THE UNIVERSITY OF CHICAGO

OPTIMIZING TENSOR LOADING FOR DISTRIBUTED MACHINE LEARNING APPLICATIONS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
KUNTAI DU

CHICAGO, ILLINOIS
AUGUST 2025

Copyright © 2025 by Kuntai Du

All Rights Reserved

TABLE OF CONTENTS

LIST OF FIGURES v

LIST OF TABLES vii

ACKNOWLEDGMENTS viii

ABSTRACT xi

1 INTRODUCTION 1

2 MOTIVATION 5
2.1 Video Analytics 5
2.1.1 Limitations of previous work 6
2.2 Large Language Model applications 8

3 DDS: SERVER-DRIVEN STREAMING FOR OBJECT DETECTION 9
3.1 DNN-Driven Video Streaming 9
3.1.1 Overall workflow 9
3.1.2 Feedback regions 11
3.2 Evaluation 13
3.2.1 Evaluation setup 13
3.2.2 Evaluation results 16

4 ONEADAPT: GENERATE REGION-OF-INTEREST VIA BACKPROPAGATION 19
4.1 Design of OneAdapt 19
4.1.1 Terminology 20
4.1.2 Adaptation goal and gradient-ascent 22
4.1.3 Fast approximation of AccGrad 24
4.1.4 Relationship between OutputGrad and AccGrad 27
4.1.5 Caveats and benefits 29
4.2 Evaluation 31
4.2.1 Evaluation setup 31
4.2.2 Experimental results 35

5 PREFILLONLY: AN INFERENCE ENGINE FOR PREFILL-ONLY WORKLOAD 40
5.1 Why prefill-only workload is huge 40
5.1.1 Preliminary of large language models 40
5.1.2 Substituting traditional deep learning models with LLMs 41
5.1.3 Underlying workload: prefill-only workload 41
5.1.4 Characteristics of prefill-only workload 43
5.1.5 Limitation of existing LLM engines 44
5.1.6 Opportunity and challenges 44

5.2 Overview of PrefillOnly 45
5.2.1 Overview of PrefillOnly 45
5.3 Scheduling in the context of prefix caching 46
5.3.1 Why not batching prefill-only requests 46
5.3.2 Limitation of traditional JCT-based scheduling 47
5.3.3 Continuous JCT calibration 49
5.4 Evaluation 51
5.4.1 Evaluation setup 51
5.4.2 Evaluation results 54

6 LIMITATION AND FUTURE WORK 59
6.1 Limitation 59
6.2 Future work 59

7 CONCLUSION 61

REFERENCES 62

LIST OF FIGURES

1.1 Machine learning pipelines are becoming distributed, and this thesis optimizes for data loading . . . 1
1.2 Illustrating the compression opportunity of video analytics application. Only 20% pixels need to be encoded, indicating a 5× compression potential. . . . 2

3.1 Contrasting the new real-time DNN-driven streaming (iterative) with traditional video streaming in video analytics. . . . 10
3.2 Illustration on how DDS generates feedback regions on two types of applications. 12
3.3 The normalized bandwidth consumption v.s. inference accuracy of DDS and several baselines on various video genres and applications. DDS achieves high accuracy with 55% bandwidth savings on object detection and 42% on semantic segmentation, and 36% on face recognition. Ellipses show the 1-σ range of results. 14
3.4 Compared to prior solutions, DDS has low additional systems overhead on both client and server. . . . 17

4.1 Illustrating how OneAdapt estimates AccGrad using InputGrad and DNNGrad using the sensor and the server. . . . 20
4.2 Comparison between calculating OutputGrad naively and calculating OutputGrad using OneAdapt. OneAdapt calculates OutputGrad with **no extra inference**. . . . 24
4.3 The CDF of the cosine similarity between AccGrad and OutputGrad across different configurations and videos. The average cosine similarity is over 0.91. . . . 29
4.4 Demonstrating the trade-off between accuracy and resource usage of OneAdapt and several baselines. OneAdapt achieves higher accuracy with 15–59% resource usage reduction or 1–5% higher accuracy with less resource usage compared to the baselines across 9 different pipelines. Each figure shows averages across 5–10 videos or 200 audios, depending on the dataset, ensuring statistical confidence. 36
4.5 When the knobs are more fine-grained, the bandwidth reduction of OneAdapt (the bandwidth usage of the region-based baseline, divided by the bandwidth usage of OneAdapt when OneAdapt is of higher accuracy) grows larger. . . . 37
4.6 Comparing the server-side compute overheads of OneAdapt against the baselines (measured by the number of frames inferred by the server per second). . . . 38
4.7 Benchmarking the effectiveness of removing unneeded computation and DNNGrad reusing on a 10-frame video chunk using pipeline ⓐ . OneAdapt reduces the GPU runtime overhead by 87% and the GPU memory overhead by 12%. . . . 39

5.1 Overview of PrefillOnly and its core techniques. . . . 45
5.2 Contrasting first-in-first-out (FIFO) scheduling, shortest-remaining-job-first (SRJF) scheduling and PrefillOnly's SRJF scheduling with continuous JCT calibration. The scheduling of PrefillOnly yields one more cache hit, achieving lower average latency. . . . 48

5.3 QPS — mean latency trade-off of PrefillOnly and baselines on four different hardware setups and two applications. PrefillOnly significantly reduces the latency when the QPS is high and only has higher QPS than tensor parallel baseline when QPS is low. Though tensor parallelism sometimes have lower latency than PrefillOnly under low QPS, it has much lower throughput than PrefillOnly due to extra communication cost and thus scales much worse than PrefillOnly in high QPS. 55
5.4 QPS — P99 latency trade-off of PrefillOnly and baselines on three different hardware setups and two applications. 56
5.5 Contrasting the throughput of PrefillOnly and baselines on credit verification workload under 2× H100. Though NVLink significantly accelerates the communication and thus enhances the throughput of communication-intensive parallelization like tensor parallel, PrefillOnly still has the highest throughput as it does not spend extra communication to parallelize the inference. 57
5.6 Illustrating the throughput of PrefillOnly and the baselines in post post-recommendation dataset under 2× H100 without NVLink. PrefillOnly has better improvement as it can maintain high throughput under high query-per-second, while the query per second of chunked prefill baseline drops because of prefix cache throttling. Parallelization-based baselines parallelize the prefix cache across GPUs and thus have sufficient prefix cache space to avoid throttling, but they have lower throughput because of extra communication and synchronization cost. 57
5.7 The CDF of request latency of PrefillOnly, under different value of fairness parameter λ. Higher λ result in better P99 latency, at the cost of inflating the average latency. 58

LIST OF TABLES

3.1 Summary of DDS datasets. 13

4.1 Summary of the notations used in OneAdapt 21
4.2 Overview of the experimental setup. Each row represents an analytic pipeline we used to evaluate OneAdapt. 31

5.1 Summarizing the dataset used in the evaluation of PrefillOnly and baselines. . . 50
5.2 Evaluating the max input length that PrefillOnly and baselines can handle under various hardware setups. WL1 indicates the post recommendation workload, and WL2 indicates the credit verification workload. × means that the max input length is insufficient to run corresponding workload. 50
5.3 The hardware and the corresponding LLM. 53

ACKNOWLEDGMENTS

Computer science itself is mostly objective — there are typically clear metrics to demonstrate success. However, a Ph.D. in computer science — the journey of developing and extending the boundaries of computer science — is deeply subjective and closely tied to the people I met along the way. It is about immersing myself in the research community and industry, building bonds with brilliant minds, drawing inspiration from conversations, observing both academia and industry to understand underlying trends and important problems, believing in an idea enough to take a leap of faith and keep pushing despite technical and personal challenges — while also doubting it just enough to let others critically examine it, revealing limitations and faulty assumptions in me and my work. It is about striving for the best work possible, while accepting that no system is unconditionally better than all others.

I want to express my deep gratitude to the community that supported me and my Ph.D. journey.

First, I feel incredibly fortunate to have worked with my advisor, Professor Junchen Jiang. He gave me broad freedom to pursue the research ideas I was passionate about, while also providing concrete advice and support to guide me throughout my PhD journey. His support extended far beyond research — many of my internships, including those at Microsoft Research and UC Berkeley, would not have been possible without his help.

Second, I want to thank Professor Ion Stoica. He taught me the importance of choosing the right problems and connecting with people working on real workloads and cutting-edge challenges, in both academia and industry.

Third, I would like to thank Ganesh Ananthanarayanan. He provided invaluable feedback on my video analytics research and helped me appreciate that great systems should be grounded in the characteristics of the workload and application itself, rather than being driven solely by algorithmic considerations.

I am more than fortunate to be guided by my PhD thesis committee, Junchen Jiang, Ion

Stoica, Ganesh Ananthanarayanan, Shan Lu and Ari Holtzman. Their valuable insights and feedback helped me to put together this thesis.

I also want to thank my lab mates — Xu Zhang, Zhengxu Xia, Xin Yuan, Yihua Cheng, Yuhan Liu, Yitian Hao, Siddhant Ray, Jiayi Yao, Haodong Wang, Yuyang Huang, Shaoting Feng, Qizheng Zhang, and Hanchen Li — for their support in research and beyond. Research can be tough and at times discouraging, but fortunately, we supported each other.

Additionally, I want to thank the vLLM team — Zhuohan Li, Woosuk Kwon, Simon Mo, Lily Liu, Kaichao You, Chen Zhang — and colleagues at Berkeley — Yifan Qiao, Yang Zhou, Jiarong Xing, Shu Liu, Bowen Wang — for their guidance and support on the vLLM project and LLM research.

A special thanks goes to my roommate, Zixuan Linliu, for driving me around, helping me explore Chicago, and teaching me how to cook.

I am also grateful to all my brilliant coauthors for their help in refining ideas and experiments. System paper requires tons of work, and it would not be possible for me to get my Ph.D. without their help.

I want to thank those who supported me before my PhD, including my previous advisors (Professor Jiaying Liu and Professor Chenren Xu), and my Olympiad in Informatics coach, Xiaoguang Wang. Without them, I would not have the chance of getting admitted by the Ph.D. program.

I want to say thank you to my family (both my parents and my aunt's family) for their ongoing support and understanding, especially during the COVID period. Specifically, I want to thank my grandmother, Xiulan Xu, who gave me unconditional love and profoundly shaped who I am. I wish she could see me get my Ph.D. The PhD journey is often a time when people begin to confront the reality of mortality — not just for me, but for many others pursuing their PhDs. I wish that all my loved ones and all Ph.D. students would find happiness, good health, and most importantly to be brave and unafraid.

I sincerely appreciate everyone who helped me during my Ph.D. journey. It was a long journey, and thank you all for your help.

At last, I want to acknowledge the funding and awards that supported my projects during my Ph.D., including NSF CNS 2146496, 2131826, 2313190, 1901466, UChicago CERES Center, Google Faculty Research Award and Siebel Scholarship.

ABSTRACT

Today, a wide range of machine learning applications, including video analytics applications and large language model (LLM) applications, are becoming distributed. Specifically, they both require loading the data from a data source to the machine learning model for efficient inference. In this thesis, we focus on two concrete applications. Video analytics, where the analytical DNNs need to load the video feeds from remote camera, and LLM inference, where the LLM inference engine need to load the KV caches from storage for faster inference. Our observation is that, by properly identifying important part of the data and loading them with high priority, the latency of the end-to-end pipeline can be greatly improved without sacrificing the other performance metrics (accuracy in video analytics, and throughput in LLM serving). Concretely, the pixels associated with objects in video analytics are more important than others, and similarly, the KV caches associated with requests with lower JCT are more important than others. However, existing works either estimates this importance too slowly or too inaccurately. This thesis leverages application-driven insights to quickly identify the important data with high accuracy. Our evaluation shows that we can reduce the latency by 2-3× without sacrificing accuracy or throughput.

CHAPTER 1

INTRODUCTION

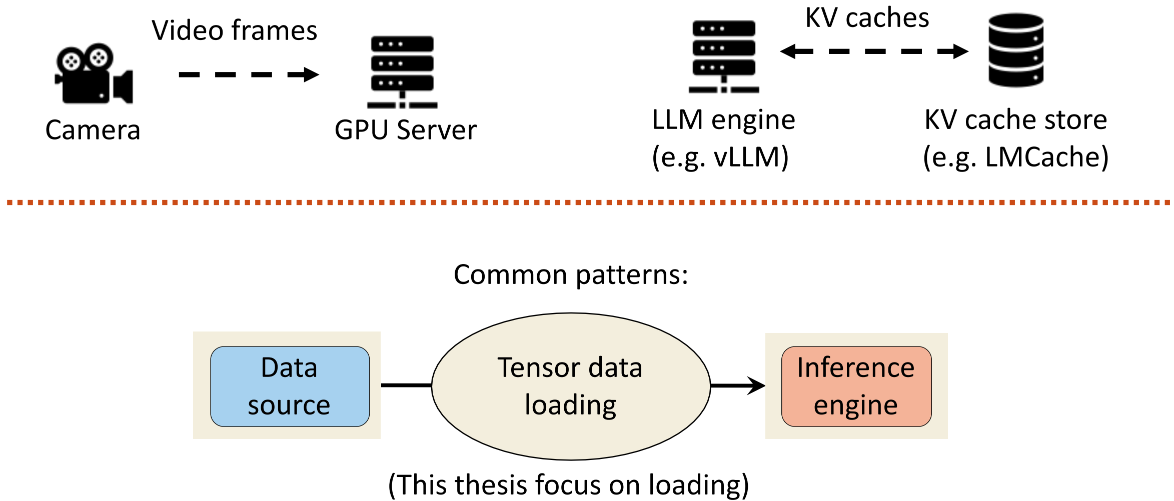


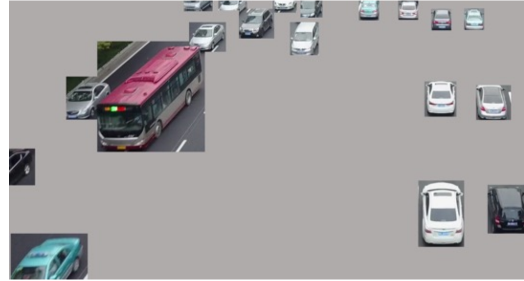
Figure 1.1: Machine learning pipelines are becoming distributed, and this thesis optimizes for data loading

Machine learning applications are increasingly becoming distributed (Zhong et al. [2024], Du et al. [2020, 2022], Li et al. [2020], Zhang et al. [2015], Chen et al. [2015], Zhang et al. [2022a,b], Liu et al. [2019], Du et al. [2025], Jiang et al. [2018], Liu et al. [2024], Yao et al. [2025]). We show this trend in Figure 1.1. This is because these applications need to load the data from a remote source (*e.g.,* camera video feeds in video analytics applications, or KV caches in large language model applications) in order to minimize the end-to-end latency of the inference system, while maximizing the performance objectives (accuracy in video analytics applications, and throughput in LLM inference applications).

Prior work shows that the latency of inference system can be greatly reduced by prioritizing the loading of critical data (Du et al. [2020, 2022], Li et al. [2020], Zhang et al. [2015], Chen et al. [2015], Zhang et al. [2022a,b], Liu et al. [2019], Du et al. [2025]). Concretely,

If the receiver runs car detection DNN…

Raw video
Car detection accuracy: 100%



Ideal compression
Only encode 20% pixel
Car detection accuracy: 100%

Figure 1.2: Illustrating the compression opportunity of video analytics application. Only 20% pixels need to be encoded, indicating a 5× compression potential.

in video analytics applications, we can stream object-relevant pixels at high fidelity while aggressively compressing irrelevant background pixels. This significantly reduces the size of the video (and thus reduces the streaming latency), while still allowing deep neural networks (DNNs) to accurately detect objects, as shown in Figure 1.2. Similarly, in large language model (LLM) applications, loading key-value (KV) caches associated with shorter job completion times (JCT) first significantly reduces the waiting delay of the requests, and thus significantly reduces the end-to-end delay when requests start to buffer.

However, existing approaches often struggle to identify such important data timely and accurately.

For example, in video analytic applicaitons, prior work identifies the important regions by using cheap object detection DNNs at the camera side (Chen et al. [2015], Du et al. [2022], Zhang et al. [2022a]). These cheap detectors cannot effectively detect small objects, because they aggressively lower the resolution of intermediate representations in the object detector to cut down the cost. Alternatively, prior work also performs profiling at the server side to

profile the video codec encoding parameters (Zhang et al. [2018, 2022a], Jiang et al. [2018]). However, such profiling is very slow — it takes minutes to finalize the profiling, which is too slow for, for example, dashcam, since the car can drive for miles away within minutes.

Similarly, in LLM applications, prior work cannot identify the job completion time (JCT) accurately, because the output length of LLM requests is highly uncertain and even non-deterministic.

Thesis statement: this thesis leverages concrete, application-driven insights to identify important data quickly and accurately. Specifically, we present and evaluate three approaches that instantiate this principle:

1. **DDS (DNN-Driven Streaming) (Du et al. [2020])**: In video object detection applications, we identify that most of the object detection DNNs inherently generate regions (*e.g.,* region proposal) that can be repurposed to propose regions that requires high-quality video encoding. To reduce the cost of obtaining such region proposal, DDS leverages an iterative approach, by first streaming low-quality videos to the object detection DNN, and then selectively re-transmitting important regions in high-quality. This allows DDS to significantly reduces the bandwidth consumption, while preserving high inference accuracy.

   That said, DDS has its own limitation: it only works for object detction DNNs. To further identify important data for wider range of DNN models, we propose our next work.

2. **OneAdapt (Du et al. [2023])**: To generalize across a wide range of DNN models, we leverage the differentiable nature of DNNs to quickly estimate which part of video data highly correlates to the inference results, by calculating the gradient of the inference results with respect to the input via backpropagation. To reduce the overhead of our backpropagation-based approach, we backpropagate only on one frame per video

chunk. We further have tailored techniques to reduce the backpropagation cost on convolution-based neural networks.

3. **PrefillOnly (Du et al. [2025])**: In LLM application, we leverage the application property of prefill-only workloads. Concretely, the prefill-only workload only outputs one token for each request, which greatly reduce the uncertainty of decoding length and potentially enables accurate JCT estimation. We further proposes continuous JCT estimation to cope with potential changes in the changes of the data source (*i.e.,* the set of available prefix caches).

Our evaluations shows that our approachs consistently reduces the latency by $2 \rightarrow 3\times$ without hurting the application-side performance (accuracy in video analytics and throughput in LLM serving).

In summary, by leveraging application-driven insights to quickly and accurately identify important data, we significantly reduces the latency of distributed machine learning inference systems, without hurting accuracy or throughput.

CHAPTER 2

MOTIVATION

Nowadays, machine learning applications are becoming distributed, due to the rising need of loading data into machine learning model for inference. In video analytics applications, people need to load thousands of video feeds from traffic camera, dashcam or drone camera to the centralized datacenter for effective inference. In large language model (LLM) applications, one need to load the KV caches, an intermediate data generated by the LLM model that can be reused across requests if they share the same prefix, to save the computation cost.

