aware DNN inference system on shared GPU via evenly-sized model splitting to improve QoS by reducing the latency violation rate and jitter. SPLIT applies a genetic algorithm to evenly split models into diverse operator combinations, or blocks, thereby minimizing the standard deviation of block execution time to reduce jitter. Furthermore, we develop a preemption method based on a greedy algorithm to swiftly assess whether an incoming request should preempt to minimize latency. We evaluate SPLIT with five common deep learning models and the experimental results reveal that SPLIT outperforms state-of-the-art approaches, reducing the latency violation rate by up to 43% and jitter by up to 69.3%.

CCS CONCEPTS

• **Computer systems organization → Embedded software**; • **Computing methodologies → Machine learning**.

KEYWORDS

model splitting, QoS, block, deep learning inference, evenly-sized

*Both authors contributed equally to this research.
†Corresponding author.
‡Also affiliated with Chongqing School, University of Chinese Academy of Science.
§Also affiliated with (1) University of Chinese Academy of Sciences, Nanjing, and (2) Nanjing Institute of Software Technology.

ACM Reference Format:
Diaohan Luo, Tian Yu, Yuewen Wu, Heng Wu, Tao Wang, and Wenbo Zhang. 2023. SPLIT: QoS-Aware DNN Inference on Shared GPU via Evenly-Sized Model Splitting. In *52nd International Conference on Parallel Processing (ICPP 2023), August 07–10, 2023, Salt Lake City, UT, USA*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605573.3605627

1 INTRODUCTION

Nowadays, edge platforms [5, 20–22] need to handle diverse deep learning inference requests (DLI requests) simultaneously, such as object detection, classification, and text generation. In these scenarios, long and short requests share the same computing processor. For example, in autonomous driving scenarios [14, 33], human detection constitutes long requests, while person tracking and pose extraction represent short requests, which are triggered at any time to assess route safety. In such cases, QoS (Quality of Service), including the latency violation rate and jitter [4], is a critical metric. However, existing DLI systems at the edge, designed to maximize throughput, face performance challenges when confronted with requests with varying QoS.

Some DLI systems, such as ClockWork [9], employ request-level preemption, allowing long requests to be interrupted for short request resource preemption. However, the preempted request must rerun, resulting in inefficiency and extra latency. As illustrated in Figure 1, a short request A and a long request B share the same computing processor. The typical Stream-Parallel approach [24] runs all tasks simultaneously on the same GPU through multiple GPU streams, which is the concurrent execution method from native GPU multi-stream support. It focuses on throughput improvement while causing serious resource contention among requests. Alternatively, Runtime-Aware approach [34] utilizes model splitting, dividing the model into operator combinations (e.g., *conv*, *relu*, *pooling*) called blocks or sub-models. This model splitting-based concurrency reduces resource contention and improves global throughput. However, if a short request A arrives, it has to be aligned with request B and wait for the completion of request B, causing significant latency violation for request A. A feasible solution involves splitting models into blocks and allowing preemption at block boundaries without alignment. However, arbitrarily splitting models into unevenly-sized blocks can lead to substantial waiting latency for short requests, as they dynamically arrive and their arrival time are uncertain.

Ideally, models should be split into evenly-sized blocks, as depicted in Figure 1, to reduce the latency violation rate by minimizing the average response ratio (normalized end-to-end latency [1]) compared to other methods. Nonetheless, determining the number of blocks to split and ensuring evenness remains challenging due to the numerous splitting candidates (including optional splitting positions and the corresponding number of splits).

In this paper, we introduce SPLIT, a QoS-aware DNN inference system on shared GPU via evenly-sized model splitting, which contributes the following advancements:

- We develop a novel genetic algorithm for offline profiling of over 20,000 splitting candidates, guiding the genetic algorithm to initialize and choose splitting candidates with the optimal number of blocks for common DNN models while maintaining evenness.
- We implement a greedy preemption method based on minimum response ratio to reduce the preemption overhead, which is aimed at addressing the preemption among blocks and avoiding the latency violation.
- Our experimental evaluation demonstrates that SPLIT can improve QoS by reducing the latency violation rate by up to 43% and jitter by up to 69.3% compared to state-of-the-art approaches.

2 PROBLEM ANALYSIS AND CHALLENGES

In this section, we first analyze the severity of interactions in concurrent inference, then discuss the importance of the latency violation rate and jitter as QoS metrics. We also examine the necessity of evenly-sized model splitting and challenges associated with it.

2.1 DLI on Edge Platforms

Multiple DLI requests sharing a single computing processor. Edge scenarios often involve long and short DLI requests sharing a single computing processor. For example, in autonomous driving scenarios, the on-board processor continuously runs person detection requests, while personal tracking and pose extraction requests are executed as individuals approach the car to determine potential route conflicts. With limited on-board processor resources, short requests preempt the execution of long requests to guarantee their QoS.

The latency violation rate and jitter are important metrics of QoS for DLI on edge. Unlike concurrent inference systems that primarily focus on throughput, the latency violation rate and jitter are essential QoS metrics for independently arriving, sequentially executed requests. According to related works [4, 29], the latency target of the request is based on their uninterrupted and isolated execution time. Real-time video processing requests, for instance, are concerned with both the latency violation rate and jitter, as a few frames violate the latency target (response ratio greater than a threshold) can reduce request stability. Hence, we consider both the latency violation rate and jitter as QoS metrics.

[1]The end-to-end latency is the time from the arrival of the request to the end of execution, including execution time and wait time.

2.2 Model Splitting

A deep learning model combines multiple operators, and its inter-operator data dependencies can be represented as a directed acyclic graph (DAG). Model splitting divides long models into smaller blocks at operator boundaries.

Splitting-based concurrency approaches overlook the latency target of short requests, leading to latency violation. Deep learning model operators have diverse resource requirements, causing traditional concurrency approaches to introduce significant contention overhead as operators compete for computing resources. This results in short requests experiencing similar end-to-end latency as long requests. To address this issue, the related work [34] proposes a splitting-based concurrency approach, aligning operators based on their resource requirements to avoid contention and reduce global end-to-end latency. This method improves throughput when requests continuously arrive without gaps, but it also generates considerable idle time for short requests due to alignment.

Sequential preemption approaches without model splitting can cause latency violation and jitter. Executing multiple requests sequentially may result in latency violation and jitter, as long requests monopolize the compute processor, causing short requests to endure long waiting latency (potentially several time longer than their execution time). In the edge scenario, the latency target for short requests are usually stricter than for long requests due to their shorter execution time, as requests perceive the latency target based on uninterrupted and isolated execution time. This is why sequential preemption approaches without model splitting are inapplicable, as the latency of waiting for long requests is substantial.

