Application-Aware Scheduling in Deep Learning Software Stacks

by

Peifeng Yu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2022

Doctoral Committee:

Assistant Professor Mosharaf Chowdhury, Chair
Associate Professor Karthik Duraisamy
Associate Professor Xin Jin, Peking University
Assistant Professor Baris Kasikci

Peifeng Yu

peifeng@umich.edu

ORCID iD: 0000-0001-7001-6647

This dissertation is dedicated to my family.

For their endless love, support and encouragement

ACKNOWLEDGMENTS

I am finally approaching the finish of this dissertation, as well as my journey as a PhD student, while I am writing down these words. It would be impossible for me to complete this without the tremendous help from everyone around me.

I would like to express my deepest appreciation to my advisor, Professor Mosharaf Chowdhury, for the continuous support of my research, for his great patience and immense knowledge. Mosharaf has been an ideal mentor and supervisor for me. While always aiming for creating something useful beyond research manuscripts, his guidance keeps me from being lost in the forest of engineering efforts and helps me develop the mindset as a researcher. I am proud of, and grateful for, my time working with him.

Besides my advisor, I am extremely grateful to the rest of my thesis committee: Professor Karthik Duraisamy, Professor Xin Jin, and Professor Baris Kasikci. This dissertation would be much worse without their constructive feedbacks and comments.

Many of my projects come from successful collaborations. I am thankful to Jiachen Liu, whose mathematical proofs make Chapter 3 much stronger. Thanks should also go to Yuqing Qiu, for her help on data preparation and experiment running while I was working on Chapter 4, which greatly reduces my burden to explore new ideas. Similarly, other people who helped me in my projects also subjects to special thanks for their inspiration and cooperation in my study.

I would also like to extend my sincere thanks to everyone in the Symbiotic Lab: Dr. Juncheng Gu, Jie You, Hasan Al Maruf, Fan Lai, Yiwen Zhang, Sanjay S. Singapuram, Jiachen Liu, Jae-Won Chung, Insu Jang, as well as the Key To Success members: Haizhong Zheng, Rui Liu, Boyu Tian. I really enjoy every discussion and brainstorm we had, and every day and sleepless night we spent together before deadlines, and all the fun we have had in the last several years. They will all be my treasured memory. In particular, I am grateful to Juncheng, who has a lot for me to learn from as the first one to graduate in our lab – including this very dissertation. Thank you for letting me using yours as an example!

I would be remiss in not mentioning all the previous authors of the `thesis-umich` LaTeX template. Thanks to everyone sharing their own modifications to the template on GitHub, upon which I am able to build my own version.

Last but most importantly, words can not express my gratitude to my family for their uncondi-

tional, unequivocal, and loving support. My accomplishments are because my mother Ling Liang, and father Jian Yu, believe in me and keep my spirits and motivation high during the process. My girlfriend Jiahui Ji, is a real gem for me, supports me emotionally, and always gives good statistical advices as a data scientist. Without her beside me, this journey would have felt much longer and more stressful.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

TABLE

LIST OF ALGORITHMS

ALGORITHM

LIST OF APPENDICES

LIST OF ABBREVIATIONS

ALU Arithmetic Logic Unit

CDF Cumulative Distribution Function

CNN Convolutional Neural Network

CV Computer Vision

DL Deep Learning

DNN Deep Neural Network

FIFO First-In, First-Out

FPGA Field-Programmable Gate Array

HOL Head-of-Line

JCT Job Completion Time

MIG Multi-Instance GPU

ML Machine Learning

MPS Multi-Process Service

NLP Natural Language Processing

OS Operating System

PDF Probability Distribution Function

PPL perplexity

RNN Recurrent Neural Network

SLO Service Level Objective

SRTF Shortest Remaining Time First

TPU Tensor Processing Unit

VM Virtual Machine

ABSTRACT

Deep Learning (DL) has pervaded many areas of computing due to the confluence of the explosive growth of large-scale computing capabilities, availability of datasets, and advances in learning techniques. However, the infrastructure that supports DL is still in its early stage, bearing mismatches among the hardware, the software stack, and DL applications. On the one hand, despite the emergence of new unique hardware and new use cases, the software stack that abstracts and schedules these hardware resources remains largely unchanged. On the other hand, user-defined performance metrics common in DL applications urge better schedulers tailored to the application's specific needs. Motivated by the mismatch, this dissertation revisits the system design across the stack, with a focus on the synergy between schedulers and application/system-specific information.

At the bottom level, the ever-growing adoption of specialized hardware like GPUs poses challenges to efficient usage. Due to the lack of operating system arbitration, applications usually assume exclusive access, making the otherwise underutilized resources unusable for other jobs on the same host. We therefore design SALUS to realize proper efficient GPU sharing. It leverages DL applications' specific usage patterns to schedule iterations and manage memory allocations, providing two missing primitives: fast job switching and memory sharing.

However, even with an efficient execution platform, it is still not trivial to harvest the hardware's full potential for higher-level applications. We investigate two such cases sitting on opposite sides of a model's lifecycle: hyperparameter tuning and inference serving.

Hyperparameter tuning – which constitutes a great portion of DL cluster usage given the proliferation of distributed resources in clusters – generates many small interdependent training trials. Existing tuning algorithms are oblivious of advanced execution strategies like intra-GPU sharing and inter-GPU execution, often causing poor resource utilization. Hence, we propose FLUID as a generalized hyperparameter tuning execution engine, that coordinates between tuning jobs and cluster resources. FLUID schedules training trials in such jobs using a water-filling approach to make the best use of resources at both intra- and inter-GPU granularity to speed up hyperparameter tuning.

Moving on, inference serving also requires careful scheduling to achieve tight latency guarantees and maintain high utilization. Existing serving solutions assume inference execution times to be data-independent and thus highly predictable. However, with the rise of dynamic neural networks,

data-dependent inferences see higher variance in execution times and become less predictable by a single, point estimation of the true running times. With ORLOJ, we show that treating and modeling inference execution times as probability distributions bring large gains for scheduling inference requests in the presence of Service Level Objective (SLO) constraints.

In this dissertation, we consider combining application/system-specific information with scheduling design as a means of efficiently supporting new hardware and new DL application use cases. Nevertheless, the pursuit of higher efficiency never ends. This dissertation tries to lay down the necessary mechanisms with the hope that our crude work may be a basis for further research to better scheduling algorithms and more efficient systems in the DL infrastructure.

CHAPTER 1

Introduction

The popularity of Deep Learning (DL) are soaring by the day, with ubiquitous adoptions in many data-driven applications, ranging from autonomous vehicles running on the road and recommendation systems supporting large commercial companies, to facial recognition and spam filters found in personal devices [86]. DL has pervaded many areas of computing due to the confluence of the explosive growth of large-scale computing capabilities [46, 49, 61], availability of datasets [69, 94], and advances in learning techniques [2, 69, 72, 95].

The core of any DL application revolves around building a mathematical model – a function that derives its outcomes from the input (*inference*) based on a set of model parameters that were fit on previously observed data (*training*). In the simplest sense, the lifecycle of a DL model can be summarized as: *a*) preparing data including any feature engineering and data augmentation; *b*) training the model on large datasets; and *c*) deploying and serving the model for inference. Deployment of the model is not the end of the lifecycle, rather, it is the beginning of the next round of preparing-training-serving, as the problem domain keeps evolving with new data and insights gained from the deployed model can influence the next one being developed.

Consequently, it has been of great interest in both industry and academia to build a whole software stack that supports the lifecycle of DL models – e.g., TensorFlow [76], PyTorch [34], Ray [48, 64], Clockwork [7] and others [38, 58, 65, 66, 83, 104, 107].

Such a software stack, or *the deep learning infrastructure*, is vital to the DL ecosystem as a whole. On the one hand, it sits on top of the underlying computing devices like CPUs, GPUs or other accelerators, providing a unified interface, thus isolating researchers and practitioners from the details and subtleties of low-level systems. On the other hand, the DL infrastructure is the building block of most DL applications and workflows, providing essential frameworks and libraries that implement common tasks and making it possible to reuse the engineering efforts when applying DL to new problem domains.

Therefore, the DL infrastructure must stay in tight integration with applications above it, as well as the hardware below it. However, when compared to the last two, the DL infrastructure is still in its early stages, creating mismatches between the advanced application and hardware requirements,

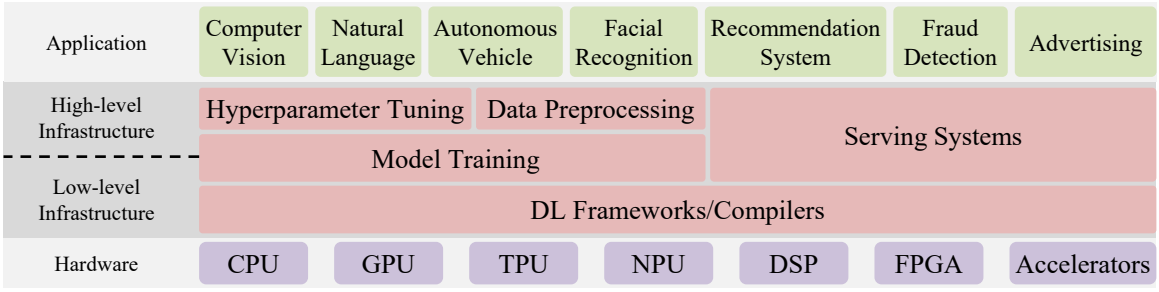


Figure 1.1: *The DL Infrastructure as a stack.*

and the inadequate infrastructure abilities. For example, despite the emergence of new unique types of hardware, the system that abstracts and schedules these hardware resources still leaves much to be desired in terms of performance, utilization and flexibility [3, 25, 27, 55, 90, 102, 110], failing to reach the full potential of the hardware. New application use cases also represent shifts in the execution characteristics at runtime [44, 47, 78]. When combined with user-defined performance metrics common in DL applications, they urge better schedulers tailored to the application's specific needs [7, 27, 58].

Motivated by the above-mentioned mismatches between the DL infrastructure and the application/hardware surrounding it, this dissertation proposes scheduling systems and techniques in different layers in the DL infrastructure, aiming to bring application and system-specific information into scheduler designs and thus alleviating gaps among the applications, the hardware and the infrastructure.

1.1 The Deep Learning Infrastructure as a Stack

One way to partition the field of systems research for DL, specifically its infrastructure, is by dividing it into high-level systems that interface with and support workflows for DL application development, and low-level systems that involve hardware or software to support training and execution of models (Fig. 1.1).

Categorizing by the model lifecycle, the high-level layer can be roughly divided into two groups: model training and model serving. Such a split in the ecosystem is a result of vastly different sets of goals while the model progressing from development to deployment. Model training values throughput (e.g., samples per second) and the ease of development, while model serving has constraints like latency or utilization.

These high-level frameworks themselves are supported by a set of common low-level libraries, like Tensorflow, PyTorch, or DL compilers and runtimes. Such low-level infrastructure layer handles specialized hardware, and often provides new types of programming interfaces to the higher level, and thus effectively decouples the model design and execution from applications.

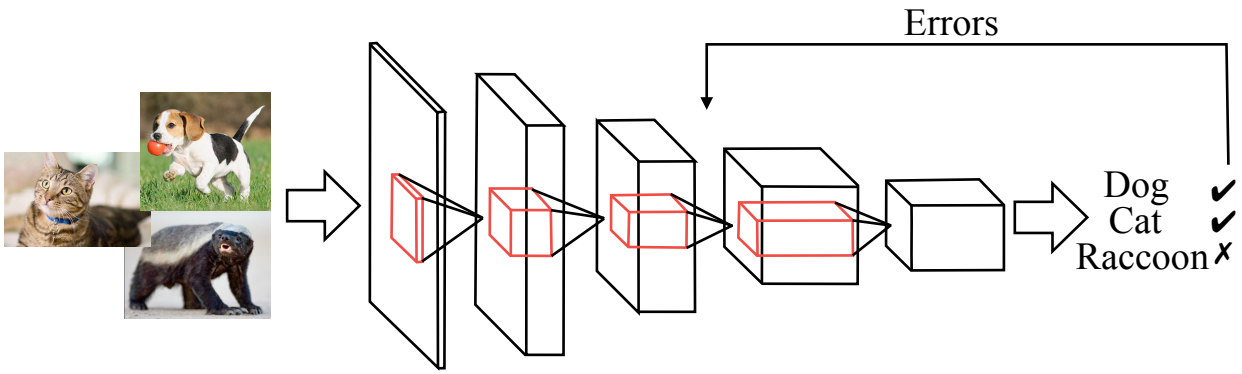


Figure 1.2: *Training a typical image classification model.*

The remainder of this section is organized as follows. Section 1.1.1 discusses important stages in a DL model's lifecycle, from hyperparameter tuning to production training, and finally model serving. Section 1.1.2 surveys the advent of specialized hardware, or accelerators, used throughout DL lifecycles and highlights the disconnect between the hardware and software stack above them. Section 1.1.3 provides an overview of large deployments of DL systems in clusters and on the cloud.

1.1.1 DL Lifecycles

Model Training DL models must be trained before they can be deployed for any practical use. The knowledge distilled from the training data is encoded in the model's parameters, which are used in addition to input data for the model to compute the final answer. Training of a DL model can be done in supervised and/or unsupervised manner, both happening in an iterative fashion. During a training iteration, the output of the model might not be correct. The magnitude of the error, which is quantified by a loss function, is used to calculate gradients such that when applied to update model parameters, the error can be minimized thereafter. This gradient calculating process is usually implemented as some form of a gradient descent algorithm [101, 121]. Figure 1.2 gives an example of an image classification model and its training data flow.

Model training is often expensive because of its need to keep and compute gradients for intermediate results. The training process can often take hours or even days, or otherwise require massive computing resources [56]. Because of the huge computational power required by DL training, it is paramount to speed up training for researcher productivity, and for increasing the scale of problems that can be tackled [59]. In addition to specialized hardware which will be discussed in Section 1.1.2, it is becoming more prevalent to leverage multiple workers working on the same model at the same time [51, 55], also known as parallel execution. For example, data parallelism partitions the input data across workers [105], model parallelism partitions model layers across workers [91, 104], and pipeline parallelism overlaps computation and communication of multiple

training iterations [31].

Hyperparameter Tuning While conceptually hyperparameter tuning is part of model training, it has its own characteristics that are worth separate explanation. The effectiveness of DL models is highly sensitive to *hyperparameters* [62], which control the model architecture and/or training process. It is important to settle down on a good set of hyperparameters before committing hours or days of hardware time to actually train the model in production.

Naturally, hyperparameter tuning has taken a central stage in machine learning clusters. Because of the high-dimensionality and non-differentiability of the search space, thousands of different hyperparameter settings often have to be evaluated before finding a final set for training in production. For instance, 2000 GPU days of reinforcement learning [75] or 3150 GPU days of evolution [52] are needed to obtain a state-of-the-art architecture for CIFAR-10 and ImageNet. In addition, a recent report suggested that over 86% of the jobs in a Microsoft GPU cluster with 5000 unique users perform hyperparameter tuning [27, 30].

Model Serving Serving model inference requests constitutes an increasingly larger portion of today's web requests [39, 45]. For example, NVIDIA estimated that 80–90% of the cloud AI workload was inference processing [138]. It is only likely to increase for the foreseeable future with the proliferation of DNN-powered APIs [133, 134] that enable applications to build on top of pre-built foundation models [1].

Inference requests are ideally handled at low latency, high throughput, and low cost, which are demanding goals to meet at scale. For example, Facebook serves over 200 trillion inference requests each day [12]. Unlike long-running model training, the underlying serving systems [7, 15, 38, 58, 66] that handle these inference requests aim to maximize throughput while reducing Service Level Objective (SLO) misses. Usually the SLO is to respond to the request within a latency budget, which may be derived from the user-facing service's deadline requirements. Due to the high throughput and low latency requirements of Deep Neural Network (DNN)-dependent applications and the large computation needs of DNN inference requests, modern serving systems often rely on expensive GPUs to serve many requests in parallel by batching them together [7, 38, 58].

1.1.2 Specialized Hardware in DL

DL models are primarily composed of many linear algebra computations (i.e. matrix-matrix, matrix-vector operations) which can be easily parallelized [36]. Specialized hardware, or accelerators are designed to accelerate those basic computation operations and reduce the execution latency and the cost of deploying DL model based applications. Despite the end of Moore's law [71], there is an explosion of innovation in DL processors and accelerators [6, 26].

While initially intended for graphics, GPUs with general purpose computing capabilities [143] has been a popular choice for DL. In addition to specialized libraries [141], NVIDIA first introduced Tensor Cores [144] in the Volta architecture [46], which enable mixed-precision computing, and have superior performance than normal CUDA cores on DL workloads. Like NVIDIA V100 [46], the later generation A100 [137] are pure computation cards intended for both inference and training, while the T4 GPU [136] is geared primarily to inference processing, though it can also be used for training.

Beyond GPUs, dataflow processors are custom-designed processors leveraging the deterministically laid out computation of DL models. They aim for dataflow processing in which computations, memory accesses and inter-Arithmetic Logic Unit (ALU) communications are all placed and routed on the physical hardware. Alibaba Hanguang 8000 [145] reported the highest inference rate for a chip when it was announced. The Inferentia chip [132] from Amazon Web Services is known for its perk inference performance with the int8 data type, while fp16 and bfloat16 types are also supported for compatibility with common DNN models. Baidu announced the Kunlun [131] AI accelerator chip with two variants for training and inference separately. Google has released 3 versions of Tensor Processing Unit (TPU) [61, 146].

In the research domain, quite a number of research teams have mapped one or more neural network models onto one or more Field-Programmable Gate Arrays (FPGAs) and collected a variety of performance and model prediction accuracy metrics [82, 89, 114].

1.1.3 DL in a Cluster and on the Cloud

More and more large-scale DL clusters have been built and shared among many users [49, 76, 129, 139]. These DL clusters are often equipped with GPUs for acceleration, and it is common for those clusters to directly adopt the resource management techniques that were designed for the traditional big-data clusters [27, 55] or need special scheduling algorithms to address the heterogeneity in the hardware [10]. DL jobs pose unique challenges on these existing resource management solutions. For example, many schedulers require resource sharing which can not be applied to GPU, since it only allows exclusive access [30, 51, 55].

Aside from internal infrastructures in large companies, there has also been the proliferation of DNN-powered APIs [133, 134] and public cloud DNN serving systems [130, 135, 139] that offer developers inference services that auto-scale based on load.

1.1.4 Summary

With the proliferation of DL-powered applications, we are likely to only see more distinctly new types of hardware, new applications, and new metrics for the foreseeable future. It naturally follows

the question, that *is the DL infrastructure stack able to adapt such emerging trends, and how?* We seek to answer this question in this dissertation, starting with identifying three mismatches below and above the DL infrastructure in the next section.

1.2 Mismatches at Boundaries Below and Above the Infrastructure

As discussed, while it is easier than ever to run state-of-the-art DL models on pre-packaged datasets, the infrastructure that supports DL in real-world applications is still in its early stages and becoming increasingly a major bottleneck in terms of adapting to new applications and hardware.

We identify three emerging trends that are creating challenges especially at the boundaries among the hardware, the applications and the infrastructure in between:

- **New Types of Hardware** There are distinctly new types of hardware beyond CPU, but many features we take for granted on CPUs, for example, time-sharing and space-sharing process scheduling, are impossible or impractical on these new hardware, due to minimal flexibility provided by accompanying low-level libraries. Such a lack of flexibility hinders the scheduler design and can lead to undesirable resource utilization.
- **New Application Use Cases** There are new application use cases beyond long-running batch processing jobs. However, their scheduling with respect to the cluster and lower-level frameworks remains largely unchanged, missing optimization opportunities from leveraging their unique runtime characteristics.
- **New Metrics** There are new metrics beyond predictive accuracy, like latency, cost, or power efficiency. They urge application-level context to be available across the software stack, so schedulers at lower levels can also enforce application defined metrics.

Thesis Statement – *As the software stack interfacing with hardware and applications, the DL infrastructure has yet to adapt to emerging new trends in layers below and above it, leading to inefficiencies across the stack. Motivated by such mismatches, this dissertation revisits the system design across the stack, with a focus on the synergy between schedulers and application/system-specific information.*

In this dissertation, we propose a suite of scheduling systems and techniques, tackling each of the aforementioned mismatches in the software stack (Fig. 1.3), showing that incorporating the combined information from applications and systems when designing schedulers can be a viable means to reduce the gap, and consequently, provide efficient support for new hardware and DL applications with broader use cases.

At the bottom level, the ever-growing adoption of specialized hardware like GPU poses challenges to efficient usage. Due to the lack of Operating System (OS) arbitration, applications usually

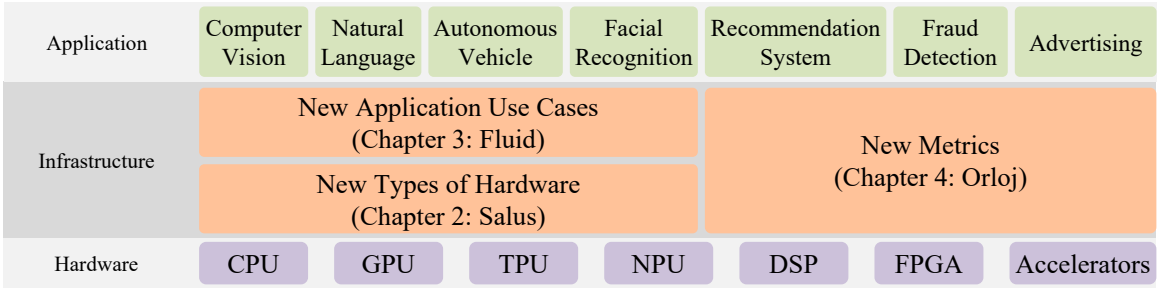


Figure 1.3: *Emerging trends create mismatches at the boundaries below and above the DL infrastructure.*

assume exclusive access, making the otherwise underutilized resource unusable for other jobs on the same host. We design **SALUS** to realize proper efficient GPU sharing. It leverages DL application specific usage patterns to schedule iterations and manage memory allocations, providing two missing primitives: fast job switching and memory sharing.

Even with an efficient execution platform, it is still not trivial to harvest the hardware's full potential for higher-level applications. We investigate two such cases sitting on opposite sides of a model's lifecycle: hyperparameter tuning and model inference serving.

Hyperparameter tuning – which constitutes a great portion of DL cluster usage given the proliferation of distributed resources in clusters – generates many small interdependent training trials. Existing tuning algorithms often ignore advanced execution strategies like intra-GPU sharing and inter-GPU execution, and thus suffer from poor resource utilization. Hence, we propose **FLUID** as a generalized hyperparameter tuning execution engine, that coordinates between tuning jobs and cluster resources. FLUID schedules training trials in such jobs using a water-filling approach to make the best use of resources at both intra- and inter-GPU granularity to speed up hyperparameter tuning.