Prior work shows that the latency of inference system can be greatly reduced by prioritizing the loading of important data segments. However, they fail to timely identify the important data with high accuracy. To provide concrete discussion on this issue, we discuss the literature of video analytics applications and LLM applications separately.

2.1 Video Analytics

Video analytics applications, such as traffic analytics, autonomous driving, drone analytics are increasingly rely on distributed inference due to the proliferation of affordable, high-definition, network-connected cameras. These cameras are often deployed across various locations (*e.g.,* crossroads, highways, and drones) with limited LTE upload bandwidth, which forces the camera to compress the video feeds before uploading them to the server. As a result, the major part of latency in distributed video analytics is the streaming latency.

To minimize the streaming latency, the key opportunity is that we can aggressively compress the pixels that are not related to the objects without hurting the object detection results.

2.1.1 Limitations of previous work

Here, we categorize previous work in four general approaches and explain why they cannot meet high accuracy and low latency simultaneously.

Local frame-filtering schemes: One of the popular techniques is to let the camera run a simple logic to identify which frames are irrelevant to the vision task and thus can be discarded (Li et al. [2020], Canel et al. [2019], Chen et al. [2015]) or encoded in low quality (Zhang et al. [2018]). This approach works well when the video content is relatively stationary, where the incidents/objects of interest are rare and easy to detect; *e.g.,* in wildlife camera feeds, animals are rare and readily detectable since they are the only moving objects on a static background. However, for frames that are not discarded, this approach encodes the entire frames with uniform quality, which can be suboptimal, since the objects of interest often occupy only a small fraction of each frame (Du et al. [2020], Liu et al. [2019]), leading to higher streaming delays than necessary.[1]

Local heuristics to lower background quality: Since objects of interest often account for a small fractions of each frame, some work (Zhang et al. [2015], Dai et al. [2021]) uses local heuristics to filter out (or lower the quality of) the background pixels and sends the remaining object-related pixels in high quality to the server-side final DNN. However, these local heuristics are constrained by the limited camera-side compute resources, giving rise to false negatives—object-related pixels are treated as background and thus filtered out or sent in low quality, causing the DNN to miss objects of interest. For instance, to detect potential object-related regions, Vigil (Zhang et al. [2015]) relies on a low-accuracy non-convolutional Haar-cascade-classifier-based object detector, and CiNet (Dai et al. [2021]) uses a very shallow convolutional network (with only 2 convolutional layers, 1 average pooling layer and 2 fully-connect layers) designed to handle only few objects per frame. There are also

1. Although CloudSeg (Wang et al. [2019]) does not perform frame filtering at the camera side, it also shares the limitation since it compresses entire frames in same encoding quality.

deeper NNs such as MobileNet-SSD (Howard et al. [2017]) that run on resource-constrained cameras, but they have to downsize frames to low resolutions (*e.g.,* 300×300) for real-time inference, thus prone to missing small objects.

Server-driven compression: To overcome the camera-side resource constraints, another approach (Du et al. [2020], Liu et al. [2019], Zhang et al. [2021]) leverages the abundant server-side compute resources to generate feedback on how videos should be encoded. This approach generally compresses videos efficiently while achieving high accuracy, but it suffers from a high inference delay. DDS (Du et al. [2020]), for instance, sends a low-quality video to the server-side DNN which returns to the camera which regions must be encoded in high quality, but under high network latency, getting such server-driven feedback can take at least two network round-trip times before the camera can actually encode the video for final DNN inference, causing high inference delay on each frame.

Local DNN compression: Instead of encoding videos on the camera, some proposals also extract the final DNN's feature maps on the camera and compress the feature maps which might contain less information than the original raw frames (Duan et al. [2020], Xia et al. [2020], Emmons et al. [2019], Kang et al. [2017], Matsubara et al. [2019]). While this approach has shown promise with classification or action recognition DNNs (Kang et al. [2017], Emmons et al. [2019], Duan et al. [2020]), these tasks do not require the spatial locations of objects, allowing aggressive aggregation of feature maps over an entire frame. In contrast, the vision tasks that we focus on (*e.g.,* object detection, semantic segmentation) are sensitive to object locations, making the intermediate feature maps much larger and much more difficult to be compressed efficiently. For example, many state-of-the-art object detectors (Detectron2) use expensive feature extractors such as ResNet101 (He et al. [2016]), and if we feed a 720p (1280×720) frame through even parts (*e.g.,* 90) of its convolution layers, the feature map still contains 2×10^7 floating-point numbers per frame, 20x more than the number of pixels in the original frame.

There are also proposals to train DNN autoencoders that compress video to a smaller size than the popular video codecs do, but these DNNs are much more compute-intensive than the video codecs and even the final DNN. For example, NLAM (Liu et al. [2020]) requires performing expensive 3D convolutions on videos for more than 30 times, while an object detector backbone (He et al. [2016]) only performs 2D convolution for 34 times.

2.2 Large Language Model applications

Nowadays, large language model (LLM) applications are getting more and more popular. As a result, people build LLM inference system to serve LLM requests with low latency.

In terms of maxially reduce the average latency, the most famous results from traditional scheduling literature is that shortest job first: if we could estimate the job completion time (JCT in short), we can put the requests with shorter JCT first, which will lead to much lower average latency overall.

However, the challenge is that the JCT of LLM requests are very difficult to estimate due to its non-deterministic output length (Zheng et al. [2023]). As a result, the key to minimize the LLM serving latency is to accurately estimate the JCT of the incoming requests.

CHAPTER 3

DDS: SERVER-DRIVEN STREAMING FOR OBJECT DETECTION

Traditional video streaming systems adopt a *source-driven* architecture, where the video source (e.g., camera) determines how to compress and transmit video frames based on local heuristics. While such designs are sufficient for human viewers or coarse machine analytics, they are suboptimal for deep learning–based video inference tasks such as object detection. This is because inference accuracy depends heavily on complex server-side DNNs, and the camera—often compute-limited—cannot accurately infer which parts of the video are critical for DNN accuracy.

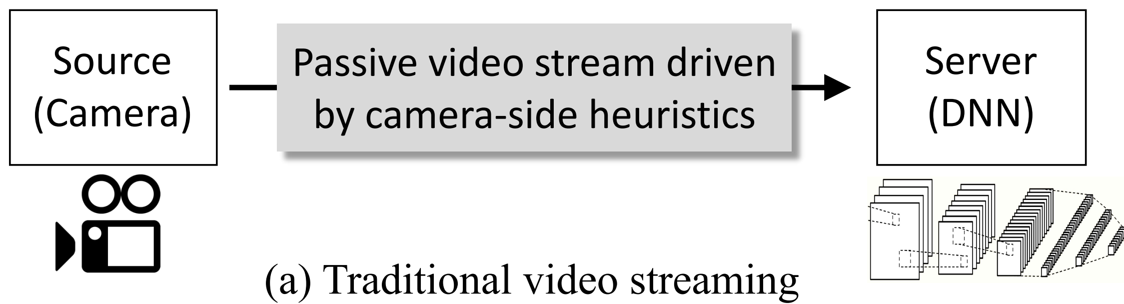
DDS (DNN-Driven Streaming) introduces a *server-driven* paradigm where compression and streaming decisions are guided by real-time feedback from the server-side DNN. The core insight is that although low-quality video may reduce DNN accuracy, it still often contains enough information to identify regions likely to be important for inference. DDS leverages this by using an initial low-quality stream to infer where to focus subsequent high-quality transmission, thereby reducing bandwidth without sacrificing accuracy.

3.1 DNN-Driven Video Streaming

In this section, we present the design of DDS and discuss its design rationale and performance tradeoffs.

3.1.1 Overall workflow

We explore an alternative approach, called *DNN-driven streaming* (DDS). In DDS, the compression and streaming behaviors are driven by the feedback judiciously generated by the server-side DNN, rather than the low-complexity local heuristics on the camera side, in or-



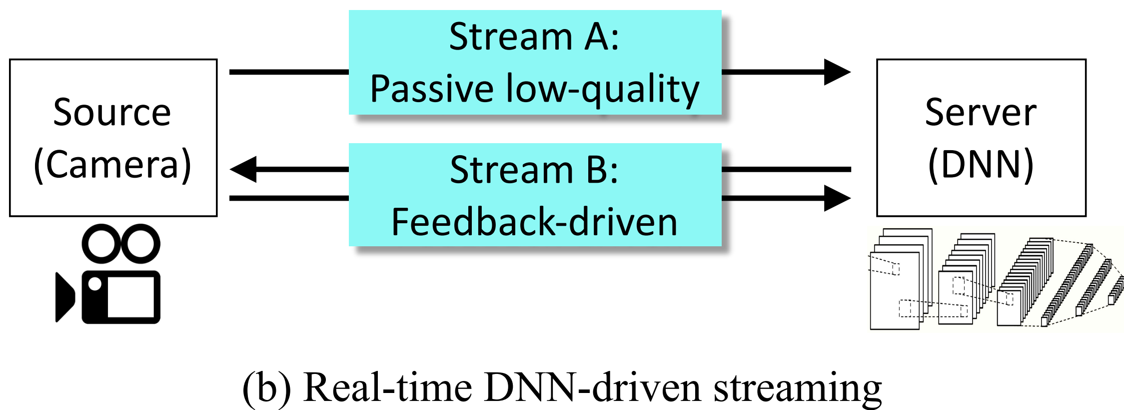


Figure 3.1: Contrasting the new real-time DNN-driven streaming (iterative) with traditional video streaming in video analytics.

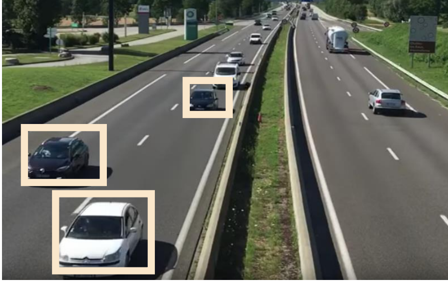
der to capture what the analytics engine needs from the real-time video content. Figure 3.1 contrasts the workflow of DDS with that of the traditional source-driven approach: source-driven streaming is "single-shot" (*i.e.,* camera using simple heuristics to determine how the video should be streamed out), but DDS is *iterative* and logically contains two streams:

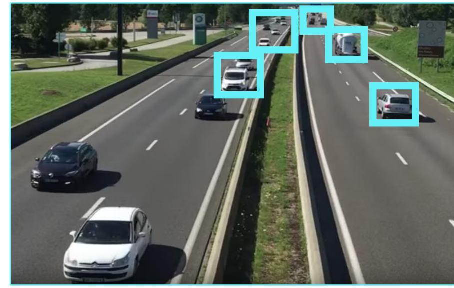
- **Stream A (passive, low quality):** The camera continuously sends the video in low quality to the server.
- **Stream B (feedback-driven):** The server frequently (*e.g.,* every handful of frames) extracts the *feedback regions* from the DNN outputs on the Stream A video and sends them back to the camera as feedback. Upon receiving the feedback from the server, the camera then re-encodes the recent history video accordingly and sends it to the server for a second-round inference on these "zoomed-in" images.

The key to DDS's success is the design of the feedback regions, which we discuss next.

3.1.2 Feedback regions

Most object detection DNNs are anchor-based (though some are anchor-free (Carion et al. [2020])). This means that a DNN will first identify regions that might contain objects and then examine each region. Each proposed region is associated with an objectness score that indicates how likely an object is in the region. For DNNs (*e.g.,* FasterRCNN-ResNet101 (Ren et al. [2017])) that use region proposal networks (RPNs), each proposed region is directly associated with an objectness score. However, not all object-detection DNNs use RPNs. For instance, Yolo (Redmon and Farhadi [2017]) does not and instead, it assigns a score for each class in each region in the final output. In this case, we sum up the scores of non-background classes as the objectness score, which indicates how likely a region includes a non-background object. We keep regions with objectness score over a threshold (*e.g.,* 0.5 for FasterRCNN-ResNet101). From these high-objectness regions, we apply two filters to remove those that are already in the DNN output on the low-quality video (Stream A).





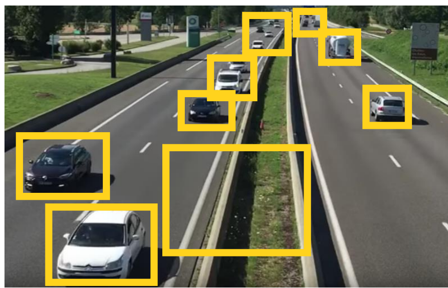


Figure 3.2: Illustration on how DDS generates feedback regions on two types of applications.

First, we filter out those regions that have over 30% IoU (intersection-over-union) overlap with the labeled bounding boxes returned by DNN on the low-quality video. We empirically pick 30% because it works well on all the videos in our dataset. Second, we remove regions that are over 4% of the frame size (roughly 20% of each dimension), because we empirically find that if an object is large, the DNN should have successfully detected it. The remaining region proposals (bounding boxes) are used as feedback regions.

Figure 3.2 shows an example: there are nine bounding boxes in high-objectness-score results, three of which overlap with inference results in Stream A and one of which is too large. The remaining five regions are the feedback regions.

Name	*Vision tasks*	*Total length*	*#videos*	*#objs/IDs*
Traffic	obj detection	2331s	7	24789
Drone	obj detection	163s	13	41678
Dashcam	obj detection	5361s	9	24691

Table 3.1: Summary of DDS datasets.

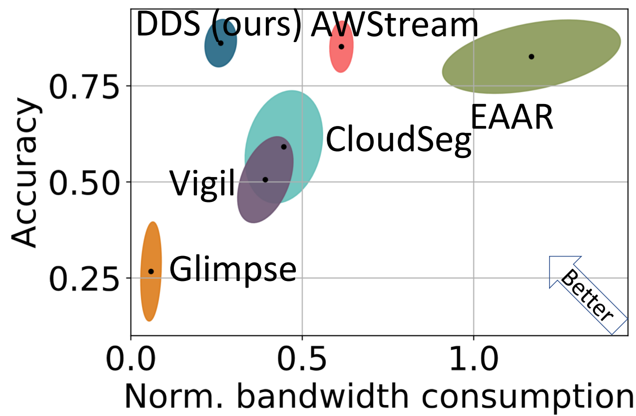
3.2 Evaluation

3.2.1 Evaluation setup

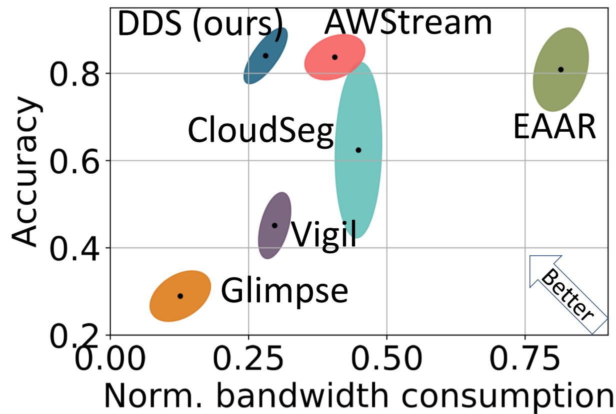
We build an emulator of video streaming that can measure the exact analytics accuracy and bandwidth usage. Although existing video analytics platforms might support DDS, we implement and test DDS and all baselines in our emulator for a fair comparison. It consists of a client (camera) that encodes/decodes locally stored videos and a fully functional server that runs any given DNN and a separate video encoder/decoder. We run DNN inference on RTX 2080 super and all other computations on Intel Xeon Silver 4100. Unless stated otherwise, we use FasterRCNN-ResNet101 (Ren et al. [2017]) as the server-side DNN for object detection.

When needed, we vary video quality along the quantization parameter and the resolution, and DDS uses 36 (QP) as low quality and 26 (QP) as high quality, with resolution scale set to 0.8 in object detection and 1.0 in semantic segmentation. We do not consider the network cost of AWStream to profile the accuracy-bandwidth relationships under different QP-resolution combinations. This makes the AWStream bandwidth usage is strictly less than its actual one. We assume a stable network connection.

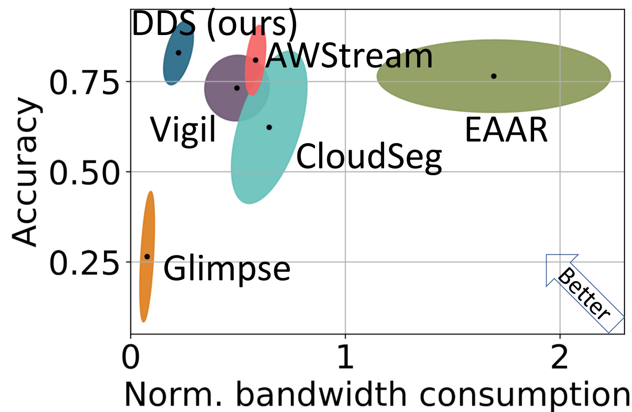
Datasets: To evaluate DDS over various video genres, we compile five video datasets each representing a real-world scenario (summarized in Table 3.1 and their links can be found in (dds)). These videos are obtained from two public sources. First, we get videos from *aiskyeye* (ais), a computer-vision benchmark designed to test DNN accuracies on drone videos. Nonetheless, DDS and the baselines can be affected by factors such as fraction of



(a) Object detection on traffic camera



(b) Object detection on dashcam



(c) Object detection on drone camera

Figure 3.3: The normalized bandwidth consumption v.s. inference accuracy of DDS and several baselines on various video genres and applications. DDS achieves high accuracy with 55% bandwidth savings on object detection and 42% on semantic segmentation, and 36% on face recognition. Ellipses show the 1-σ range of results.

frames with objects of interest or size of the regions with objects. Therefore, we try to cover a range of values along these factors (including objects of various sizes and frames with various number of objects) by adding YouTube videos as follows. We search keywords (*e.g.,* "highway traffic video HD") in private browsing mode (to avoid personalization biases); among the top results, we manually remove the videos that are irrelevant (*e.g.,* news report that mentions traffic), and we download the remaining videos in their entirety or the first 10-minutes (if they exceed 10 minutes). The vision tasks are (1) to detect (or segment) vehicles in traffic and dashcam videos, (2) to detect humans in drone videos, and (3) to recognize identities in sitcom videos. Because many of these videos do not have human-annotated ground truth, for fairness, we use the DNN output on the full-size original video as the reference result to calculate accuracy. For instance, in object detection, the accuracy is defined by the F1 score with respect to the server-side DNN output in highest resolution (original) with over 30% confidence score.