Evenly-sized model splitting is difficult. There are dozens or hundreds of operators in the model. When dividing a model with M operators into N blocks, the number of candidates is C_{M-1}^{N-1}. For example, dividing ResNet50 into 3 blocks results in 287,980 candidates, requiring over 80 hours for profiling all possibilities.

2.3 Greedy Preemption

The preemption with static priority is difficult to guarantee QoS for all requests. In the realm of request preemption, existing works [4, 10] adopt fixed priority schemes or approaches that handle best-effort and real-time requests together. These methods, which assign the static priority level to requests, are intrinsically unable to guarantee a well-balanced QoS for all requests.

Conversely, assigning dynamic priority level to requests can offer a more equitable distribution of service quality. Nonetheless, this technique necessitates recalculating the priority level of all tasks each time a new request arrives, aiming to achieve the optimal preemption outcome. This process is exceedingly time-consuming and often surpasses the processing time of the requests themselves.

To address this issue, we propose a greedy-based preemption method that not only reduces the response ratio of the requests but also substantially diminishes the time spent on preemption. With a time complexity of O(n) in the worst case, this approach presents a more efficient and scalable solution for managing request preemption at the edge.

For combinatorial optimization problems, the abundance of available splitting candidates results in substantial search overhead when

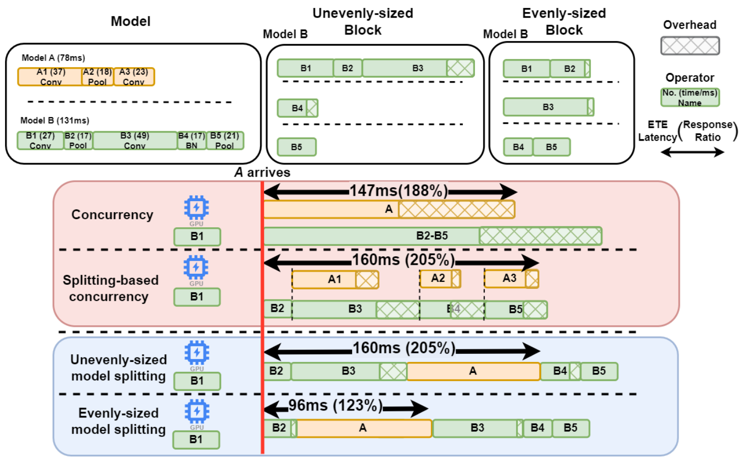
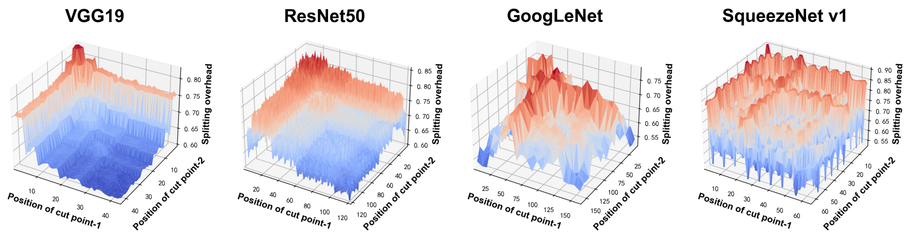
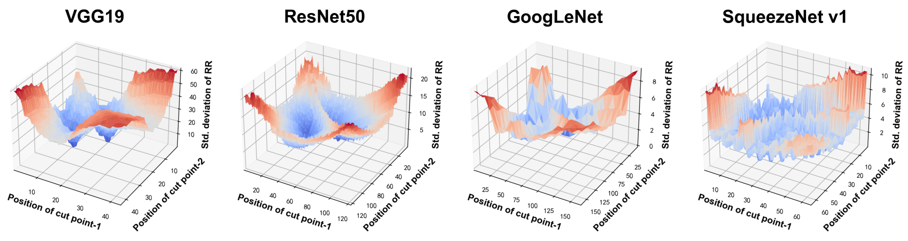


Figure 1: An example of multiple DLI requests scheduling with different preemption schemes.



(a) The relationship between position of cut points and splitting overhead. Splitting the model on the earlier operators can incur a large splitting overhead.



(b) The relationship between position of cut points and standard deviation of block execution time. Splitting the model at the beginning or last few operators will result in a larger standard deviation of block execution time.

Figure 2: The relationship of splitting overhead and standard deviation to position of cut points. The X-axis and Y-axis show position of the first cut point and the second cut point.

employing heuristic methods or reinforcement learning approaches, which often renders them unsuitable for DLI. However, by leveraging prior knowledge to accelerate the learning process, we can enhance the feasibility of these methods. Guided by the observations obtained from analysis, the algorithm can efficiently explore the solution space while avoiding the exhaustive search. Consequently, we can use the genetic algorithm and significantly reduce the search space, enabling us to identify effective splitting candidates in a more practical and efficient manner for real-world DLI requests.

2.4 Observations

We conducted a comprehensive analysis of deep learning models and derived observations regarding the impact of position of cut points on execution time and jitter:

The influence of position of cut points on execution time. Figure 2(a) illustrates the relationship between position of cut points and splitting overhead [2]. A key finding from the Figure 2(a) is that splitting the model on earlier operators leads to a larger splitting overhead. This is because, after splitting the model, data transmission between previously adjacent operators contributes to the overhead. During the inference of the model, the volume of data

[2]Splitting overhead is the ratio of the additional execution time of the blocks to the original model's execution time.

decreases with convolution and encoding, resulting in reduced data transmission overhead.

Consequently, splitting closer to the end of the model yields a smaller overhead and has a minimal impact on execution time.

The influence of position of cut points on jitter. Figure 2(b) depicts the relationship between position of cut points and standard deviation of block execution time. We use the standard deviation as a measure of splitting evenness, or jitter; a smaller standard deviation indicates a more even splitting. The figure reveals that splitting the model at the beginning or the last few operators results in uneven splitting, while splitting closer to the middle but slightly towards the beginning yields a more even splitting outcome. This is because the data volume is larger at the beginning of the model, causing the operator execution time to be longer compared to later operators.

As a result, after splitting, the execution time of the front block with fewer operators is comparable to that of the back block containing more operators, leading to a more even splitting, which means less jitter.