Moving on, model inference serving also requires careful scheduling to achieve tight latency guarantees and maintain high utilization. Existing serving solutions assume inference execution times to be data-independent and thus highly predictable. However, with the rise of dynamic neural networks, data-dependent inferences see higher variance in execution times and become less predictable by a single, point estimation of the true running time. With **ORLOJ**, we show that probability distributions of execution times can be effectively exploited for scheduling inferences in the presence of SLO constraints.

With SALUS, FLUID, and ORLOJ, this dissertation redesigns important aspects in the ecosystem and aims to be a step towards the final answer of the question: *How should the DL infrastructure stack be adapted to new applications and new types of hardware?*

In the next section, we highlight our contributions and summarize the results.

1.3 Summary of Contributions and Results

SALUS The minimum granularity of GPU allocations today is often the entire GPU, which leads to two major inefficiencies. First, the coarse-grained, one-at-a-time GPU allocation model hinders the scheduling ability of GPU cluster managers, because a running DL job must be fully purged from the GPU before another one can start, incurring large performance overhead. Second, not all DL jobs can fully utilize a GPU all the time, but GPU's exclusive access model makes the otherwise underutilized resource unusable for other jobs on the same host.

We address these issues by presenting SALUS [18], which enables fine-grained sharing of individual GPUs with flexible scheduling policies among co-existing, unmodified DL applications. SALUS exposes two GPU sharing primitives: *fast job switching* and *memory sharing*. The former ensures that we can quickly switch the current active DL job on a GPU, enabling efficient time-sharing and preemption. The latter ensures high utilization by packing more small DL jobs on the same device. The unique memory usage pattern of DL applications is the key to why such primitives can be efficiently implemented in SALUS: we identify three different memory usage types and apply different management policies when handling them.

We have integrated SALUS with TensorFlow and evaluated it on a collection of DL workloads consisting of popular DL models. Our results show that SALUS improves the average completion time of DL training jobs by $3.19\times$ through efficiently implementing the Shortest Remaining Time First (SRTF) scheduling policy to avoid Head-of-Line (HOL) blocking. In addition, SALUS shows $2.38\times$ improvement on GPU utilization for the hyperparameter tuning workload, and $42\times$ over not sharing the GPU and $7\times$ over NVIDIA MPS for DL inference applications with small overhead.

FLUID Hyperparameter tuning constitutes a great portion of DL cluster usage, generates many small interdependent training trials. It however suffers from poor resource usage in today's distributed computation environment due to the oblivion of advanced execution strategies like intra- and inter-GPU sharing. We observe that the root cause of such suboptimal uses is hyperparameter tuning solutions' indifference to how trials are executed. But it is also unrealistic to leave the burden to Machine Learning (ML) researchers, expecting them to manually determine good execution strategies when tuning hyperparameters. However, the *wide variety of algorithms*, the *dynamicity of hyperparameter tuning workloads* and the *differences in training trial profiles* make it non-trivial to design a generalized hyperparameter tuning execution engine that can make efficient use of resources and speed up evaluating a group of hyperparameter configurations.

We propose FLUID [4], an algorithm- and resource-aware hyperparameter tuning execution engine that coordinates between the cluster and hyperparameter tuning algorithms. By abstracting hyperparameter tuning job as a sequence of *TrialGroup*s, FLUID provides a generic high-level

interface for hyperparameter tuning algorithms to express their execution requests for training trials. FLUID then automatically schedules the trials considering both the current workload and available resources to improve utilization and speed up the tuning process.

FLUID models the problem as a strip packing problem where each rectangle has different shapes and the goal is to minimize the height of the strip. By combining techniques like elastic training and GPU multiplexing, FLUID is able to change the trial size, scaling out across many GPUs, or scaling in sharing one GPU with other trials. We also propose heuristics in FLUID to solve the strip packing problem efficiently and prove that FLUID's performance is bounded within $2\times$ of the optimal.

ORLOJ Existing DNN serving solutions can provide tight latency SLOs while maintaining high throughput via careful scheduling of incoming requests, whose execution times are assumed to be highly predictable and data-independent. However, inference requests to emerging dynamic DNNs – e.g., popular Natural Language Processing (NLP) models and Computer Vision (CV) models that skip layers – are *data-dependent*. They exhibit poor performance when served using existing solutions because they experience large variance in request execution times depending on the input – the longest request in a batch inflates the execution times of the smaller ones, causing SLO misses in the absence of careful batching.

We develop ORLOJ, a dynamic DNN serving system, that captures this variance in dynamic DNNs using empirical distributions of expected request execution times, and then efficiently batches and schedules them without knowing a request's precise execution time. Going beyond prior distribution-based solutions for cluster scheduling and query processing [25, 50, 97], ORLOJ addresses two challenges unique to model serving. First, batching is vital to achieve high throughput, but it also affects execution times, as all requests in the same batch start and finish at the same time, regardless of their individual execution times. ORLOJ proposes a batch-aware priority score that derives batch execution time distributions given those of individual requests, and uses the score to guide its batching decisions. Second, because a model can receive requests from different applications, the joint distribution can be multimodal with even higher variance. To this end, ORLOJ tags each request with its originating application and relies on probability theory to accurately estimate batch execution times even when execution times of requests in the same batch follow different distributions.

ORLOJ significantly outperforms state-of-the-art serving solutions for high variance dynamic DNN workloads by 51–80% in finish rate under tight SLO constraints, and over 100% under more relaxed SLO settings. For well-studied static DNN workloads, ORLOJ keeps comparable performance with the state-of-the-art.

1.4 Thesis Structure

The rest of this dissertation is organized as follows. Chapter 2 introduces new GPU sharing primitives to enable more flexible scheduling and higher utilization through SALUS. Following the route of improving resource utilization, Chapter 3 presents FLUID, a hyperparameter tuning execution engine that mediates between the cluster and hyperparameter tuning algorithms to automatically adapt training trials to make the best use of available resources. Chapter 4 discusses ORLOJ, a dynamic neural network inference serving scheduler to improve inference scheduling finishing rate with tight SLO bounds under the condition of request execution time being data-dependent. We finally conclude the dissertation and discuss future work in Chapter 5.

CHAPTER 2

SALUS: Fine-Grained GPU Sharing Primitives for Deep Learning Applications

2.1 Introduction

GPUs have emerged as a popular choice in the context of DL training because of their excellence of highly parallelizable matrix operations common in DL jobs [46, 61, 76, 82]. Unfortunately, the minimum granularity of GPU allocation today is often the entire GPU – *an application can have multiple GPUs, but each GPU can only be allocated to exactly one application* [27]. While such exclusiveness in accessing a GPU simplifies the hardware design and makes it efficient in the first place, it leads to two major inefficiencies.

First, the coarse-grained, one-at-a-time GPU allocation model [1] hinders the scheduling ability of GPU cluster managers [24, 25, 55, 90, 102, 110, 143]. For flexible scheduling, a cluster manager often has to suspend and resume jobs (i.e., preempt), or even migrate a job to a different host. However, a running DL job must be fully purged from the GPU before another one can start, incurring large performance overhead. As such, GPU clusters often employ non-preemptive scheduling, such as First-In, First-Out (FIFO) [24, 27], which is susceptible to the HOL blocking problem; or they suffer large overhead when using preemptive scheduling [25].

Second, not all DL jobs can fully utilize a GPU all the time (Section 2.2). On the one hand, DL training jobs are usually considered resource-intensive. But for memory-intensive ones (e.g., with large batch sizes), our analysis shows that the average GPU memory utilization is often less than 50% (Section 2.2.1) due to varied memory usage over time and between iterations. Similar patterns can also be observed in compute-intensive training jobs. DL model serving also calls for finer-grained GPU sharing and packing. As the request rate varies temporally within the day as well as across models, the ability to hold many DL models on the same GPU during low request rates can significantly cut the cost by decreasing the number of GPUs needed in serving clusters [58, 63].

[1] It is possible to forcibly run multiple processes on the same GPU, but that leads to high overhead compared to exclusive mode.

Additionally, the increasingly popular trend of automatic hyperparameter tuning of DL models [68, 78, 96] further emphasizes the need to improve GPU utilization. This can be viewed as "pre-training." One exploration task usually generates hundreds of training jobs in parallel, many of which are killed as soon as they are deemed to be of poor quality. Improved GPU utilization by spatio-temporal packing of many of these jobs together results in shorter makespan, which is desirable because exploration jobs arrive in waves and the result is useful only after all jobs are finished.

We address these issues by presenting SALUS, which enables fine-grained sharing of individual GPUs with flexible scheduling policies among co-existing, unmodified DL applications. While simply sharing a GPU may be achievable, doing so efficiently is not trivial (Section 2.2.2). SALUS achieves this by exposing two GPU sharing primitives: *fast job switching* and *memory sharing* (Section 2.3). The former ensures that we can quickly switch the current active DL job on a GPU, enabling efficient time-sharing and preemption. The latter ensures high utilization by packing more small DL jobs on the same device. The unique memory usage pattern of DL applications is the key to why such primitives can be efficiently implemented in SALUS: we identify three different memory usage types and apply different management policies when handling them (Section 2.3.2). Combining these two primitives, we implement a variety of GPU scheduling solutions (Section 2.4).

We have integrated SALUS with TensorFlow and evaluated it on a collection of DL workloads consisting of popular DL models (Section 2.5). Our results show that SALUS improves the average completion time of DL training jobs by $3.19\times$ by efficiently implementing the SRTF scheduling policy to avoid HOL blocking. In addition, SALUS shows $2.38\times$ improvement on GPU utilization for the hyperparameter tuning workload, and $42\times$ over not sharing the GPU and $7\times$ over NVIDIA MPS for DL inference applications with small overhead.

2.2 Background and Motivation

2.2.1 DL Workloads Characteristics

We analyzed a collection of 15 DL models (Table A.1 in Appendix) to understand the resource usage patterns of DL jobs. This set of models are compiled from the official TensorFlow Convolutional Neural Network (CNN) benchmarks [147] and other selected popular models in respective fields.

In order to cover a wider range of use cases, while keeping the native input characteristics, we varied the batch size to create 45 distinct workloads, as shown in Table A.1. Note that the batch size specifies the number of samples (e.g., images for CNNs) trained in each iteration and affects the size of model parameters. Thus, the larger the batch size, the longer it takes to compute an iteration. Throughout the paper, we uniquely identify a workload by the model name plus the input batch size.

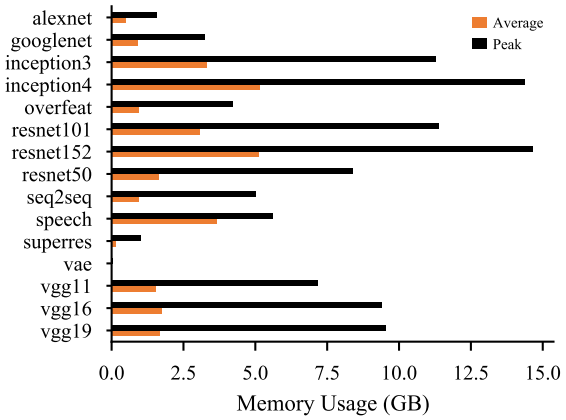


Figure 2.1: *Average and peak GPU memory usage per workload.* [1]

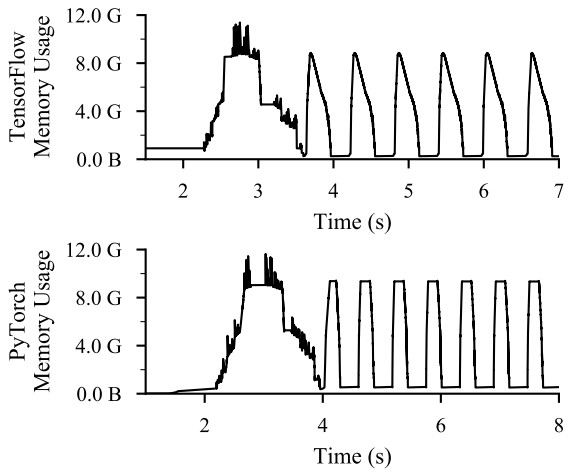


Figure 2.2: *Part of the GPU memory usage trace showing the spatio-temporal pattern.* [2]

For example, `alexnet_25` means a job training `alexnet`, with a batch size of 25.

In terms of GPU resource usage, one can consider two high-level resources: (i) GPU computation resources (primarily in terms of computation time, often referred to as GPU utilization in the literature) and (ii) GPU memory. We found that both are often correlated with the complexity of the DL model. However, GPU memory is especially important because *the entire DL model and its associated data must reside in memory for the GPU to perform any computation*; in contrast, computations can be staggered over time given sufficient GPU memory.

In the following, we highlight a few key GPU memory usage patterns in DL workloads that lead to memory underutilization issues and/or opportunities for improvements.

Heterogeneous Peak Memory Usage Across Jobs DL workloads are known for heavy memory usage [76, 91, 93]. Figure 2.1 visualizes the average and peak memory usages of our workloads. As models become larger (with more and wider layers) and the batch size increases, memory requirements of DL jobs increase as well. For example, we observed peak memory usages as high as 13.8 GB for `resnet152` and as low as less than 1 GB for `vae`. Such high variations suggest that even during peak allocation periods, it may be possible to run multiple models on the same GPU instead of FIFO.

Temporal Memory Usage Variations Within a Job Within each job, however, each iteration of a DL training job is highly predictable with a well-defined peak memory usage and a trough in

[1] Measured in TensorFlow and running on NVIDIA P100. The average and peak usage for vae is 22 MB, 35 MB, which are too small to show in the figure. The appendix also includes the measurement in PyTorch (Fig. A.1), which shares a similar pattern.

[2] Measured when training `resnet101_75` on NVIDIA P100.

between iterations. Figure 2.2 shows an example. This is because DL jobs go through the same sequence of operations and memory allocations in each iteration. The presence of predictable peaks and troughs can help us identify scheduler invocation points.

Low Persistent Memory Usage Another important characteristic of GPU memory usage of DL jobs is the use of persistent memory to hold parameters of a model – this corresponds to the consistent troughs across iterations. Even though the peak usage can be very high, most of it is temporary data created and destroyed within the same iteration. Fortunately, the size of persistent memory is often very low in comparison to the peak, ranging from 110.9 MB for `googlenet_25` to 822.2 MB for `resnet152_75`. *As long as the model parameter is already in GPU memory, we can quickly start an iteration of that model.* This gives us an additional opportunity to improve sharing and utilization.

2.2.2 Existing Techniques for Sharing GPUs

Given that DL workloads leave ample room for GPU sharing, a straw man approach would be disabling the exclusive access mode and statically partitioning the GPU memory among DL jobs. This cannot completely address the underutilization problem due to high peak-to-average memory usage of DL jobs.

NVIDIA's Multi-Process Service (MPS) [140] can be used to speed up the static partitioning approach for GPU sharing by avoiding costly GPU context switches. Nonetheless, MPS has limited support for DL frameworks or companies' in-house monitoring tool according to our experiments and various bug reports.

Static partitioning and MPS also fail to provide performance isolation. Co-located DL jobs can cause large and hard to predict interferences. A recent work, Gandiva [55] approaches this by trial and error and fallback to non-sharing mode. Xu et al. [40] propose to use machine learning model to predict and schedule GPU-using Virtual Machines (VMs) in the cluster to minimize interferences.

NVIDIA's TensorRT Inference server [63] and the prior work by Samanta et al. [37] achieve simultaneous DL inference in parallel on a single GPU. But they lack support for DL training.

Finally, earlier works on fine-grained GPU sharing fall into two categories. Some attempt to intercept GPU driver API calls and dynamically introduce concurrency by time-slicing kernel execution at runtime [87, 100, 112]. Others call for new APIs for GPU programming [57, 73, 81]. These solutions are designed for jobs with a few GPU kernels; as such, they are not scalable to DL applications, where the number of unique kernels can easily go up to several hundreds.

2.3 SALUS

SALUS [1] is our attempt to build an ideal solution to GPU sharing. It is designed to enable efficient, fine-grained sharing while maintaining compatibility with existing frameworks (Section 2.3.1). Its overall design is guided by the unique memory usage characteristics of DL jobs. Packing multiple jobs onto one GPU changes the combined memory allocation patterns and special care must be taken to mitigate increased fragmentation, because existing DL frameworks are designed for the job-exclusive GPU usage scenario. SALUS addresses both temporal and spatial aspects of the memory management problem by enabling two GPU sharing primitives:

- Fine-grained time-sharing via *efficient job switching* among ongoing DL jobs (Section 2.3.2);
- Dynamic memory sharing via the *GPU lane* (Section 2.3.3).

Together, these primitives open up new scheduling and resource sharing opportunities. Instead of submitting one job at a time, which can easily lead to HOL blocking, one can perform preemption or run multiple DL jobs in a time- or space-shared manner – all of which can be utilized by a GPU cluster scheduler [25, 55]. We demonstrate the possibilities by implementing common scheduling policies such as preempting jobs for SRTF or fair sharing, and packing many jobs in a single GPU to increase its utilization (Section 2.4).

2.3.1 Architectural Overview

At the highest level, SALUS is implemented as a singleton *execution service*, which consolidates all GPU accesses, thus enabling sharing while avoiding costly context switch among processes on the GPU. As a result, any unmodified DL job can leverage SALUS using a DL framework-specific *adaptor* (Fig. 2.3).

From a framework's point of view, the adaptor abstracts away low level details, and SALUS can be viewed as another (virtual) computation device; From a user's perspective, the API of the framework does not change at all. All scripts will work the same as before.

It is perhaps better to explain the architecture via an example of the life cycle of a DL job. When a job is created in a user script, SALUS *adaptor* in the DL framework creates a corresponding session in SALUS (1a). The computation graph of the DL job is also sent to SALUS during the creation. The session then proceeds to request a lane from the *memory manager* (1b). Depending on existing jobs in the system, this process can block, and the session will be queued (Section 2.3.3). During the job's runtime, either training or inferencing, iterations are generated by the user script and forwarded to the corresponding session in SALUS (2a). They are then scheduled according to their associated GPU lanes by the iteration scheduler (2b), and send to the GPU for execution.

[1] SALUS is available as an open-source software at https://github.com/SymbioticLab/Salus.

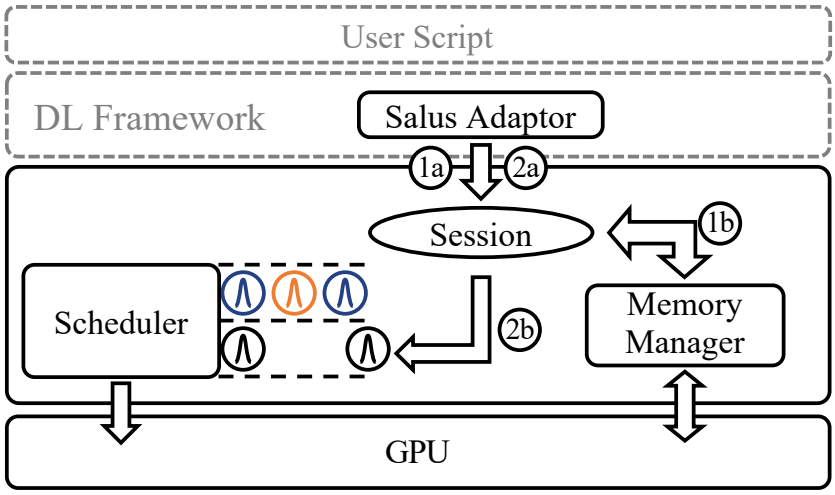


Figure 2.3: SALUS *sits between frameworks and the hardware in the DL stack, transparent to users.*

The SALUS execution service thus achieves GPU sharing via iteration-granularity scheduling of DL jobs. We elaborate on a performance-efficiency tradeoff in choosing this granularity (Section 2.3.2.2).

2.3.2 Efficient Job Switching

The ability to switch between jobs is paramount to implement time-sharing and preemption – two techniques extensively used by modern schedulers in many contexts. Suspending a running job and resuming the same or another one have always been possible on GPU as well. Modern DL frameworks extensively use checkpoints to mitigate data and computation loss due to the long-running nature of DL training jobs. The same technique is applied by Gandiva [55] to achieve second-scale suspend/resume. Nevertheless, checkpoints can result in large data transfers from and to the GPU memory, even in the best case when only model parameters are transferred, the communication time is still non-negligible. It even becomes unacceptable if the system ever wants to support inference workloads: the theoretical minimal transfer time can be even several times longer than the inference latency itself, according to the measurement on our collection of workloads (Fig. 2.4).

Observation 1. *Transferring GPU memory back and forth is not practical to achieve low latency given current GPU communication bandwidth.*

2.3.2.1 Characterizing DL Memory Allocations

We observe that one can push things further by taking a close look at different types of memory allocations in a DL job. Specifically, we define three types of memory allocations with unique

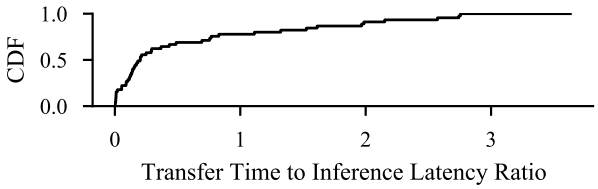


Figure 2.4: *CDF of theoretical minimal transfer time to model inference latency ratio for 15 models, assuming* 30 *GB/s transfer speed.*

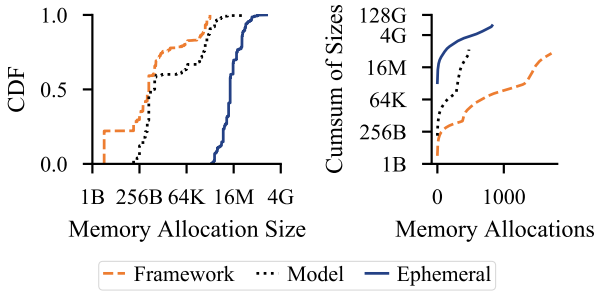


Figure 2.5: *Allocation size distribution per memory type in* `inception3_50`.

characteristics.

1. *Model:* These mostly hold model parameters and typically consist of a few large chunks of memory. They are persistent because they have to be available throughout the whole job's lifetime. Because the model size is typically fixed during the entire training process, model data has little or no temporal variations and is predictable.
2. *Ephemeral:* These are the scratch memory needed during each iteration. They usually hold intermediate layers' outputs as well as temporary data generated by the algorithm itself. Ephemeral allocations are only needed during computations and are released between iterations, giving rise to the temporal memory usage patterns of DL jobs. They are often large memory allocations as well.
3. *Framework-internal:* These are usually used by the DL framework for bookkeeping or for data preparation pipeline. They often persist across iterations.

Collectively, model and framework-internal memory are *persistent* across iterations. As an example, Fig. 2.5 gives the memory allocation size distribution for a popular CNN workload: `inception3_50`.