Baselines: We use five baselines to represent two state-of-the-art techniques: camera-side heuristics (Glimpse (Chen et al. [2015]), Vigil (Zhang et al. [2015]), EAAR (Liu et al. [2019])) and adaptive streaming (AWStream (Zhang et al. [2018]), CloudSeg (Wang et al. [2019])). We made a few minor modifications to ensure the comparison is fair. First, all baselines and DDS use the same server-side DNN. Second, although DDS needs no more camera-side compute power than encoding, camera-side heuristics baselines are given sufficient compute resource to run more advanced tracking (Henriques et al. [2014]) and object detection algorithm (Sandler et al. [2018]) than what Glimpse and Vigil[1] originally used, so these baselines' performance is strictly better than their original designs. Third, all DNNs used in baselines and DDS are pre-trained (*i.e.,* not transfer-learned with samples from the test dataset); In particular, our implementation of CloudSeg uses the pre-trained super-

1. Our implementation of Vigil does not include the optimization of setting the background pixels in same RGB color, but in a separate experiment, we find that on the object detection videos (Table 3.1), this optimization only reduces Vigil's bandwidth usage by 10-20% and leads to similarly low accuracy.

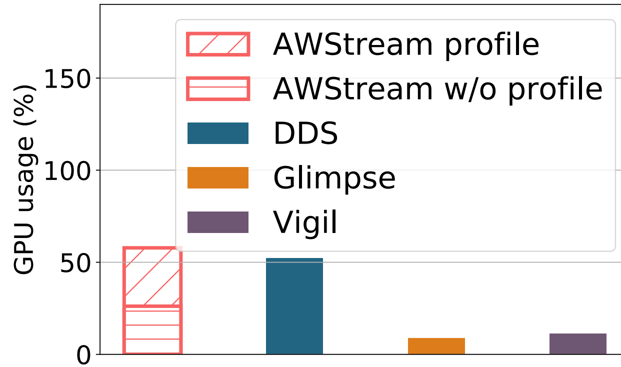
resolution model (Ahn et al. [2018]) from the website (car). This ensures the gains are due to the video streaming algorithm, not due to DNN fine-tuning, and it also helps reproducibility. Finally, although DDS could lower the frame rate, to ensure that the accuracies are always calculated on the same set of images, we do not sample frames and only vary the resolution and QP in DDS.

Performance metrics: We use accuracy and average response delay. To avoid impact of video content on bandwidth usage, we report bandwidth usages of DDS and the baselines after normalizing them against the bandwidth usage of each original video.

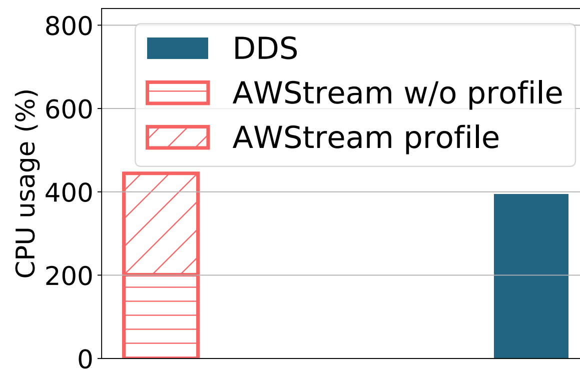
3.2.2 Evaluation results

We start with DDS's overall performance gains over the baselines along bandwidth savings, accuracy, and average response delay.

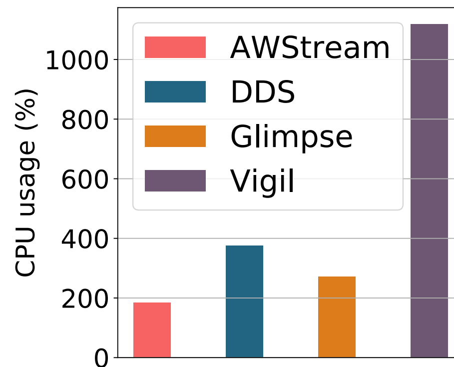
Bandwidth saving: Figure 3.3 compares the bandwidth-accuracy tradeoffs of DDS with those of the baselines, when the total available bandwidth is set to match the size of the original video. DDS achieves higher or comparable accuracy than AWStream and CloudSeg but uses 55% less bandwidth in object detection. Glimpse and Vigil do sometime use less bandwidth but they have much lower accuracy. Overall, even if DDS is less accurate or uses more bandwidth, it always has an overwhelming gain on the other metric. Figure 3.3 compares the bandwidth-accuracy tradeoffs of DDS with those of the baselines. In each application, we use a fixed DDS configuration and normalize the bandwidth usage against the size of the highest-quality videos (which we use to derive the ground truth). We also lower the frame rate to 1 FPS to speed up the experiments. Across three vision tasks, DDS achieves higher or comparable accuracy than AWStream but uses 55% less bandwidth in object detection. Glimpse sometimes uses less bandwidth but has much lower accuracy. Vigil, Glimpse, CloudSeg and EAAR consumes more bandwidth than DDS with lower accuracy. Overall, even if DDS is less accurate or uses more bandwidth, it always has an overwhelming



(a) Server GPU



(b) Server CPU



(c) Client CPU

Figure 3.4: Compared to prior solutions, DDS has low additional systems overhead on both client and server.

gain on the other metric.

Camera-side and server-side overheads: Figure 3.4 compares the systems overheads of DDS with the baselines. We benchmark their performance on our server, with one RTX

2080 Super and one 16-core Intel Xeon Silver 4100. We scale the CPU and GPU usage by normalizing their runtime (*e.g.,* 2x runtime on the same number of fully used CPUs will be translated to 2x more CPU usage). Figure 3.4 shows that compared to AWStream (when the profiling cost is excluded), DDS has 2x more overheads at both client-side CPU (the CPU usage may exceed 100% since it may leverage more than one CPU cores), server-side CPU and server-side GPU. This is because DDS invokes extra encoding, decoding and inference costs in Stream B. However, the profiling cost of AWStream is a substantial server-side CPU and GPU costs. We estimate it by profiling 30 configurations (compared to 216 claimed in (Zhang et al. [2018])) over a 10-second video every 4 minutes (which is much less often than profiling a 30-second video every 2 minutes as used in (Zhang et al. [2018])), the server-side CPU and GPU costs of AWStream have already become higher than DDS. That said, we acknowledge that if AWStream updates the profile less frequently (*e.g.,* every tens of minutes), its GPU usage could be lower than DDS, but that might cause its profile to be out of date and less accurate. On the server side, both Vigil and Glimpse incur minimal CPU overheads (since they do not need to decode the video) and much less GPU overheads than DDS and AWStream (since their camera-side logics reduce the need for server-side inference).

CHAPTER 4

ONEADAPT: GENERATE REGION-OF-INTEREST VIA BACKPROPAGATION

In distributed video analytics, reducing inference costs while maintaining accuracy is essential for deploying real-time applications at scale. Traditional solutions either rely on fully offloading inference to the cloud, which can be bandwidth- and latency-intensive, or processing entirely on edge devices, which are compute-limited. **OneAdapt** proposes a hybrid strategy: use the server to process only a small number of video frames and then adaptively offload future frames to the edge based on the similarity to previously processed frames.

The core idea is that many frames in a video stream exhibit high temporal and spatial redundancy. If two frames are visually similar and the scene dynamics remain stable, their DNN inference results are also likely to be similar. OneAdapt exploits this by learning to detect such similarity online, allowing the system to reuse inference results from server-processed frames for similar edge frames without re-running the DNN.

4.1 Design of OneAdapt

We present OneAdapt, a configuration adaptation system that aims at improving along the three requirements mentioned above — frequent, near-optimal adaptation on various configuration knobs. The basic idea of OneAdapt is to frequently estimate the gradient of how inference accuracy changes with each configuration knob, which we refer to as AccGrad. OneAdapt then performs gradient ascent based on AccGrad to update the configuration, with an objective metric that combines inference accuracy and resource usage (§4.1.2). To efficiently estimate AccGrad, OneAdapt approximates it by estimating OutputGrad, another metric that can be efficiently calculated by leveraging DNN's inherent differentiability (§4.1.3) and statistically correlated with AccGrad (§4.1.4). Finally, we will discuss the caveats

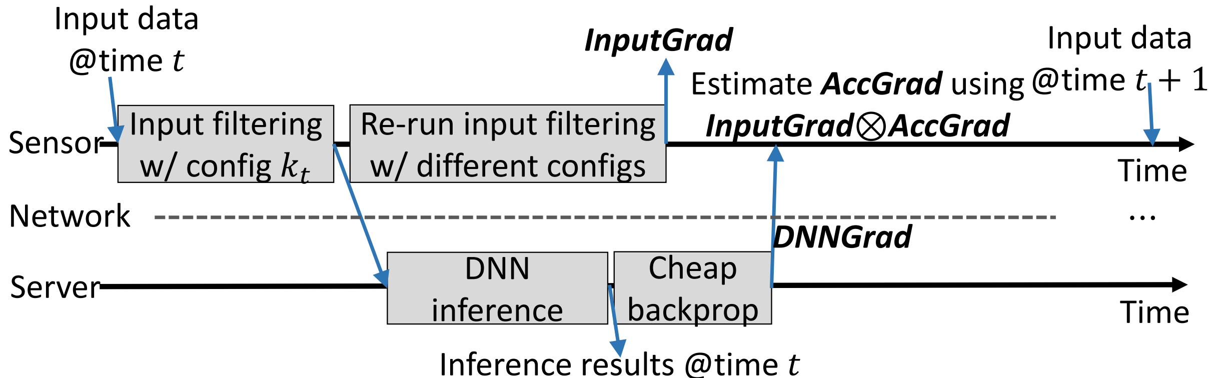


Figure 4.1: Illustrating how OneAdapt estimates AccGrad using InputGrad and DNNGrad using the sensor and the server.

of OneAdapt's gradient-based adaptation (§4.1.5).

4.1.1 Terminology

To adapt configuration for various DNN pipelines , we introduce a unified terminology to describe their input and output. Table 4.1 summarizes the notations and their meanings.

We split the input data into fixed-length (by default, one-second-long) intervals, called *adaptation intervals*. In the tth adaptation interval, the input data $\mathbf{x}_t$ is first preprocessed with the current configuration $\mathbf{k}_t$ (*e.g.,* video resolution and frame rate) to get the DNN input $\mathbf{y}(\mathbf{k}_t;\mathbf{x}_t)$, and the DNN takes it as input and returns $Res(\mathbf{k}_t;\mathbf{x}_t)$ as output. DNN output $Res(\mathbf{k}_t,\mathbf{x}_t)$ contains multiple *elements*. For the DNNs in our considered tasks, each element is associated with a confidence *score*, and only elements with a score over a confidence *threshold* θ will be counted towards accuracy.

The definitions of an element e and input data $\mathbf{x}$ vary with the considered application. An element is a detected object in object detection, a detected text in audio-to-text translation, or a segmented mask of a detected instance in instance segmentation. Input data can be a sequence of RGB frames in videos, point-cloud frames in LiDAR, or an audio wave segment in audio data.

Notation	Definition	Example
n	Number of knobs	$n = 2$
t	t^{th} adaptation interval (by default, each interval is 1 second)	$t = 1$
T	Total number of adaptation intervals	$T = 60$
$\mathbf{k}_t$	Configuration: a vector of knobs with their selected values at interval t	$\mathbf{k}_1$ =(frame rate=10, resolution=480p)
$k_{i,t}$	The value of ith knob in configuration $\mathbf{k}_t$	$k_{2,1}$=480p
Δk_i	A small increase on the ith knob	Δk_2=120p
$\mathbf{x}_t$	input data at interval t	
$\mathbf{r}(\mathbf{k}_t; \mathbf{x}_t)$ or $\mathbf{r}(\mathbf{k}_t)$	Resource usage of config $\mathbf{k}_t$ under $\mathbf{x}_t$. We omit $\mathbf{x}_t$ for simplicity	$\mathbf{r}(\mathbf{k}_1)$ =5Mbps
$Res(\mathbf{k}_t; \mathbf{x}_t)$ or $Res(\mathbf{k}_t)$	Inference results of config $\mathbf{k}_t$ under $\mathbf{x}_t$. We omit $\mathbf{x}_t$ for simplicity.	$Res(\mathbf{k}_1) = \{$ (obj1, "car", score: 0.2), (obj2, "bike", score: 0.8) $\}$
e	An element in the inference results (*e.g.,* a detected object). Each element is associated with a confidence score.	$e =$ (obj1, "car", score: 0.2)
θ	Confidence threshold	$\theta = 0.5$ (default)
$\mathbf{y}(\mathbf{k}_t; \mathbf{x}_t)$ or $\mathbf{y}(\mathbf{k}_t)$	DNN input generated by configuration $\mathbf{k}_t$ using input data $\mathbf{x}_t$. We omit $\mathbf{x}_t$ for simplicity.	
$\mathbf{z}(\mathbf{k}_t; \mathbf{x}_t)$ or $\mathbf{z}(\mathbf{k}_t)$	Output utility: number of above-confidence-threshold elements. We omit $\mathbf{x}_t$ for simplicity.	$\mathbf{z}(\mathbf{k}_1) = 1$
$Acc(\mathbf{k}_t; x_t)$ or $Acc(\mathbf{k}_t)$	Accuracy of the configuration $\mathbf{k}_t$ under input data $\mathbf{x}_t$, defined as the similarity between the current inference result $Res(\mathbf{k}_t)$ and the inference results generated using the most resource-demanding configuration.	$Acc(\mathbf{k}_1) = 100\%$
α	Learning rate of OneAdapt	$\alpha = 0.5$ (default)
λ	The hyperparameter that trade-off between accuracy and resource usage.	$\lambda = 1$

Table 4.1: Summary of the notations used in OneAdapt

Based on these notations, we define:

- $Acc(\mathbf{k}_t; \mathbf{x}_t)$ is the accuracy of the inference results generated using input data $\mathbf{x}_t$ and configuration $\mathbf{k}_t$. Following prior work (Zhang et al. [2018], Jiang et al. [2018], Du et al. [2020, 2022], Bhardwaj et al. [2022], Agarwal and Netravali [2023]), we define accuracy as the ***similarity*** between the current inference results and the inference results generated from the most resource-consuming configuration $\mathbf{k}^*$.[1]
- $\mathbf{z}(\mathbf{k}_t; \mathbf{x}_t)$ is the output utility of the inference results generated using input data $\mathbf{x}_t$ and configuration $\mathbf{k}_t$. We define output utility as the number of elements with confidence scores above the confidence threshold (see §4.1.3).
- $\mathbf{r}(\mathbf{k}_t)$ is the resource usage (bandwidth and/or GPU cycles, defined per application) at the tth adaptation interval.

We may omit the time label t and the input data $\mathbf{x}_t$ for simplicity when the corresponding variables are of the same interval as the current input data $\mathbf{x}_t$.

4.1.2 Adaptation goal and gradient-ascent

The adaptation goal of OneAdapt is to pick the configurations for all adaptation intervals such that the following weighted sum between the accuracy Acc and the resource usage $\mathbf{r}$ is maximized:

$$\sum_{t=1}^{T} \underbrace{Acc(\mathbf{k}_t)}_{\text{accuracy of } t^{th} \text{ interval}} - \lambda \underbrace{\mathbf{r}(\mathbf{k}_t)}_{\text{resource usage of } t^{th} \text{ interval}}, \tag{4.1}$$

where λ is a hyperparameter that governs the tradeoff between accuracy and resource usage: higher λ will emphasize resource-saving and lower λ will emphasize accuracy improvement. Note that this objective can be extended to perform *budgeted* optimization (*e.g.,* to maximize

1. This notion of accuracy is well studied in the literature. While it does not compare DNN output with human-annotated ground truth, it better captures the impact of adapting configurations on inference results.

accuracy under a budget of resource usage, we can add a change the term $\mathbf{r}(\mathbf{k}_t)$ to a large penalty when resource usage extends the budget). We leave this extension to future work.

To optimize towards this goal, OneAdapt uses an online ***gradient-ascent strategy*** (illustrated in Figure 4.1). Concretely, OneAdapt derives a new configuration $\mathbf{k}_{t+1}$ based on the current configuration $\mathbf{k}_t$ using the following Equation:

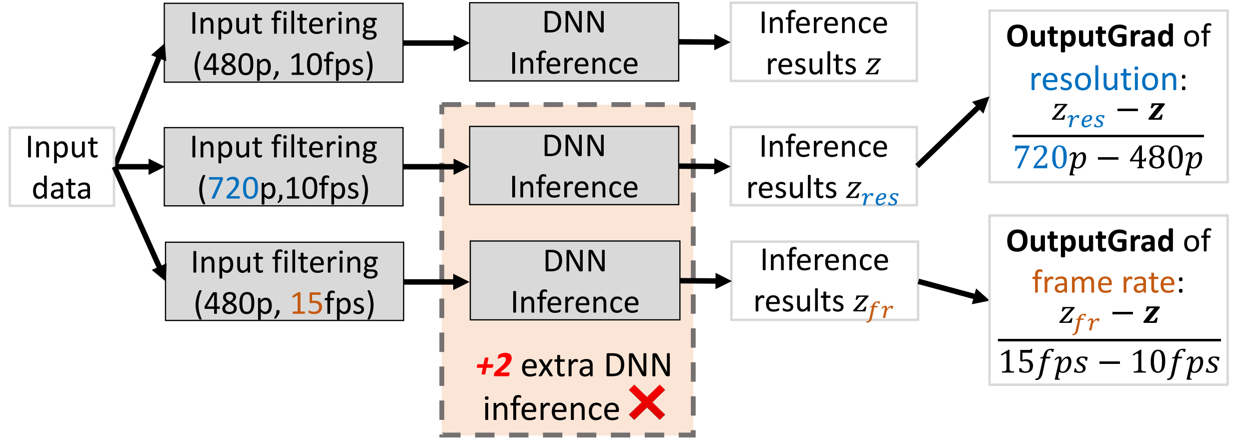
$$k_{t+1,i} = k_{t,i} + \alpha \left(\underbrace{\frac{Acc(\mathbf{k}_t + \Delta k_i) - Acc(\mathbf{k}_t)}{\Delta k_i}}_{\textsf{grad. of accuracy } (\textbf{\textsf{AccGrad}})} - \lambda \underbrace{\frac{\mathbf{r}(\mathbf{k}_t + \Delta k_i) - \mathbf{r}(\mathbf{k}_t)}{\Delta k_i}}_{\textsf{grad. of resource usage}} \right), \tag{4.2}$$

where $k_{t+1,i}$ is the configuration of the ith knob in the next adaptation interval, Δk_i is a small increase on the ith knob, α is the learning rate. We will discuss the convergence of this gradient-ascent strategy and extend it to discrete configuration values in §4.1.5. For now, we assume the configuration of each knob can be tuned continuously. Also, while more advanced gradient-based optimization (*e.g.,* Nesterov's Accelerated Gradient Descent (Nesterov [2013])) exists, OneAdapt chooses a standard gradient-ascent strategy to make sure the source of improvement is from the gradient itself instead of advanced optimization techniques.