2.5 Objectives and Challenges

Given the difficulties in splitting models based on operator execution time and the vast number of candidates, we must address the following challenges:

- **How to choose cut points for evenly-sized model splitting.** We will investigate the relationship of execution time and jitter to position and number of cut points , aiming to split models evenly.
- **How to preempt quickly to minimize latency.** While evenly-sized model splitting can decrease waiting latency of requests, additional overhead during preemption is also crucial. A high time complexity preemption algorithm may increase latency, causing short requests to violate the latency target due to extended preemption. Thus, a fast preemption algorithm is necessary.

3 DESIGN

We introduce SPLIT, a DLI resource allocator designed to enhance request QoS by reducing the latency violation rate and jitter. First, we present an evaluation to understand the impact of position of cut points on execution time and jitter. Then, we characterize the effects of evenness and the number of model splits on waiting latency of short requests. Finally, we propose an evenly-sized model splitting method based on a genetic algorithm in §3.3 and a fast greedy preemption algorithm in §3.4.

3.1 Learning the Impact of Cut Points on Execution Time and Jitter from a Large-Scale Evaluation.

Observations on the impact of position of cut points on execution time and jitter (§2.4) provide inspiration for evenly-sized model splitting. To analyze commonly used deep learning models and learn this relationship across a wide range of operators as cut point candidates, we carry out a large-scale evaluation on NVIDIA Jetson Nano. Specially, we evaluate 11 typical deep learning models including CNN and Transformer from the model zoo of ONNX, including following categories:

- **Image classification.** It includes VGG19, ResNet50, AlexNet, SqueezeNetv1, ShuffleNet, DenseNet and GoogLeNet.
- **Object detection.** It includes YOLOv2 and EfficientNet.
- **Text generation.** It includes GPT-2.

We use ONNX Runtime [6] as the runtime framework. To profile the impact of position of cut points on execution time and jitter, we analyze and split the model using ONNX. Then we tune position of cut points and collect the execution time to calculate the splitting overhead and the standard deviation of block execution time, which is shown in Figure 2.

Base on the observations, we first analyze the waiting latency $Latency_{wait}$ of a newly arrived request and characterize the impact of model splitting options on execution time.

Assuming that a long model is split into n blocks with execution time $\{t_1, t_2, ..., t_n\}$ and a short request arrives randomly at $(0,\ \sum_{i=1}^{n} t_i)(1 \le i \le n)$, the waiting latency $Latency_{wait}$ can be given by:

$$\begin{aligned} Latency_{wait} &= \frac{\int_0^{t_1}(t_1 - t)\mathrm{d}t + \sum_{j=2}^{n}\int_{\sum_{i=1}^{j-1} t_i}^{\sum_{i=1}^{j} t_i}(\sum_{i=1}^{j} t_i - t)\mathrm{d}t}{\sum_{i=1}^{n} t_i} \\ &= \frac{1}{2}\cdot\frac{\sum_{i=1}^{n} t_i^2}{\sum_{i=1}^{n} t_i} = \frac{1}{2}\cdot(\frac{\sigma^2}{\bar{t}} + \bar{t}). \end{aligned} \quad (1)$$

where σ denotes the standard deviation of the execution time and $\bar{t}$ represents the average execution time of blocks. The term $\frac{\int_{\sum_{i=1}^{j-1} t_i}^{\sum_{i=1}^{j} t_i}(\sum_{i=1}^{j} t_i - t)\mathrm{d}t}{\sum_{i=1}^{n} t_i}$ indicates the average waiting latency of the arriving short request when the kth block is being executed. The execution time $\{t_1, t_2, ..., t_n\}$ can be profiled within 1s since a request's execution time is less than 100ms. As the number of blocks increases, the average execution time of blocks decreases, so we use the average execution time to represent the number of splits. Eq. 1 demonstrates that a high standard deviation can cause significant waiting latency. For a given standard deviation, the relationship between splitting overhead and average latency is hyperbolic, indicating that an optimal number of splits exists and more blocks may not be beneficial.

3.2 Guiding Model Splitting Using Genetic Algorithms Based on Observations

Based on the observations derived from our analysis of deep learning models in § 2.4, we designed a genetic algorithm approach to achieve evenly-sized model splitting, initialing population and choosing offspring based on observations. Our observations inform the development of this strategy in the following ways:

Position of cut points and execution time: As we observed that splitting at early operators incurs a larger overhead, the genetic algorithm is designed to prioritize choosing splitting position far from the front of the model, where the data volume and transmission overhead are reduced. This approach ensures that the impact of splitting on request execution time is minimized, leading to improved performance.

Position and jitter: Our observation that splitting closer to the middle but slightly towards the beginning of the model results in more even splitting outcomes informs the design of the fitness function of the genetic algorithm. The fitness function aims to minimize the standard deviation of block execution time, thereby promoting splitting evenly. This optimization criterion guides the evolution of the population in the genetic algorithm, encouraging the selection of splitting solutions with more evenly distributed execution time among blocks.

In light of these observations, we employ a genetic algorithm with an appropriately designed fitness function and crossover strategy, effectively exploring the search space for optimal splitting solutions.

By leveraging our observations to inform the design of genetic algorithm, we are able to identify splitting solutions that achieve a balance between evenness and reduced execution time overhead, thus optimizing the overall performance of DLI.

3.3 Evenly-sized Model Splitting Based on Genetic Algorithm

We introduce a genetic algorithm-based model splitting method in this work. Evenly-sized model splitting selects $m - 1$ cut points to divide vanilla models into m blocks. Since waiting latency depends on the evenness of blocks and the number of blocks (Eq. 1), the fitness function of the genetic algorithm considers both evenness and the number of blocks. We define the fitness function $fitness$ as:

$$fitness = -1 \cdot (e^{\frac{\sigma}{T}-1} + e^{\frac{overhead}{m}-1}) \tag{2}$$

where m denotes the number of blocks and T represents the execution time of the vanilla model. Given the relationship between block standard deviation and model splitting options as a black-box function, the genetic algorithm can achieve the desired model splitting options. Here are the steps involved in the genetic algorithm:

- Generate an initial population of model splitting options with randomly chosen cut points.
- Split the models and profile overhead and standard deviation of blocks. Calculate the $fitness$ of each population.
- Select the best model splitting options with the maximum $fitness$ and create offspring using crossover probability; otherwise, copy the parents as offspring.
- Mutate the latest offspring with a mutation probability.
- Retain a certain elite population based on the elite percentage for the next generation.
- Repeat until reaching the number of generations or the result remains unchanged for a certain number of iterations.