Observation 2. *There is significantly less persistent memory than ephemeral ones in a DL job. It is possible to keep more than one job's persistent memory in GPU while still having enough space for either one's ephemeral memory.*

The above two observations naturally lead to the conclusion that fast job switching can be enabled by not removing persistent memory from GPU at all. Thus unlike existing works [55], SALUS is designed to enable significantly faster suspend/resume operations by keeping persistent memory around, and then an iteration-granularity job scheduler (e.g., time-sharing or preemption-based) decides which job's iteration should be run next.

2.3.2.2 Scheduling Granularity

Given that iterations are typically short in DL jobs (ranging from tens of milliseconds to a few seconds), with an even finer granularity, e.g., at the GPU kernel level, it may be possible to further utilize GPU resources. However, finer-grained scheduling also adds more overhead to the execution service. Indeed, there is a tradeoff between maximum utilization and efficiency for a given scheduling granularity.

To understand this tradeoff, we prototyped a GPU kernel-level switching mechanism as well only to find that scheduling at that level incurs too much overhead for little gain. It requires all GPU kernels to go through a central scheduler, which, in addition to becoming a single bottleneck, breaks common efficiency optimizations in DL frameworks such as kernel batching and pipelining.

2.3.3 Spatial Sharing via GPU Lane

Although DL jobs' memory usages have spatio-temporal variations, many cannot reach the total capacity of a GPU's memory. Naturally, we must consider ways to better utilize the unused memory.

Built on top of the efficient job switching, we design a special memory layout scheme, the *GPU Lane*, that achieves memory sharing and improves memory utilization.

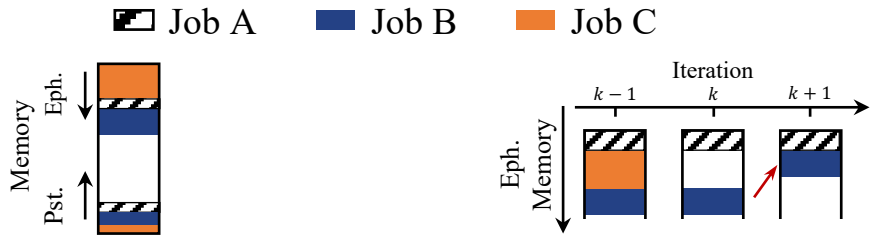
First, learning from classic memory management techniques of stack and heap to separate dynamic allocations from static ones, we divide GPU memory space into *ephemeral* (Eph.) and *persistent* (Pst.) regions, growing from both end of the memory space (Fig. 2.6a). A DL job's ephemeral memory goes into the ephemeral region, while other types of memory is allocated in the persistent region.

The ephemeral region is further divided into *lanes*, which are continuous memory spaces that can contain ephemeral memory allocation for iterations. Lanes are not only about memory, though. Iteration execution is serialized within a lane and parallelism is achieved across lanes, which is implemented using GPU streams. Each lane can be assigned to multiple DL jobs, which are time-shared within the lane.

The lane's restriction on execution is necessary because different from the other two types of memory, ephemeral allocations happens in small chunks and cannot be predicted ahead. As a result, simply putting two iterations together may cause deadlock because there is no swapping [1] for the oversubscribed memory.

Even if enough memory is ensured for both peak memory usage for two iterations, memory fragmentation can still cause superfluous out-of-memory errors if not handled correctly. More specifically, while the framework-internal memory allocations are small in size, they can have a large impact on the overall memory layout and may create more memory fragments when multiple

[1] The existing memory overcommit technique Unified Memory Access is too slow to use. See Section 2.5.4.



(a) *Memory regions* **(b)** *Auto defragmentation at iteration boundaries.*

Figure 2.6: *The memory layout of the GPU lane scheme.*

iterations are allocating simultaneously. While there are works implementing a memory planner before actually starting the iteration [83], they are not available to all frameworks.

Since our goal is to fully support existing workloads with minimal impact on the user, we approach the problem by limiting the dynamic allocation in the ephemeral region and isolate memory allocations across lanes to ensure maximum compatibility while achieving adequate flexibility.

2.3.3.1 Lane Auto Defragmentation

Having lanes does not eliminate memory fragmentation, it moves fragmentation within lane to fragmentation at the lane level. However, defragmentation is much easier at this level. Traditionally, defragmentation is achieved by first moving data out of memory and later moving it back again. In case of lanes, the allocations are released completely at the end of each iteration – they are ephemeral memory after all. Therefore, defragmentation happens almost automatically at no cost: no extra memory movement is needed.

Consider the situation illustrated in Fig. 2.6b, when job C stops, its lane space is quickly reclaimed (the red arrow) at the iteration boundary by job B that was allocated below it.

2.3.3.2 Lane Assignment

It is vital to determine the size and number of lanes in the GPU, as well as how lanes are assigned to jobs. SALUS uses a simple yet efficient algorithm to decide between opening a new lane and putting jobs into existing lanes.

Throughout the process, the following "safety" condition is always kept to make sure the persistent region and ephemeral region do not collide into each other:

$$\sum_{\text{job } i} P_i + \sum_{\text{lane } l} \max_{\text{job } j \text{ in } l} (E_j) \leq C \tag{2.1}$$

where P and E are respectively the persistent (model and framework-internal) and ephemeral

Algorithm 2.1: Find GPU Lane for Job

Input:
$P \leftarrow$ the job's persistent memory requirement
$E \leftarrow$ the job's ephemeral memory requirement
$C \leftarrow$ total memory capacity of the GPU
$P_i \leftarrow$ persistent memory usage of existing job i
$L_j \leftarrow$ lane size of existing lane j
$\mathbb{L} \leftarrow$ set of existing lanes

1 **if** $\sum_i P_i + P + \sum_j L_j + E \leq C$ **then**
2 $lane \leftarrow$ new GPU lane with capacity E
3 **return** $lane$
4 **foreach** $j \in \mathbb{L}$ **do**
5 **if** $L_j \geq E$ *and is the best match* **then**
6 **return** j
7 **foreach** $r \in \mathbb{L}$ *in* L_r *ascending order* **do**
8 **if** $\sum_i P_i + P + \sum_j L_j - L_r + E \leq C$ **then**
9 $L_r \leftarrow E$
10 **return** r
11 **return** *not found*

memory usage of a job. C is the capacity of the GPU. The second term is the sum of all lanes' size, which is defined as the maximum ephemeral memory usage of all jobs in the lane.

By ensuring enough capacity for persistent memory of all the admitted jobs and enough remaining for the iteration with the largest temporary memory requirement, SALUS increases the utilization while making sure that at least one job in the lane can proceed.

Implementation-wise, the system is event-driven, and reacts when there are jobs arriving or finishing, or at iteration boundaries when auto defragmentation happens. The lane finding logic is shown in Algorithm 2.1, which outputs a suitable lane given a job's memory requirement.

It is beyond the scope of this work to find the best strategy to reorganize lane assignments. We find the one implemented in our algorithm works fairly well in practice, but there are more possibilities about finding the optimal number of lanes given a set of jobs.

2.4 Scheduling Policies in SALUS

The state-of-the-art for running multiple DL jobs on a single GPU is simply FIFO, which can lead to HOL blocking. Although recent works [25, 55] have proposed time-sharing, they enforce sharing over many minutes due to high switching overhead.

Thanks to its fine-grained GPU sharing primitives, SALUS makes it possible to pack jobs

together to increase efficiency, or to enforce any priority criteria with preemption. It opens up a huge design space to be explored in future works.

To demonstrate the possibilities, in our current work, we have implemented some simple scheduling policies, with SALUS specific constraints (i.e., safety condition). The `PACK` policy aims to improve resource utilization and thus makespan, the `SRTF` policy is an implementation of SRTF, and the `FAIR` policy tries to equalize resource shares of concurrent jobs.

2.4.1 `PACK` to Maximize Efficiency

To achieve higher utilization of GPU resources, many jobs with different GPU memory requirements can be packed together in separate GPU lanes based on their memory usages. However, packing too many lanes exceeding the GPU memory capacity will either crash the jobs or incur costly paging overhead (if memory overcommit is enabled), both of which would do more harm than good. Consequently, this policy works with "safety" condition to ensure that the total peak memory usage across all lanes is smaller than the GPU memory capacity. No fairness is considered among lanes.

Apart from running training jobs or hyperparameter searching jobs in parallel, this can also enable highly efficient inference serving. By simultaneously holding many models in the same GPU's memory, SALUS can significantly decrease the GPU requirements of model serving systems like Clipper [58].

2.4.2 `SRTF` to Enable Prioritization

Developing DL models are often an interactive, trial-and-error process where practitioners go through multiple iterations before finding a good model. Instead of waiting for an ongoing large training to finish, SALUS can enable preemption – the large job is paused – to let the smaller one finish faster. This way, SALUS can support job priorities based on arbitrary criteria, including size and/or duration to implement the SRTF policy. The higher priority job is admitted as long as its own safety condition is met – i.e., at least, it can run alone on the GPU – regardless of other already-running jobs.

Note that we assume the job execution time is known, and it is thus possible to implement SRTF. While there are works on how to estimate such job execution time [51], the subject is beyond the scope of this paper, and we only focus on providing primitives to enable the implementation of such schedulers.

2.4.3 `FAIR` to Equalize Job Progress

Instead of increasing efficiency or decreasing the average completion time, one may want to share the GPU among DL jobs during high contention periods. Note that there may be different definitions of *fairness*, and we demonstrate the feasibility of implementing one or more of them instead of proposing the optimal fairness policy. Specifically, we admit new jobs into the GPU while maintaining the safety condition, and equalize total service over time for jobs in each lane.

2.5 Evaluation

We have integrated SALUS with TensorFlow and evaluated it using a collection of training, hyperparameter tuning, and inference workloads [80, 92, 95, 99, 147] to understand its effectiveness and overhead. The highlights of our evaluation are as follows:

- SALUS can be used to implement many popular scheduling algorithms. For example, the preemptive SRTF scheduler implemented in SALUS can outperform FIFO by $3.19\times$ in terms of the average completion time of DL training jobs (Section 2.5.1).
- SALUS can run multiple DL jobs during hyperparameter tuning, increasing GPU utilization by $2.38\times$ (Section 2.5.2).
- Similarly, for inference, SALUS can improve the overall GPU utilization by $42\times$ over not sharing the GPU and $7\times$ over NVIDIA MPS (Section 2.5.3).
- SALUS has relatively small performance overhead given its flexibility and gains (Section 2.5.4).

Environment All experiments were done on a x86_64 based Intel Xeon E5-2670 machine with 2 NVIDIA Tesla P100 GPUs available in CloudLab [23]. Each GPU has 16 GB on-chip memory. TensorFlow v1.5.0 and CUDA 8.0 are used in all cases.

Baseline (s) Our primary baseline is the FIFO scheduling commonly used in today's GPU clusters [55]. We also compare against NVIDIA MPS.

2.5.1 Long-Running Training

We start by focusing on SALUS's impact on training. To this end, we evaluate SALUS using a job trace of 100 workloads, generated using the jobs described in Table A.1. We considered multiple batch sizes and durations of each training job in the mix. The overall distribution followed one found in a production cluster [25].

We compare four different schedulers:

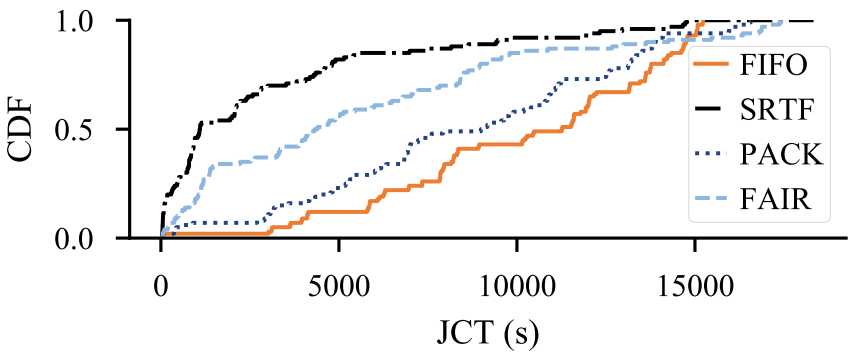


Figure 2.7: *CDFs of JCTs for all four scheduling policies.*

Table 2.1: *Makespan and aggregate statistics for different schedulers.*

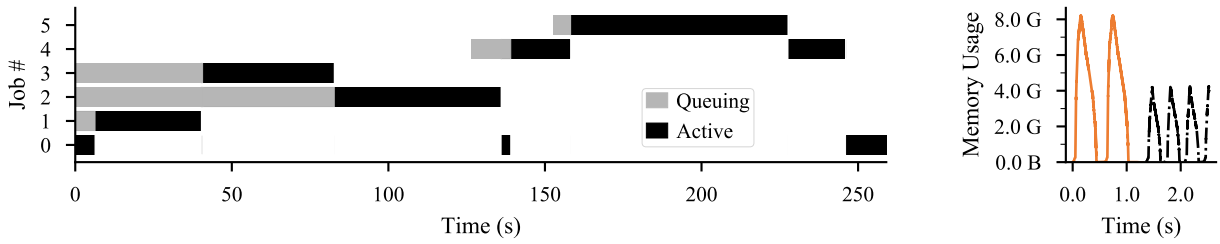
Scheduler	Makespan	Average Queuing	Average JCT	95% JCT
FIFO	303.4 min	167.6 min	170.6 min	251.1 min
SRTF	306.0 min	28.6 min	53.4 min	217.0 min
PACK	287.4 min	129.9 min	145.5 min	266.1 min
FAIR	301.6 min	58.5 min	96.6 min	281.2 min

1. **FIFO** refers to processing jobs in order of their arrival. This is the de facto mechanism in use today.
2. **SRTF** is a preemptive shortest-remaining-time-first scheduler. We assume that the duration is known or can be estimated [51].
3. **PACK** attempts to pack as many jobs as possible in to the GPU. The goal is to minimize the makespan.
4. **FAIR** uses time-sharing to equally share the GPU time among many jobs.

2.5.1.1 Overall Comparison

Figure 2.7 presents the distributions of Job Completion Times (JCTs) for all four policies, while Table 2.1 presents makespan and aggregate statistics. Given the similarities of makespan values between FIFO, SRTF, and FAIR, we can conclude that SALUS introduces little overhead. Furthermore, packing jobs can indeed improve makespan. Note that because of online job arrivals, we do not observe large improvement from PACK in this case. However, when many jobs arrive together, PACK can indeed have a larger impact (Section 2.5.2).

These experiments also reestablishes the fact that in the presence of known completion times, SRTF can indeed improve the average JCT – 3.19× w.r.t. FIFO in this case.



(a) *A slice of 6 jobs switching between each other. Gray areas represent the waiting between a job arrives, and it actually gets to run. Black areas represent active execution.*

(b) *Memory usage during a job switching.*

Figure 2.8: *Details of a snapshot during the long trace running with* `SRTF`*. In both slices, time is normalized.*

2.5.1.2 Impact of Fast Job Switching

We evaluate SALUS's ability to perform fast job switching in two contexts. First, we show that it allows cheap preemption implementation, which, in turn, makes possible the SRTF scheduling policy. Second, we show fast job switching can achieve fair sharing among DL jobs in seconds-granularity – instead of minutes [55]. In both cases, we consider a single GPU lane.

SRTF Consider the following scenario: a large training job has been running for a while, then the user wants to quickly do some test runs for hyperparameter tuning for smaller models. Without SALUS, they would have to wait until the large job finishing – this is an instance of HOL blocking. SALUS enables preemption via efficient switching to run short jobs and resumes the larger job later.

We pick a segment in the long job trace, containing exact the scenario, and record its detailed execution trace, showing in Fig. 2.8a. When job #1 arrives, the background job #0 is immediately stopped and SALUS switches to run the newly arrived shorter job. Job #2 comes earlier than job #3, but since #3 is shorter, it is scheduled first. And finally since job #5 is shorter, #4 is preempted and let #5 run to completion. During the process, the background job #0 is only scheduled when there is no other shorter job existing.

Figure 2.8b is another example demonstrating SALUS's ability to fast switch. It visualizes memory allocations in the scale of seconds: at the moment of a job switching, the second job's iteration starts immediately after the first job stops.

Time Sharing/Fairness To better illustrate the impact of fairness, we show another microbenchmark, demonstrating SALUS's ability to switch jobs efficiently using 3 training jobs and focusing on the fair sharing of GPU throughput in Fig. 2.9.

For ease of exposition, we picked three jobs of the same DL model `inception3_50` – this allows us to compare and aggregate training throughput of the three models in terms of images processed per second. In this figure, in addition to the throughput of individual jobs, the black dashed line shows the aggregate throughput.

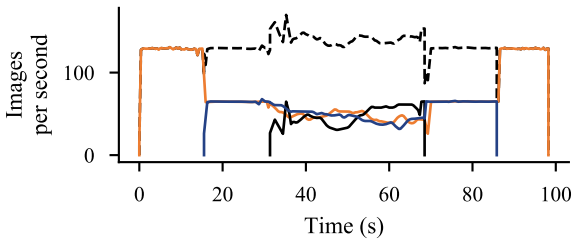


Figure 2.9: *Fair sharing among three* `inception3_50` *training jobs. Black dashed line shows the overall throughput.*

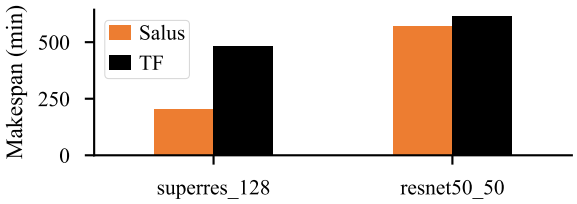


Figure 2.10: *Makespan of two hyperparameter tuning multi-jobs each of which consists of 300 individual jobs.*

The training jobs start at time 0s, 15s and 30s. At 15s, when the second job starts, while the total throughput remains unchanged, each job's share is halved. It further reduces to about a third when the third job arrives. Similarly, the reverse happens when jobs finishes in the reverse order. The system throughput roughly remains the same throughout the experiment. Note that SALUS reacts almost immediately for job arriving and leaving events.

In contrast, FIFO scheduling or other sharing policies (e.g., MPS) cannot enforce fair sharing.

2.5.2 Hyperparameter Exploration

Using SALUS to `PACK` many jobs is especially useful when many/all jobs are ready to run. One possible use case for this is automatic hyperparameter tuning. Typically, hundreds of training jobs are generated in parallel for parameter exploration. Most of the generated models will be killed shortly after they are deemed to be of poor quality. In this case, increasing the concurrency on GPU can help improve the parameter exploration performance by running multiple small jobs together, whereas today only FIFO is possible.

We evaluate two sets of hyperparameter exploration jobs: `resnet50_50` and `superres_128`, for image classification and resolution enhancement, respectively. Each set has 300 jobs, and each one completes after all 300 complete. A comparison of achieved makespan using FIFO (in TensorFlow) and SALUS is shown in Fig. 2.10. In the `resnet50_50` case, there is $1.07\times$ makespan improvement while it is $2.38\times$ for `superres_128`.

Little improvement is seen for `resnet50_50` because while the GPU has enough memory, computation becomes the bottleneck under such heavy sharing. Consequently, the makespan does not see much improvement.

2.5.3 Inference

So far we have only discussed DL training, but we note that serving a trained model, i.e., inference, can also be a good – if not better – candidate for GPU memory sharing. Rather than

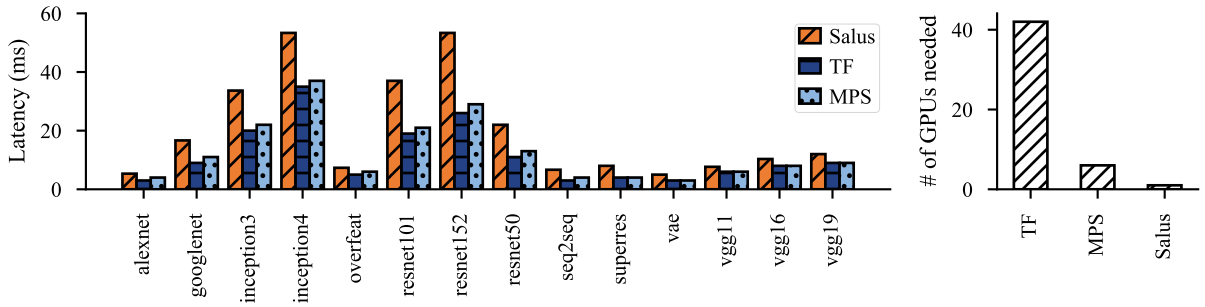


Figure 2.11: *The latencies and number of GPUs needed to host 42 DL models for inference at the same time. 3 instances of each model is created. Each instance has a low request rate.*

focusing on throughout when training, latency of individual inference request becomes a more important requirement when serving DL models [58, 63].

In order to keep responsive to requests, DL models have to be online 24×7 hours. In the traditional setting, each model must reside on a dedicated GPU. However, the traffic of serving requests is not always constant throughout the day, and there are times when the request rate is significantly lower compared to peak. Consolidating DL models into fewer GPUs while remain responsive can save the maintains cost for service providers.

We demonstrate SALUS's ability to reduce the number of GPUs needed while maintaining reasonable response latency in Fig. 2.11. 42 DL inference jobs are selected consisting of 14 different models, 3 instances for each model. Without MPS or SALUS, 42 GPUs are needed to hold these DL models. In contrast, SALUS needs only 1 GPU, achieving $42\times$ improvement, while the average latency overhead is less than 5ms. For comparison, MPS needs 6 GPUs.

A future work is to detect current request rate for inference jobs and automatically scale up or down horizontally. Nevertheless, SALUS provides the essential primitives that makes the implementation possible.

2.5.4 Overhead

SALUS has to be efficient, otherwise the benefits gained from sharing can be easily offset by the overhead. Figure 2.12 shows per iteration training time in SALUS, normalized by per iteration training time in baseline TensorFlow.

For most CNN models, SALUS has minimal overhead – less than 10%, except for a few. The common point of these high-overhead DL models is that they also perform large portion of CPU computation in addition to heavy GPU usage. Since SALUS implements its own execution engine, the CPU computation is also redirected and sent to SALUS for execution, which is not yet heavily optimized.

We finally proceed to compare the performance to run two jobs on a single GPU using existing

Figure 2.12: *Per iteration time per workload in* SALUS*, normalized by that of TensorFlow. Only the largest batch size for each model is reported, as other batch sizes have similar performance.*

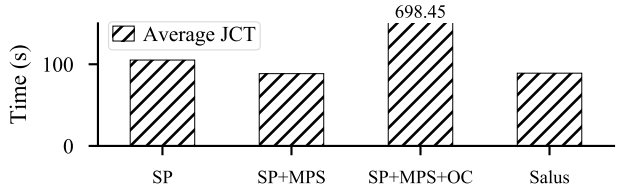


Figure 2.13: *Two concurrent* `alexnet_25` *training jobs for 1 min.*

solutions. Two `alexnet_25` training jobs are started at the same time and each runs for a minute. The jobs share a single GPU using SALUS, *static partitioning* (SP), *static partitioning with MPS* (SP+MPS), and *static partitioning with MPS and memory overcommit* (SP+MPS+OC). We collect and compare the average JCT and report the result in Fig. 2.13.