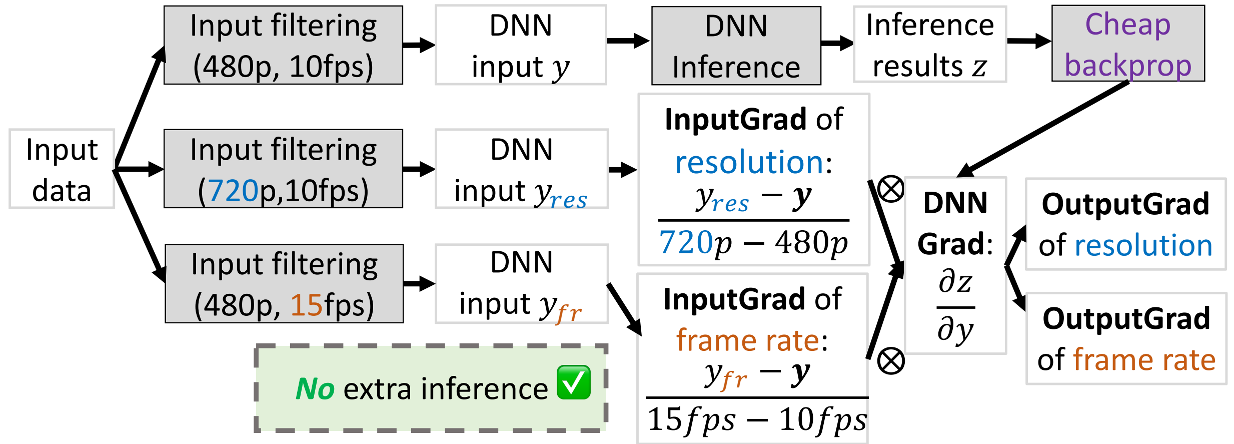
The gradient-ascent strategy in Equation 4.2 requires calculating the gradient of accuracy (hereinafter AccGrad) and the gradient of resource usage along each knob i. We calculate the gradient of resource usage by using its definition (*i.e.,* we directly calculate the bandwidth and GPU computation consumption of configuration $\mathbf{k}_t + \Delta k_i$ and then calculate$\frac{\mathbf{r}(\mathbf{k}_t+\Delta k_i)-\mathbf{r}(\mathbf{k}_t)}{\Delta k_i})$[2].

However, if we calculate AccGrad by its definition, it would be prohibitively expensive (see Figure 4.2a). Obtaining $Acc(\mathbf{k})$ of any configuration $\mathbf{k}$ will require running inference on

2. Note the calculating the resource usage of a configuration does not require DNN inference, as calculating the bandwidth usage requires no inference, and the GPU computation largely depends on the shape of DNN input rather than the content of the input.



(a) Calculating OutputGrad using its definition



(b) Calculating OutputGrad using OneAdapt

Figure 4.2: Comparison between calculating OutputGrad naively and calculating OutputGrad using OneAdapt. OneAdapt calculates OutputGrad with **no extra inference**.

the most resource-intensive configuration $\mathbf{k}^*$. Moreover, to get the AccGrad of each knob would require running an extra inference to test the impact of a small change in configuration on DNN output.

4.1.3 Fast approximation of AccGrad

To estimate AccGrad efficiently, we introduce a new metric, called OutputGrad, to approximate it. Here, we define OutputGrad and explain how to calculate it efficiently. The next subsection will show its statistical correlation with AccGrad.

DNN output utility: Recall from §4.1.1 that the DNN output contains multiple *elements*, each associated with a confidence *score*. An element can be a detected object in object detection or a detected text in audio-text translation. Since only elements with a score over the confidence *threshold* will be counted towards accuracy, we define *output utility* of a DNN output by the number of elements in an adaptation interval whose confidence scores exceed the confidence threshold. We use $\mathbf{z}(\mathbf{k}_t)$ to denote the output utility of DNN output $Res(\mathbf{k}_t; \mathbf{x}_t)$ under configuration $\mathbf{k}_t$.

Output utility offers a single-value summarization of the DNN output whose change indicates the change in inference accuracy (More rationale of output utility in §4.1.4).

OutputGrad definition: OutputGrad is defined as how much a small change in the current value of each knob changes the output utility. The OutputGrad of knob k_i can be written as

$$\frac{\Delta \mathbf{z}}{\Delta k_i} = \frac{\mathbf{z}(\ldots, k_{t,i} + \Delta k_i, \ldots) - \mathbf{z}(\ldots, k_{t,i}, \ldots)}{(k_{t,i} + \Delta k_i) - k_{t,i}}.$$

To help understand OutputGrad, we use a simple video analytics system that feeds each video frame to a vehicle detection DNN, using a confidence score of 0.5. Figure 4.2a illustrates OutputGrad in this example. Following prior work (Jiang et al. [2018], Xu et al. [2019], Zhang et al. [2017]), we assume the system can tune two configuration knobs, frame rate, and resolution. Suppose that the current configuration is 10 frames per second (fps) and 480p. When taking 1-second input, the DNN outputs 34 vehicle bounding boxes with confidence scores greater than 0.5 on these 10 frames, an average of 3.4 vehicles per frame. If we slightly increase the frame rate from 10fps to 15fps, the DNN outputs 66 vehicle bounding boxes with confidence scores greater than 0.5, on average 4.4 vehicles per frame. Then the OutputGrad of the current frame rate is

$$\frac{\Delta \mathbf{z}}{\Delta \text{framerate}} = \frac{\mathbf{z}(480p, 15fps) - \mathbf{z}(480p, 10fps)}{15fps - 10fps} = \frac{4.4 - 3.4}{15 - 10} = 0.2.$$

Similarly, the OutputGrad of the resolution is:

$$\frac{\Delta \mathbf{z}}{\Delta \text{resolution}} = \frac{\mathbf{z}(720p, 10fps) - \mathbf{z}(480p, 10fps)}{720p - 480p} = \frac{5.8 - 3.4}{720 - 480} = 0.01.$$

How to calculate OutputGrad efficiently: The insight of OneAdapt is that OutputGrad can be computed efficiently *with no extra DNN inference* by taking advantage of the **differentiability of DNNs**. As our considered application pipelines affect the inference results by altering the DNN input through knobs, the OutputGrad of a knob can be written as the inner product of two separate gradients (illustrated in Figure 4.1), each can be computed efficiently:

- DNNGrad: How the DNN's output changes with respect to the DNN's input.
- InputGrad: How the DNN's input changes with respect to the knob's configuration.

The decoupling makes the calculation of OutputGrad much more resource-efficient for three reasons:

- DNNGrad can be estimated by running one backpropagation . This is a constant GPU-time operation without extra inference as it reuses the layer-wise features computed as part of the DNN inference of the current configuration.
- DNNGrad only needs to be calculated once and can be re-used to derive OutputGrad of different knobs without extra DNN inference. This is because the DNNGrad describes DNN's sensitivity to its input and thus remains largely similar if the change in DNN input is not dramatic. For instance, the DNNGrad of an object-detection DNN is high on pixels associated with key visual features of an object.
- Finally, computing InputGrad does not require running the final DNN.

To reuse the same example from Figure 4.2a, Figure 4.2b shows how OutputGrad is calculated from DNNGrad and InputGrad:

$$\frac{\mathbf{z}_{(480p,15fps)} - \mathbf{z}_{(480p,10fps)}}{15fps - 10fps} \approx \frac{\mathbf{y}_{(480p,15fps)} - \mathbf{y}_{(480p,10fps)}}{15fps - 10fps} \otimes \frac{\Delta \mathbf{z}}{\Delta \mathbf{y}}\Big|_{\mathbf{y}=\mathbf{y}_{(480p,10fps)}},$$

$$\frac{\mathbf{z}_{(720p,10fps)} - \mathbf{z}_{(480p,10fps)}}{720p - 480p} \approx \frac{\mathbf{y}_{(720p,10fps)} - \mathbf{y}_{(480p,10fps)}}{720p - 480p} \otimes \frac{\Delta \mathbf{z}}{\Delta \mathbf{y}}\Big|_{\mathbf{y}=\mathbf{y}_{(480p,10fps)}},$$

where $\otimes$ represents inner product and $\frac{\Delta \mathbf{z}}{\Delta \mathbf{y}}|_{\mathbf{y}=\mathbf{y}_{(480p,10fps)}}$ is DNNGrad of the current configuration $(480p, 10fps)$. Figure 4.2b shows that OneAdapt saves extra GPU inference by running backpropagation once and re-using the backpropagation results for different knobs. This example has two knobs, thus the saving may seem marginal, but our evaluation shows more savings when OneAdapt is used to adapt more knobs in more realistic DNN analytics systems.

4.1.4 Relationship between OutputGrad and AccGrad

The definitions of OutputGrad and AccGrad differ, yet they are closely related both theoretically and empirically This can be intuitively explained as follows: a high OutputGrad means a great change in the inference output, which often causes a greater change in accuracy and thus a high AccGrad.

Theoretical correlation: To formalize this correlation between OutputGrad and AccGrad, we prove that they are statistically correlated under specific ***assumptions***.

- First, we define the accuracy of DNN output as the number of correctly-identified elements (including both true positive and true negative), minus the number of wrongfully-identified elements. Acknowledging that this definition of accuracy differs from the metric that is widely used in prior work (*e.g.,* F1 score (Du et al. [2020], Liu et al. [2019], Zhang et al. [2018], Jiang et al. [2018], Du et al. [2022], Zhang et al. [2022a], Xu et al. [2019], Bhardwaj et al. [2022], Zhang et al. [2017])), we observe that, if a configuration has higher accuracy than another configuration under one accuracy metric, it is likely that this configuration

also has higher accuracy under other accuracy metrics.

- Second, the accuracy of DNN output increases when a knob changes towards using more resources (*e.g.,* higher resolution or selecting more frames). This observation aligns with prior work (Zhang et al. [2018], Jiang et al. [2018], Du et al. [2020], Liu et al. [2019], Du et al. [2022], Xu et al. [2019], Zhang et al. [2017], Li et al. [2020], Agarwal and Netravali [2023]).
- Third, different elements do not overlap (*e.g.,* object bounding boxes do not mutually overlap, or the detected words in the audio do not overlap). This assumption holds as a wide range of DNNs perform non-maximum suppression (Neubeck and Van Gool [2006], Detectron2, Han et al. [2016]) to remove overlapped elements.

Formally, if these assumptions hold, we can formally prove that

$$\underbrace{\frac{\partial Acc(\mathbf{k})}{\partial \mathbf{k}}}_{\textsf{AccGrad}} = \left| \underbrace{\frac{\partial \mathbf{z}(\mathbf{k})}{\partial \mathbf{k}}}_{\textsf{OutputGrad}} \right|$$

.

While there can be exceptions to the conditions required by our proof, these conditions corroborate the observations in previous studies. For instance, higher encoding resolution leads to higher bandwidth usage and likely more accurate inference, or higher frame rate leads to higher GPU usage and likely more accurate results (Zhang et al. [2018], Jiang et al. [2018], Du et al. [2020], Liu et al. [2019], Du et al. [2022], Xu et al. [2019], Zhang et al. [2017], Li et al. [2020], Agarwal and Netravali [2023]).

Empirical correlation: We empirically validate the correlation between AccGrad and OutputGrad using a streaming media analytics pipeline (pipeline ⓐ) from the Downtown dataset. For each configuration in pipeline ⓐ , we derive AccGrad from its definition and OutputGrad through backpropagation on the first 10 seconds of each video in Downtown dataset. Since both AccGrad and OutputGrad are vectors (with each element being the

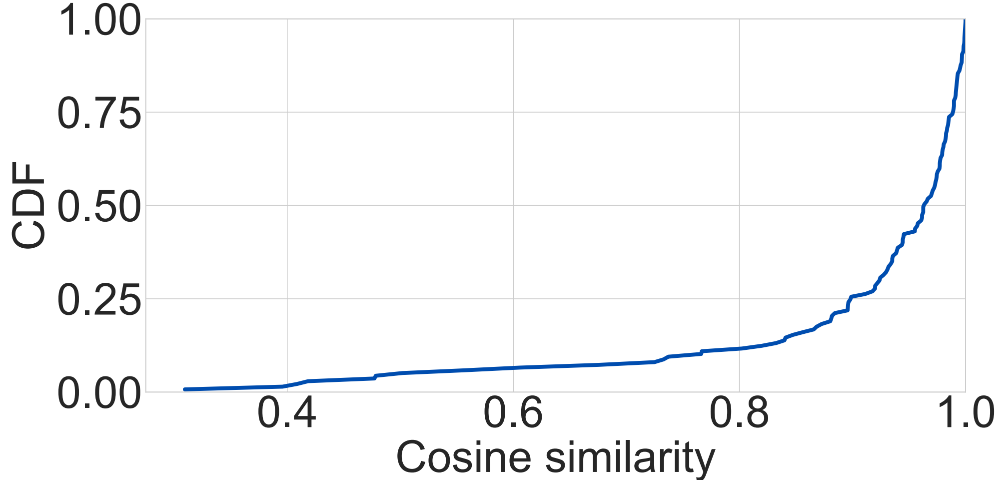


Figure 4.3: The CDF of the cosine similarity between AccGrad and OutputGrad across different configurations and videos. The average cosine similarity is over 0.91.

gradient for a knob), we measure their correlation by cosine similarity. Figure 4.3 displays the CDF of cosine similarity across different configurations and videos. The average cosine similarity exceeds 0.91, indicating a strong correlation between AccGrad and OutputGrad.

4.1.5 Caveats and benefits

While our evaluation shows OneAdapt's gradient-ascent strategy performs well for a wide range of DNN tasks, input data types, and configuration knobs, it is important to understand its limits and why it works in practice.

Convergence of OneAdapt on changing input data:

OneAdapt takes a gradient-ascent strategy to adapt configurations, where OneAdapt derives a new configuration based on the input data of the current adaptation interval and applies it to the next interval. This approach will not converge if the input changes dramatically between intervals. That said, the input data in our benchmark (and other papers (Zhang et al. [2018], Jiang et al. [2018])) incurs drastic change on the scale of tens of seconds (*e.g.,* when driving in the countryside, the vehicle appears on a scale of tens of seconds), whereas OneAdapt empirically converges to a near-optimal configuration within 3-5

intervals (seconds). Thus, as long as the best configuration lasts for at least a few seconds, OneAdapt can identify the best configuration faster than profiling-based methods.

The reason behind the fast convergence of OneAdapt is that: OneAdapt can obtain the gradient of every configuration knob within one backpropagation, which effective translates to having the knowledge of n configurations, where n is the number of knobs. However, the baseline can only obtain the knowledge of one configuration by one inference.

Why OneAdapt beats alternatives: OneAdapt outperforms previous work on all three requirements (*i.e.,* adapt frequently, converge to a near-optimal configuration, and generalize to different types of knobs and analytic tasks).

First, compared to profiling-based work, OneAdapt adapts much more frequently (as OneAdapt updates its configuration *at every second* but profiling-based work updates its configuration once every minute (Zhang et al. [2018])) and earlier (as OneAdapt adapts to the change of input content right at the next second but the profiling-based work need to wait till the next profiling).

Second, compared to heuristics-based methods, OneAdapt's gradient ascent can converge to a *closer-to-optimal* configuration. Most heuristics used in prior work adapt configuration by analyzing only the input data, rather than how the data might affect the final DNN's output and accuracy. By contrast, AccGrad directly indicates that DNN accuracy varies with a small change in the configuration.

Third, we show that OneAdapt can generalize to 4 analytic tasks (vehicle detection, human detection segmentation, audio-to-text), 5 types of input data (*e.g.,* LiDAR videos, audio waves), and 5 types of knobs (*e.g.,* video codec knobs, frame filtering thresholds) in our evaluation. That said, we have not tested if OneAdapt will work for applications outside streaming media analytics (*e.g.,* text generation and image generation) and knobs that alter the final DNN itself (*e.g.,* DNN selection).

We want to clarify that similar to prior work (Zhang et al. [2018], Jiang et al. [2018],

ID	Target application	Analytic task	Dataset	Streaming media type	DNN	Configuration knobs
ⓐ	Autonomous driving	Vehicle detection	Downtown	RGB	Faster-RCNN	Fine-grained encoding quality assignment
ⓑ						Video codec knobs
ⓒ			Country			Frame filtering knobs
ⓓ	Autonomous driving	Vehicle detection	KITTI	LiDAR	Point-Pillars	Fine-grained encoding quality assignment
ⓔ	Home security	Human detection	PKU-MMD	RGB, Depth, InfraRed	Yolo	Fine-grained encoding quality assignment on DNN features
ⓕ						
ⓖ						
ⓗ	Smart home	Audio-to-text	Google AudioSet	Audio	Wav2Vec	Audio sampling rate
ⓘ	Traffic analytics	Vehicle segmentation	Traffic	RGB	Mask-RCNN	Fine-grained encoding quality assignment

Table 4.2: Overview of the experimental setup. Each row represents an analytic pipeline we used to evaluate OneAdapt.

Zhang et al. [2022a]), OneAdapt leverages idle time to perform adaptation. Thus, OneAdapt does not delay the generation of inference results.

Running gradient ascent over discrete configuration space: Before running OneAdapt, we first linearly normalize all knob values to [0,1] and make sure that the increase in knob value corresponds to the increase in resource usage. We then calculate the updated knob value (Equation 4.2) and configure each knob using the configuration value closest to the updated value. We want to clarify that running gradient ascent on top of discrete configuration space can converge to a near-optimal configuration when the objective function (*i.e.,* the weighted sum between accuracy and resource usage) is concave (Ramazanov [2011], Chistyakov and Pardalos [2015]).

4.2 Evaluation

4.2.1 Evaluation setup

Applications and DNNs: We target four types of applications: vehicle detection with FasterRCNN for autonomous driving (Ren et al. [2015], Detectron2), vehicle segmentation

employing MaskRCNN for traffic analytics (Detectron2, He et al. [2017]), human detection using YoLo for home security (Redmon and Farhadi [2017], yol) and audio-to-text utilizing Wav2Vec for smart home applications (Baevski et al. [2020], wav). Note that both detection and segmentation applications categorize objects as either of interest or not.

Accuracy metrics: We use F1 score (Du et al. [2020], Zhang et al. [2018], Jiang et al. [2018], Zhang et al. [2022a]) for vehicle detection, vehicle segmentation, and audio-to-text. For human detection, we use mean IoU (mIoU (Liu et al. [2019])). Following prior work (Zhang et al. [2018], Jiang et al. [2018], Du et al. [2020, 2022], Bhardwaj et al. [2022], Agarwal and Netravali [2023]), we define accuracy as the similarity between the current inference results and the inference results generated from the most resource-consuming configuration to measure the impact of adapting configurations on inference results.

Input settings: For all our applications we use 10FPS from the respective sensors (except for audio-to-text, where we chunk the raw audio into one-second segments and send them to the DNN for analytics). Note that while 30FPS and 60FPS are typical for video content intended for human viewing (vid), 10FPS is prevalent in real-time analytic applications such as autonomous driving (aut [a,b], Xiao et al. [2020], Ghasemieh and Kashef [2022]).