By leveraging our observations to inform the design of the genetic algorithm, we are able to identify splitting solutions that achieve a balance between evenness and splitting overhead, thus optimizing the overall performance of DLI. Now we have now obtained evenly-sized blocks with low overhead using the genetic algorithm-based model splitting. In order to further guarantee the QoS of requests, a fast preemption algorithm is required.

Limitation of evenly-sized model splitting and elastic model splitting in SPLIT. Although evenly-sized model splitting can improve the QoS in most situations, it also introduces additional splitting overhead. To address this, SPLIT employs an elastic model splitting mechanism: under conditions of particularly high request density, model splitting is temporarily disabled to avoid the impact of the extra splitting overhead on QoS. Similarly, when an excessive number of requests of the same type are present, splitting is also temporarily suspended. This is because requests of the same type adhere to the FIFO principle, rendering splitting and preemption unnecessary between them. The elastic model splitting mechanism allows SPLIT to flexibly optimize task QoS based on the distribution of incoming requests, striking a balance between the benefits of splitting and the potential drawbacks of the associated overhead. By dynamically adjusting its splitting strategy, SPLIT is better equipped to ensure near-optimal QoS under a variety of workload conditions.

3.4 Fast Greedy Preemption Method Based on Response Ratio

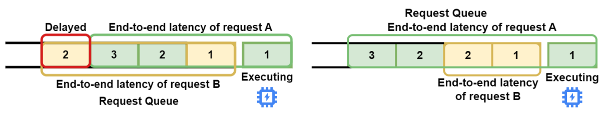
Since the preemption algorithm is executed every time a request arrives, it is called frequently. So, DLI in edge platforms often demands low latency, making high time complexity preemption algorithms infeasible. Considering that DLI requests typically complete in milliseconds, a complex preemption algorithm may significantly increase lock contention, resulting in higher latency.

We make two observations on preemption: (1) The request completion time depends on the end time of the last block. All blocks of a higher priority request should preempt together (see Figure 3(b)), or the request may suffer additional waiting latency caused by the last block (Figure 3(a)). (2) Adjusting the execution order of neighboring requests does not change their respective execution time, so it does not impact other requests' waiting time. For requests from the same task, since they have identical execution time and QoS requirements, the request arriving first should be executed first to achieve a lower response ratio.

Based on these observations, we introduce a greedy preemption algorithm based on response ratio. To evaluate the impact of preemption on QoS, we define the response ratio RR of each request when a new request is appended to the request queue:

$$\begin{gathered} latency_{wait} = latency_{waited} + latency_{waiting} \\ RR = \frac{latency_{wait} + t_{ext}}{t_{ext}} = \frac{t_{ete}}{t_{ext}} \end{gathered} \tag{3}$$

where $latency_{waited}$ denotes the waiting latency of the request, $latency_{waiting}$ represents the predicted latency the request will continue to wait, t_{ext} is the request's execution time, and t_{ete} refers to the end-to-end latency, and the $latency_{waiting}$ is the sum of t_{ext} of all previous requests.


Delayed
End-to-end latency of request A
2
3
2
1
1
End-to-end latency of request B
Request Queue
Executing
Request Queue
End-to-end latency of request A
3
2
2
1
1
End-to-end latency of request B
Executing

(a) The partial preemption. (b) The full preemption.

Figure 3: Comparison of between partial preemption and full preemption. The partial preemption produces straggler and increases total latency of request A. The full preemption can reduce latency of request B.

We conclude the following characteristics for block-level preemption:

- All blocks of one request executing preemption together is better than partial preemption.
- Swapping the order of two neighbors does not influence other requests.
- First-Input-First-Output (FIFO) for requests from the same task.
- The total latency of a request is related to its position in the request queue.

Based on these conclusions, we propose a fast greedy preemption algorithm based on response ratio to achieve near-optimal preemption at microsecond-scale. Algorithm 1[3] describes the greedy preemption algorithm. We find that, for n requests with k models split into m blocks on average, SPLIT has a worst-case time complexity of $O(n)$ and an average time complexity of $O(k)$.

Preemption can only be executed between neighbors since swapping the order of two neighbors does not influence others. They only exchange the order when it can reach a lower response ratio and repeat it until:

- No requests are ahead, meaning the request has the highest priority.
- Both requests are from the same task.
- Exchanging cannot reduce the average response ratio of the two requests.

4 IMPLEMENTATION

We implemented and deployed SPLIT on NVIDIA Jetson Nano with CUDA. The architecture of SPLIT is depicted in Figure 4. Our implementation consists of around 9000 lines of C++ code. We use ONNX Runtime [6] as the deep learning runtime framework and convert models to *.onnx* format to provide support for most deep learning frameworks.

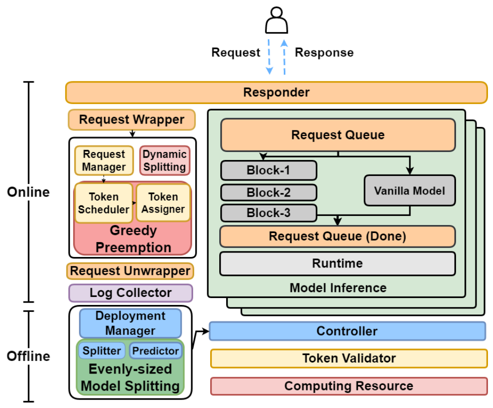


Figure 4: System workflow.

[3]Inspired by PREMA [4], we define *the latency target*, the maximum allowed end-to-end latency, as $\alpha \times Ext(t_i)$.)