The result confirms that MPS is indeed better than SP due to the avoidance of GPU context switching. Unfortunately, the SP+MPS+OC solution has significantly bad performance that is beyond useful at the moment. SALUS manages to achieve almost the same performance as MPS while providing much more flexibility in scheduling policy. As shown before, in lightly-loaded inference scenarios, it can significantly outperform MPS in terms of utilization.

2.6 Concluding Remarks

GPUs have emerged as the primary computation devices for DL applications. However, modern GPUs and their runtimes do not allow efficient multiple coexisting processes in a GPU. As a result, unused memory of a DL job remains inaccessible to other jobs, leading to large efficiency, performance loss, and HOL blocking.

SALUS enables fine-grained GPU sharing among complex, unmodified DL jobs by exposing two important primitives: (1) *fast job switching* that can be used to implement time-sharing and preemption; and (2) the *GPU lane* abstraction to enable dynamic memory partitioning, which can

be used for packing multiple jobs on the same GPU. Together, they can be used to implement unforeseen new policies as well.

However, SALUS is only a first attempt, and it opens many interesting research challenges. First, SALUS provides a mechanism but the question of policy – what is the best scheduling algorithm for DL jobs running on a shared GPU? – remains open. Second, while not highlighted in the paper, SALUS can be extended to multiple GPUs or even other accelerators on the same machine. Finally, we plan to extend it to GPUs across multiple machines leveraging RDMA.

CHAPTER 3

FLUID: Resource-Aware Hyperparameter Tuning Engine

3.1 Introduction

The effectiveness of DL models, is highly sensitive to *hyperparameters* [62], which control the model architecture and/or training process, and have to be set before training. Naturally, hyperparameter tuning has taken a central stage in machine learning clusters. Because of the high-dimensionality of the search space, thousands of different hyperparameter settings often have to be evaluated before finding a final set for training in production.

Hyperparameter tuning is an optimization loop in order to find the best set of hyperparameters that are likely to produce the highest validation accuracy. A hyperparameter tuning job contains a large group of training trials, each with its own configuration. It gets feedback from running previous trials before selecting new ones to explore until reaching a target accuracy or stopped by the user. Distributed computation is commonly used to speed up the tuning process [44, 47, 111].

Unfortunately, current hyperparameter tuning solutions cannot efficiently leverage distributed computation. Instead of planning a group of training trials as a whole, each training trial is often independently submitted to a cluster manager, most of the time as a single task running on a single worker GPU [25, 27]. The cluster manager then launches those trials without considering the possibility of intra-worker and inter-worker sharing, failing to make good use of the available resources. For example, there can be more workers than training trials, which can lead to resource underutilization if each training trial only uses one worker. Moreover, a single training trial may not fully occupy one worker; the single-trial-to-single-worker mapping then leaves room to further improve resource utilization when there are more training trials than workers. Even the state-of-the-art cluster managers, which support users to submit a collection of jobs [30], focus on fair sharing of resources between multiple users but leave it up to the users to decide how to execute them.

In an attempt to better utilize cluster resources, some recent works have proposed fully asynchronous execution methods [44, 47] that launch a new trial whenever there is an idle worker in their allocation of resources. However, this execution strategy tightly couples the concurrency of the tuning algorithm itself to the number of available workers. In addition to problems caused

by the single-trial-to-single-worker mapping mentioned above, it fails to concentrate resources on promising hyperparameter configurations. Although all resources are used in this case, many configurations do not necessarily do useful work – i.e., their results are discarded rather than used to guide the generation of the final configuration.

In this paper, we observe that the root cause of the suboptimal use of resources in current hyperparameter tuning solutions is their indifference to how trials are executed. But it is also unrealistic to leave the burden to ML researchers expecting them to manually determine good execution strategies when tuning hyperparameters. We, therefore, take a different approach and propose to decouple the execution strategy from hyperparameter tuning algorithms into a separate execution engine. This has the following advantages: *a*) by separating the concern between tuning algorithms and execution engines, both components can evolve independently; *b*) resource usage can be optimized, which results in faster tuning speed for any tuning algorithms, benefiting a wider range of applications.

However, the *wide variety of algorithms*, the *dynamicity of hyperparameter tuning workloads* and the *differences in training trial profiles* make it non-trivial to design a generalized hyperparameter tuning execution engine that can make efficient use of resources and speed up evaluating a group of hyperparameter configurations.

To this end, we propose FLUID, an algorithm- and resource-aware hyperparameter tuning execution engine that coordinates between the cluster and hyperparameter tuning algorithms. By abstracting hyperparameter tuning job as a sequence of **TrialGroup**s, FLUID provides a generic high-level interface for hyperparameter tuning algorithms to express their execution requests for training trials. FLUID then automatically schedules the trials considering both the current workload and available resources to improve utilization and speed up the tuning process.

FLUID models the problem as a strip packing problem where each rectangle has different shapes and the goal is to minimize the height of the strip. The intuition behind FLUID is to grant more resources to more demanding/promising configurations, such as those with larger training budget, higher resource requirement, or higher priority. By combining techniques like elastic training and GPU multiplexing, FLUID is able to change the trial size, scaling out across many GPUs, or scaling in sharing one GPU with other trials. We also propose heuristics to solve the strip packing problem efficiently and prove that their performance is bounded within $2\times$ of the optimal.

To the best of our knowledge, we make the following contributions:

- FLUID is the first generalized hyperparameter tuning execution engine. It captures the characteristics of a hyperparameter tuning algorithm as a sequence of TrialGroups and models TrialGroup scheduling as a strip packing problem;
- FLUID proposes efficient heuristics with theoretical guarantees to solve the packing problem, and it applies elastic training and GPU multiplexing to enforce the solutions;

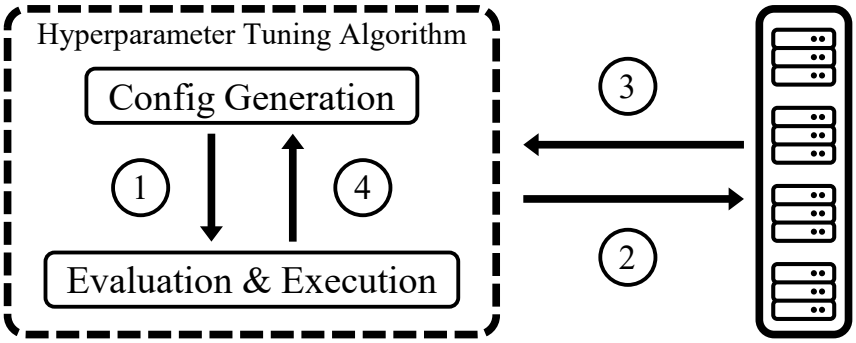


Figure 3.1: *Hyperparameter tuning today: The tuning algorithm which implicitly contains execution logics interacts with the cluster directly.*

- FLUID can boost the performance of various hyperparameter tuning algorithms with higher utilization and shorter end-to-end time. According to our experiment, FLUID can speed up synchronous BOHB by 100%, BOHB and ASHA by 30%, with similar final accuracy.

3.2 Motivation

3.2.1 Background and Related Work

3.2.1.1 Hyperparameter Tuning Algorithms

Figure 3.1 gives an overview of how hyperparameter tuning algorithms work in general. ① The hyperparameter configurations are generated and passed on for evaluation; ② the evaluation logic creates corresponding training trials and submits them directly to the cluster; ③ the training trial finishes execution and the tuning algorithm gets notified; ④ the results are passed back to the generation mechanism for future trial generations.

There are five primary trends in existing hyperparameter tuning algorithms tweaking parts of the aforementioned process to accelerate the evaluation and searching process for a good set of hyperparameter.

1. **Parallel search:** Improved upon grid or random search, Parallel hyperparameter search approaches [103, 122] introduce parallelism to speed up evaluation. However, because finding the best configurations in a large space requires guidance, random-based methods cannot quickly converge to good configurations.
2. **Model-based search:** Model-based hyperparameter search approaches [79, 106, 116] sequentially generate better hyperparameter configurations based on feedback from previous evaluation results. However, such adaptive selecting and evaluating process is inherently sequential and thus not suitable for the large-scale regime.

Table 3.1: *An overview of common hyperparameter tuning algorithms and how they employ the techniques mentioned in Section 3.2.1.1. The last column indicates if they have dedicated execution logic.*

Method	Parallel	Model-based	Early-stopping	Async	Dedicated Execution
Grid/Rand.	✓				
SMBO		✓			
Hyperband	✓		✓		
BOHB	✓	✓	✓	✓	Async
ASHA	✓		✓	✓	Async
PBT	✓	✓			
HyperSched	✓		✓	✓	✓

3. **Early-stopping:** Early-stopping evaluation strategies [84] aim at detecting and stopping poor configurations earlier in order to avoid wasting resources on unpromising configurations. Successive Halving [98], for example, iteratively kills the poor trials and allocates more training time to the top fraction of trials. However, under this iterative process, only a few promising configurations end up being evaluated end-to-end, creating triangular shaped resource usage pattern over time, which degrades the resource efficiency in distributed environments.
4. **Asynchronous search:** Fully asynchronous algorithms [47] always align the number of training trials with the number of workers, which aim at fully utilize all the resources to evaluate configurations. However, these algorithms fail to concentrate resources on promising configurations but spend them on exploring hyperparameter search space.
5. **Hybrid approach:** Hybrid approaches combine the ideas of the four trends aforementioned [19, 44, 47, 60, 78].

3.2.1.2 The Default Execution Logic

As already discussed, current hyperparameter tuning methods focus on speeding up searching mostly in the algorithm, while hardly considering their interactions with the cluster.

In fact, as shown in Table 3.1, only a few algorithms have dedicated execution logic. Most others simply assign one pending job on one idle worker and maintain a FIFO queue. Some newer ones, such as ASHA and BOHB [44, 47], use a simple fully asynchronous strategy to launch a new trial whenever there is an idle worker. Although HyperSched [11] has its own execution logic on the top of ASHA that makes use of distributed training when deadline approaches, this logic is too specific to generalize to other tuning algorithms.

Overall, the execution logic is tightly coupled with the evaluation logic in tuning algorithms,

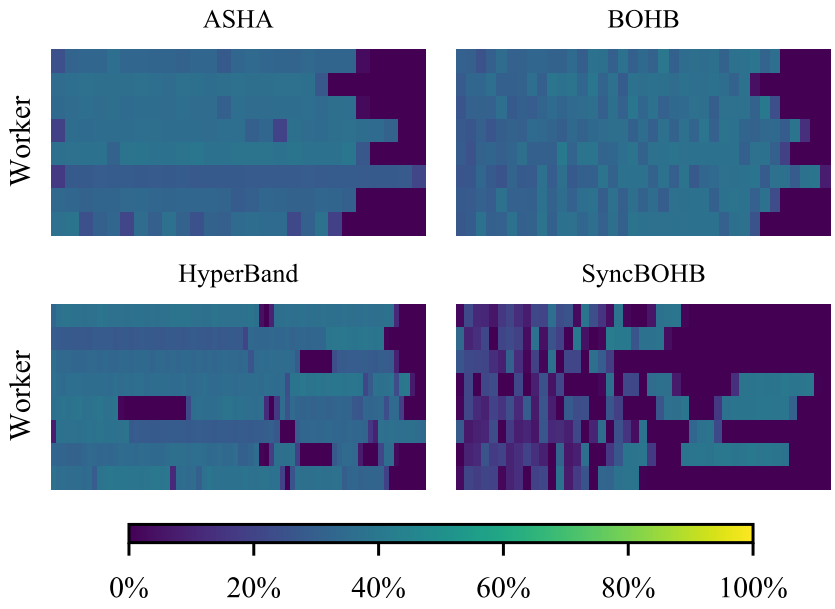


Figure 3.2: *GPU utilization of 4 algorithms on the CIFAR-10 task, running to completion. Each algorithm has 8 workers and each data point is averaged over a 30 seconds window.*

and they lack any generalization to be efficiently applied to every algorithm.

3.2.2 Motivation

Our work is motivated by the low resource utilization in existing hyperparameter tuning algorithms. Here we illustrate that the default execution strategy fails to efficiently use all available resources, while asynchronous execution strategies aim to increase utilization but not all resource usage contribute to selecting the final configuration.

3.2.2.1 Resource (Under-)Utilization

We consider four representative algorithms – ASHA, Hyperband, BOHB, and Synchronous BOHB – to understand their resource usage characteristics.

In Fig. 3.2, we observe ample room for improvement in terms of resource usage. The underutilization has two root causes: *a*) No trial is running during the purple-shaded time slots in a particular worker. Distributed training can be used to balance workloads from other workers onto these free ones; *b*) Even when a GPU is used, the overall utilization is at most 60%, suggesting that a single trial often cannot fully saturate the GPU. Multiple trials can be stacked/packed together to reduce the average trial completion time.

Table 3.2: *Resource utilization and runtime over different number of workers with Successive Halving.*

# Workers	Average Utilization	Runtime
2	81.20%	2356
4	63.00%	1432
8	45.80%	1073
16	47.00%	475
32	25.20%	432

3.2.2.2 Case Study: Lack of Elasticity Reduces Utilization

In the default setting of many early-stopping based algorithms like Successive Halving, the number of available workers is static while the number of trials is diminishing. Training trials are executed on the workers in a FIFO order.

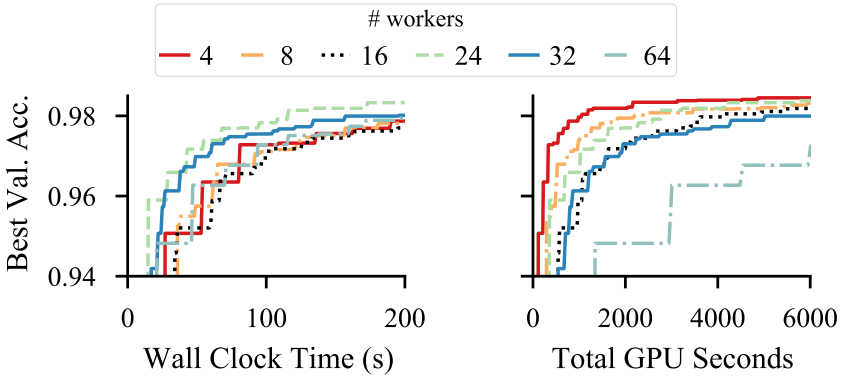
In Table 3.2, we set up a basic Successive Halving based tuning session and measure the average resource utilization as well as tuning speed over varying number of workers. (See Section 3.6 for details about the CIFAR-10 task.) As we can see, when the number of workers increases, the tuning process is indeed faster, but the resource utilization is lower. While more workers can help with the beginning stages of Successive Halving, latter stages only have a smaller number of trials; even though there are idle workers, the algorithm is unable to make use of them. Furthermore, increasing resource allocation further is of no use.

3.2.2.3 Case Study: High Utilization $\neq$ Useful Work

To increase utilization, a common fully asynchronous execution strategy is starting a new trial whenever there is an idle worker. As a result, the tuning algorithm can use all available workers, and its training concurrency is bounded by the number of workers.

Unfortunately, our analysis of ASHA, a popular asynchronous tuning algorithm, shows that not all work is useful work. In this experiment, we measure the best accuracy vs total GPU seconds and wall clock time over different training trial concurrency levels (Fig. 3.3a and Fig. 3.3b). We observe that as we increase the training trial concurrency, the best configuration is not necessarily identified faster; however, more GPU seconds are consumed to reach the same search target.

This is because the hyperparameter tuning is an exploration and exploitation process. Even though more workers are kept busy working on something fresh (exploration), not necessarily all the work done contribute to the final configuration's generation (exploitation), which makes it fail to concentrate resources on promising configurations. To this end, instead of blindly improve the resource utilization, hyperparameter tuning should spend its resource where it counts the most. An



(a) *Wall clock time* **(b)** *GPU Seconds*

Figure 3.3: *ASHA best validation accuracy over wall clock time and total GPU seconds (averaged across 3 ASHA jobs respectively).*

ideal evaluation and execution strategy should achieve the target accuracy with shorter wall clock time and smaller total GPU seconds.

3.3 FLUID

In this section, we start with the problem statement and review the challenges FLUID must solve to become a generalized execution engine for hyperparameter tuning. Then we introduce the high-level interface used by FLUID and how it interacts with other components in a hyperparameter tuning job. Finally, we illustrate the intra- and inter-GPU sharing considerations in FLUID.

3.3.1 Problem Statement

Given a hyperparameter tuning job and available resources, the objective of FLUID is to carefully allocate resources to each training trial in the job such that resources are efficiently used and the makespan is minimized. In addition, FLUID should generalize to most hyperparameter tuning algorithms and react to cluster resource changes.

FLUID must address the following challenges.

1. **The wide variety of tuning algorithms** There are many strategies for hyperparameter tuning, each of which generates configurations and evaluates their performance in different ways (Section 3.2.1.1). It is challenging to design a general execution engine that can take algorithm-aware scheduling decisions and minimize the makespan of a collection of training trials (Section 3.3.2).
2. **Highly dynamic training workloads and resources** Training trials generated by hyperparameter tuning dynamically change over time due to the use of early-stopping strategy (Section 3.2.1.1). Cluster resource allocation can also be dynamic for fairness and efficiency reasons.

Hence, it is challenging to capture the dynamic resource usage to reallocate correspondingly.

3. **Heterogeneity in training trial profiles** Differences in hyperparameters across training trials may cause different resource demands and be given different amounts of training budgets. Hyperparameter configurations may react differently for different resource allocations too. This challenges the execution engine to treat different trial profiles accurately (Section 3.3.4 and Section 3.4.2.2).

3.3.2 The TrialGroup Abstraction

To generalize different hyperparameter tuning algorithms, we introduce an abstraction called **TrialGroup** between the tuning algorithms and FLUID. Algorithms can use this simple-yet-rich interface to express their training requests, and FLUID works with this consistent model to schedule executions.

A TrialGroup is a group of training trials with a training budget associated to each trial in the group. At any time, trials may be removed from the group due to completion or requested termination from the algorithm. The optimization goal for such a TrialGroup is to minimize its makespan, such that all the results are available as early as possible. TrialGroup is the basic unit of scheduling in FLUID.

Although the definition of TrialGroup is simple, we find it quite expressive for modeling hyperparameter tuning algorithms' training trial execution requirements. In the most simple case – grid/random search, where all training trials are created at the same time and have a fixed amount of budget, all trials fit nicely in one TrialGroup. For more involved algorithms discussed in Section 3.2.1.1, where new iterations of the algorithm may be added based on previous feedbacks, all trials from a single iteration forms a TrialGroup. This essentially creates a sequence of TrialGroups, each has its own makespan minimized individually.

3.3.3 FLUID Overview

Figure 3.4 illustrates FLUID's position in the stack. FLUID coordinates between the cluster and tuning algorithms, decoupling the execution logic from any single tuning algorithm. The tuning algorithm (2a) submits TrialGroup to FLUID over time. During the tuning process, intermediate results are (3b) reported back so that new trials may be created or existing ones removed.

The action of FLUID will only be triggered when new TrialGroups are added or some resources are freed. Based on the (3a) real-time resource usage reported by the cluster, FLUID uses `StaticFluid` (Section 3.4.2) to schedule any new TrialGroups onto idle resources. It then reactively waits for more events to handle. When some resources are freed up and there are not pending jobs in the queue towards the end of the TrialGroup, FLUID adapts `DynamicFluid` (Section 3.4.3)

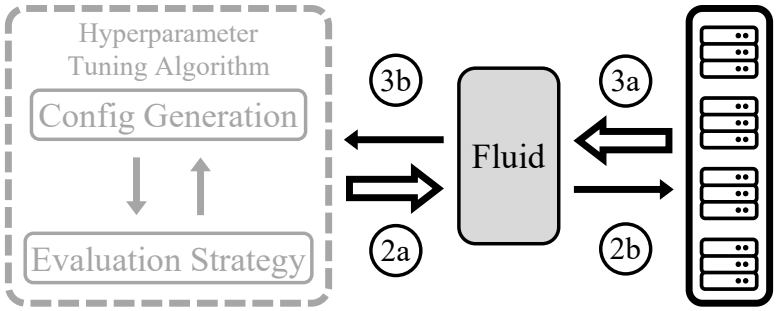


Figure 3.4: *With* FLUID*, the tuning algorithm (2a) submits* ***TrialGroup*** *according to its evaluation strategy and (3b) gets feedbacks back anytime they are available.* FLUID *itself manages (2b) the job execution and handles (3a) resource changing events.*

to reallocate resources with the concern of any overhead.

3.3.4 Parallelism in Multiple Granularities

Minimizing the TrialGroup makespan ultimately translates to making better use of the underlying hardware resources in parallel. However, as shown in Section 3.2.2, relying on creating massive amount of trials as the single source of parallelism is neither enough to saturate individual workers nor applicable to many model-based tuning algorithms.

In FLUID, in addition to the number of trials, parallelism is sourced from within the training trial by using existing techniques like MPS [140] (or Multi-Instance GPU (MIG) [142] more recently) and automatic distributed training. NVIDIA MPS allows multiple processes to run on a single GPU at the same time with different CUDA streams, providing *intra-GPU* parallelism. Distributed training is an established technique to use multiple parallel workers to reduce the training trial's job completion time, providing *inter-GPU* parallelism [104]. In addition, resource elasticity [13] realizes resource reallocation on the fly providing more flexibility in scheduling.

FLUID uses both *inter-GPU* and *intra-GPU* parallelism to perform resource allocation in a water-filling scheme. However, distributed training has communication overhead, while GPU sharing inevitably creates interference; both degrade performance. Therefore, we incorporate dynamic overhead measurement into FLUID's design (Section 3.4.2.2) and assess the *marginal benefit* before using both techniques to increase parallelism in intra- and inter-GPU granularities.

3.4 FLUID Algorithm

The scheduling problem of a hyperparameter tuning job, a sequence of TrialGroups, can be broken down into solving several independent TrialGroup scheduling problems. In this section, we begin with formulating this single TrialGroup scheduling as a strip packing problem (Section 3.4.1).

Since the hyperparameter tuning can be simplified as two actions (Section 3.3.3), trial arrival and departure, we then propose two heuristics to make full use of resources respectively under two conditions: *a*) new TrialGroup is launched to be scheduled; *b*) resource is freed to be allocated.

We first introduce `StaticFluid` and how it uses GPU sharing to schedule incoming TrialGroup with the concern of sharing overheads (Section 3.4.2). We provide theoretical analysis for our `StaticFluid` in the appendix. Then, we introduce `DynamicFluid` which uses resource elasticity to reallocate freed resources while being robust to overheads (Section 3.4.3).