Dataset: We use the following datasets to evaluate OneAdapt, with the goal of covering various application scenarios and streaming media:

- *Autonomous driving*: our dataset covers two driving contexts (downtown driving and country driving) and two main types of sensors (RGB video sensor and LiDAR sensor). Specifically, we obtain 10 downtown driving RGB videos (goo) and 8 country driving RGB videos (goo) using an anonymous YouTube search, and 6 urban driving LiDAR videos from KITTI dataset (Geiger et al. [2013]).
- *Traffic analytics:* We collect 5 traffic camera video footag by anonymous YouTube searche (goo).
- *Home security:* We use three types of sensor data (RGB video sensor, InfraRed sensor, and depth sensor), with 10 videos each from PKU-MMD dataset (Chunhui et al. [2017]). The

inclusion of InfraRed and depth data is vital for enhancing night-time human detection accuracy in home security applications.

- *Smart home:* We randomly sample 200 audio clips in Google AudioSet (aud).

Pipelines: We use nine analytic pipelines (pipeline ⓐ -ⓘ). These analytic pipelines show the applicability of OneAdapt across configuration knobs (pipeline ⓐ -ⓒ), types of streaming media (pipeline ⓓ -ⓖ) and applications (pipeline ⓗ -ⓘ). Table 4.2 summarizes these pipelines, including their target applications, analytic tasks, datasets, streaming media types, DNNs, and accuracy metrics.

Knobs and baselines: We describe the knobs and the corresponding baselines of these pipelines one by one:

- Pipeline ⓐ : This pipeline saves bandwidth by adjusting the encoding quality within each 16x16 pixel macroblocks (Wiegand et al. [2003], Du et al. [2022]). We benchmark against three methods. DDS (Du et al. [2020]) uses a low-quality video for initial inference, then refines specific regions with high-quality encoding. EAAR (Liu et al. [2019]) uses results from the previous frame to identify current frame regions needing high-quality encoding. AccMPEG (Du et al. [2022]) runs a sensor-side neural network to determine regions for high-quality encoding.
- Pipeline ⓑ : This pipeline optimizes bandwidth using three knobs: resolution, QP, and B-frame selection likelihood (Zhang et al. [2018], Jiang et al. [2018], Xu et al. [2019], Zhang et al. [2017], Du et al. [2020], Liu et al. [2019], Du et al. [2022], Fischer et al. [2021], Zhang et al. [2015], Li et al. [2020], Canel et al. [2019], Chen et al. [2015], Zhang et al. [2022a])), all supported in prevalent video codecs (ffm, Wiegand et al. [2003], Coding and Rec [2013]). We implement three baselines for this pipeline. Chameleon (Jiang et al. [2018]) periodically profiles top-k configurations and picks the one with the highest accuracy under the current bandwidth budget. AWStream (Zhang et al. [2018]) periodically profiles a downsampled subset of configurations and chooses the one that maximizes accuracy within bandwidth

limits. CASVA (Zhang et al. [2022a]) runs a reinforcement learning model on the sensor to determine the new configuration.

- Pipeline ⓒ : This pipeline reduces the GPU computation usage by running frame filtering using two frame filtering knobs (pixel difference threshold and area difference threshold (Li et al. [2020])). As for baselines, we implement Reducto (Li et al. [2020]), together with a profiling-based baseline (Zhang et al. [2018]).
- Pipeline ⓓ : This pipeline saves bandwidth when streaming LiDAR point clouds by segmenting the 3D space around the LiDAR sensor into spatial blocks and streaming k% of LiDAR points from each. The value of k can vary between blocks. While this encoding mechanism is basic, our goal is not to establish a best practice, but to showcase the advantage of configuration adaptation. To ensure robustness, we average results from five repeated experiments. We compare against two baselines: a region-based method extended from (Liu et al. [2019]) and a uniform-quality approach, which applies a fixed encoding quality across blocks.
- Pipeline ⓔ ⓕ ⓖ : These pipelines transform videos into feature vectors using a sensor-side DNN and then stream them for server analysis (vcm). To reduce bandwidth usage, we vary encoding qualities across the spatial blocks of feature vectors. Traditional heuristics are not directly applicable on DNN-generated features, so we introduce a region-based heuristic that prioritizes blocks with recent human activity (Du et al. [2020], Liu et al. [2019]). Note that given the 16 configuration knobs in this pipeline, profiling methods fall short, even when configurations are aggressively downsampled. As evidence, we implement Chameleon (Jiang et al. [2018]) in pipeline ⓔ as the profiling baseline, which, despite 8x more GPU resources than OneAdapt and only tuning 4 knobs, still fails to outperform OneAdapt.
- Pipeline ⓗ : This pipeline optimizes the delay between a user's speech and its transcription into text by reducing bandwidth usage. We use the audio sampling rate as our knob. We

implement a profiling-based baseline (Chameleon (Jiang et al. [2018])) and a heuristics baseline that raises the audio sampling rate when detecting human voice (Janbakhshi and Kodrasi [2021]).

- Pipeline (i) : This pipeline aims to reduce the bandwidth usage of running traffic analytics, by assigning different encoding quality to different spatial areas. We use the baselines same as pipeline (a) except for EAAR, as EAAR is not directly applicable to vehicle segmentation DNNs.

Resource usage: For those pipelines that aim to minimize bandwidth usage, we measure the bandwidth usage by the bandwidth needed to send one-second worth of data, while constraining the GPU computation as being able to analyze 1.5-second worth of data per second (we relax this constraint for one baseline (DDS (Du et al. [2020])) as it needs to examine the same one-second data twice for each second). For those pipelines that aim to minimize GPU computation usage, we measure the GPU computation by the number of video frames that need to be analyzed per second[3]) and we do not constraint the bandwidth usage of OneAdapt and other baselines.

Hardware settings: We use one Intel Xeon 4100 Silver CPU as the CPU and NVIDIA RTX 2080 as the GPU.

4.2.2 Experimental results

Better trade-off between accuracy and resource: We show that OneAdapt achieves a better trade-off between accuracy and resource usage across 9 different pipelines in Figure 4.4. We see that across these applications, OneAdapt achieves 15-59% resource usage reduction compared to the baselines without decreasing inference accuracy, or improves the accuracy by 1-5% without inflating resource usage. We highlight that in Figure 4.4e, a profiling-based

3. The backpropagation overhead of OneAdapt is also transformed to the number of frames analyzed per second (by using backpropagation runtime divided by the runtime of analyzing one frame).

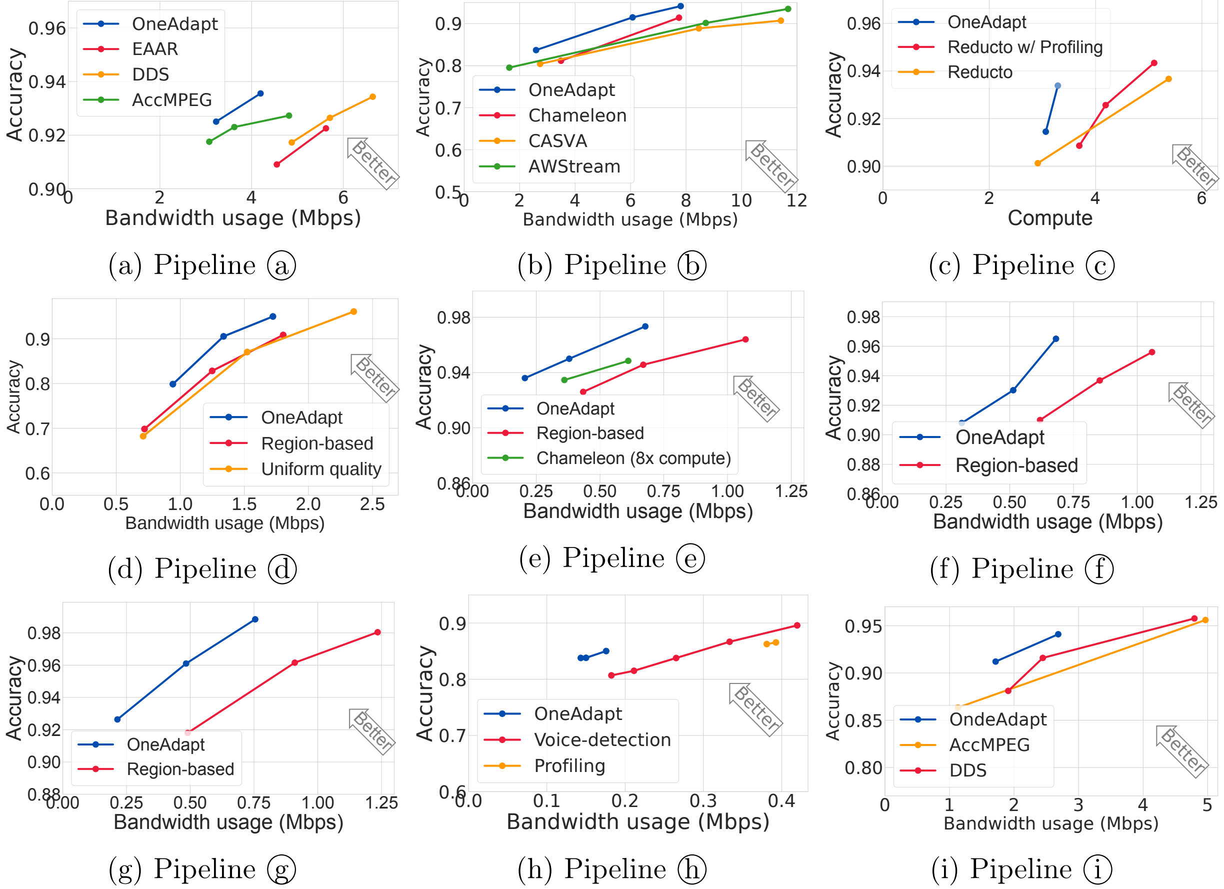


Figure 4.4: Demonstrating the trade-off between accuracy and resource usage of OneAdapt and several baselines. OneAdapt achieves higher accuracy with 15–59% resource usage reduction or 1–5% higher accuracy with less resource usage compared to the baselines across 9 different pipelines. Each figure shows averages across 5–10 videos or 200 audios, depending on the dataset, ensuring statistical confidence.

baseline (Chameleon) with 8x more GPU compute than OneAdapt still has a sub-optimal trade-off between resource usage and accuracy.

Please refer to our full paper (Du et al. [2023]) for further evaluation results.

More knobs, more gain: We show that OneAdapt achieves higher bandwidth reduction[4] compared to the baseline in pipeline ⓔ . In Figure 4.5, encoding qualities are assigned to 4, 16, 64 spatial blocks, resulting in 4, 16, 64 knobs to adapt. We show that the bandwidth

4. We define bandwidth reduction as the bandwidth usage of the region-based approach, divided by the bandwidth usage of OneAdapt when OneAdapt has higher accuracy

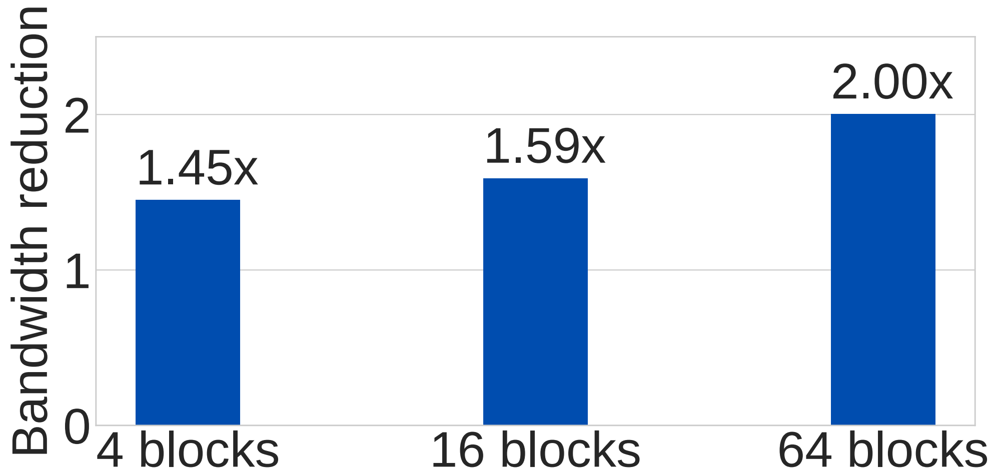


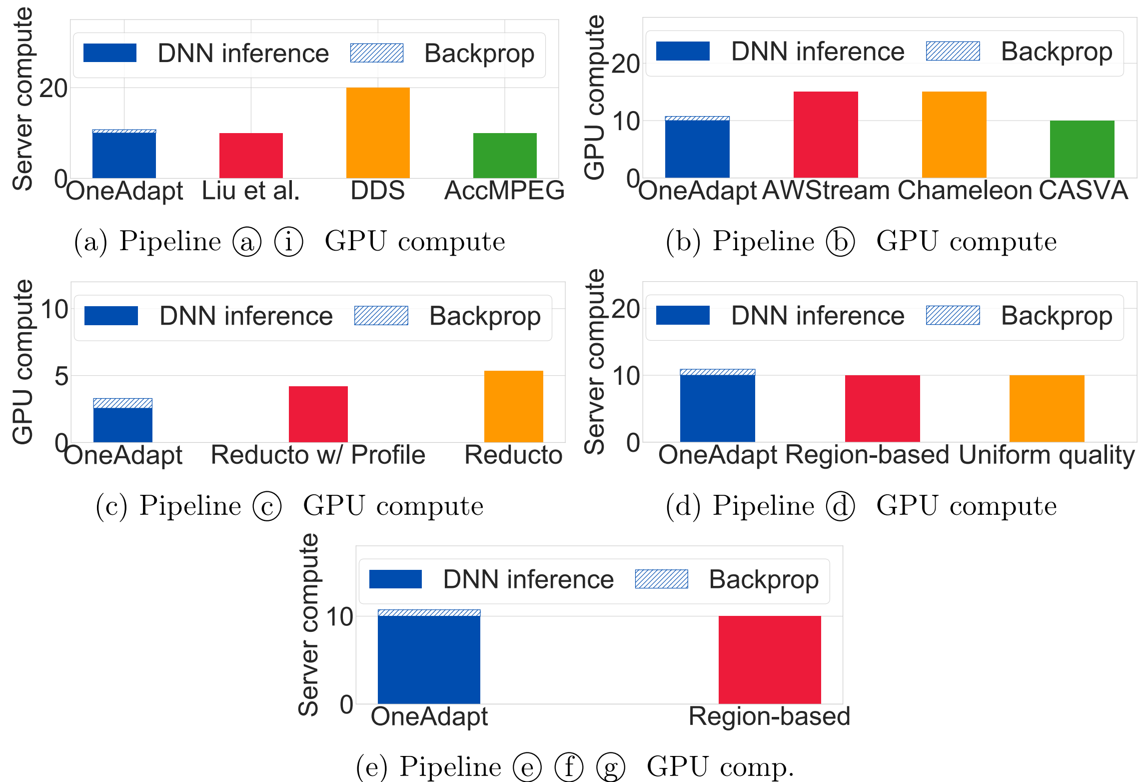
Figure 4.5: When the knobs are more fine-grained, the bandwidth reduction of OneAdapt (the bandwidth usage of the region-based baseline, divided by the bandwidth usage of OneAdapt when OneAdapt is of higher accuracy) grows larger.

reduction of OneAdapt grows larger when there are more configuration knobs. This is because OneAdapt near-optimally handles more knobs without adding GPU computation (since OneAdapt only runs one backpropagation, regardless of the number of knobs) or CPU computation. However, the heuristics encode similar areas in high quality regardless of number of knobs and thus cannot significantly improve the resource–accuracy trade-off when there are more knobs.

Overhead of OneAdapt: We measure the sensor-side CPU computation and the server-side GPU computation of OneAdapt[5] using the CPU computation divided by the CPU computation of processing one frame (or 1/10 worth of data in other data formats), and the GPU computation divided by the GPU computation of analyzing one frame. We mark the adaptation overhead of OneAdapt in the hatched area. From Figure 4.6, we see that the server-side GPU computation of OneAdapt is comparable to or lower than the baselines, and the adaptation overhead of OneAdapt is negligible.

That said, the sensor-side CPU overhead of OneAdapt is high. Though OneAdapt has

5. Note that the bandwidth overhead of streaming DNNGrad from the server to the client is negligible as it contains >7000x less amount of data after sampling and compression of DNNGrad.



(a) Pipeline ⓐ ⓘ GPU compute (b) Pipeline ⓑ GPU compute

(c) Pipeline ⓒ GPU compute (d) Pipeline ⓓ GPU compute

(e) Pipeline ⓔ ⓕ ⓖ GPU comp.

Figure 4.6: Comparing the server-side compute overheads of OneAdapt against the baselines (measured by the number of frames inferred by the server per second).

equal or lower CPU overhead in pipeline ⓒ ⓗ than the baselines, OneAdapt needs to encode the input data 3 times (in pipeline ⓐ ⓓ ⓔ ⓕ ⓖ ⓘ , even 4 times in pipelineⓑ), resulting in more CPU computation overhead than the baselines. One may worry that OneAdapt imposes too much CPU overhead on the sensor that may exceed the sensor-side computation capability. To answer this question, we measure the encoding speed on an Intel Xeon 4100 Silver CPU and find that it has enough compute to encode 103.2 video frames per second, which translates to encoding the input data 10 times (as the input data is 10FPS) and is sufficient for OneAdapt in our evaluation setup.

Effectiveness of GPU overhead reduction: OneAdapt removes unneeded computation and DNNGrad reusing and significantly reduces the GPU overhead. Figure 4.7 tests how these two techniques reduce the backpropagation overhead of OneAdapt in terms of GPU runtime and GPU memory on pipeline ⓐ , on a 10-frame video chunk. OneAdapt reduces

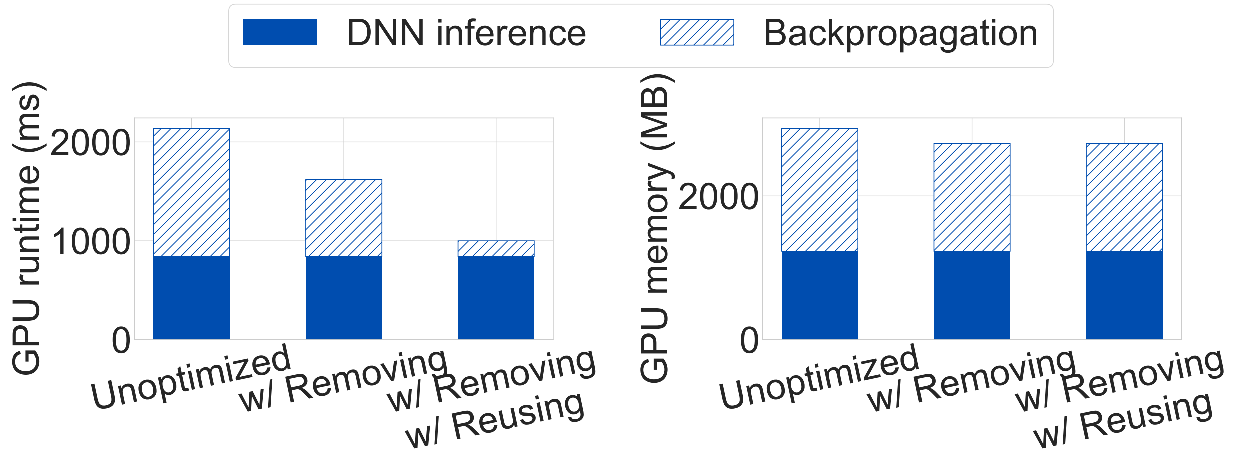


Figure 4.7: Benchmarking the effectiveness of removing unneeded computation and DNNGrad reusing on a 10-frame video chunk using pipeline ⓐ . OneAdapt reduces the GPU runtime overhead by 87% and the GPU memory overhead by 12%.

the GPU runtime overhead of backpropagation by 87% and the GPU memory overhead by 12%.