Algorithm 1 Greedy preemption

```
Input:
  • $T$:Inference requests with $N$ requests
  • $t_{new}$:New-arrived request
  • $R(t_i, l)$:Response ratio of request $t_i$ with end-to-end latency $l$
  • $Ext(t_i)$:Execution time of $t_i$
  • $Ext_{left}(t_i)$:Left executing latency of $t_i$
  • $Start(t)$:Time the request $t$ starts to be executed
Output: $T$
Function ResponseRatio($l^{waiting}, t_i, T$)
1:  $l^{waited} \leftarrow clock() - Start(t_i)$
2:  $Target \leftarrow \alpha \times Ext(t_i)$
3:  return $\frac{l^{waited}+l^{waiting}+Ext_{left}(t_i)}{Target}$
Function Preemption($T, t_{new}$)
4:  $l_{waiting} = \sum_{n=1}^{N} Ext(t_n)$
5:  for $i$ requests in $1, 2, ..., N$ do
6:      if $type(t_{new}) = type(t_i)$ then
7:          return $T$
8:      end if
9:      $RR_{t_{new}}^{back} \leftarrow ResponseRatio(l_{waiting}, t_{new}, T)$
10:     $l_{waiting} \leftarrow l_{waiting} - Ext_{left}(t_i)$
11:     $RR_{t_{new}}^{front} \leftarrow ResponseRatio(l_{waiting}, t_{new}, T)$
12:     $RR_{t_i}^{back} \leftarrow ResponseRatio(l_{waiting} + Ext_{left}(t_i), t_i, T)$
13:     $RR_{t_i}^{front} \leftarrow ResponseRatio(l_{waiting}, t_i, T)$
14:     if $RR_{t_{new}}^{front} - RR_{t_{new}}^{back} \geq RR_{t_i}^{front} - RR_{t_i}^{back}$ then
15:         $T$.insert($t_i, t_{new}$)
16:         return $T$
17:     end if
18: end for
19: $T$.insert($T.end, t_{new}$)
20: return $T$
```

4.1 System Workflow

Figure 4 illustrates the system workflow of SPLIT: (1) Users deploy various tasks with multiple deep learning models, each generating requests independently. (2) SPLIT accepts models from popular deep learning frameworks (e.g., TensorFlow [1], PyTorch [26], PaddlePaddle [19]) and converts them to *.onnx* format. (3) SPLIT divides models into evenly-sized blocks based on the genetic algorithm and stores the blocks as *.onnx* files. (4) SPLIT deploys models according to the splitting results and preempts based on the greedy algorithm. (5) SPLIT returns inference results to the user.

In particular, processes (1), (2), (4), and (5) are online, while process (3) is offline. As models deployed on the edge remain constant, lengthy models only need to be split once.

4.2 System Components

Responder accepts user requests using the RPC protocol and appends them to the request queue. It reads the inference result from the output queue and replies to users. To minimize waiting latency, the responder runs on a separate thread and secures asynchronous r/w with a thread lock. **Request wrapper** analyzes computational graphs and wraps requests using the given deep learning runtime

Table 1: Evaluated deep learning models.

Model	Operators	Domain	Latency(ms)	Type
YOLOv2	84	Object Detection	10.8	Short
GoogLeNet	142	Image Classification	13.2	Short
ResNet50	122	Image Classification	28.35	Long
VGG19	44	Image Classification	67.5	Long
GPT-2	2534	Text Generation	20.4	Short

framework. **Request unwrapper** converts model files to *.onnx* format. **Deployment manager** deploys the blocks on the processor, executing each request as a separate thread to ensure resource efficiency. **Token scheduler** orders all requests in the request queue based on the greedy algorithm. **Token assigner** assigns the token to the highest priority request for execution. **Responder** collects the inference result and responds to users.

5 EVALUATION

In this section, we assess the effectiveness of SPLIT by addressing the following questions:

- Can evenly-sized model splitting divide various models into evenly-sized blocks with minimal splitting overhead?
- Can SPLIT improve QoS by reducing the latency violation rate?
- Can SPLIT reduce jitter by reducing the standard deviation of block execution time in various scenarios?

5.1 Experimental Setup

Benchmarks. We create various DLI scenarios with the following deep learning models: YOLOv2 [28], GoogLeNet, ResNet50, VGG19, and GPT-2 [27]. The evaluated models, including image classification, object detection, and text generation, exhibit different execution time, as shown in Table 1. Though some models belong to the same application domain, their execution time and performance differ due to distinct computing graph topologies and operators. Consequently, they are deployed and used independently for different requests.

Table 2: Scenarios that simulate various DLI applications running on an edge system.

Name	Average arrival interval(λ)	Load
Scenario1	160ms	Low
Scenario2	150ms	
Scenario3	140ms	↓
Scenario4	130ms	
Scenario5	120ms	High
Scenario6	110ms	

Workload. Real-world request generation scenarios are crucial for evaluating computing resource allocation methods. However, research about DLI on a single edge computing resource is still nascent, resulting in few publicly available datasets for experimental purposes. Thus, we generate random request queries using *Poisson distribution* to emulate real-world request generation scenarios, inspired by prior works [2, 30, 35]. The λ of the Poisson distribution represents the average request arrival interval. For instance, $\lambda = 150$ indicates an average arrival interval of $150ms$. Based on hardware tolerance[4], we evaluate SPLIT and other works under six scenarios, detailed in Table 2. The "Average arrival interval(λ)" column denotes λ for each scenario's corresponding requests. The total number of requests is set to 1000.

Testbed. Our experiments utilize an NVIDIA Jetson Nano. The software environment consists of ONNX Runtime 1.12.1 and Ubuntu 18.04.

5.2 Metrics

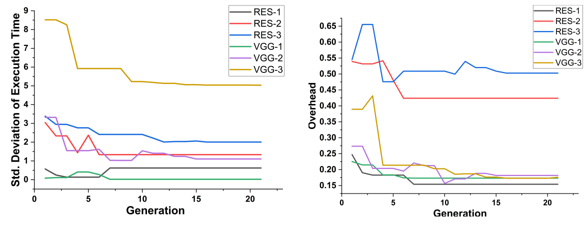
We employ the following metrics to gauge SPLIT's effectiveness:

The latency violation rate. The unique latency target characterize each request and serve as vital QoS metrics. As discussed, requests perceive QoS based on uninterrupted and isolated execution time (§2.1). We define the latency target as $\alpha \times Ext(t_i)$ in §3.4. If a request's response ratio exceeds α, it violates the latency target. Since the request's latency target must be greater than its execution time, we sweep α from 2 to 20 and measure the fraction of latency-violated requests as a function.

Standard deviation of block execution time. This metric measures inference jitter by evaluating data dispersion, which is essential for assessing request stability.

5.3 Baselines

We compare SPLIT with typical DLI resource allocation methods. ClockWork [9] sequentially executes DLI requests on the GPU with static priority. PREMA [4] is a temporal multi-requesting algorithm allowing requests to preempt with passive priority. Runtime-aware approach (RT-A) [34] concurrently runs all requests on the same GPU through multiple GPU streams, splitting and aligning models based on resource requirement of operators.



(a) Standard deviation of optimal model splitting option of each generation.
(b) Overhead of the optimal model splitting option of each generation.