3.4.1 Problem Definition

Let the TrialGroup scheduling be represented as a strip packing problem $I = \{A, M\}$. Each rectangle a_i in $A = \{a_1, \ldots, a_k\}$ with width and height corresponds to a trial with allocated resources and remaining runtime. It is worth noting that in our problem setting, *each rectangle's width w_i determines its height $h_{i,w}$*. $h_{i,w}$ thus implies the relationship between different resource allocation and its corresponding runtime for this trial. Strips in $M = \{m_1, \ldots, m_n\}$ with identical width 1 and infinite height represents n available identical resources for current hyperparameter tuning jobs.

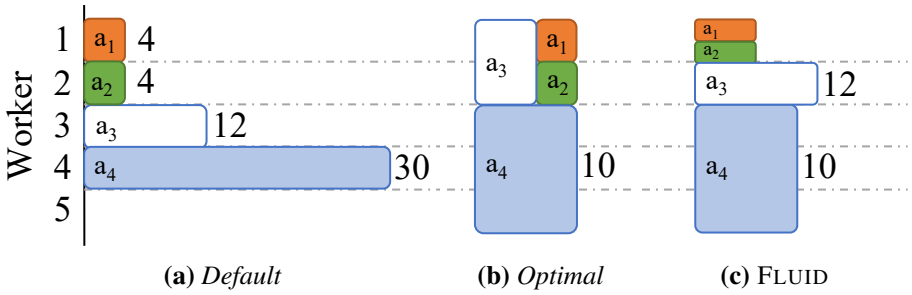
FLUID utilizes both intra- and inter-GPU sharing to achieve higher utilization of GPU resources and minimize the TrialGroup makespan. Hence, a training trial can be assigned to a real number amount of resources, where the fractional part of the resources represents a worker that will be shared with other trials, and an overall > 1 resources means the trial must be placed across workers using distributed training. The goal is to find a non-overlapping orthogonal packing of these k rectangles (taking into account different size options) into n strips such that the maximum height of strips is minimized.

3.4.2 StaticFluid

3.4.2.1 Algorithms

Since minimizing the height of strip packing is NP-hard [126], FLUID proposes an efficient heuristic `StaticFluid` to find an approximate solution. At a high level, FLUID resource allocation is performed using a water-filling scheme to balance the relationship between workloads and resources by "evenly" allocating resources to current evaluation trials in order to minimize the TrialGroup makespan.

Consider a grid search example with 5 GPUs and 4 trials $\{a_i\}_{i=1}^{i=4}$ arrived at the beginning with training budget 4s, 4s, 12s and 30s respectively. As shown in Fig. 3.5a, the default FIFO scheduler treats each training trial independently without considering the different impact on the TrialGroup makespan; thus, it performs poorly because of stragglers and resource underutilization.



(a) *Default* **(b)** *Optimal* **(c)** FLUID

Figure 3.5: *Toy example: default, optimal and* FLUID *for 4 training trials scheduled on 5 workers.*

Algorithm 3.1: STATICFLUID

Input: TrialGroup A, Idle Resources M'

Sort a_i by $h_{i,1}$ in non-increasing order

foreach $a_i \in A$ **do**

  $w_i \leftarrow \min(\max(\lfloor \frac{h_{i,1}}{\sum_j h_{j,1}} n \rfloor, \frac{1}{c}), d)$

  Allocate a_i with w_i resources

Intuitively, a good schedule would prioritize resources to long training trials to mitigate the straggler and minimize the makespan. As shown in Fig. 3.5c, FLUID distributes the longest training trial a_4 onto three workers and packs shorter training trials a_1 and a_2 together on one worker. In this way, straggler is mitigated and resources are fully utilized. FLUID's static heuristic comes close to the optimal schedule for this example (Fig. 3.5b).

As shown in Algorithm 3.1, FLUID allocates resources w_i based on the ratio of each trial's runtime $h_{i,1}$ to the sum of runtime $\sum h_{j,1}$ in the TrialGroup. FLUID then schedules the trials in non-increasing order of resources onto the idle worker set. To avoid performance degradation from GPU sharing, we limit the maximum intra-GPU sharing to c and maximum inter-GPU training to d, which we discuss in more details in Section 3.4.2.2.

Formally, we give Theorem 1 without concerning any overhead, whose proof is included in Appendix B.1.3.

Theorem 1. *In the ideal situation, FluidStatic is a 2-approximation algorithm.*

3.4.2.2 Accounting for Overheads

FLUID adapts MPS and distributed training to realize intra-GPU sharing and inter-GPU training. These inevitably causes interference and overheads that hurt training performance. We account for such possibilities by tracking marginal benefits.

Intra-GPU Overhead Under the assumption that trials of the same hyperparameter tuning job are similar to each other, FLUID regards similar trends for diminishing marginal benefit for packing one more trial from the same TrialGroup. FLUID determines the optimal number of concurrent trials c on a GPU by finding out the inflection point where marginal benefit degrades below a threshold O_{th}.

We define the marginal benefit O^p for packing the p^{th} trial on a worker as

$$O^p = 1 - \frac{p-1}{p}\frac{\text{avg}(T^p)}{\text{avg}(T^{p-1})} \quad (p > 1) \tag{3.1}$$

where $T^p = \{t_i\}_{i=1}^{i=p}$ is the set of training time per iteration for p training trials. The marginal benefit of reduced average trial completion time is usually diminishing with the increasing packing overhead. FLUID ensures the packing benefit over packing overhead by limiting the number of concurrent training trials under the optimal number c where $c = \arg\max_p O^p > O_{th}$.

Inter-GPU Overhead The speed-up brought by distributed training is not linear because of communication overheads. Such communication overheads are often determined by the size of model parameters and the number of workers [70].

FLUID determines the maximum degree of parallelism d for an evaluated configuration using the Paleo framework [67] to estimate the cost of training neural networks in parallel. It ensures the scaling benefit over scaling overhead by limiting the number of distributed worker to less than d.

Problem Setup To consider these realistic factors for our strip packing problem, we define the relationship between runtime $h_{i,w}$ and resources w for trial a_i as

$$h_{i,w} = \begin{cases} h_{i,1}\alpha_i^{\frac{1}{w}-1} & w \in (0,1) \\ \frac{h_{i,1}}{w}\beta_i^{w-1} & w \in [1,d] \end{cases} \tag{3.2}$$

where $h_{i,1}$ is the trial runtime on one worker. $\alpha_i \in \left[1, \frac{c}{c-1}\right)$ is a measure of the packing overhead for trial a_i. $\beta_i \in \left[1, 1+\frac{1}{d}\right)$ is a measure of the scaling overhead for trial a_i. This definition of $h_{i,w}$ ensures the following result.

Theorem 2. *In the real situation,*

$$\text{StaticFluid}(I) < \max(2\,\text{OPT}(I) + \max(h_1)\alpha^{\frac{1}{\alpha-1}}, 2\,\text{OPT}(I)\beta^{\frac{1}{\beta-1}-1}) \tag{3.3}$$

We prove that `StaticFluid` has theoretical guarantee even when practical resource sharing overheads is considered (Appendix B.1.3).

Algorithm 3.2: DynamicFluid

Input: TrialGroup A, Total Res. M

Sort a_i by $h_{i,1}$ in non-increasing order

foreach $a_i \in A$ **do**

  $w_i' \leftarrow \min(\max(\lfloor \frac{h_{i,1}}{\sum_j h_{j,1}} n \rfloor, \frac{1}{c}), d)$

  if $w_i' > w_i$ *and* $h_{i,w_i'} + \epsilon < h_{i,w_i}$ **then**

    ▷ `Scale up`

    Update a_i with w_i resources

  else if $w_i' < w_i$ *and* $w_i'(h_{i,w_i'} + \epsilon) < w_i h_{i,w_i}$ **then**

    ▷ `Scale down`

    Update a_i with w_i resources

3.4.3 DynamicFluid

In addition to scheduling incoming TrialGroup, FLUID also need to handle the dynamic changing resource usage caused by job departure and cluster resource changes.

As shown in Fig. 3.5c, there is still a resource gap before the TrialGroup completes, which leaves space to further improve the utilization and minimize makespan. Therefore, we extend the heuristic `StaticFluid` to `DynamicFluid` that reallocates the incoming idle resources on the fly with the help of resource elasticity.

Ideally, `DynamicFluid` updates the resource allocation plan w' using resource elasticity with the same formula in Algorithm 3.1 when new resources are freed. However, using resource elasticity to adjust parallelism will inevitably incur overhead. Hence, FLUID considers scaling overhead ϵ to avoid performance degradation and frequent parallelism adjustment. As shown in Algorithm 3.2, FLUID updates resources $w_i \leftarrow w_i'$ for trial a_i at the end of current iteration only when it does not lead to performance degradation.

In addition to resource usage change caused by job departure, some cluster schedulers [30] may dynamically change resource allocation for fairness or efficiency. FLUID can handle such change by triggering `DynamicFluid` when resources are updated.

3.5 FLUID Implementation

We implemented FLUID as an executor for Ray Tune [48]. In addition to the implementing our execution algorithm, tuning algorithms in Tune were adapted to make use of the TrialGroup interface, which translates to one extra function call per algorithm class when applicable, to signify the creation of new TrialGroups.

In order to adjust training trials' number of worker at runtime with lower overhead, we also

Table 3.3: *List of workloads for* FLUID.

Task	Base Model	# of Arch. Params.	# of Training Param.	Target
CIFAR-10	AlexNet	3	4	Acc. $\geq 90\%$
WLM	RNN	4	6	PPL ≤ 140
DCGAN	CNN	0	2	Inception ≥ 5.2

implemented training elasticity technique similar to the one proposed by recent work [13].

One important input of the FLUID algorithm is the packing overhead measurement $\alpha_i \in \left[1, \frac{c}{c-1}\right)$ and distributed overhead measurement $\beta_i \in \left[1, 1 + \frac{1}{d}\right)$. These numbers depend on many factors of the training process and are affected by the hardware in use. It is an active field of research to predict them given a particular configuration. In FLUID, instead of predicting, we use a trial-and-error approach by measuring these numbers on real hardware. Leveraging the iterative nature of deep learning training process, only a few iterations is enough to have reasonable measurements. FLUID thus uses a small fraction of resources to profile current trials and reuse the result throughout the whole tuning session.

3.6 Evaluation

We evaluate FLUID with a range of hyperparameter optimization algorithms, including Grid/Random Search, PBT, Successive Halving, Hyperband, BOHB and ASHA [44, 47, 60, 78, 98, 103, 122]. Our evaluation shows the following key highlights:

- FLUID can speed up diverse hyperparameter tuning workloads around 10%–70%, while improve cluster efficiency up to 10%–100%.
- FLUID can improve the trade-off between resource efficiency and hyperparameter searching speed, and optimize both simultaneously.
- FLUID's benefits are robust under different kinds of environment setup and training workloads.

3.6.1 Experiment Setup

Testbed We built our testbed on Chameleon Cloud [9]. Each node has an Intel Xeon Gold 6126 CPU with NVIDIA Quadro RTX 6000 GPU. The interconnection is 10 G Ethernet.

Workloads We create our set of workloads (Table 3.3) using different deep learning tasks including computer vision, natural language processing and adversarial learning. The CIFAR-10 task includes the tuning of a variation of AlexNet on an image classification task maximizing accuracy; The

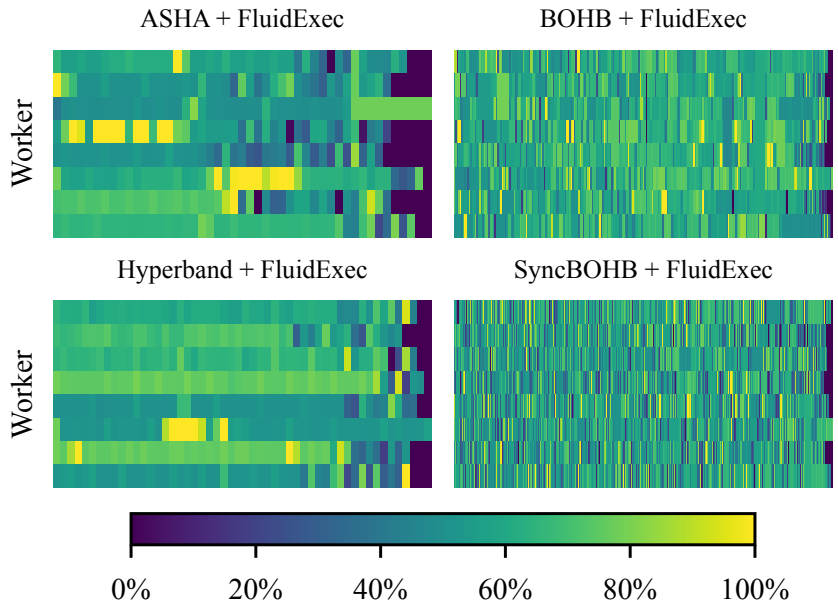


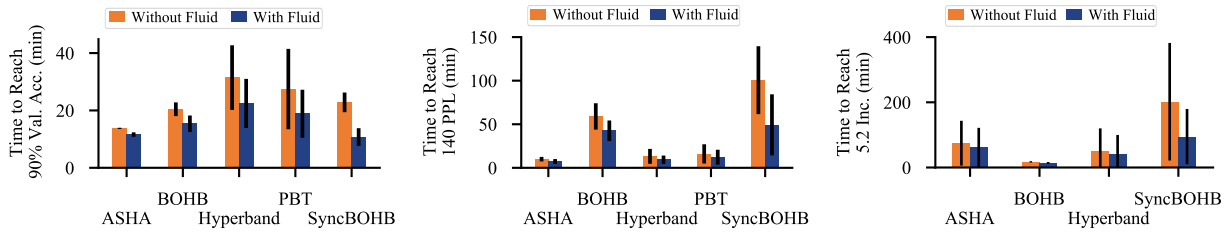
Figure 3.6: *GPU utilization of 4 algorithms on the CIFAR-10 task using* FLUID*, running to completion. Each algorithm has 8 workers and each data point is averaged over a 30 seconds window.*

WLM task tunes the training of a multi-layer Recurrent Neural Network (RNN) on the word-level language modeling task minimizing perplexity (PPL); The DCGAN task tunes two multiple-layer CNN network, creating a generative model on the MNIST dataset. The tuning target is maximizing the inception score.

In all tasks, the tuned hyperparameters include training parameters like learning rate, batch size, etc., as well as architectural parameters like number of CNN layers, size of layers, type of models, etc.

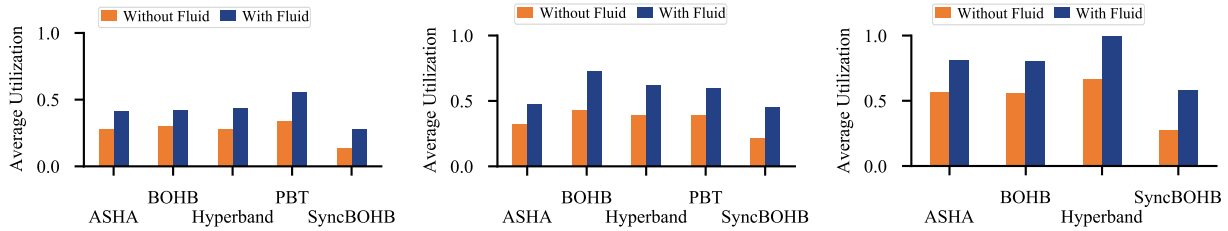
Tuning algorithms We compare the performance of 5 hyperparameter tuning algorithms with and without FLUID: PBT, Hyperband, ASHA, synchronous BOHB (SyncBOHB) and BOHB. These algorithms include both synchronous stage-based strategies and asynchronous strategies, covering most of the situation that may appear during hyperparameter tuning process, including stopping, promotion and so on.

Metrics The improvement in hyperparameter tuning job makespan is our key metric. The makespan is defined as the end-to-end time needed for a given problem to reach a certain metric target. For example, the time needed to reach 90% accuracy for the CIFAR-10 task and the time needed to reach 140 ppl for language models. We also measure resource utilization improvement to indicate FLUID's effort on improving resource usage.



(a) *CIFAR-10* **(b)** *WLM* **(c)** *DCGAN*

Figure 3.7: *Time to reach target. Data averaged over 5 runs. Error bar represents standard deviation. PBT on DCGAN did not finish in reasonable time and thus excluded from the report.*



(a) *CIFAR-10* **(b)** *WLM* **(c)** *DCGAN*

Figure 3.8: *Average utilization.*

3.6.2 Macrobenchmarks

We first report the performance improvement of FLUID over 5 tuning algorithms on all three tasks in Fig. 3.7. The average GPU utilization of those runs are reported in Fig. 3.8. In addition, we report the utilization heatmap similarly as in Section 3.2.2 on the CIFAR-10 task in Fig. 3.6.

The benefit of FLUID varies over algorithms. We see a pattern that synchronous stage-based strategies (SyncBOHB, Hyperband) can gain more benefit 30%–100% on job makespan and resource usage with the help of FLUID due to underutilization of resources, while asynchronous strategies (ASHA, BOHB), in spite of their original high resource utilization, see 10%–30% improvements. PBT always has a constant number of trials running, so the benefit of FLUID is limited. But FLUID can still help PBT to scale out to more concurrent trials, which improves the overall time to reach target.

FLUID's benefit comes from the following: *a*) Resource under-utilization due to mismatch between training trials and available resources over time. *b*) Stragglers due to algorithm's synchronous nature. *c*) Insufficient concurrency when the number of running trials is tied to the number of workers.

The rest of this section gives detailed insights into the improvement of synchronous stage-based tuning algorithms and fully asynchronous algorithms. We then move on to microbenchmarks in which we break down the improvement of individual techniques.

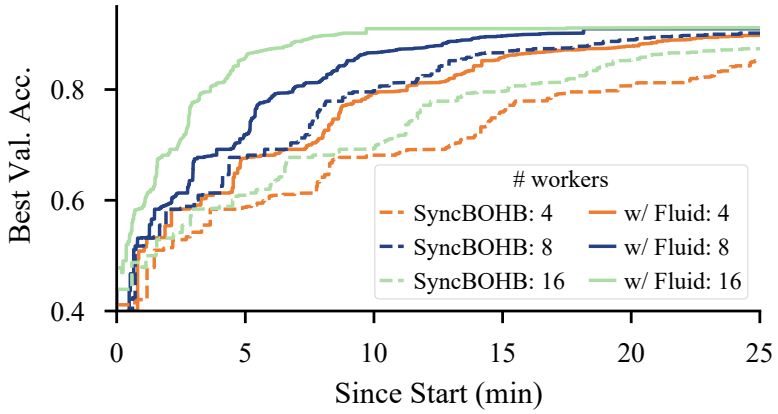


Figure 3.9: *Validation accuracy over time for SyncBOHB w/ and w/o* FLUID *on the CIFAR-10 task. Data averaged over 5 runs.*

3.6.2.1 Benefit on Synchronous Stage-based Strategy

In Fig. 3.9, we measure the best accuracy over time for synchronous stage-based strategy BOHB using a different number of workers. Our results show that the tuning speed up is not in proportion to the increase of available workers. However, with the help of FLUID, we are able to achieve better scalability. The experiment was done on the CIFAR-10 task using SyncBOHB on 4, 8, 16 GPUs respectively.

3.6.2.2 Benefit on Fully Asynchronous Strategy

In Fig. 3.10, we measure the max accuracy over time for fully asynchronous strategy ASHA using 4, 8, 16 GPU workers respectively. Our results show that the performance of fully asynchronous strategy largely depends on the number of workers. And in this particular case, 8 workers works the best. With the help of FLUID, we can easily set the concurrency for ASHA regardless of the number of physical workers, because FLUID is able to adjust the number of concurrent running trials. ASHA can therefore achieve the optimal balance. As a result, with FLUID, we are able to achieve similar performance as 8 GPUs using only 4 or better performance with more GPUs.

3.6.3 Microbenchmarks

3.6.3.1 Benefit of Intra-GPU Sharing

In Fig. 3.11a, we compared the performance of hyperparameter optimization algorithms with and without MPS packing. We show the speed-up of FLUID with MPS as the only enabled mechanism, compared to the original algorithm. For the sake of discussion, we choose Grid Search in this experiment to avoid influences from other factors.

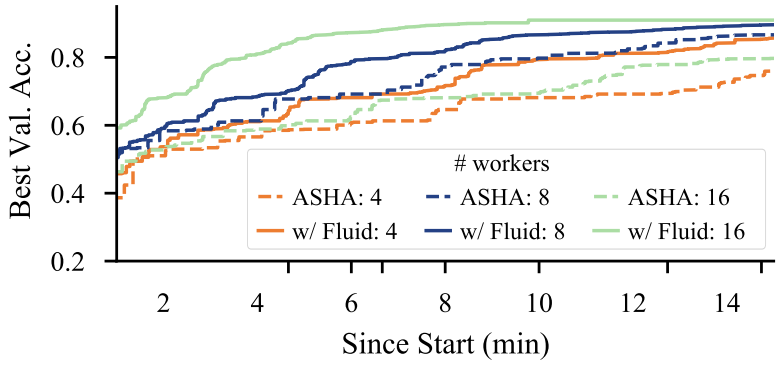
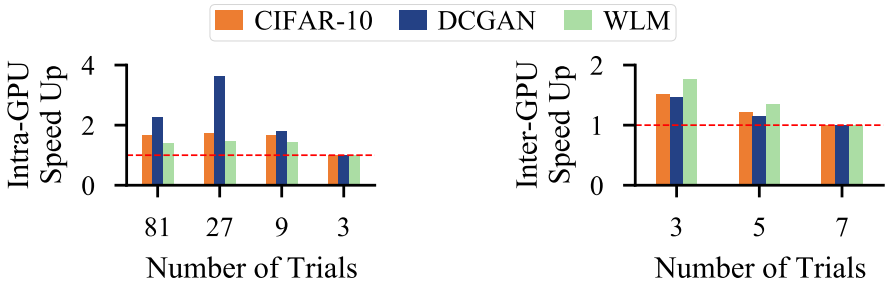


Figure 3.10: *Validation accuracy over time for ASHA w/ and w/o* FLUID *on the CIFAR-10 task. Data averaged over 5 runs.* FLUID *is set to have 8 concurrent trials regardless of # workers.*



(a) *Intra-GPU only* **(b)** *Inter-GPU only*

Figure 3.11: *Speed-up break down of intra- and inter-GPU training.*

Our results show that MPS Packing provides a significant performance increase especially on tuning relatively large TrialGroup with small model size. We experiment with Grid Search on three training workloads from small to large, and for each training workloads, we evaluate 9, 27, 81 configurations on 8 GPUs.