CHAPTER 5

PREFILLONLY: AN INFERENCE ENGINE FOR PREFILL-ONLY WORKLOAD

To reduce the uncertainty, we leverage the property of prefill-only workload. We first introduce the rationales about why prefill-only workload is a substantial workload, and then introduce techniques related to job completion time calculation in prefill-only workload.

5.1 Why prefill-only workload is huge

5.1.1 Preliminary of large language models

This section introduces the basic concepts in large language model (LLM) inference.

Prefilling and decoding: LLM inference contains two phases. The prefilling phase of LLM processes the input and generates one output token. Then, the LLM engine runs multiple rounds of the decoding phase, where each decoding phase forwards the request through the LLM and generates one more output token.

KV caches: As LLM inference incurs multiple rounds, when processing one request, the LLM engine will store the intermediate tensors produced by the attention layers inside the GPU, known as KV caches, to accelerate future rounds of LLM inference. The size of KV caches can be large—*e.g.,* the KV cache size of a request with 100,000 tokens is around 12 GB for a medium-sized LLM model (Llama-3.1-8B).

Prefix caching: The KV caches generated by one request can be reused by another request if they share part of the prefix, which is commonly referred to as prefix caches. As a result, existing LLM inference engines (Kwon et al. [2023a], Zheng et al. [2024]) will not immediately free these KV caches after request execution, but instead cache them in the GPU so that future requests can potentially reuse the KV caches.

5.1.2 Substituting traditional deep learning models with LLMs

In addition to generative LLM applications that generate new content for people to read and use, (*e.g.,* ChatGPT (OpenAI [2025]), Character.AI (cha), GitHub Copilot (GitHub [2025]), Cursor (GitHub [2025]), Perplexity (Perlexity AI [2025]) and more), both industry and academia have started to use LLMs to replace traditional deep learning pipelines in applications such as recommendation, credit verification, and data labeling mainly for two reasons.

LLM streamlines the development: Developing traditional deep learning pipelines is time-consuming and complex, as it requires co-optimizing multiple stages in the pipeline, from data cleaning to model fine-tuning, which requires cross-team collaboration and extensive infrastructure support, and eventually accumulates maintenance debt. In contrast, LLM can take raw text as input and chat with the developer, and is general enough to obviate fine-tuning. This allows the developer to interactively debug and improve the pipeline by just chatting with the LLM. This argument is supported by recent literature (Firooz et al. [2025]), which shows that a single LLM model can serve over 30 tasks across over 8 different domains.

LLM achieves higher decision quality: Further, by properly selecting the size of the LLM model and engineering the prompt, LLM can generalize to out-of-domain tasks and surfaces, and achieves comparable performance similar to or better than a production model (Firooz et al. [2025]).

5.1.3 Underlying workload: prefill-only workload

We observe that people use LLMs in these applications in a different way from those generative applications. Concretely, these applications only let the LLM generate one single output token for each request. We call these requests *prefill-only requests* (as they only require the LLM engine to run prefilling to generate a single output token), and name such a workload

a prefill-only workload.

Single token suffices to express LLM's preference: Generating one single token still allows LLM to express its preference between different options. To illustrate this, we use post recommendations as an example. Assume that we are a social media platform that aims to recommend social media posts to a specific user via LLM. In this case, the recommendation system will first gather user's browsing history as the profile. Then, the system selects, for example, fifty posts that the user might be interested in using traditional heuristics like embedding-based similarity search. Then, the recommendation system sends fifty LLM requests to the LLM, one per document. Here is an example of such LLM request:

You are a recommendation assistant to use user's profile and history to recommend the item that the user is most interested in. Here is the user profile:

[User profile]

Here is the browsing history of an user:

[User browsing history]

If we recommend the following article to this user, will the user be interested in reading it? Please response using Yes or No.

[Article]

Your answer is:

Then, we constraint the output of LLM to only `Yes` and `No`, and let LLM prefill this request to yield two probability numbers: $\mathbb{P}(\texttt{Yes})$ and $\mathbb{P}(\texttt{No})$, where their sum equals to 1. After that, the recommendation system will use $\mathbb{P}(\texttt{Yes})$ as the recommendation score.

Lower latency: Generating one single token also significantly reduces LLM inference latency. This is because the input processing of LLMs is much faster than output generation. Concretely, using Llama 3.1-8B-model and one NVIDIA H100, we measure that handling a request with 2048 tokens input and 256 tokens output is 1.5× slower than handling a request with 2048 tokens input and 1 token output.

Clearly-defined and controlled output behavior: Further, we argue that prefill-only workload allows the developer to simply define and control the output behavior by passing over a list of acceptable tokens to the LLM engine and then let the LLM engine to sample only output from this list. In contrast, it is difficult to clearly define and guarantee the expected LLM output behavior in traditional generative requests, which motivates a long line of research (*e.g.,* (Dong et al. [2024], Ye et al. [2025])).

5.1.4 Characteristics of prefill-only workload

We conclude two characteristics of prefill-only workload.

First, the input length of prefill-only requests is typically long. For example, in a documentation recommendation application, the user profile potentially contains months of the user's browsing history, which can easily reach tens of thousands of tokens, and even more.

Second, different from a traditional LLM workload that is GPU-memory-bound, a prefill-only workload is bounded by GPU computation.

Sharing GPU resources with traditional generative workload is impractical: To meet the stringent application requirements, instead of sharing the GPU with traditional generative requests, one needs to allocate GPU dedicated to prefill-only workloads. This is because the inter-workload interference of LLM applications is significant. For example, serving both prefill-only requests and traditional generative requests on the same GPUs may greatly increase the average and P99 time-per-output-token in generative use cases, as the decoding jobs will be batched with prefill jobs more frequently compared to not mixing these two workloads.

Further, the volume of prefill-only workload is large enough that it deserves dedicated GPU resources. For example, in recommendation workload, the typical queries per second is at the scale of tens of thousands, which demands hundreds or even thousands of H100 GPUs to serve.

5.1.5 *Limitation of existing LLM engines*

In practice, we observe two issues that limit the capacity of existing LLM inference engines in prefill-only workloads.

Scheduling algorithm is unaware of JCT: Existing LLM engines typically leverage JCT-agnostic scheduling, such as first-come-first-serve scheduling, as the output length of LLM requests can be non-deterministic and difficult to predict. If the JCT can be accurately estimated, one can leverage JCT-aware scheduling (like shortest remaining job first) to further reduce the latency of requests.

5.1.6 *Opportunity and challenges*

In prefill-only workload, the opportunity is that the JCT of prefill-only requests is deterministic and predictable as the output length is always 1.

A naive solution to leverage the opportunities above is by offline profiling how the JCT changes with respect to request length, and in the online phase, the LLM engine uses this JCT profile to obtain the JCT of each incoming request, and then schedules the waiting requests using JCT-aware algorithms (e.g., shortest remaining job first).

However, this solution achieves limited performance improvement in practice, because the JCT does not only depend the request itself, but the prefix cache condition as well. Existing LLM engines store the KV caches in the LLM engine so that future requests with the same prefix can be accelerated. This technique is known as prefix caching. Fully discarding the KV caches prevents such optimization.

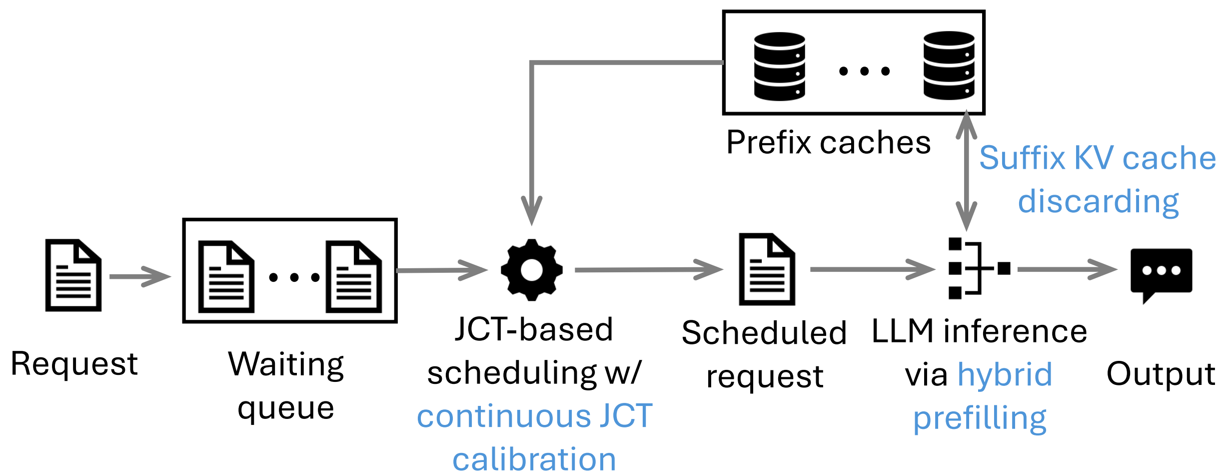


Figure 5.1: Overview of PrefillOnly and its core techniques.

5.2 Overview of PrefillOnly

5.2.1 Overview of PrefillOnly

PrefillOnly is an inference engine that serves prefill-only requests online. PrefillOnly opens an HTTP server compatible with the OpenAI API protocol for the user to send their prefill-only requests during runtime. When a new request arrives, PrefillOnly tokenizes the request and sends it to the waiting queue of the scheduler process using ZeroMQ-based RPC. The scheduler process then schedules the requests in the granularity of *step*. During each step, PrefillOnly enumerates the requests in the waiting queue to find the request with minimum JCT. Then, PrefillOnly pops out this request and sends it to the executor processes. Finally, the request executor processes then execute the request and return the prefill-only output all the way back to the user.

We summarize the workflow of PrefillOnly in Figure 5.1

5.3 Scheduling in the context of prefix caching

The key challenge of PrefillOnly, is that the JCT of each request changes over time, since the JCT depends on the prefix cache storage, which is constantly changing: new request introduces new prefix cache and evicts old ones.

To handle this case, PrefillOnly handles one request at a time, and performs JCT estimation everytime when PrefillOnly schedules a new request (we call this continuous JCT calibration). This allows PrefillOnly to obtain accurate JCT at the time when the request is being scheduled, and significantly improves the overall prefix cache hit rate.

In this section, we first discuss why PrefillOnly does not choose to batch prefill-only requests, and then discuss continuous JCT calibration as the key technique to handle JCT changes.

5.3.1 Why not batching prefill-only requests

In traditional LLM workloads, to maximize the output generation throughput (*i.e.* decoding throughput), the inference engine needs to maintain a large batch size by continuously batching new requests to the LLM engine and removing finalized requests from the batch. This technique significantly improves throughput, because the output generation phase is bounded by GPU memory accessing bandwidth, and batching 2$\times$ requests only marginally increases the total GPU memory needs to be accessed (as the major part is the LLM model weights) and thus marginally increases the runtime, but doubles the generation throughput.

However, the inference in prefill-only workload is typically GPU computation-bound, and thus, batching does not significantly improve the throughput of processing Thus, batching prefill-only requests increases the average latency compared to processing the requests one by one, and does not improve the throughput. As a result, PrefillOnly chooses to schedule the requests one by one instead of batching them.

5.3.2 Limitation of traditional JCT-based scheduling

We observe that, traditional JCT-based scheduling algorithm has a low prefix cache hit rate in the context of prefix caching, resulting in high latency and low throughput.

This is because the JCT changes over time when prefix caching is enabled: the JCT of one request reduces when the prefix cache related to this request enters the LLM engine, and increases when this prefix cache is evicted. As a result, this approach fails to timely prioritize those requests that can hit the prefix cache, and when the LLM engine executes these requests, the prefix cache might already be evicted.

This is because traditional JCT-based scheduling algorithms make decisions based on the JCT when the request arrives, and thus fail to timely prioritize the requests when the LLM engine receives prefix cache related to these requests, and deprioritize the requests when the corresponding prefix cache of these requests is evicted.

Example: To further understand why the traditional JCT-based scheduling algorithm has low prefix cache hit rates. We provide an illustrative example. In this example, we assume that four requests (A, B, C, D) come into the LLM inference engine altogether, with request length A $<$ C $<$ B $<$ D. Further, we assume A and D share the same prefix, and so do B and C. Also, we assume that the prefix cache space is limited and thus can only store the state of one request.

In this case, traditional shortest remaining job first scheduling will schedule the job based on its JCT, which is proportional to the request length. As a result, it will schedule the request in the order of A, C, B, D. In this case, request B can hit the prefix cache of request C, and thus request B can be accelerated. However, request D that could have hit the prefix cache of A, cannot be accelerated (as request C evicted the state of A), leading to high inference latency.

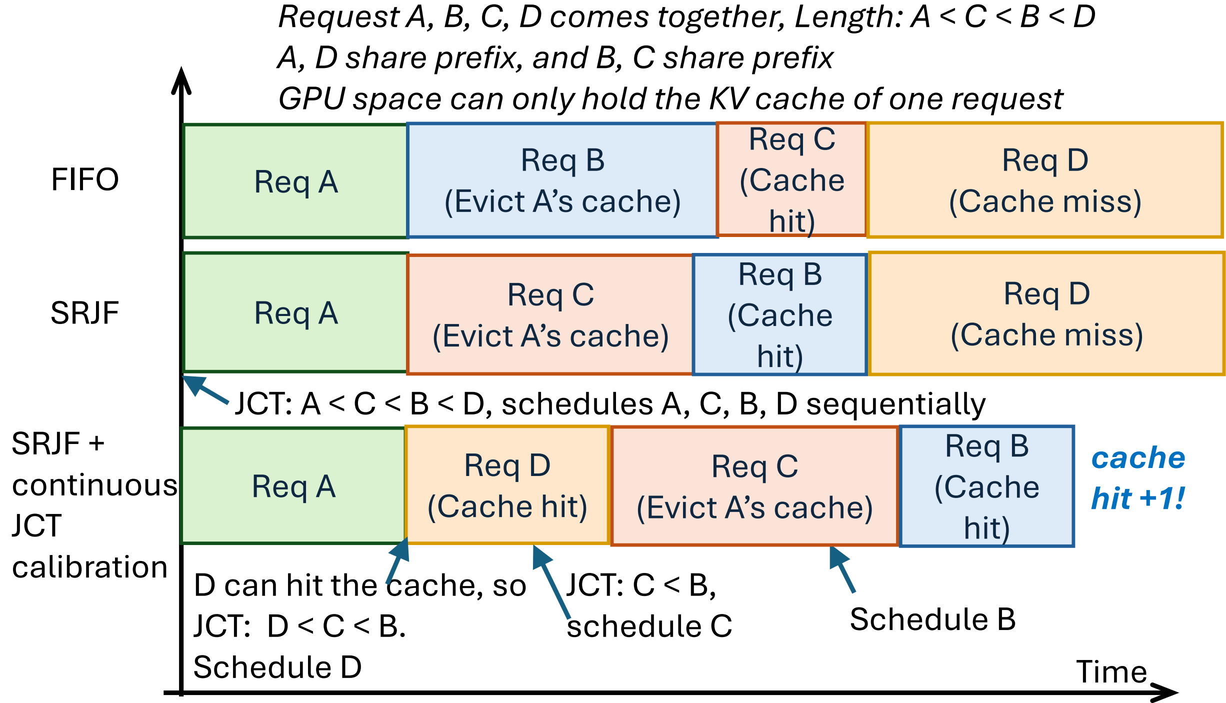


Figure 5.2: Contrasting first-in-first-out (FIFO) scheduling, shortest-remaining-job-first (SRJF) scheduling and PrefillOnly's SRJF scheduling with continuous JCT calibration. The scheduling of PrefillOnly yields one more cache hit, achieving lower average latency.

Algorithm 1 SRJF with continuous JCT calibration

1: **Input:** Waiting queue Q of requests
2: **Output:** Request to schedule in the next step
3: $r_{\text{shortest}} \leftarrow$ None
4: $score_{\text{min}} \leftarrow \infty$
5: **for** each request $r \in Q$ **do**
6: $\quad n_{\text{input}} \leftarrow$ the number of input tokens in r
7: $\quad n_{\text{cached}} \leftarrow$ the number of tokens in r that hits prefix cache
8: $\quad T_{\text{queue}} \leftarrow$ the queuing time of r
9: $\quad score \leftarrow \text{get_jct}(n_{\text{input}}, n_{\text{cached}}) - \lambda \cdot T_{queue}$
10: $\quad$ **if** $score < score_{\text{min}}$ **then**
11: $\quad\quad score_{\text{min}} \leftarrow jct$
12: $\quad\quad r_{\text{shortest}} \leftarrow r$
13: $\quad$ **end if**
14: **end for**
15: **Schedule** r_{shortest}

5.3.3 Continuous JCT calibration

To improve the prefix cache hit rate of JCT-based scheduling algorithm, we propose PrefillOnly employs *continuous JCT calibration*, where PrefillOnly calibrates the JCT of waiting requests every time before scheduling.

Such calibration significantly improves the prefix cache hit rate, as it allows the scheduling algorithm to timely prioritize those requests that can hit the prefix cache (as they typically have much lower JCT than other requests), which makes these requests much more likely to hit the prefix cache.

Example: To further illustrate why continuous JCT calibration helps JCT-based scheduling algorithms, like SRJF, improve latency, we give an illustrative example using the same setup as §5.3.2. The first scheduled job will be A. When scheduling the next job, the scheduler will perform JCT calibration, where it finds out that the JCT of D is significantly lowered as D can hit the prefix cache of A. As a result, the second scheduled job will be D. Then, the scheduler calibrates the JCT of B and C again, and their JCT remains unchanged. As a result, the scheduler then schedules C as it is shorter (and thus has lower JCT). After that, the scheduler will then schedule request B. In this case, the total number of cache hits is 2, where the total number of cache hits is 1 in both FIFO scheduling and naive SRJF scheduling. Thus, SRJF with continuous calibration achieves lower latency and higher throughput.