Figure 5: The minimum standard deviation and its overhead of each generation.

5.4 Effectiveness of Evenly-sized Model Splitting Based on Genetic Algorithm

This experiment evaluates if evenly-sized model splitting can divide **long models** into evenly-sized blocks and provide optimal model splitting options for ResNet50 and VGG19 with varying numbers of blocks.

Figure 5 depicts the standard deviation and overhead of model splitting options in each generation with different numbers of

[4]Shorter intervals (e.g., 90ms) result in a growing request queue and following requests will always violate the latency target. As for longer intervals (e.g., 200ms), requests can be handled sequentially and no scheduling strategy is required.

Table 3: Optimal model splitting options for different numbers of blocks.

Model	Blocks	Std. Deviation	Overhead	Range(Percentage)
ResNet50	2	0.62	15.4%	5.69%
	3	1.33	42.4%	14.70%
	4	2	50.3%	23.40%
VGG19	2	0.02	19.8%	0.09%
	3	1.1	18.1%	5.37%
	4	5.03	27.6%	24.8%

blocks (RES-1 means splitting ResNet50 into 2 blocks). Figure 5(a) demonstrates that nearly all models obtain optimal model splitting options within 12 generations. After 15 generations, all models find the optimal options with minimal standard deviation to split models evenly. Figure 5(b) shows the overhead reduction during the search, decreasing as standard deviation decreases, with final overheads lower than the initial values. Table 3 presents the optimal model splitting options for different numbers of model splits. According to Eq. 1, the optimal split numbers for ResNet50 and VGG19 are 2 and 3 for the minimum splitting overhead, respectively. Due to the discrete execution time of operators, increasing the number of model splits raises the standard deviation and may result in higher overhead.

5.5 Effectiveness of SPLIT in Various Scenarios

We assess latency and jitter of inference using the latency violation rate and standard deviation of execution time, two crucial QoS factors. In this experiment, we compare SPLIT with other baselines across six scenarios in Table 2.

Effectiveness in reducing the latency violation rate of requests. Figure 6 displays the latency violation rate of SPLIT and baselines in different scenarios. SPLIT significantly reduces the latency violation rate to below 10% beyond an latency target of $\alpha = 4$, a marked improvement over the 26% latency violation under RT-A. SPLIT lowers the latency violation rate in all six scenarios.

Effectiveness in reducing jitter. We employ the standard deviation of block execution time to measure inference jitter. Figure 7 showcases the standard deviation of execution time for SPLIT and baselines across various scenarios. For low workloads, such as scenario 1, SPLIT reduces the standard deviation of short requests by 55.3%, 46.8%, and 68.9% compared to ClockWork [9], PREMA [4], and RT-A [34]. For high workloads, the reductions are 56.0%, 50.3%, and 69.3%. Model splitting enhances the preemption capability of short requests like YOLOv2, GPT-2, and GoogLeNet. SPLIT achieves the best stability compared to related works, though it sacrifices stability for longer requests like ResNet50 and VGG19. However, the standard deviation of long requests is still slightly lower than short requests, indicating that the stability of all requests is approximately at the same level.

6 DISCUSSION

Overhead of model splitting. We observe that the total execution time of all blocks is greater than that of the vanilla model. Model splitting options significantly influence the splitting overhead, which is strongly correlated with the intermediate input and output data at the splitting boundaries. The considerable splitting overhead also explains the difficulty in accurately predicting model execution time based on operators. Approaches like REEF [10], which split requests based on GPU kernel, can alleviate this problem at the cost of higher hardware dependency. In contrast, SPLIT is more flexible and benefits from its insensitivity to hardware.

Predictability of DLI latency. Latency predictability is a crucial concern for DLI in edge platforms, especially in real-time systems. However, latency becomes unstable when various DLI requests run concurrently on a single computing resource due to hardware interactions. SPLIT infers requests sequentially to avoid interactions and achieve predictable latency. This approach may appear to sacrifice spatial resource utilization, but it still outperforms the concurrent approach(RT-A), as demonstrated in §5.5.

Limitations of model splitting. SPLIT's model splitting supports CNNs and Transformer networks due to their static DAG structure and the static knowledge of the vanilla model's execution time. However, the execution time of RNNs depends on the input data size, rendering them unsuitable for offline splitting.

7 RELATED WORK

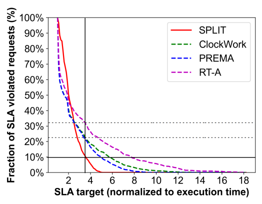
Numerous works have been proposed to optimize resource allocation for DLI systems, primarily focusing on throughput or total resource utilization. SPLIT, however, emphasizes the QoS of each task.

Task-level resource allocation. These approaches treat a task as a resource management unit and are insensitive to the model structure details of deep learning models, simplifying scheduling [11, 15]. ClockWork [9] employs FCFS task-level scheduling, dropping tasks predicted to be stragglers upon arrival. However, this is unsuitable for tasks with random arrival times. PREMA [4] combines offline records and online token-based task scheduling for predictive multi-task scheduling. Such approaches are suitable for cluster-based DLI but have limited impact on single computing resources with coarse-grained tasks.

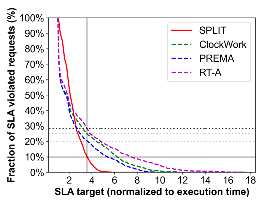
Graph-level resource allocation. Compared to task-level allocation, graph-level allocation allows for more fine-grained and flexible allocations for various models [12, 13, 17]. Band [16] splits deep learning models into operators and allocates resources according to affinity between the operator and hardware. EOP [32] partitions operators based on operator execution estimation using three performance variation patterns. Runtime-aware [34] merges multiple models into a single model with parallel branches. Although these approaches can improve throughput, short requests may experience increased latency as they wait for longer requests to complete, negatively impacting QoS.

Kernel-level resource allocation. Works in this category improve task inference efficiency through thread blocking, loop tiling, and other techniques. For instance, REEF [10] employs thread blocking for microsecond-scale kernel preemption and controlled concurrent execution. [31] blocks new threads to passively preempt GPU streaming multiprocessors. TVM [3] is a kernel-level auto-tuning framework optimizing configurations to maximize GPU utilization. SPLIT can be integrated with these approaches to further enhance performance.