3.6.3.2 Benefit of Inter-GPU Training

Similarly, we enable only inter-GPU mechanism in FLUID, and compare the performance gain using the Grid Search. With 8 GPU workers, we limit the trial number to smaller than that to explicitly trigger the inter-GPU distributed training. Our results show that inter-GPU distributed training provides a significant performance increase especially on tuning relatively small TrialGroup with large model size. The results are shown in Fig. 3.11b. FLUID effectively utilizes the idle distributed resources and achieve tuning speed up especially when the gap between number of trials and number of workers is large.

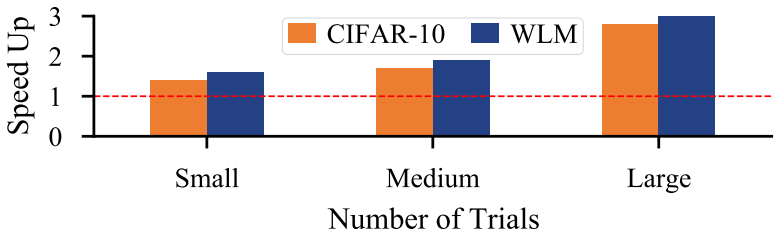


Figure 3.12: *Relative speed-up given trial runtime variance.*

3.6.4 Sensitivity Analysis

3.6.4.1 Effect of Runtime Variance

In Fig. 3.12, we show how FLUID performs with different trial runtime variance. Our results show that FLUID can achieve better speed up or resource utilization improvement especially when trials' runtime variance is large, which can be attributed to `DynamicFluid` which adjusts resources at runtime. We experiment with Grid Search on tuning different training workloads, and for each training workloads, we evaluate 9 configurations with different degree of job runtime variance on 8 GPUs.

3.6.4.2 Effect of Packing Overhead

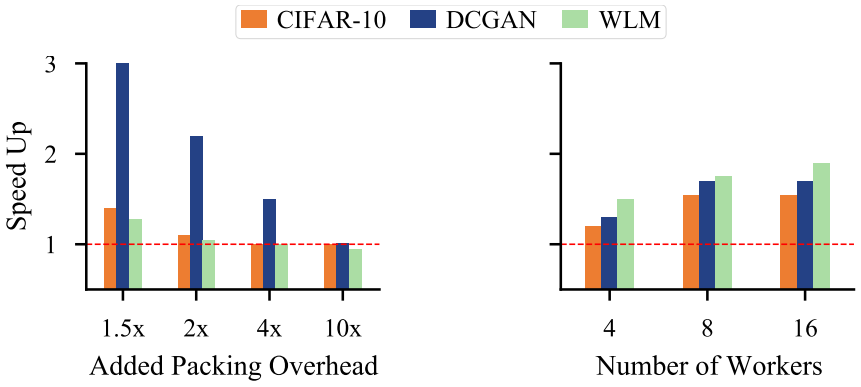
By manually modify tasks to include controllable artificial packing overhead, we are able to assess FLUID's reaction under different packing conditions.

The results reported in Fig. 3.13a shows the speed-up ratio of completion time of Grid Search with FLUID, with $1.5, 2, 10$ times packing overhead, relative to those without using FLUID. The experiment is given 27 hyperparameter configurations and has 8 workers in total.

When the overhead is relatively small, FLUID still sees positive marginal benefit to packing. With $10x$ overhead, the intra-GPU packing is effectively disabled and FLUID's performance becomes the same as the original algorithm.

3.6.4.3 Effect of Scaling Overhead

In Fig. 3.13b, we show how FLUID reacts across various scaling overhead. Our results show that FLUID can achieve different degrees of speed-up under different scaling overhead by detecting model's scalability. We experiment with Grid Search on scaling different training workloads, and for each training workloads, we evaluate 3 configurations on 4, 8, 16 GPUs. In this experiment we disable the intra-GPU packing mechanism. Going from 4 GPUs to 8 gains sizable performance benefit across all 3 tasks. But CIFAR-10 and DCGAN does not benefit from adding more GPUs. In fact, the added GPUs are not used at all, because FLUID detects there will be high scaling overhead



(a) *27 trials Grid Search on 8 workers* **(b)** *3 trials Grid Search*

Figure 3.13: *The speed-up of* FLUID *with Grid Search with varying packing and scaling overhead.*

associated if these workloads scaling beyond enough. In real settings, those idle GPUs will be used by other trials.

3.7 Conclusion

FLUID is a generic hyperparameter tuning execution engine that decouples execution logic from tuning algorithms, with the high-level TrialGroup interface for tuning algorithms to express their execution needs. FLUID can boost the performance of diverse hyperparameter tuning solutions.

CHAPTER 4

ORLOJ: Predictably Serving Unpredictable DNNs

4.1 Introduction

The underlying serving systems [7, 15, 38, 58, 66] that handle DL inference requests aim to maximize throughput while reducing SLO misses. Due to the high throughput and low latency requirements of DNN-dependent applications and the large computation needs of DNN inference requests, modern serving systems often rely on expensive GPUs to serve many requests in parallel by batching them together [7]. All the requests in the same batch experience the same request execution time. This works well because the state-of-the-art DNN serving systems [7, 38, 58] have so far assumed *data-independence* of incoming requests; i.e., the amount of computation required for each request is the same regardless of the input data. For example, for an image classification model, whether the input image contains a dog or a cat, the model performs exactly the same computation to derive the answer. Consequently, the request execution time of a model can be accurately profiled and used to precisely schedule inference requests [7].

We observe that the recent rise of a new class of *dynamic* DNNs [2, 72] challenges this assumption (Section 4.2.2). Unlike static DNNs, dynamic DNNs can adapt their structures or parameters to the input during inference and are thus inherently *data-dependent*. For example, SkipNet [54] dynamically skips layers depending on the input sample; RDI-Nets [8] allows for each input to adaptively choose one of the multiple output layers to output its prediction; and various NLP models exhibit recurrent structures or loops [14, 17, 29, 32, 35, 42]. The result is unpredictable execution times for individual requests. Because request execution times come from a distribution instead of being a single constant, existing systems that use a single mean or tail execution time from historical data to *plan ahead* perform poorly [7, 38]. They fail to capture the high variance in incoming requests' execution times, and when they batch multiple requests together, one long request in the batch slows down many short ones, leading to large number of SLO misses (Section 4.2.3). Serving systems that do not assume data independence [5, 15, 58] and instead perform *reactive* adjustment to dynamically provision workers at runtime perform even worse, because they treat SLOs as long-term reactive targets and cannot effectively curtail tail

latency, especially under stringent SLO constraints [7].

In this paper, we present ORLOJ[1], a distribution-aware dynamic DNN serving system, to provide high throughput and low SLO misses. ORLOJ also takes a plan-ahead approach, but unlike recent solutions, it uses a random variable to capture the variance in request execution times of dynamic DNNs, rather than assuming a constant mean or tail latency for all requests. This gives ORLOJ more flexibility to account for uncertainty in execution times when batching them together to achieve high throughput.

Going beyond prior distribution-based solutions for cluster scheduling and query processing [25, 50, 97], ORLOJ addresses two challenges unique to model serving. First, batching is vital to achieve high throughput, but it also affects execution times, as all requests in the same batch start and finish at the same time, regardless of their individual execution times. ORLOJ proposes a batch-aware priority score that derives batch execution time distributions given those of individual requests, and uses the score to guide its batching decisions. Second, because a model can receive requests from different applications, the joint distribution can be multimodal with even higher variance. To this end, ORLOJ tags each request with its originating application and relies on probability theory to accurately estimate batch execution times even when execution times of requests in the same batch follow different distributions.

We have implemented and evaluated ORLOJ using production traces and a large range of possible input execution time distributions (Section 4.5). In comparison to Clockwork [7], Nexus [38], and Clipper [58], ORLOJ can improve the finish rate when serving dynamic DNNs by 51–80% under tight SLO constraints, and over 100% under more relaxed SLO settings. For well-studied static DNN workloads, ORLOJ keeps comparable performance with the state-of-the-art.

Overall, we make the following contributions in this paper:

- To the best of our knowledge, ORLOJ is the first system to systematically analyze the inference performance of dynamic DNNs and associated challenges.
- We present a batch-aware distribution-based scheduling algorithm to handle batching and multimodal distributions in dynamic DNN serving systems.
- ORLOJ can handle both static and dynamic workloads with high throughput and tight SLO guarantees.

4.2 Background and Motivation

In this section, we overview model serving systems and the recent rise of dynamic DNNs, and then move on to the limitations of existing state-of-the-art solutions when serving such dynamic networks.

[1] Orloj (pronounced /ɔrlɔj/) means astronomical clock.

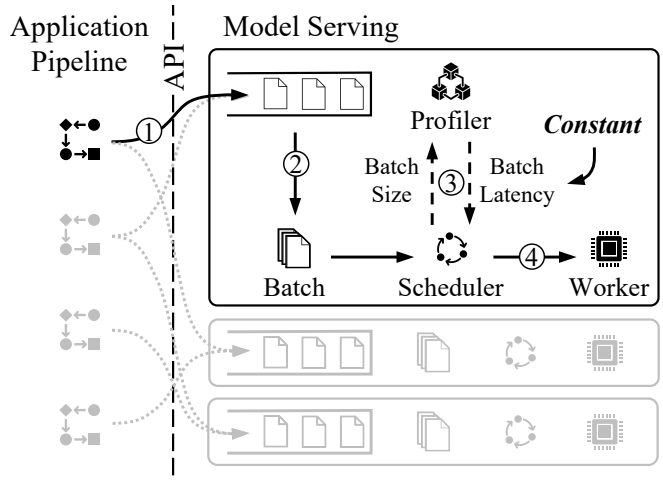
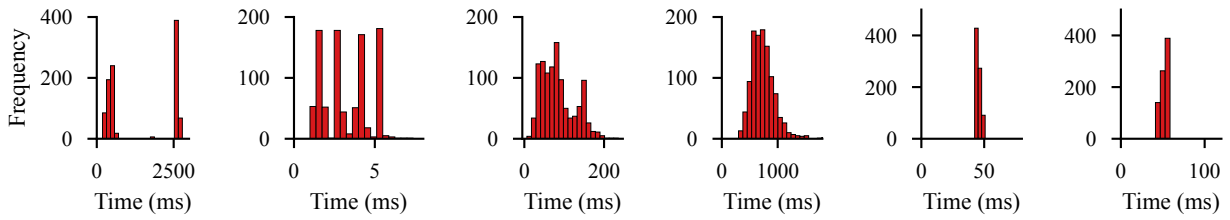


Figure 4.1: *A model serving service multiplexes requests from multiple applications and schedules batches of requests on workers.*

4.2.1 Model Serving

Increasingly more models are deployed on the critical paths of online interactive services [39]. They may even comprise dependent computations across multiple models, forming pipelines [5]. For example, video analytics pipelines first detect certain objects from each image and then recognize individual objects [38, 74]. A serving system in the backend handles inference requests for individual models by running a model replica on the input, usually providing APIs for specific tasks such as detection, translation, or prediction [133, 134]. Similar to other datacenter services [77], it multiplexes workloads of different applications and load balances requests across multiple workers [15, 58].

Lifecycle of Serving Inference Requests Figure 4.1 illustrates the lifecycle of (a batch of) inference requests in a worker of such serving systems. ① Incoming requests first go through a priority queue, usually ordered by deadline, but it may vary depending on a system's optimization goal. ② The dynamic batcher will extract requests from the queue to create batches, while respecting the deadline requirement of the requests at the top of the queue. ③ The size of the formed batch will be queried against historical profiling data to decide an estimated batch latency. ④ This single latency value will be used in the scheduler to derive an execution plan on the worker. Of course, there is not exactly one way to divide work between the stages shown here, and there may be additional interactions between various components. For example, Nexus [38] uses a pre-computed plan ahead of time, while Clockwork [7] employs multiple "Batch Queues" to select the best batch size at runtime. Nevertheless, the active state in these systems depends on a *single, point estimation of the batch latency*, oblivious to individual request-specific data at runtime.



(a) *RDI-Nets* **(b)** *SkipNet* **(c)** *GPT* **(d)** *BART* **(e)** *Inception V3* **(f)** *ResNet*

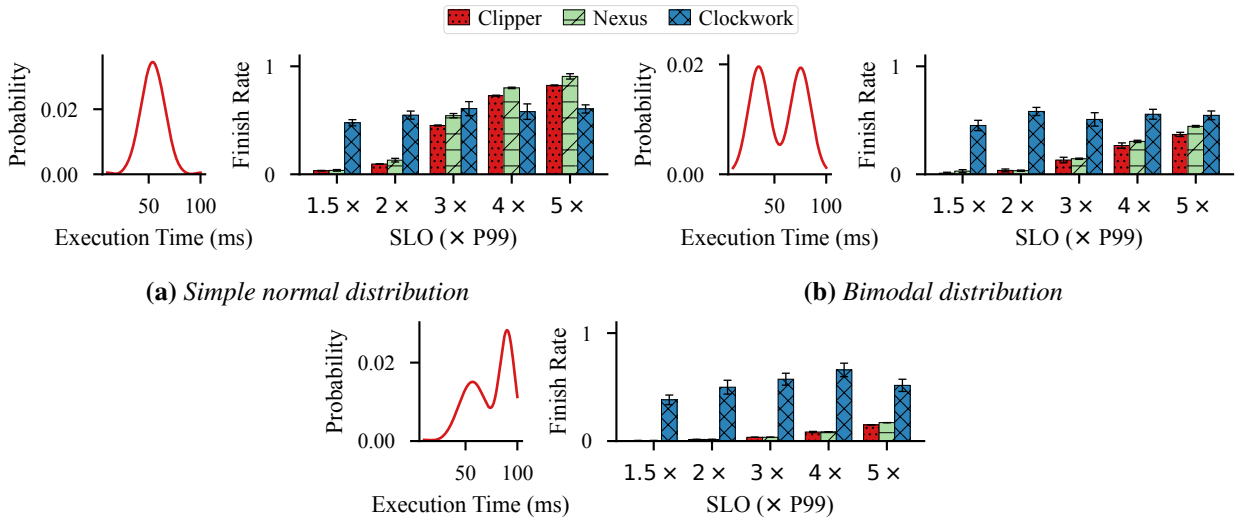
Figure 4.2: *Inference request execution time varies widely in dynamic CV and NLP models. (Inception V3 and ResNet are shown for comparison.)*

Batching and the Throughput-Latency Tradeoff To achieve high throughput, model serving systems [5, 7, 15, 38, 58] rely on batching multiple requests. Requests arriving within a specific time window are batched together to ensure that worker resources do not idle [38, 58]. There is, however, a tradeoff in picking a batch size. While a larger batch size may increase throughput, it also means longer time window; the latter can lead to higher (tail) latency and missed SLOs. In contrast, a smaller batch size can result in underutilized hardware. Existing solutions focus on finding the sweet spot to reduce SLO misses, typically for each model individually [7, 15, 38, 58], while pipeline-aware solutions break down the end-to-end SLO into smaller pieces [5]. The overall objective of a model serving system can, therefore, be described as *maximize throughput while reducing SLO misses*.

4.2.2 Dynamic DNNs

Point estimations in existing solutions is sufficient only because they focus on static DNNs (e.g., CNNs), where each request roughly takes the same amount of time and is highly predictable [7]. However, we observe that dynamic DNNs are becoming increasingly more popular in recent years [1, 2]. Dynamic DNNs adapt their structures or parameters to different inputs, leading to notable advantages in terms of accuracy, computational efficiency, adaptiveness, etc., compared to static DNNs that have fixed computational graphs and parameters at the inference stage [2]. Examples of dynamic DNNs include various language models [21, 72] as well as early-exit techniques used in emerging CNN models [54].

Observation: Dynamic DNN Inference is Unpredictable As the name suggests, the computation requirement of inference requests to a dynamic DNN model can be *dynamic*. It naturally follows that the request execution time is no longer constant for different inputs anymore. We report inference execution time histograms for four common dynamic networks (SkipNet [54], RDI-Nets [8] for image recognition, GPT [42], BART [29] for NLP) in Fig. 4.2. For comparison, we also include the execution time for two common static CV models: Inception V3 [88] and ResNet [85]. Note that



(a) *Simple normal distribution*

(b) *Bimodal distribution*

(c) *Bimodal distribution w/ inequal peaks*

Figure 4.3: *Performance of existing model serving solutions for dynamic DNNs. Similar results hold for more diverse distributions as well (Section 4.5).*

the absolute number of requests in these histograms is less important, as it merely represents the testing dataset we used. However, the existence of a large range of values in the x-axis reflects the possibility of different execution times.

For the image recognition models, there are a few distinct clustered ranges, which represent multiple code paths with different execution times, created by skipping/choosing parts of the model. Similar observations hold for NLP models too. However, instead of having distinct code paths, execution time for NLP models falls in a continuous range, reflecting the impact of the input sequence length on execution time. Nevertheless, it can be seen that the difference in inference latency can be as large as $10\times$, with some requests finishing in 10ms while many others taking more than 100ms.

Challenges in Serving Dynamic DNNs The presence of a large variance in execution times of dynamic DNNs as well as the shared nature of model-serving-as-a-service systems lead to two key challenges when serving dynamic DNNs.

1. **Effective batching:** Batching is a must for high throughput and worker utilization. However, all requests in the same batch start and finish at the same time, regardless of their individual execution times. When requests in the same batch vary widely in their execution times, the longest one becomes the straggler and slows down the entire batch.
2. **Handling multimodal distribution:** A model built for a specific task (e.g., classification, translation, etc.), is often used by multiple applications, especially when it is exposed as

a service. As a result, input requests and their corresponding execution times often follow different application-specific distributions. The combined multimodal distribution has even higher variance, which can introduce even more stragglers during batching.

4.2.3 Limitations of Existing Solutions

Indeed, state-of-the-art model serving solutions [7, 38, 58], which have been optimized for static DNNs, suffer from high SLO violation rates when applied to dynamic DNNs. Fig. 4.3 shows the finish rates of three recent model serving systems when the input execution time follows various distributions, under different SLO settings. For each case, the Probability Distribution Function (PDF) of the input execution times is shown to the left. For all cases, the incoming rate trace is derived from the Microsoft Azure Functions workload trace [16] similar to Clockwork [7]. Section 4.5 provides more details on the methodology.

The high-level takeaway is that all these systems have undesirable performance. As most batches contain both long requests and short ones, the execution time for the whole batch is almost always longer than the average. This causes Clockwork [7] to often mispredict a batch's latency, which in turn leads to frequent time-out error in its scheduler, causing the subsequent batch to fail. This explains its close-to-half finish rate. Nexus [38] pre-computes an execution plan ahead of time using the average execution time, but due to the variance in input execution, it cannot reach a stable state. Clipper [58] monitors request execution time reactively, but it cannot keep up under tight SLO settings. Fundamentally, we observe the effect of existing solutions failing to batch requests effectively.

Distribution-Based Schedulers Existing distribution-based schedulers such as 3Sigma [50] or Shepherd score [97], proposed for cluster scheduling and query processing respectively, do not fare well either. They do not consider sub-second level latency constraints or inference serving-specific challenges like batching and lack of preemption.

4.3 ORLOJ Overview

ORLOJ is an inference serving system that serves inference requests to a dynamic DNN model while maximizing the number of requests that can be served within the SLO. In this section, we provide an overview of how ORLOJ fits in the inference life cycle of dynamic DNNs to help the reader follow the subsequent sections.

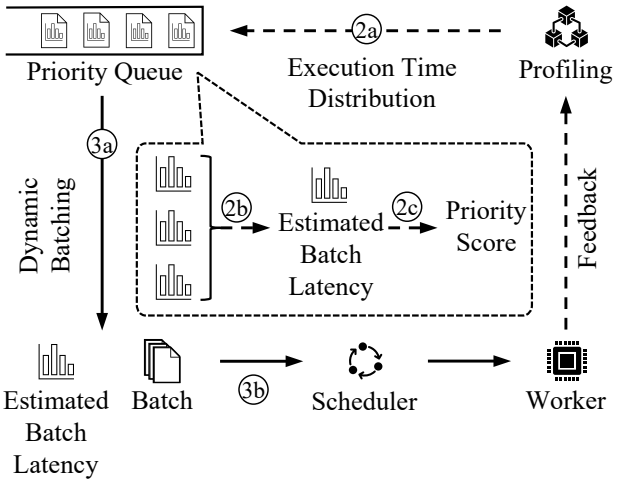


Figure 4.4: ORLOJ *architecture.*

4.3.1 Problem Statement

Each inference request in ORLOJ is defined by its *release time* and *deadline* (release time plus SLO), and has a minimum *execution time* that is measured when the request is executed alone. Multiple applications with diverse use cases and corresponding input distributions send requests to the same dynamic DNN model served by ORLOJ. Each GPU worker processes these requests one batch at a time. Note that, to scale out to a pool of workers in a cluster setting, different models and their replicas can use ORLOJ in parallel.

Given a set of pending inference requests, the ORLOJ scheduler must decide *which subset of them should be included in the next batch* submitted to the GPU to maximize throughput while reducing SLO misses, under the following constraints:

- **Partial information**: the execution time of a request, and thus a batch, is unknown to the scheduler. However, its probabilistic distribution can be learned from historical data.
- **Non-preemption**: inference execution of a batch cannot be preempted after it is submitted to the GPU.

4.3.2 ORLOJ Architecture

Unlike existing model serving systems, ORLOJ represents the request execution time of a dynamic DNN model as a random variable, which is described using an empirical distribution over a time window. Thereafter, instead of using simply the mean or the max of the population, it makes scheduling decisions using the knowledge of the entire distribution.

ORLOJ addresses the challenges arising from multimodal distribution and effective batching by proposing a *time-varying priority score* (detailed in Section 4.4) by considering a batch's combined

distribution. It determines this priority score for all requests that potentially can be put together to form a batch, and then it performs priority-based scheduling.

Inference Lifecycle As shown in Fig. 4.4, ORLOJ follows the same overall process as other model serving systems, but with updated scheduling steps ② and ③ in Fig. 4.1:

(2a) incoming requests are tagged per application, and the application-specific execution time distribution is associated with each request using the information collected by ORLOJ's online profiler;

(2b) execution time distributions from requests are combined to derive estimated batch execution time (latency);

(2c) next, the priority score for all requests are calculated using the estimated request latency, deadlines and the current time as input;

(3a) ORLOJ calculates estimated batch latency for all potential batch sizes;

(3b) the scheduler loop selects a feasible batch size to actually create the batch.

After obtaining a batch, ORLOJ submits it to the worker for execution, following the same step ④ as existing model serving systems.

Next, we elaborate on ORLOJ's scheduling loop and its batch handling.

Batch Size Selection It is not always possible to execute requests using the maximum possible batch size, because some requests may have tighter deadlines than others and waiting for the maximum batch size can take too long.

ORLOJ tracks the set of feasible batch sizes for each request. These sets of feasible batch sizes are updated over time. When the deadline is approaching, batch sizes that are too large to meet the deadline will be dropped, so the corresponding request will only be considered for small batches. In addition, for each batch size, ORLOJ keeps note of the earliest deadline for requests suitable for the batch size. The batch size with the overall earliest deadline will be chosen by the scheduler to lazily create a batch and send to the worker for execution.