Calibration details: As shown in Algorithm 1, PrefillOnly calibrates the JCT of a given request r by calculating the number of input tokens n_{input} and the number of tokens that hits the prefix cache n_{cached}, and generate the JCT of this request by calling $jct(n_{\text{input}}, n_{\text{cached}})$, where we obtain jct by profiling how the JCT varies with respect to different pairs of n_{input} and n_{cached} that covers the maximum input length with the granularity of 1000 tokens, and trains a small linear model using linear regression.

Empirically, however, we found that the number of tokens that do not hit the prefix

Dataset	Why evaluating this dataset	#users	User profile length	Post length
Post recommendation	Evaluate the ability of PrefillOnly under frequent prefix cache reuse	20	11, 000 tokens — 17, 000 tokens	150 tokens
Credit verification	Evaluate the ability of PrefillOnly under long input length	60	40, 000 tokens — 60, 000 tokens	N/A

Table 5.1: Summarizing the dataset used in the evaluation of PrefillOnly and baselines.

Config	**Max input length (measured by number of tokens)**		
	L4	**A100**	**H100**
Paged Attention	24,000 WL1: ✓, WL2: ×	11,000 WL1: ×, WL2: ×	15,000 WL1: ×, WL2: ×
Chunked Prefill	46,000 WL1: ✓, WL2: ×	17,000 WL1: ✓, WL2: ×	25,000 WL1: ✓, WL2: ×
Pipeline Parallel	72,000 WL1: ✓, WL2: ✓	38,000 WL1: ✓, WL2: ✓	183,000 WL1: ✓, WL2: ✓
Tensor Parallel	195,000 WL1: ✓, WL2: ✓	77,000 WL1: ✓, WL2: ✓	238,000 WL1: ✓, WL2: ✓
PrefillOnly (ours)	130,000 WL1: ✓, WL2: ✓	87,000 WL1: ✓, WL2: ✓	97,000 WL1: ✓, WL2: ✓

Table 5.2: Evaluating the max input length that PrefillOnly and baselines can handle under various hardware setups. WL1 indicates the post recommendation workload, and WL2 indicates the credit verification workload. × means that the max input length is insufficient to run corresponding workload.

cache (*i.e.,* $n_{\text{input}} - n_{\text{cached}}$) is a good proxy of JCT: we measure that, on 1× A100 the Pearson correlation coefficient between the actual JCT and the number of cache miss tokens $n_{\text{input}} - n_{\text{cached}}$ is 0.987 on Qwen 32B with FP8 quantization (where 1 means perfectly correlated). As a result, PrefillOnly uses this JCT proxy by default.

Preventing starvation: In order to prevent starvation, PrefillOnly will reduce the JCT by $\lambda \cdot T_{req}$, where T_{req} is the queuing time of the request and λ is a hyperparameter: increasing the λ result in better worst-case latency at the trade of worse average latency.

We summarize the scheduling algorithm in Algorithm 1.

5.4 Evaluation

Our evaluation shows that:

- PrefillOnly handles 1.4−4.0× larger query-per-second without inflating the average latency and P99 latency compared to baselines.
- PrefillOnly expands the maximum request length by upto 5× without requiring parallelizing the LLM inference.

5.4.1 Evaluation setup

In this subsection, we introduce the dataset, GPUs, LLM models, evaluation metrics, and baselines in our evaluation.

Existing LLM datasets mainly focus on evaluating the LLM accuracy instead of the performance of the LLM engine. As a result, in our evaluation, we use two simulated datasets, covering two tasks: post recommendation on a social media platform and credit verification for a bank application.

We summarize our evaluation dataset in Table 5.1.

Post recommendation dataset: In this dataset, we aim to evaluate the benefit of PrefillOnly under a short context scenario, where the major benefit of PrefillOnly comes from its scheduling. Concretely, we simulate a post recommendation scenario, where we recommend 10 out of 50 posts for a given user, based on the browsing history of this user. The key attributes from the perspective of LLM engine performance are the following parameters:

- Post length: To get a rough estimation of the post length, in terms of number of tokens, we take X as an example, and measured that the number of tokens for a short X post is less than 150 tokens. As a result, we use 150 tokens as the post length.
- Number of posts to be recommended per user: We set the number of posts to be recommended per user as 50. We assume that these 50 posts are given by the underlying

recommendation systems using heuristics like embedding-based similarity search.

- User profile length: As for the user profile length, we focus on the click history of one user, where the user has already been engaged with the social media four times a week, and for four weeks. We assume that each time the user only clicks on five or six posts. As a result, the total length of the profile history is roughly 11,000 to 17,000 tokens. As a result, we use a normal distribution to simulate the user profile length, with a mean as 14,000 and a standard deviation of 3,000.
- Number of users: We evaluated 20 users in total.

Credit verification dataset: In this dataset, our goal is to simulate a credit verification scenario, where we verify the credit of one user based on the credit history of one user. We measure that the length of credit history for one month is about 4,000 to 6,000 tokens. We simulate ten months of credit history, resulting in a credit history length from 40,000 to 60,000 for each user. We consider 60 users in total.

Request arrival pattern: We assume that the user arrival pattern is a Poisson process. We further vary the rate in the Poisson process to vary the query-per-second.

Hardware and LLM setup: We summarize the hardware and the LLM setup in Table 5.3.

PrefillOnly: We implement PrefillOnly based on state-of-the-art LLM serving engine vLLM (vll), with 4.6k lines of Python code to implement the core techniques of PrefillOnly. We set the fairness parameter $\lambda = 500$ by default.

Baselines: We pick four baselines, where two of them parallelize the LLM inference (tensor parallel and pipeline parallel) and the other two do not parallelize the inference (PagedAttention (Kwon et al. [2023b]) and chunked prefill (Agrawal et al. [2024])):

- *Tensor parallel.* In this baseline, we parallelize the inference onto 2 GPUs with the degree of tensor parallelism equal to 2 using the existing implementation available in production-grade inference engine vLLM (vll).
- *Pipeline parallel.* In this baseline, we parallelize the inference onto 2 GPUs with the degree

Scenario	GPU Type	LLM Model
Low-end GPU	2x NVIDIA L4 PCIe (24 GB)	meta-llama/ Llama-3.1-8B
Middle-end GPU	2x NVIDA A100 PCIe (40 GB)	RedHatAI/ DeepSeek-R1-Distill-Qwen-32B-FP8-dynamic
High-end GPU	2x NVIDIA H100 PCIe (80 GB)	Infermatic/ Llama-3.3-70B-Instruct-FP8-Dynamic
High-end GPU w/ NVLink	2x NVIDIA H100 NVLink (80 GB)	Infermatic/ Llama-3.3-70B-Instruct-FP8-Dynamic

Table 5.3: The hardware and the corresponding LLM.

of pipeline parallelism equal to 2. We also use the implementation in vLLM (vll).

- *PagedAttention* (Kwon et al. [2023b]). This baseline manages the KV caches using a page table to minimize fragmentation and employs first-come-first-serve scheduling.
- *Chunked prefill* (Agrawal et al. [2024]). This baseline processes the LLM input chunk-by-chunk to allow handling longer requests.

Note that we enable prefix caching for both PrefillOnly and all these baselines. Also, some baselines cannot handle some workloads as their maximum input length is too short, we show this in Table 5.2

Routing: We note that, for PrefillOnly and non-parallelization-based baselines, in order to utilize multiple GPUs, we launch multiple instances of LLM inference engines, one on each GPU, and then perform *user-id-based routing*, where we route the request from the same user to the same instance, and decide which user should be assigned to which instance in a round-robin manner.

5.4.2 Evaluation results

QPS–latency trade-off: We show the trade-off between query per second (QPS) and latency (mean latency and p99 latency) across three different hardware setups and two different applications in Figure 5.3. In this figure, we determine the evaluation QPS by running PrefillOnly with all requests in the dataset coming at once, and then obtain the throughput (requests per second) of PrefillOnly in this situation, where we denote this value as x. We then evaluate QPS $1/4x, 1/2x, x, 2x, 3x, 4x$. This approach allows us to show the full spectrum performance of PrefillOnly.

In Figure 5.3, we see that PrefillOnly always achieves lowest latency when the QPS is high, indicating that the throughput of PrefillOnly is significantly higher than the baselines. However, in low QPS the latency of PrefillOnly can be higher than parallelization-based baselines (as baselines use multiple GPUs to serve one request but PrefillOnly just uses one).

We also show in Figure 5.4 that PrefillOnly achieves better P99 latency than baselines, indicating that the JCT-based allocation of PrefillOnly does not hurt P99 latency after applying the fairness twist mentioned in §5.3.3.

Varying the fairness parameter λ: We vary the CDF of request latency of PrefillOnly, under different values of fairness parameter λ in Figure 5.7. Higher λ results in better P99 latency, at the cost of inflating the average latency

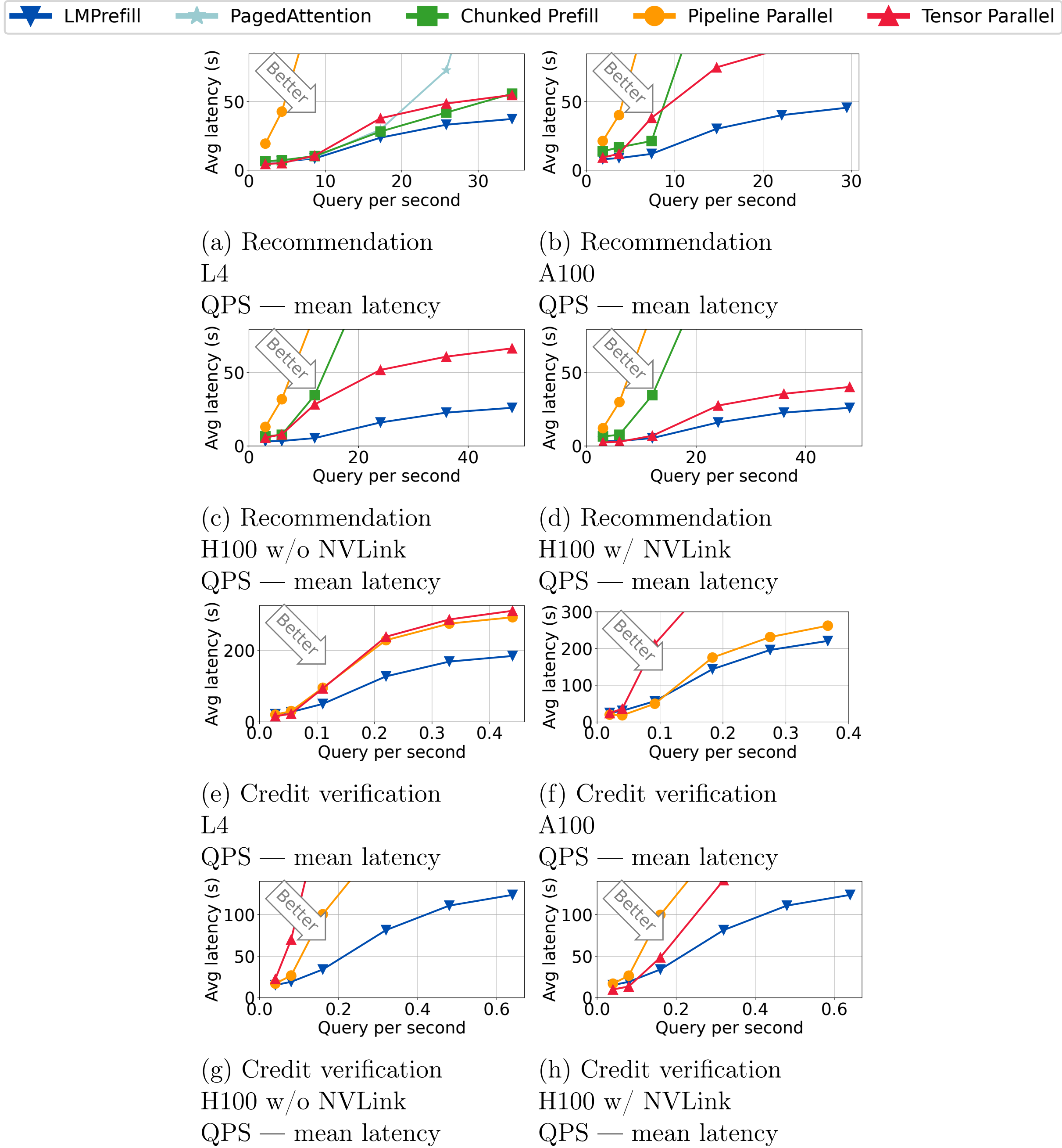
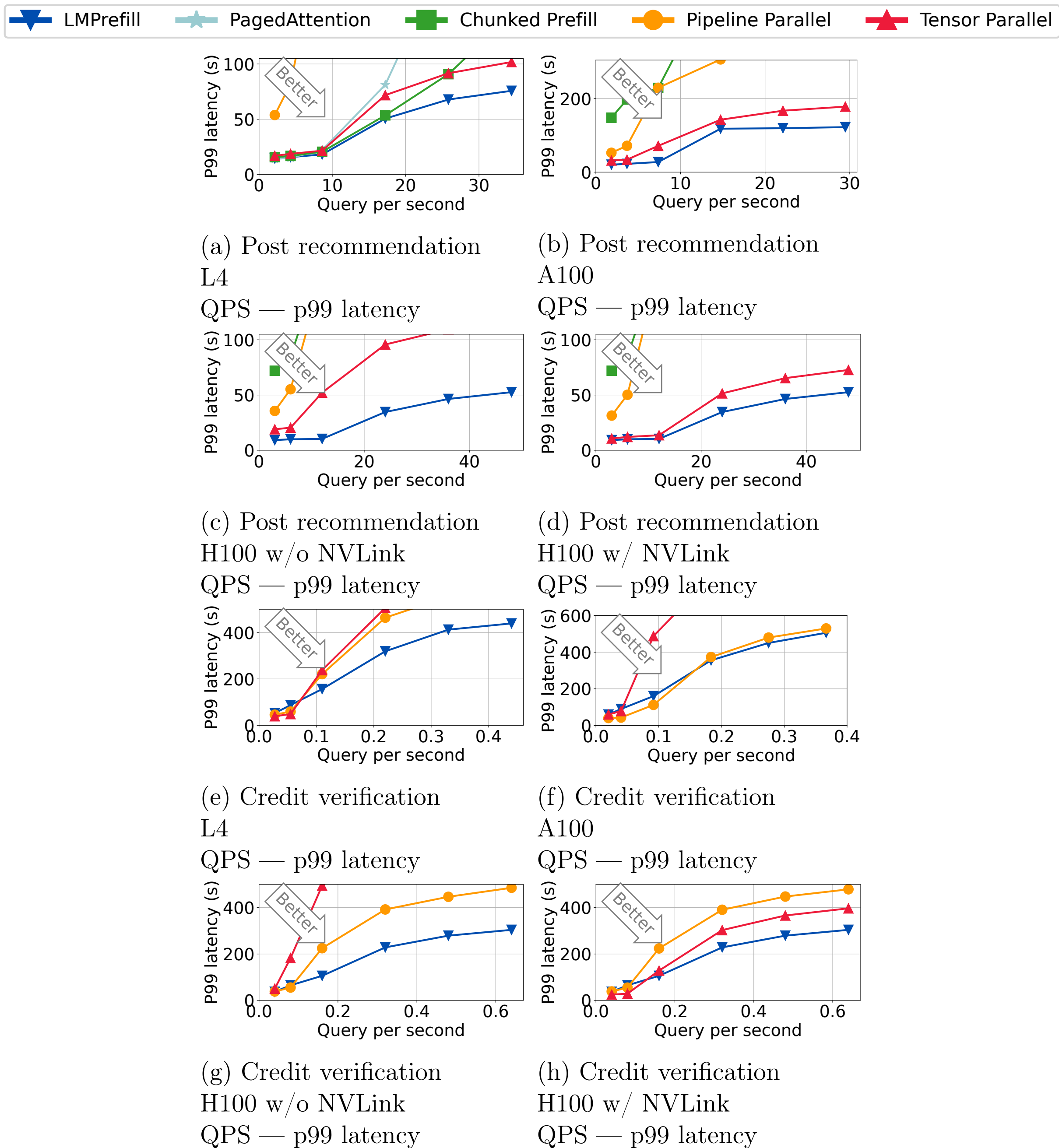


Figure 5.3: QPS — mean latency trade-off of PrefillOnly and baselines on four different hardware setups and two applications. PrefillOnly significantly reduces the latency when the QPS is high and only has higher QPS than tensor parallel baseline when QPS is low. Though tensor parallelism sometimes have lower latency than PrefillOnly under low QPS, it has much lower throughput than PrefillOnly due to extra communication cost and thus scales much worse than PrefillOnly in high QPS.



(a) Post recommendation L4 QPS — p99 latency

(b) Post recommendation A100 QPS — p99 latency

(c) Post recommendation H100 w/o NVLink QPS — p99 latency

(d) Post recommendation H100 w/ NVLink QPS — p99 latency

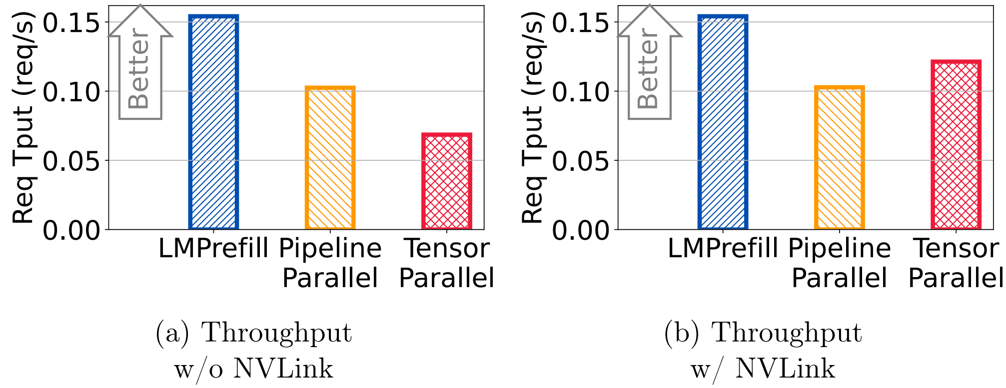
(e) Credit verification L4 QPS — p99 latency

(f) Credit verification A100 QPS — p99 latency

(g) Credit verification H100 w/o NVLink QPS — p99 latency

(h) Credit verification H100 w/ NVLink QPS — p99 latency

Figure 5.4: QPS — P99 latency trade-off of PrefillOnly and baselines on three different hardware setups and two applications.



(a) Throughput w/o NVLink

(b) Throughput w/ NVLink

Figure 5.5: Contrasting the throughput of PrefillOnly and baselines on credit verification workload under 2× H100. Though NVLink significantly accelerates the communication and thus enhances the throughput of communication-intensive parallelization like tensor parallel, PrefillOnly still has the highest throughput as it does not spend extra communication to parallelize the inference.