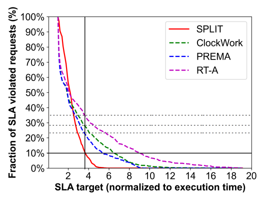
Resource-level management. With the emergence of fine-grained computing resource sharing and partitioning techniques like multi-process services (MPS [25]) and multi-streams, many works aim to manage fine-grained resources and avoid task interactions [18, 23]. GSLICE [7] builds on MPS to enable controlled spatial



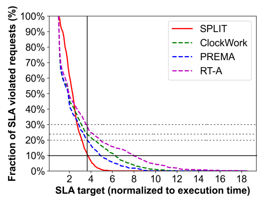
(a) Scenario 1.



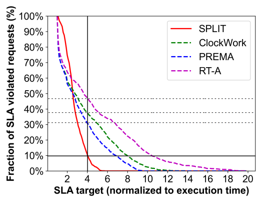
(b) Scenario 2.



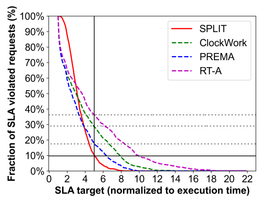
(c) Scenario 3.



(d) Scenario 4.

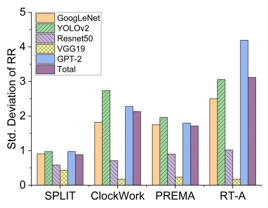


(e) Scenario 5.

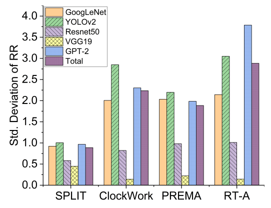


(f) Scenario 6.

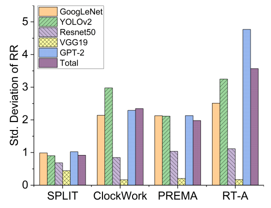
Figure 6: The latency violation rate for all requests as a function of latency target (normalized to execution time) on the X-axis.



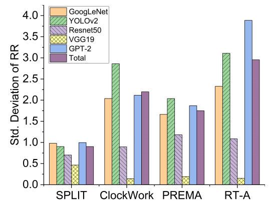
(a) Scenario 1.



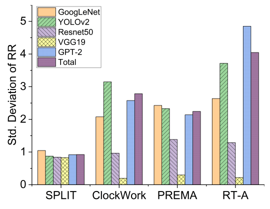
(b) Scenario 2.



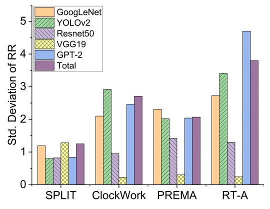
(c) Scenario 3.



(d) Scenario 4.



(e) Scenario 5.



(f) Scenario 6.

Figure 7: Standard deviation of block execution time of each model in various scenarios.

GPU sharing across multiple inference function models, using a self-learning method to dynamically adjust GPU resource allocation ratios for workloads. Planaria [8] dynamically allocates computing resources with a deadline-aware scheduler. SPLIT could leverage these optimizations to improve resource utilization.

8 CONCLUSION

This paper presents SPLIT, a QoS-aware DNN inference system which splits models into evenly-sized blocks to improve QoS by reducing the latency violation rate and jitter of each request. We propose an approach that splits models into evenly-sized blocks based on genetic algorithm to reduce the overhead of requests. Next, we propose the latency violation rate and jitter to evaluate QoS from the perspective of requests. Finally, we use greedy preemption to minimize response ratio of requests to improve QoS at microsecond-scale. Experiments show that SPLIT can reduce response ratio of requests by up to 43% and jitter by up to 69.3%.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback and suggestions to improve this work. This work was partially supported by Provincial Key Research and Development Program of Shandong, China (2021CXGC010101) and National Natural Science Foundation of China (No.61972386).

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: A System for {Large-Scale} Machine Learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] Shane Bergsma, Timothy Zeyl, Arik Senderovich, and J. Christopher Beck. 2021. Generating Complex, Realistic Cloud Workloads using Recurrent Neural Networks. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 376–391. https://doi.org/10.1145/3477132.3483590

[3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos

Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[4] Yujeong Choi and Minsoo Rhu. 2020. PREMA: A Predictive Multi-Task Scheduling Algorithm For Preemptible Neural Processing Units. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*. IEEE, 220–233. https://doi.org/10.1109/HPCA47549.2020.00027

[5] Prafulla N. Dawadi, Diane Joyce Cook, and Maureen Schmitter-Edgecombe. 2016. Automated Cognitive Health Assessment From Smart Home-Based Behavior Data. *IEEE J. Biomed. Health Informatics* 20, 4 (2016), 1188–1194. https://doi.org/10.1109/JBHI.2015.2445754

[6] ONNX Runtime developers. 2021. ONNX Runtime. https://onnxruntime.ai/. Version: x.y.z.

[7] Aditya Dhakal, Sameer G. Kulkarni, and K. K. Ramakrishnan. 2020. GSLICE: controlled spatial sharing of GPUs for a scalable inference platform. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 492–506. https://doi.org/10.1145/3419111.3421284

[8] Soroush Ghodrati, Byung Hoon Ahn, Joon Kyung Kim, Sean Kinzer, Brahmendra Reddy Yatham, Navateja Alla, Hardik Sharma, Mohammad Alian, Eiman Ebrahimi, Nam Sung Kim, Cliff Young, and Hadi Esmaeilzadeh. 2020. Planaria: Dynamic Architecture Fission for Spatial Multi-Tenant Acceleration of Deep Neural Networks. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*. IEEE, 681–697. https://doi.org/10.1109/MICRO50266.2020.00062

[9] Arpan Gujarati, Reza Karimi, Safya Alzayat, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. *CoRR* abs/2006.02464 (2020). arXiv:2006.02464 https://arxiv.org/abs/2006.02464

[10] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 539–558. https://www.usenix.org/conference/osdi22/presentation/han

[11] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. 2019. GRNN: Low-Latency and Scalable RNN Inference on GPUs. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, George Candea, Robbert van Renesse, and Christof Fetzer (Eds.). ACM, 41:1–41:16. https://doi.org/10.1145/3302424.3303949

[12] Zhaowu Huang, Fang Dong, Dian Shen, Junxue Zhang, Huitian Wang, Guangxing Cai, and Qiang He. 2021. Enabling Low Latency Edge Intelligence based on Multi-exit DNNs in the Wild. In *41st IEEE International Conference on Distributed Computing Systems, ICDCS 2021, Washington DC, USA, July 7-10, 2021*. IEEE, 729–739. https://doi.org/10.1109/ICDCS51616.2021.00075