Algorithm 4.1 shows ORLOJ's scheduling loop, which implements the above batch size selection scheme (Line 10 to Line 22). It is worth noting that ORLOJ uses a separate priority queue Q_{bs} for each batch size bs. The earliest deadline for requests in Q_{bs} is tracked by an additional Fibonacci heap to allow online deletion.

Batch Priority The highest priority batch is the one to be scheduled next, and it depends on the priorities of individual requests. ORLOJ uses a time-varying score for request priority, i.e., the priority of a request changes over time. Naively sorting the pending requests' queue in $O(n \log n)$ time for each scheduling iteration in the hot path is inefficient. Instead, we use an $O(\log^2 n)$ priority

Algorithm 4.1: ORLOJ Scheduler Iteration

Input:
$\mathcal{R} \leftarrow$ set of pending requests,
$t \leftarrow$ current time,
$\mathcal{S} \leftarrow$ set of batch sizes suported by the model,
$Q_{bs} \leftarrow$ set of requests viable for batch size bs,
$D_r \leftarrow$ deadline of request r,
$D_{Q_{bs}} \leftarrow \min\{D_r | r \in Q_{bs}\}$,
Output: Batch $\mathcal{B} \subseteq \mathcal{R}$

▷ `Update priority scores (Section 4.4.4)`
$U \leftarrow \emptyset$ ▷ `requests needs to be updated`
if *need reset base time* **then**
  reset base time
  $U \leftarrow \mathcal{R}$
for $r \in \mathcal{R}$ **do**
  if $t \geq$ `Milestone(`r`)` **then**
    $U \leftarrow U \cup \{r\}$
for $r \in U$, $bs \in \mathcal{S}$ **do**
  update priority score of r in Q_{bs}
▷ `Drop requests from queue if too late`
for $bs \in \mathcal{S}$, $r \in Q_{bs}$ *ordered by* D_r **do**
  if $t +$ `EstimateBatchLatency(`r, bs`)` $> D_r$ **then**
    $Q_{bs} \leftarrow Q_{bs} \setminus \{r\}$
  if bs *is the last feasible batch size for* r **then**
    Mark r as timed out
▷ `Determine candidate batch size`
$candidate \leftarrow$ nil
for bs *ordered by* $(D_{Q_{bs}}, bs)$ *in descending order* **do**
  if $|Q_{bs}| \geq bs$ **then**
    $candidate \leftarrow bs$
    break
if $candidate$ *is* nil **then**
  return
▷ `Select top ones ordered by ORLOJ score`
return $\mathcal{B} \leftarrow$ `PopBatch(`$Q_{candidate}$`)`

queue [97, 108]. We also address issues related to floating-point overflow when using such queue in practice (Section 4.4.4).

Before proceeding to the details of ORLOJ's algorithm in the next section, we highlight a few other components in ORLOJ.

Per-Application Tracking As shown in (2a), ORLOJ associates each request an execution time distribution using application-specific historical data. First, it is possible to distinguish requests from application as there are usually certain application IDs involved when the model is exposed as a service. Second, such tracking is also necessary, because applications may solve problems in different domains despite using the model for the same task. As a result, input requests execution times often follow different distributions. While ORLOJ does not assume any pre-defined distribution for its input and only tracks empirical distributions, the combined multimodal distribution has higher variance. This hurts scheduling abilities even if the scheduler has perfect information of its distribution, because the scheduler has to account for different possibilities when scheduling.

Long-Term Feedback Loop Incoming requests can change their arrival pattern and volume over time, either due to the diurnal nature of the service or due to shifts in general interests. ORLOJ therefore needs to track per application execution time data for requests over time. However, our calculation needs the execution time for requests when they execute alone, which cannot be guaranteed if simply measuring the time online. Instead, the profiler in ORLOJ takes an asynchronous approach. Finished requests are sampled and send to the profiler to evaluate individually. The execution time data will then be asynchronously picked up and accumulated by the scheduler periodically, completely off the critical path. In order to adapt to drifts in the input, ORLOJ resets its profiling memory every once a while. The exact window is configurable and is determined by domain knowledge.

4.4 Batch-Aware Distribution-Based Scheduling

At its core, ORLOJ is a priority-based scheduler where the priorities of individual requests are determined using a cost function that captures the distribution of request execution times. Highest priority requests are then put in a batch to achieve the maximum level of parallelism, which is then submitted to the worker. To achieve this, ORLOJ relies on probability theory to accurately estimate request execution times even when requests in the same batch affect each other and their executions are no longer independent.

In this section, after a brief introduction of cost function and the definition of priority on a single request (Section 4.4.1), we dive into the derivation of a vital term in the batch-aware priority score – batch execution time, given request execution time following the same distribution (Section 4.4.2.1) and different distributions (Section 4.4.2.2). Finally, we discuss how to break the cyclic dependency between batch formation and priority score computation (Section 4.4.3), as well as floating-point overflow handling when applying the algorithm in practice (Section 4.4.4).

4.4.1 Preliminaries

Cost Function Instead of directly optimizing for metrics such as average/tail latency and throughput, we model SLO deadlines using a cost function that captures the opportunity cost differences between two important scheduling decisions. On the one hand, executing a request involves costs for resource usage (e.g., server utilization) and opportunity cost (e.g., it may postpone other requests). On the other hand, missing deadline may have a monetary penalty according to the SLO. Throughout the rest of the paper, we use SLO cost functions similar to that in Fig. 4.5. Meaning, for requests arriving at time T with deadline D, there is a penalty c for missing that deadline.

Scheduler Objective The objective of our scheduler is to minimize the overall cost, or as we set out to do, maximize the number of requests that finish within corresponding deadlines. Selecting a request to put into the next batch reduces the expected cost that would have been incurred if it were delayed. Therefore, our goal is to find requests for which the ratio of expected cost reductions are the greatest.

Background: Priority of a Single Request Consider a request whose execution time L is a random variable and its cost function is $C(t)$. Cost reduction for this request boils down to the difference between the costs of two scheduler decisions: $C_{now}(t)$, including the request in the next batch and executing it right away, or $C_{delay}(t)$, selecting another one and thus delaying this request.

Its priority $p(t)$ can thus be defined as:

$$p(t) = \frac{1}{\mathbb{E}[L]} \left(\mathbb{E}[C_{delay}(t)] - \mathbb{E}[C_{now}(t)]\right) \tag{4.1}$$

Note that $C_{now}(t) = C(t + L)$ is a random variable, as well as $C_{delay}(t) = C(t + \tau + L)$ (τ is the anticipated delay).

Given that $C(t)$ has the same shape as in Fig. 4.5, and assuming τ follow an exponential distribution with parameter b,[1] prior work [97] has shown that, when L can be described using a histogram, $p(t)$ can be derived by computing on each histogram bin separately and combining the results:

[1]The probability that a request will be selected for execution, given that it is already queued, does not change with time unless there is a change in the state of the queue. Therefore, the anticipated delay is an exponential [125].

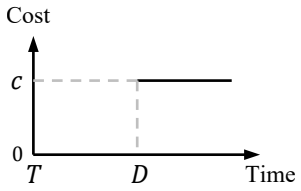


Figure 4.5: *An example of SLO cost function.*

$$
\begin{aligned}
p(t) &= \sum_i p_i(t) \\
p_i(t) &= \begin{cases}
\frac{hc}{\mathbb{E}[L]b}\left(e^{bl_2^{(i)}} - e^{bl_1^{(i)}}\right)e^{-bD}e^{bt} & t < D - l_2^{(i)} \\
\frac{hc}{\mathbb{E}[L]b} - \frac{hc}{\mathbb{E}[L]b}e^{bl_1}e^{-bD}e^{bt} & D - l_2^{(i)} \le t < D - l_1^{(i)} \\
0 & D - l_1^{(i)} \le t
\end{cases}
\end{aligned} \tag{4.2}
$$

where $p_i(t)$ is the score for the i-th bin in the histogram with range $[l_1^{(i)}, l_2^{(i)})$ and frequency h. We represent the deadline for the request in consideration using D, with c being the cost when missing the deadline.

4.4.2 Batch Latency Estimation

We still need to find out the distribution of L, as well as $\mathbb{E}[L]$. As we discussed in Section 4.2.1, the scheduler must schedule *batches* of requests to ensure high throughput. This is an important detail, because up until now, we assumed the execution time L as an intrinsic property of the request, depending solely on the request itself. However, under the batch execution model, it is no longer the case: *requests do not execute alone, and they affect each other's execution time in the same batch.* The purpose of the term $\frac{1}{\mathbb{E}[L]}$ in Eq. (4.1) is to account for the worker time usage of the request. As such, to accurately represent a request's potential worker time usage during batching, L must now be the execution time of the whole batch the request is in.

When the request execution time itself is a constant (i.e., in static DNN scenarios), this is trivial: requests are homogeneous; so is the batch. It is thus possible to profile the batch execution time for all possible batch sizes ahead of time [7]. In our case, however, not only are requests' execution times random variables, but *a batch may also contain requests from different duration distributions.* Next, we describe how ORLOJ handles batches of requests following the same or different distribution separately.

4.4.2.1 Requests in Batch Follow the Same Distribution

For a batch B of k requests, if requests are of the same duration l, the batch execution time l_B could be fairly assumed (within reasonable range) as

$$l_B = c_0 + c_1 k l \tag{4.3}$$

where c_0 and c_1 are constant parameters specific to a model and hardware.

In the case of dynamic models, requests in a batch are padded to the largest one and therefore, it can be viewed as if the whole batch's requests have the same length:

$$l = \max_{r \in B} l_r \tag{4.4}$$

Then the execution time of B becomes a new random variable L_B. With L_r as the random variable of request r's execution time, we have

$$\begin{aligned} \mathbb{E}[L_B] &= c_0 + c_1 k \, \mathbb{E}[\max_{r \in B} l_r] \\ &= c_0 + c_1 k \int_0^\infty L_{r(k)} f_{L_{r(k)}}(l) \, \mathrm{d}l \end{aligned} \tag{4.5}$$

where $L_{r(k)}$ is L_r's max order statistics over k samples, and $f_{L_{r(k)}}(l)$ is its PDF. While it is possible to directly find out $f_{L_{r(k)}}(l)$, it is easier to go through the Cumulative Distribution Function (CDF) of $L_{r(k)}$ first, denoted as $F_{L_{r(k)}}(l)$, which is related to the CDF of L_r via a simple equation:

$$F_{L_{r(k)}}(l) = [F_{L_r}(l)]^k \tag{4.6}$$

And we can obtain $F_{L_r}(l)$ from L_r's histogram. Note that while it is possible to directly use $F_{L_{r(k)}}(l)$ to calculate $\mathbb{E}[\max_{r \in B} l_r]$, the result would be far too inaccurate, as we only have a discrete histogram to start with.

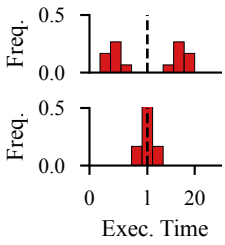
4.4.2.2 Requests in Batch Follow Different Distributions

Taking one step further, if requests $r_i (i = 1, 2, \ldots, k)$ in B come from different distributions, the problem becomes finding the max order statistics $L_{r(k)}$ for k random variables L_{r_i} that are independent, but not necessarily identically distributed.

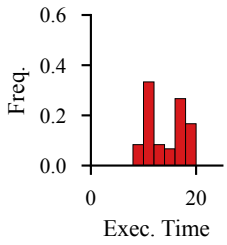
Let F_i and f_i be the CDF and PDF for L_{r_i}, respectively, and define F^s, f^s as

$$\begin{aligned} F^s &= \frac{1}{n_s} \sum_{i \in s} F_i \\ f^s &= \frac{1}{n_s} \sum_{i \in s} f_i \end{aligned} \tag{4.7}$$

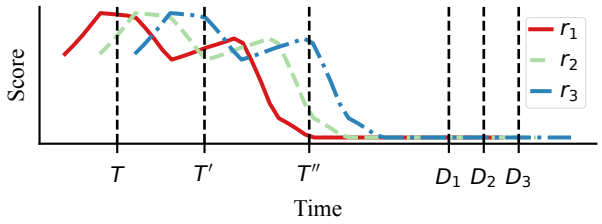
where s is a subset of requests ($s \subseteq B$) with $n_s \geq 1$ elements. The PDF of the maximum (i.e. k-th order statistics) of these k variables $f_{(k)}$ is given by Özbey et al. [33]:



(a) *Execution time of two types of requests.*



(b) *Execution time histogram for the batch.*



(c) *The priority score of r_1, r_2, r_3 entering the system one after another.*

Figure 4.6: *Toy example of batch execution time estimation and priority score computation.*

$$f_{L_{r(k)}} = \sum_{\kappa=1}^{k} (-1)^{k-\kappa} \frac{\kappa^k}{k!} \sum_{n_s=\kappa} k[F^s]^{k-1} f^s \tag{4.8}$$

Here $\sum_{n_s=\kappa}$ means summation over all possible $s \subseteq B$ where $n_s = \kappa$.
With $f_{L_{r(k)}}$, we can describe the PDF of L_B using Eq. (4.3)

$$f_{L_B}(l) = f_{L_{r(k)}}(\frac{l - c_0}{c_1 k}) \tag{4.9}$$

Similarly, by plugging Eq. (4.8) into Eq. (4.5), we can also find out $\mathbb{E}[L_B]$. We now have complete ingredients to compute the priority score $p(t)$.

A Toy Example Let us use an example to illustrate how $p(t)$ changes over time t, to put the above discussed equations into perspective.

Consider two types of requests in total, whose execution time follow two different distributions, as shown in Fig. 4.6a. While they all have the same mean execution time l, the first distribution has higher probability of finishing exactly at l, and the second one may either finish very early, or very late. Further, assuming that the batch size in consideration is 2 and there is no overhead for batching (i.e. $c_0 = 0, c_1 k = 1$).

Figure 4.6b shows the histogram of L_B that is derived using Eq. (4.9). As expected, the execution time for the whole batch is skewed to the right, because it is not possible for the shorter execution time in the first distribution to ever become the whole batch's execution time.

Figure 4.6c illustrates the priority score changing over time for three requests r_1, r_2, r_3 entering the system one after another. At time T, r_1 and r_2 are the most urgent and should be executed. As the deadline is approaching, r_1 and r_3 become the top ones at time T'. Finally, r_2 and r_3 would have been selected as the score for r_1 drops to 0.

4.4.3 Batch Formation

The last step is forming the batch, which must address a key challenge: the circular dependency between priority calculation and batch formation. $\mathbb{E}[L]$ is used in the priority calculation, so the list of possible distributions must be known. However, the batch is determined *after* priority is computed, and thus it is impossible to know what other requests are in the batch. Assuming that the queue most likely contain requests from all types of applications the model is serving, we therefore always use all execution time distributions associated with the model to compute $\mathbb{E}[L]$. As this list of distributions is available ahead of time, and only depends on the batch size, this approach has the additional benefit that the relatively heavy computation can be moved away from the critical path.

4.4.4 Efficient Computation

As shown before, the priority score for a request varies depending on the remaining time before its deadline. It thus has to be re-computed continuously. Combined with the effort to re-sort all pending requests, the naive implementation is not scalable to large number of requests.

It is possible to convert the problem to dynamic convex hull querying, which can achieve $O(\log^2 n)$ complexity for each reschedule, where n is the number of pending requests [108]. The dynamic convex hull problem is defined by rewrite each request's priority Eq. (4.2) in the form of $p_i(t) = \alpha e^{bt} + \beta$, and consider each request to be on a 2D plane with the coordinate (α, β). Then, the first point on the convex hull hit by affine lines of slope $-e^{bt}$ corresponds to the request with the highest score at time t.

This way, the problem is divided into one time-invariant part where the relative positions of requests only change a few times (when the relationship between $t, D, l_1^{(i)}, l_2^{(i)}$ changes, corresponding to the `Milestone` function in Algorithm 4.1), and one time-varying part – querying the convex hull with a line. Therefore, the priority queue can be implemented by maintaining a convex hull containing all pending requests.

However, while querying a static convex hull is trivial, our convex hull changes over time as requests come and go. In ORLOJ, we use the convex hull algorithm proposed by Overmars and von Leeuwen [127], which supports dynamically adding/removing points on the convex hull in $O(\log^2 n)$ complexity and can be queried with a line in $O(1)$ time.

Overflow Handling of Exponential Values While the theory works out, there is still a non-trivial challenge that we faced when implementing the priority score in practice.

In the original Eq. (4.2), the score only depends on $D - t$ which is the remaining time before the deadline, and is bounded assuming requests too far in the future should not enter the system. However, the clever 2D plane mapping breaks the component into e^{-bD} and e^{bt} individually. Because D and t can be large timestamps (usually represented as elapsed seconds/milliseconds since UNIX epoch), this leads to floating-point overflow when the system tries to compute and store these very large exponential values.

We compensate this by using relative timestamps for D and t, and then choose b wisely. If the time resolution is in milliseconds, and $b = 10^{-4}$, we can sustain about 1000 s of scheduling before overflows of 64-bit floating-point numbers and having to reset the relative timestamps' reference point and thus re-calculate everything. Note that the exact value of b does not matter because it does not change the relative ordering of requests as long as it is kept constant.

4.5 Evaluation

We evaluate ORLOJ against three existing serving systems (Clipper [58], Nexus [38], and Clockwork [7]). Our primary findings are as follows:

- Compared to the state-of-the-art, ORLOJ can improve the finish rate when serving dynamic DNNs by 51–80% under extremely tight SLO constraints, or over 100% under more relaxed SLO settings (Section 4.5.3). At the same time, ORLOJ keeps comparable performance when serving traditional static models (Section 4.5.4).
- ORLOJ can sustain thousands of pending requests in its priority queue with less than 0.5 ms per-request insertion time (Section 4.5.5).
- Our choice of b, the anticipated delay distribution parameter (Section 4.4.1) in the priority score is safe, as ORLOJ is not sensitive to the value of b (Section 4.5.6).
- ORLOJ has minimal overheads and can manage requests with execution time varying in ranges as low as 2 ms–20 ms (Section 4.5.7).

4.5.1 ORLOJ Implementation

We implemented ORLOJ on top of Clockwork [7], a state-of-the-art serving system with fewer than 4000 lines of C++ code. One-fourth of the new code is for implementing the dynamic convex hull data structure, as there is no established library available for solving the dynamic convex hull problem. Specifically, we implemented the inner *concatenate queue* as a 2–3 tree extending from the left-leaning-red-black-tree [115, 128].

4.5.2 Experimental Methodology

Testbed We built our testbed on Chameleon Cloud [9]. The host has 2 Intel Xeon Gold 6230 CPUs with NVIDIA V100 GPU. In order to have stable results, we fix the GPU clock speed to its maximum 1380 MHz and memory clock speed to 877 MHz.

We use Ubuntu 20.04 as the base OS environment with the latest NVIDIA GPU driver. We use CUDA versions matching the published original source code, which means CUDA 11.1.1 with CuDNN 8 for Clockwork and ORLOJ, CUDA 10.0 with CuDNN 7 for Nexus. For Clipper, its latest commit `9f25e3f` is used.

During experiments, each evaluated system's server is deployed on the host. Clockwork and ORLOJ additionally have their serving threads set to high-priority and pinned to physical cores as per Clockwork's host setup instructions. In addition, the model is modified to allow us to explicitly control its execution time via input for the purpose of evaluation.

An open loop (no wait for requests completion before issuing the next one) client is used to drive all experiments on the same host to minimize the impact of networking.

Input Trace Similar to workload traces for static models, the request incoming rate trace determines how fast requests coming in and needs to match the system's load. Same as Clockwork's evaluation, we adapt the Azure trace [16], which is published by Microsoft for lambda functions. The trace was scaled down such that the incoming rate matches the system load. The incoming rate trace is kept the same across all experiments.

Request Execution Time Distribution Unlike a single number for one model/dataset combination in evaluations in static serving systems, we need a full distribution. We group the model's associated dataset into short-running and relatively long-running requests (or more groups in case of higher modality), then randomly choose from them to get a mixture of both. To get a fair comparison, the generation is done once among different runs, we then record the arrival time and the input, which will be replayed for subsequent runs.

Real World Dataset We evaluate ORLOJ on a set of real world learning tasks covering both CV ones like image classification, and NLP ones including chatbot, summarization and translation (Table 4.1). For each workload, the input execution time distribution is determined according to the above-mentioned method, and the P99 of real execution time used to determine SLO is reported in the table.

Metrics We focus on the *finish rate*, which is defined as the ratio of the number of requests finished in time to the total number of requests. We assume that the SLO is set to a reasonable

Table 4.1: *List of workloads for* ORLOJ.

Task	Model	Dataset	Mean Exec. (ms)	P99 Exec. (ms)
Image classification	RDI-Nets [8]	CIFAR [113]	683.15	2667.54
Image classification	SkipNet [54]	ImageNet [94]	3.24	5.56
Chatbot	Blenderbot [14]	convAI [22]	200.39	242.27
Chatbot	Blenderbot	Cornell [109]	203.22	247.04
Chatbot	GPT [42]	convAI	79.47	143.40
Chatbot	GPT	Cornell	94.84	161.69
Summarization	BART [29]	CNN [69]	774.66	1101.99
Summarization	T5 [35]	CNN	552.91	797.28
Translation	FSMT [32]	WMT [43]	189.30	319.31
Translation	mBART [17]	WMT	432.38	729.87

value manually given historical data, similar to serving static models. Using the P99 tail of all input requests' real execution time as a measurement, we vary the SLO to be multiples of P99 for most our experiments.