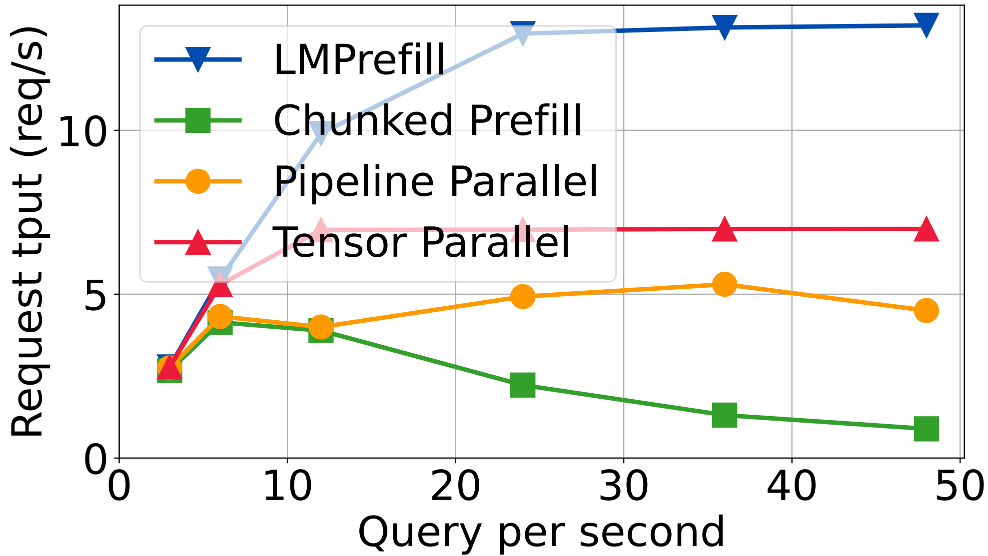


Figure 5.6: Illustrating the throughput of PrefillOnly and the baselines in post post-recommendation dataset under 2× H100 without NVLink. PrefillOnly has better improvement as it can maintain high throughput under high query-per-second, while the query per second of chunked prefill baseline drops because of prefix cache throttling. Parallelization-based baselines parallelize the prefix cache across GPUs and thus have sufficient prefix cache space to avoid throttling, but they have lower throughput because of extra communication and synchronization cost.

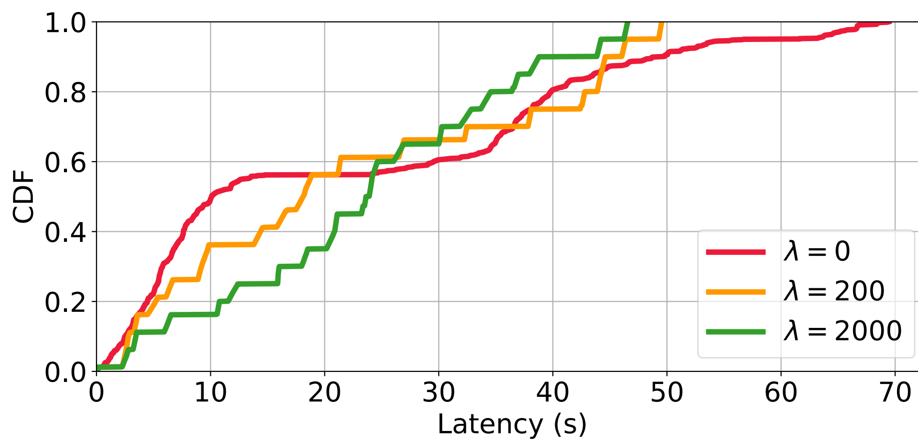


Figure 5.7: The CDF of request latency of PrefillOnly, under different value of fairness parameter λ. Higher λ result in better P99 latency, at the cost of inflating the average latency.

CHAPTER 6

LIMITATION AND FUTURE WORK

6.1 Limitation

In this chapter, we focus on discussing the large language model aspects, as this attracts more and more attentions these days and become the centroid of machine learning system research.

My previous work, PrefillOnly, specifically targets prefill workloads and significantly enhances throughputy. However, it is limited in its general applicability. For instance, PrefillOnly inherently assumes that applications do not preprocess the incoming requests (*e.g.,* sorting the requests).

Additionally, the trend in LLM serving is increasingly oriented toward inference-time scaling, where decoding phases typically dominate runtime due to longer generated texts. Examples include agentic workloads, thinking LLM models and chain-of-thought tasks. Consequently, optimizations that exclusively target prefill phases, like PrefillOnly, might see diminished benefits and adoption as inference-time scaling continues to grow.

Finally, although PrefillOnly effectively reduces latency and improves throughput by leveraging continuous job completion time (JCT) estimations, its effectiveness reduces when the KV cache storage is distributed and very large, as it makes the KV cache eviction less likely to happen.

6.2 Future work

In this section, I outline several promising future research directions to address current limitations and emerging challenges in distributed machine learning and large language model inference.

Faster decoding: : With the rising popularity of inference-time scaling techniques, such as thinking LLMs, agentic workflows, and chain-of-thought, decoding phases become increasingly critical. Future work should focus on developing novel algorithms and hardware accelerations specifically aimed at boosting decoding efficiency, minimizing latency, and maximizing throughput during long-form text generation.

Tight integration with tool calls: : A recent trend is that LLMs interact with external tools and APIs to enhance their functionality in agentic tasks. Integrating these tool calls in a streaming and interactive manner (instead of creating a barrier between LLM and the function call) to ensure low latency and high responsiveness remains a significant challenge. Addressing this will require combining chunked prefills with the streaming behavior of tool calls, and also require "chunked prefill" of the tool using LLM output.

Scheduling for agentic workloads: Agentic workloads often produce significant amounts of intermediate text that are not directly visible to users, generating opportunities for more flexible scheduling policies. We can shoot for high throughput (instead of balancing between latency and throughput) without sacrificing user-perceived application response time, which can substantially improve overall system efficiency.

Disaggregated serving: : The growing heterogeneity of LLM architectures (e.g., multimodal encoders, heterogeneous attention mechanisms, disaggregating prefill and decoding modules, mixture-of-experts) and diverse hardware platforms (e.g., NVIDIA GPUs, AMD GPUs, Google TPUs) motivates the need for fully disaggregated inference architectures. Future work should focus on building LLM "micro-services" for each heterogeneous LLM components, and building robust and high-performance tensor transmission frameworks and resource allocation mechanisms to efficiently connect these components.

CHAPTER 7

CONCLUSION

As machine learning applications, including video analytics and large language model (LLM) inference, become increasingly distributed, they face the critical challenge of efficiently loading data from remote sources to the machine learning models. Though prior work already observes the performance benefit of identifying the important data and load them first, this thesis emphasizes that accurately and rapidly prioritizing important data can further improve the performance of the end-to-end inference pipeline. Concretely, in video analytics, we demonstrate the advantage of selectively streaming pixels most relevant to object detection tasks and other tasks, achieving 2$\times$ latency reductions without sacrificing detection accuracy. In LLM applications, we show the benefit of prioritizing requests based on precise job completion time (JCT) estimations, significantly reducing latency while maintaining throughput.

REFERENCES

Vision meets drones: A challenge. http://www.aiskyeye.com/.

Audioset. http://research.google.com/audioset/index.html. (Accessed on 02/09/2023).

Vision navigates obstacles on the road to autonomous vehicles | automate. https://www.automate.org/industry-insights/vision-navigates-obstacles-on-the-road-to-autonomous-vehicles, a. (Accessed on 06/09/2023).

End-to-end deep learning for self-driving cars | nvidia technical blog. https://developer.nvidia.com/blog/deep-learning-self-driving-cars/, b. (Accessed on 06/09/2023).

Official implementation of efficient cascading residual network for sr. https://github.com/nmhkahn/CARN-pytorch.

character.ai | personalized ai for every moment of your day. https://character.ai/. (Accessed on 09/07/2024).

Benchmarking videos used in dds. https://github.com/KuntaiDu/dds.

x264 ffmpeg options guide - linux encoding. https://sites.google.com/site/linuxencoding/x264-ffmpeg-mapping. (Accessed on 09/10/2022).

OneAdapt: Driving Videos — docs.google.com. https://docs.google.com/spreadsheets/d/1KwRDkt2B7h_WemrK5K86MRz6_Y1czK3bz9u4kBnhvk4. [Accessed 02/15/2023].

Video coding for machines (the moving picture experts group). https://mpeg.chiariglione.org/standards/exploration/video-coding-machines.

The best frame rate for video. https://photographylife.com/best-frame-rate-for-video. (Accessed on 09/22/2023).

Extensions in arc: How to import, add, & open – arc help center. URL https://resources.arc.net/hc/en-us/articles/19434259167767-Extensions-in-Arc-How-to-Import-Add-Open. [Online; accessed 2025-04-17].

fairseq/readme.md at main · facebookresearch/fairseq · github. https://github.com/facebookresearch/fairseq/blob/main/examples/wav2vec/README.md. (Accessed on 02/09/2023).

Pytorch_yolov3/yolov3.py at master · dena/pytorch_yolov3 · github. https://github.com/DeNA/PyTorch_YOLOv3/blob/master/models/yolov3.py. (Accessed on 02/09/2023).

Neil Agarwal and Ravi Netravali. Boggart: Towards {General-Purpose} acceleration of retrospective video analytics. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 933–951, 2023.

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, 2024.

Namhyuk Ahn, Byungkon Kang, and Kyung-Ah Sohn. Fast, accurate, and lightweight super-resolution with cascading residual network. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 252–268, 2018.

Alexei Baevski, Yuhao Zhou, Abdelrahman Mohamed, and Michael Auli. wav2vec 2.0: A framework for self-supervised learning of speech representations. *Advances in neural information processing systems*, 33:12449–12460, 2020.

Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. Ekya: Continuous learning of video analytics models on edge compute servers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 119–135, 2022.

Christopher Canel, Thomas Kim, Giulio Zhou, Conglong Li, Hyeontaek Lim, David G Andersen, Michael Kaminsky, and Subramanya R Dulloor. Scaling video analytics on constrained edge nodes. *arXiv preprint arXiv:1905.13536*, 2019.

Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. *arXiv preprint arXiv:2005.12872*, 2020.

Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 155–168, 2015.

Vyacheslav V Chistyakov and Panos M Pardalos. Stability analysis in discrete optimization involving generalized addition operations. *Journal of Optimization Theory and Applications*, 167:585–616, 2015.

Liu Chunhui, Hu Yueyu, Li Yanghao, Song Sijie, and Liu Jiaying. Pku-mmd: A large scale benchmark for continuous multi-modal human action understanding. *arXiv preprint arXiv:1703.07475*, 2017.

High Efficiency Video Coding and ITUT Rec. H. 265 and iso, 2013.

Xin Dai, Xiangnan Kong, Tian Guo, and Yixian Huang. Cinet: Redesigning deep neural networks for efficient mobile-cloud collaborative inference. In *Proceedings of the 2021 SIAM International Conference on Data Mining (SDM)*, pages 459–467. SIAM, 2021.

Detectron2. Detectron2 model zoo. `https://github.com/facebookresearch/detectron2`.

Yixin Dong, Charlie F Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. Xgrammar: Flexible and efficient structured generation engine for large language models. *arXiv preprint arXiv:2411.15100*, 2024.

Kuntai Du, Ahsan Pervaiz, Xin Yuan, Aakanksha Chowdhery, Qizheng Zhang, Henry Hoffmann, and Junchen Jiang. Server-driven video streaming for deep learning inference. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 557–570, 2020.

Kuntai Du, Qizheng Zhang, Anton Arapin, Haodong Wang, Zhengxu Xia, and Junchen Jiang. Accmpeg: Optimizing video encoding for accurate video analytics. *Proceedings of Machine Learning and Systems*, 4, 2022.

Kuntai Du, Yuhan Liu, Yitian Hao, Qizheng Zhang, Haodong Wang, Yuyang Huang, Ganesh Ananthanarayanan, and Junchen Jiang. Oneadapt: Fast adaptation for deep learning applications via backpropagation. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, pages 158–176, 2023.

Kuntai Du, Bowen Wang, Chen Zhang, Yiming Cheng, Qing Lan, Hejian Sang, Yihua Cheng, Jiayi Yao, Xiaoxuan Liu, Yifan Qiao, et al. Prefillonly: An inference engine for prefill-only workloads in large language model applications. *arXiv preprint arXiv:2505.07203*, 2025.

Lingyu Duan, Jiaying Liu, Wenhan Yang, Tiejun Huang, and Wen Gao. Video coding for machines: A paradigm of collaborative compression and intelligent analytics. *IEEE Transactions on Image Processing*, 29:8680–8695, 2020.

John Emmons, Sadjad Fouladi, Ganesh Ananthanarayanan, Shivaram Venkataraman, Silvio Savarese, and Keith Winstein. Cracking open the dnn black-box: Video analytics with dnns across the camera-cloud boundary. In *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*, pages 27–32, 2019.

Hamed Firooz, Maziar Sanjabi, Adrian Englhardt, Aman Gupta, Ben Levine, Dre Olgiati, Gungor Polatkan, Iuliia Melnychuk, Karthik Ramgopal, Kirill Talanine, et al. 360brew: A decoder-only foundation model for personalized ranking and recommendation. *arXiv preprint arXiv:2501.16450*, 2025.

Kristian Fischer, Felix Fleckenstein, Christian Herglotz, and André Kaup. Saliency-driven versatile video coding for neural object detection. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1505–1509. IEEE, 2021.

Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*, 2013.

Alireza Ghasemieh and Rasha Kashef. 3d object detection for autonomous driving: Methods, models, sensors, data, and challenges. *Transportation Engineering*, 8:100115, 2022.

GitHub. Github copilot - write code faster. `https://copilot.github.com/`, 2025.

Wei Han, Pooya Khorrami, Tom Le Paine, Prajit Ramachandran, Mohammad Babaeizadeh, Honghui Shi, Jianan Li, Shuicheng Yan, and Thomas S Huang. Seq-nms for video object detection. *arXiv preprint arXiv:1602.08465*, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

João F Henriques, Rui Caseiro, Pedro Martins, and Jorge Batista. High-speed tracking with kernelized correlation filters. *IEEE transactions on pattern analysis and machine intelligence*, 37(3):583–596, 2014.

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

Parvaneh Janbakhshi and Ina Kodrasi. Experimental investigation on stft phase representations for deep learning-based dysarthric speech detection, 2021. URL `https://arxiv.org/abs/2110.03283`.

Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 253–266, 2018.

Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *Acm Sigplan Notices*, 52(4):615–629, 2017.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023a.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023b.

Yuanqi Li, Arthi Padmanabhan, Pengzhan Zhao, Yufei Wang, Guoqing Harry Xu, and Ravi Netravali. Reducto: On-camera filtering for resource-efficient real-time video analytics. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 359–376, 2020.

Haojie Liu, Han Shen, Lichao Huang, Ming Lu, Tong Chen, and Zhan Ma. Learned video compression via joint spatial-temporal correlation exploration. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 11580–11587, 2020.

Luyang Liu, Hongyu Li, and Marco Gruteser. Edge assisted real-time object detection for mobile augmented reality. In *The 25th annual international conference on mobile computing and networking*, pages 1–16, 2019.

Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 38–56, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706141. doi:10.1145/3651890.3672274. URL `https://doi.org/10.1145/3651890.3672274`.

Yoshitomo Matsubara, Sabur Baidya, Davide Callegaro, Marco Levorato, and Sameer Singh. Distilled split deep neural networks for edge-assisted real-time systems. In *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*, pages 21–26, 2019.

Yu Nesterov. Gradient methods for minimizing composite functions. *Mathematical programming*, 140(1):125–161, 2013.

Alexander Neubeck and Luc Van Gool. Efficient non-maximum suppression. In *18th international conference on pattern recognition (ICPR'06)*, volume 3, pages 850–855. IEEE, 2006.

OpenAI. Chatgpt: Conversational language model. `https://chat.openai.com`, 2025.

Perlexity AI. Perplexity is a free ai search engine. `https://www.perplexity.ai/`, 2025.

AB Ramazanov. On stability of the gradient algorithm in convex discrete optimisation problems and related questions. 2011.

Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 39(6):1137–1149, 2017.

Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

Yiding Wang, Weiyan Wang, Junxue Zhang, Junchen Jiang, and Kai Chen. Bridging the {Edge-Cloud} barrier for real-time advanced vision analytics. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the h.264/avc video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13 (7):560–576, 2003. doi:10.1109/TCSVT.2003.815165.

Sifeng Xia, Kunchangtai Liang, Wenhan Yang, Ling-Yu Duan, and Jiaying Liu. An emerging coding paradigm vcm: A scalable coding approach beyond feature and signal. In *2020 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6. IEEE, 2020.

Yi Xiao, Felipe Codevilla, Akhil Gurram, Onay Urfalioglu, and Antonio M López. Multimodal end-to-end autonomous driving. *IEEE Transactions on Intelligent Transportation Systems*, 23(1):537–547, 2020.

Tiantu Xu, Luis Materon Botelho, and Felix Xiaozhu Lin. Vstore: A data store for analytics on large videos. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019.

Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 94–109, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400711961. doi:10.1145/3689031.3696098. URL `https://doi.org/10.1145/3689031.3696098`.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*, 2025.

Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A Lee. Awstream: Adaptive wide-area streaming analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 236–252. ACM, 2018.

Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. Live video analytics at scale with approximation and {Delay-Tolerance}. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 377–392, 2017.

Miao Zhang, Fangxin Wang, and Jiangchuan Liu. Casva: Configuration-adaptive streaming for live video analytics. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 2168–2177. IEEE, 2022a.

Qizheng Zhang, Kuntai Du, Neil Agarwal, Ravi Netravali, and Junchen Jiang. Understanding the potential of server-driven edge video analytics. In *Proceedings of the 23rd Annual International Workshop on Mobile Computing Systems and Applications*, page 8–14, 2022b.

Tan Zhang, Aakanksha Chowdhery, Paramvir Victor Bahl, Kyle Jamieson, and Suman Banerjee. The design and implementation of a wireless video surveillance system. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 426–438. ACM, 2015.

Wuyang Zhang, Zhezhi He, Luyang Liu, Zhenhua Jia, Yunxin Liu, Marco Gruteser, Dipankar Raychaudhuri, and Yanyong Zhang. Elf: accelerate high-resolution mobile deep vision with content-aware parallel offloading. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 201–214, 2021.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs, 2024. URL `https://arxiv.org/abs/2312.07104`.

Zangwei Zheng, Xiaozhe Ren, Fuzhao Xue, Yang Luo, Xin Jiang, and Yang You. Response length perception and sequence scheduling: An llm-empowered llm inference pipeline. *Advances in Neural Information Processing Systems*, 36:65517–65530, 2023.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.