[13] Arpan Jain, Tim Moon, Tom Benson, Hari Subramoni, Sam Adé Jacobs, Dhabaleswar K. Panda, and Brian Van Essen. 2021. SUPER: SUb-Graph Parallelism for TransformERs. In *35th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Portland, OR, USA, May 17-21, 2021*. IEEE, 629–638. https://doi.org/10.1109/IPDPS49936.2021.00071

[14] Wonseok Jang, Hansaem Jeong, Kyungtae Kang, Nikil D. Dutt, and Jong-Chan Kim. 2020. R-TOD: Real-Time Object Detector with Minimized End-to-End Delay for Autonomous Driving. In *41st IEEE Real-Time Systems Symposium, RTSS 2020, Houston, TX, USA, December 1-4, 2020*. IEEE, 191–204. https://doi.org/10.1109/RTSS49844.2020.00027

[15] Beomyeol Jeon, Linda Cai, Pallavi Srivastava, Jintao Jiang, Xiaolan Ke, Yitao Meng, Cong Xie, and Indranil Gupta. 2020. Baechi: fast device placement of machine learning graphs. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 416–430. https://doi.org/10.1145/3419111.3421302

[16] Joo Seong Jeong, Jingyu Lee, Donghyun Kim, Changmin Jeon, Changjin Jeong, Youngki Lee, and Byung-Gon Chun. 2022. Band: coordinated multi-DNN inference on heterogeneous mobile processors. In *MobiSys '22: The 20th Annual International Conference on Mobile Systems, Applications and Services, Portland, Oregon, 27 June 2022 - 1 July 2022*, Nirupama Bulusu, Ehsan Aryafar, Aruna Balasubramanian, and Junehwa Song (Eds.). ACM, 235–247. https://doi.org/10.1145/3498361.3538948

[17] Emre Kilcioglu, Hamed Mirghasemi, Ivan Stupia, and Luc Vandendorpe. 2021. An Energy-Efficient Fine-Grained Deep Neural Network Partitioning Scheme for Wireless Collaborative Fog Computing. *IEEE Access* 9 (2021), 79611–79627. https://doi.org/10.1109/ACCESS.2021.3084689

[18] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. 2019. μLayer: Low Latency On-Device Inference Using Cooperative Single-Layer Acceleration and Processor-Friendly Quantization. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, George Candea, Robbert van Renesse, and Christof Fetzer (Eds.). ACM, 45:1–45:15. https://doi.org/10.1145/3302424.3303950

[19] Yanjun Ma, Dianhai Yu, Tian Wu, and Haifeng Wang. 2019. PaddlePaddle: An open-source deep learning platform from industrial practice. *Frontiers of Data and Domputing* 1, 1 (2019), 105–115.

[20] Emiliano Miluzzo, Tianyu Wang, and Andrew T. Campbell. 2010. EyePhone: activating mobile phones with your eyes. In *Proceedings of the 2ndt ACM SIGCOMM Workshop on Networking, Systems, and Applications for Mobile Handhelds, MobiHeld 2010, New Delhi, India, August 30, 2010*, Landon P. Cox and Alec Wolman (Eds.). ACM, 15–20. https://doi.org/10.1145/1851322.1851328

[21] Mehdi Mohammadi and Ala I. Al-Fuqaha. 2018. Enabling Cognitive Smart Cities Using Big Data and Machine Learning: Approaches and Challenges. *IEEE Commun. Mag.* 56, 2 (2018), 94–101. https://doi.org/10.1109/MCOM.2018.1700298

[22] Mehdi Mohammadi, Ala I. Al-Fuqaha, Mohsen Guizani, and Jun-Seok Oh. 2018. Semisupervised Deep Reinforcement Learning in Support of IoT and Smart City Services. *IEEE Internet Things J.* 5, 2 (2018), 624–635. https://doi.org/10.1109/JIOT.2017.2712560

[23] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 1–15. https://doi.org/10.1145/3341301.3359646

[24] NVIDIA. 2020. Cuda streams. https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

[25] NVIDIA. 2021. NVIDIA Multi-Process Service. https://docs.nvidia.com/deploy/mps

[26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*. 8024–8035.

[27] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[28] Joseph Redmon and Ali Farhadi. 2017. YOLO9000: Better, Faster, Stronger. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 6517–6525. https://doi.org/10.1109/CVPR.2017.690

[29] Wonik Seo, Sanghoon Cha, Yeonjae Kim, Jaehyuk Huh, and Jongse Park. 2021. SLO-Aware Inference Scheduler for Heterogeneous Processors in Edge Platforms. *ACM Trans. Archit. Code Optim.* 18, 4 (2021), 43:1–43:26. https://doi.org/10.1145/3460352

[30] 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, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 322–337. https://doi.org/10.1145/3341301.3359658

[31] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramírez, Nacho Navarro, and Mateo Valero. 2014. Enabling preemptive multiprogramming on GPUs. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 193–204. https://doi.org/10.1109/ISCA.2014.6853208

[32] Yuanjia Xu, Heng Wu, Wenbo Zhang, and Yi Hu. 2022. EOP: efficient operator partition for deep learning inference over edge servers. In *VEE '22: 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Virtual Event, Switzerland, 1 March 2022*, John Criswell, Dan Williams, and Yubin Xia (Eds.). ACM, 45–57. https://doi.org/10.1145/3516807.3516820

[33] Ming Yang, Shige Wang, Joshua Bakita, Thanh Vu, F. Donelson Smith, James H. Anderson, and Jan-Michael Frahm. 2019. Re-Thinking CNN Frameworks for Time-Sensitive Autonomous-Driving Applications: Addressing an Industrial Challenge. In *25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2019, Montreal, QC, Canada, April 16-18, 2019*, Björn B. Brandenburg (Ed.). IEEE, 305–317. https://doi.org/10.1109/RTAS.2019.00033

[34] Fuxun Yu, Shawn Bray, Di Wang, Longfei Shangguan, Xulong Tang, Chenchen Liu, and Xiang Chen. 2021. Automated Runtime-Aware Scheduling for Multi-Tenant DNN Inference on GPU. In *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2021, Munich, Germany, November 1-4, 2021*. IEEE, 1–9. https://doi.org/10.1109/ICCAD51958.2021.9643501

[35] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 1049–1062. https://www.usenix.org/conference/atc19/presentation/zhang-chengliang