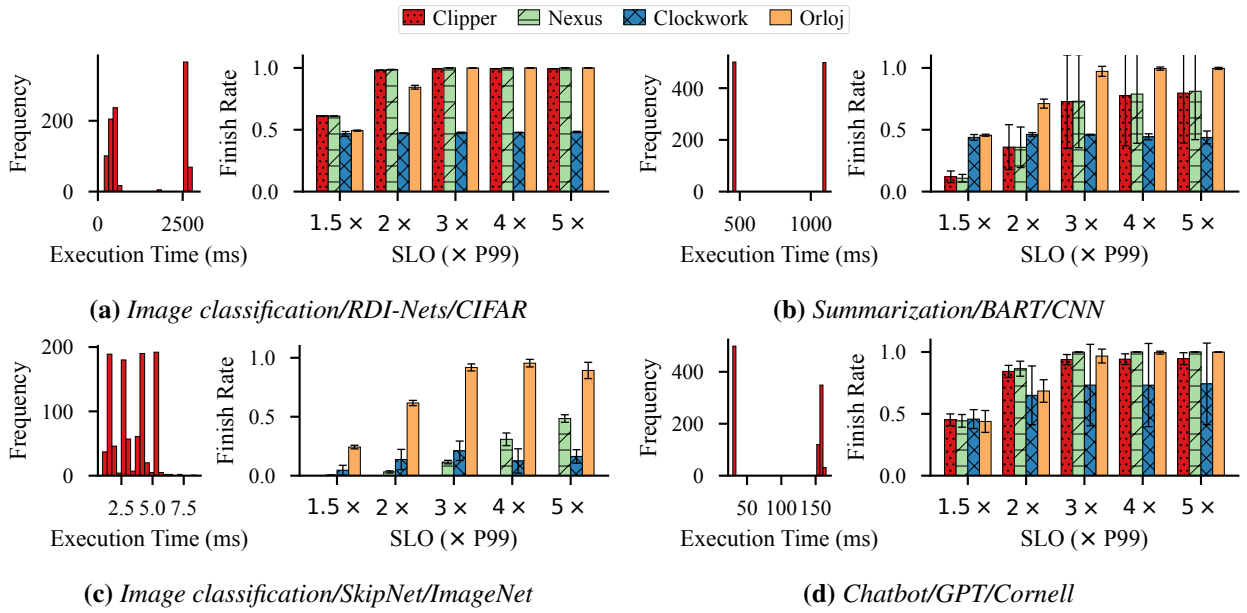
4.5.3 Improvements for Dynamic Workloads

Real World Dataset We report representative cases in Fig. 4.7, while the complete results can be found in Appendix C.1. In most workloads, existing systems can barely make it due to the mixture of long and short requests which rarely match the mean execution time those systems use for scheduling. For RDI-Nets/CIFAR, BART/CNN and GPT/Cornell, ORLOJ can reach near 100% finish rate with sufficient SLO settings. When requests become extremely short (e.g., Fig. 4.7c), none of the system can finish many requests due to the too tight latency requirement. However, ORLOJ still manages to finish more requests than others.

Different Distributions We then evaluate ORLOJ's performance under more diverse distributions using the same BART model and with a synthesized dataset to control execution times.

In Fig. 4.8, we increase the number of modalities of the distribution to simulate the effect of multiple applications. With the number of modalities increases, the variation in execution time increases. ORLOJ keeps relatively good finish rate and see performance gain as high as $2\times$. The result is consistent with even higher modalities, and we report additional results for up to 8 modals in Appendix C.1.

The distributions in Fig. 4.9 are the same except for mirrored inequal peak locations. However,



(a) *Image classification/RDI-Nets/CIFAR* **(b)** *Summarization/BART/CNN*

(c) *Image classification/SkipNet/ImageNet* **(d)** *Chatbot/GPT/Cornell*

Figure 4.7: ORLOJ *performance on real world tasks. Error bars represent standard deviation across 5 runs.*

Clipper and Nexus suffer more in Fig. 4.9a because exceptional longer than expected requests are definitely timed out while exceptional shorter than expected requests can still meet the deadline.

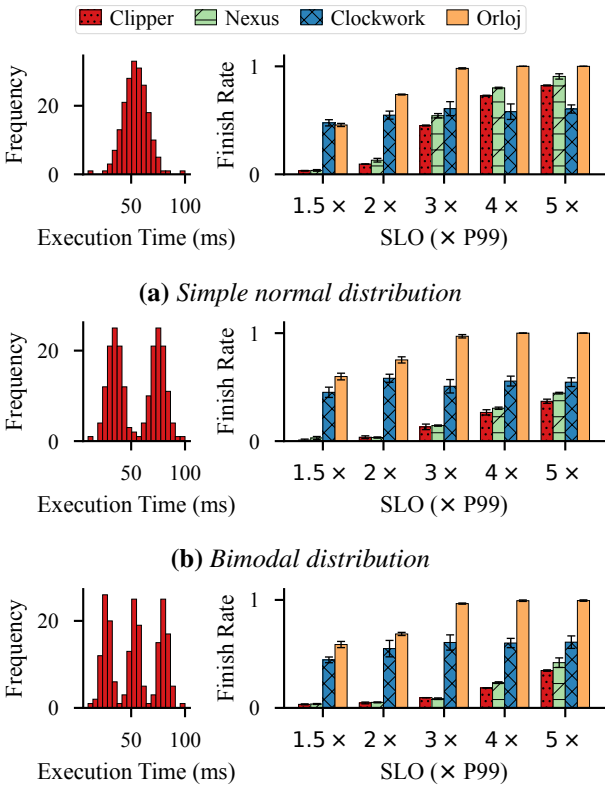
Figure 4.10 extends on Fig. 4.8b, whose input request execution time distribution is generated with $\sigma = 1$, and explores the effect of smaller ($\sigma = 0.5$) and larger ($\sigma = 1$) values. Larger σ means the peaks are less distinguishable and the degrees of longer requests blocking shorter ones is less severe. ORLOJ's performance remains stable while others see slightly higher finish rate when the separation between requests become less extreme and vice versa.

Clockwork's performance is not affected by changes in distributions. As long as the execution time is not constant, it suffers from the same fail-every-other-batch pattern as we discussed in Section 4.2.3.

4.5.4 Improvements for Traditional Workloads

We next verify ORLOJ's performance under traditional workloads using static models where there is no variance in request execution time.

Using the ImageNet [94] dataset, we measure the finish rate for four systems when serving the ResNet [85] model and Inception V3 [88] model (Fig. 4.11). ORLOJ sees significant improvement over Nexus and Clipper under tight SLOs ($1.5\times$ and $2\times$), thanks to its plan-ahead scheduling. When compared to Clockwork, upon which ORLOJ is built, due to differences in request handling mechanisms, ORLOJ performs slightly better when SLO is higher while Clockwork has higher finish rate under tight SLOs, albeit with higher variances.

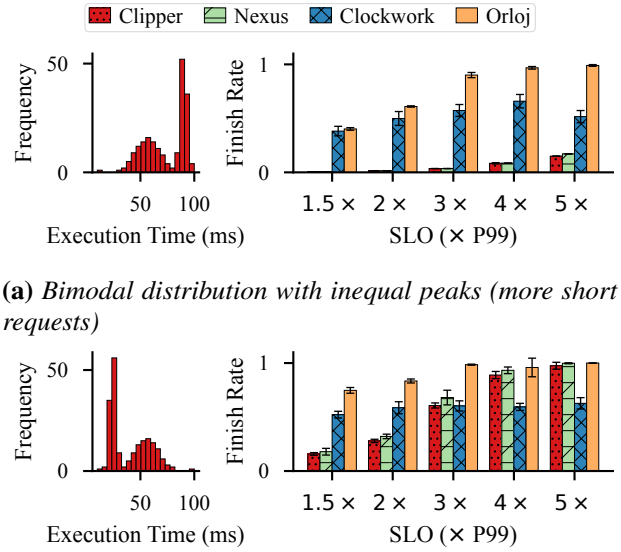


(a) *Simple normal distribution*

(b) *Bimodal distribution*

(c) *Three-modal distribution*

Figure 4.8: *Finish rate under different modality distributions.*



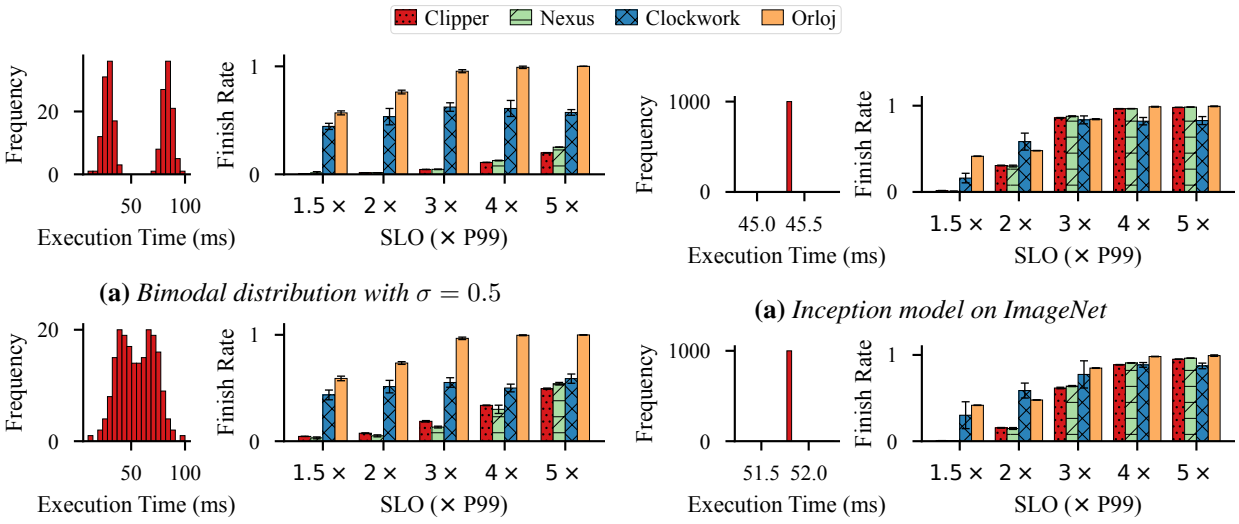
(a) *Bimodal distribution with inequal peaks (more short requests)*

(b) *Bimodal distribution with inequal peaks (more long requests)*

Figure 4.9: ORLOJ*'s performance under inequal-peak distributions.*

4.5.5 Efficiency of the Priority Queue

To study the efficiency of our priority queue implementation, we evaluate two of the most common operation of the priority queue in isolation: insertion and query. To measure the time it takes to insert a request into the queue, we run micro-benchmarks that fill the queue to certain number of requests, and compute the average insertion time per-request. For query, we first fill the queue and measure the time it takes to query the queue against a line of random slope – the equivalent of finding the request with the highest priority, as discussed in Section 4.4.4. We vary the number of requests in queue from 10 to 10000, and each data point is averaged over 100 samples. As reported in Fig. 4.12, the insertion operation takes longer as the queue becomes larger, and the overall complexity trend fits the theory $O(\log^2 n)$ line pretty well. Query time sees large variation when the number of requests is small, but stabilizes and remains constant as the queue size increases. Overall, we can see that thanks to the efficient implementation, thousands of requests can be handled in negligible time, and thus ORLOJ is able to schedule large number of pending requests.



(a) *Bimodal distribution with* $\sigma = 0.5$

(b) *Bimodal distribution with* $\sigma = 2$

Figure 4.10: *Finish rate under different distribution parameters.*

(a) *Inception model on ImageNet*

(b) *ResNet model on ImageNet*

Figure 4.11: ORLOJ *keeps comparable performance under workloads where there is no variance in request execution time.*

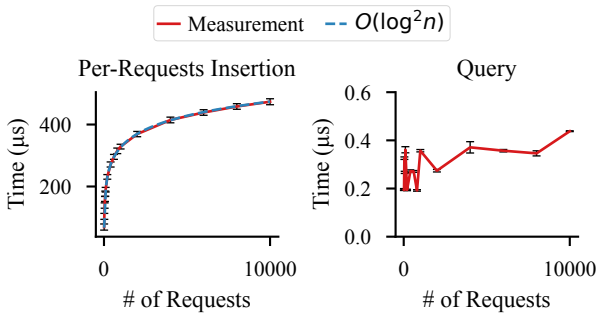


Figure 4.12: *The insertion time of* ORLOJ *priority queue under different numbers of requests in queue.*

4.5.6 Sensitivity to the Anticipated Delay Distribution

In the priority score calculation in Section 4.4.1, we introduce a parameter b when describing the distribution of the anticipated delay. And we discussed that in order to avoid floating-point overflow during calculation, we choose $b = 10^{-4}$ in Eq. (4.2). To verify that our choice of b is reasonable and ORLOJ's scheduling is not sensitive to the value of b, we do a parameter sweep with $b = 10^{-6}, 10^{-5}, \ldots, 10^{-1}$, and measure the performance of ORLOJ using the three-modal distribution as shown in Fig. 4.8c.

Each line in Fig. 4.13 represents the trend of finish rate under a given SLO setting (as a multiple of P99 execution time). And it can be seen that under all SLO settings, ORLOJ indeed keeps stable finish rate regardless of the choice of b. It is therefore safe to choose b to account for floating-point

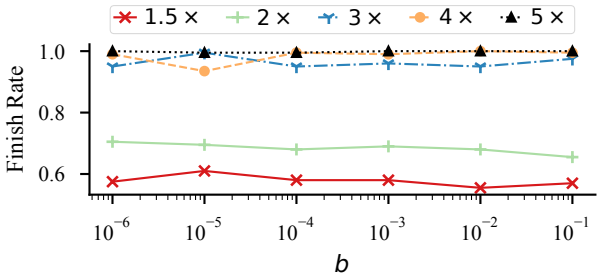


Figure 4.13: *Finish rate as we vary b. Note the x-axis is in log scale.*

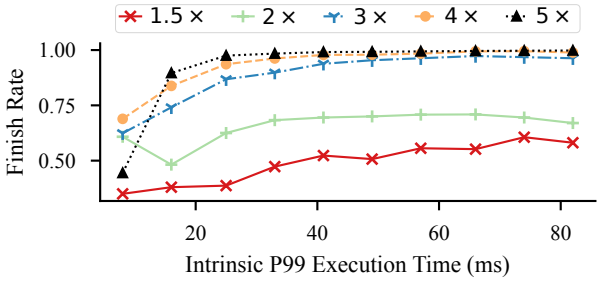


Figure 4.14: *Finish rate as we vary incoming requests' minimum execution time.*

overflow, as we do in ORLOJ's implementation.

4.5.7 Overheads

To understand ORLOJ's scheduling overhead, we evaluate the lower limit on SLOs that ORLOJ can achieve by measuring the finish rate while varying incoming requests' minimum execution time (time to execute one request alone). In this experiment, we use the same workload with three-modal distribution as shown in Fig. 4.8c, and scale the whole execution time distribution down until ORLOJ's finish rate drops significantly.

Fig. 4.14 reports the trends of finish rates under different SLO settings. Similar to other experiments, we set SLOs according to the P99 of minimum execution time and report results under different SLO to P99 ratios from $1.5\times$ to $5\times$. ORLOJ keeps consistent and stable finish rate. Its performance only starts to degrade when the minimum P99 execution time is approaching 20 ms. At that time, due to variance in request execution time, the mean approaches 10 ms and requests can go as low as less than 2 ms.

4.6 Related Work

Model Serving We directly compare ORLOJ to Clockwork [7], Nexus [38] and Clipper [58]. While Clockwork's assumption about static DNNs no longer holds for the dynamic DNNs, its idea

of proactively planning ahead and consolidating choices across layers to reduce disturbance still applies in ORLOJ. In addition, existing systems propose several orthogonal concepts that can be seen as complementary to ORLOJ. Clipper's idea of model selection and INFaaS's [15] model variant concept could be applied in front of ORLOJ. Nexus' model prefix-sharing and InferLine's [5] inference pipelines are also compatible with ORLOJ's idea of tracking request execution time as random variables.

In cloud or serverless platforms there are projects focusing on serving models at scale [20, 28, 41]. TensorFlow Serving [66] provides one of the first production environments for models trained using the TensorFlow framework. SageMaker [130], Vertex AI [135] and Azure ML [139] are public cloud DNN serving systems that offer developers inference services that auto-scale based on load.

Cost-Aware Scheduling One family of the well-studied scheduling algorithms are cost-aware or utility-based algorithms. The decisions in these algorithms are made to optimize certain costs, which could be defined in many ways: they could be fixed or time-varying values [118, 123], costs of rolling back transactions [124], or derived from SLOs [117, 119, 120]. It is however common in these algorithms to assume the exact execution time to be available during scheduling, which is not the case in serving dynamic DNNs.

Unknown-Sized Job Scheduling in Cluster Scheduling jobs with unknown duration has been studied in cluster computing. 3Sigma [50] uses the job length distribution in the scheduling and enumerates all possible choices to find the optimal scheduling decision. Age-based mechanisms [25, 53] gradually update jobs' priorities based on sustained service time. There are also various techniques used to mitigate mis-prediction in case the job's length exceeds expectation [51].

However, inference request serving differs from the above in its timescale and properties. Inference requests usually complete in less than a second, whereas cluster jobs can last hours or even days. So our scheduler has to make decision very quickly, unlike cluster schedulers that can use extensive searching before settling on a schedule. Furthermore, while cluster jobs are usually preemptable, inference requests are not, which rules out age-based algorithms.

4.7 Conclusion

Dynamic DNNs adapt their structures or parameters to the input, and thus experiencing rapid development thanks to notable advantages in terms of accuracy, computational efficiency, adaptiveness, etc. This challenges existing DNN serving solutions that assume data-independence of incoming requests, and they suffer from poor performance due to the large variance in request execution times. We propose ORLOJ, a dynamic DNN serving system, to meet these challenges. ORLOJ

captures the variance in dynamic DNNs by modeling request execution time as random variables, and then efficiently batches and schedules them without knowing a request's precise execution time. We demonstrated that ORLOJ significantly outperforms states-of-the-art serving solutions for high variance dynamic DNN workloads while maintaining nearly identical performance for static workloads.

While we take a first step in this paper, we hope that ORLOJ will inspire further research not only on dynamic DNN inference serving systems but other aspects of dynamic DNN lifecycle as well.

CHAPTER 5

Conclusions

Given newly emerging hardware and applications in the era of the advent of DL, this dissertation demonstrates that there are mismatches between the DL infrastructure and the hardware below it, as well as the applications above it. By leveraging application/system specific information in system design, we first tackle the problem of *cooperative sharing on GPUs during model training*. Moving up in the software stack, we then systematically design an execution engine for *efficient hyperparameter tuning in the cluster with adaptive execution plans*, and finally, we break one of the fundamental assumptions in existing works on model serving and *use distribution information and probability theory to serve inference requests for dynamic DNNs with tight SLO constraints*.

GPU Sharing Primitives for DL Applications SALUS observes that the coarse-grained, one-at-a-time GPU allocation model in DL model training is prohibitive for high GPU utilization and insufficient for flexible scheduler design in the cluster. By providing a consolidated execution service, SALUS implements two missing primitives for GPU execution: *fast job switching* and *memory sharing*. Such primitives enable fine-grained GPU sharing among complex, unmodified DL jobs. In addition to allowing the packing of multiple jobs on the same GPU, they can be used to implement unforeseen new policies as well.

Unified Hyperparameter Tuning Interface and Elastic Trial Execution As a flexible and unified high-level interface for hyperparameter tuning algorithms, the proposed TrialGroup abstraction decouples execution logic from tuning algorithm designers and users, enabling us – the systems researchers – to focus on efficient execution engine design. The resulting execution engine, FLUID, avoids common pitfalls in parallelizing tuning algorithms and implements complex execution plans like elastically scaling evaluation trials up and down during runtime, in reaction to cluster resource changes. Thanks to TrialGroup, the rich dependency information among trials are kept, and all improvements brought by FLUID in efficient execution can thus be applied to a broad range of algorithms to boost their performance.

Breaking the Data-Independency Assumption in Model Serving Existing model serving solutions can provide tight latency SLOs while maintaining high throughput by carefully scheduling of incoming requests, whose execution times are assumed to be highly predictable and data-independent. In ORLOJ, we break such assumptions and propose to model request execution time as random variables. The scheduler then efficiently batches and schedules requests without knowing their precise execution times. By using distribution information and the probability theory, ORLOJ is able to handle inference requests for emerging dynamic DNNs, whose inference requests are data-dependent due to their runtime changing model structure or parameters. ORLOJ can significantly outperform states-of-the-art serving solutions for high variance dynamic DNN workloads while maintaining nearly identical performance for static ones.

In the rest of this chapter, we discuss the limitations in Section 5.1, propose several directions for future work in Section 5.2, and finally, conclude.

5.1 Limitations

SALUS SALUS sidesteps the GPU hardware black box by performing all scheduling in software, before entering the GPU. However, due to performance reasons, SALUS focuses on memory allocations and does not handle computation scheduling decision enforcement. In addition, SALUS assumes a cooperative environment due to the lack of true isolation of execution of jobs from different users.

FLUID The TrialGroup concept proposed by FLUID is powerful enough to express a large body of hyperparameter tuning algorithms. However, it still requires the algorithms to opt in to such interface to maximize its benefits in scheduling. Also, FLUID currently only considers workers as homogenous resources, and is agnostic to location information. In practice, clusters may contain heterogeneous hardware and communication topologies. Placing jobs arbitrarily therefore is prone to suboptimal performance due to unnecessary data transfers or unbalanced worker execution speeds.

ORLOJ ORLOJ tracks per-application execution time distribution, only to later combine them all together when estimating batch execution times and calculating its priority score. This is limited by the circular dependency between the priority score and the actual batch creation. The current implementation in ORLOJ assumes that each application served by the model contributes roughly equally to pending requests. However, when it is no longer the case, for example, when the proportion of applications are highly skewed, the inaccurate estimations used in priority score can lead to over-conservative or over-optimistic scheduling decisions, resulting in suboptimal performance.

5.2 Future Work

In this section, we identify several avenues for future work to further bring closer the gap among the hardware, the application, and most importantly, the DL infrastructure stack in between.

Improving OS Abstraction for Accelerators SALUS takes a step toward bringing back the traditional OS-level abstraction for GPUs, but it is only a first attempt, and it opens many interesting new research challenges. First, SALUS provides a mechanism for sharing but the question of policy – what is the best scheduling algorithm for DL jobs running on a shared GPU? – remains open. Second, we only discussed SALUS on a single GPU, it naturally follows that what is the best strategy to integrate SALUS's idea with multiple GPUs on the same host, or GPUs across multiple hosts.

Abstracting DL Execution FLUID helps with the execution strategy selection in hyperparameter tuning jobs. The problem of automated execution strategy selection for general DL training jobs is still open. ML practitioners are forced to implement strategies like data/model/pipeline parallelism, select batch sizes, select the degree of parallelism when launching a job. Ideally, the cluster job manager should be able to figure out the execution plan on itself, only given an input of whether to optimize for job completion time, energy usage, target accuracy, etc.

Serving Dynamic Model in Pipeline Instead of a single SLO for the model, the pipeline may have an entire end-to-end SLO, which needs to be divided between multiple models. Existing pipeline-aware DL serving systems still rely on reactive provisioning or single-point estimations to perform scheduling. Applying the idea of using distribution information for each single dynamic model to the pipeline represents new challenges, such as how to combine and divide the end-to-end SLO when each stage's latency is a random variable.

Efficient Dynamic Model Execution Currently, padding is the de facto way of executing dynamic models while enabling batching. That however is neither necessarily the only nor the best way, due to the wasted computation on all padded areas. Further research are needed to discover more efficient means to execute a dynamic model, which will inevitably exhibit very different runtime characteristics when serving the model, and necessitate another round of re-designing of the serving system.

5.3 Final Remarks

It is impossible to address all challenges in the DL infrastructure in one dissertation. Indeed, this work only focuses on a small fraction of systems in the ecosystem while leaving many other areas

untouched. Nevertheless, with the ever accelerating growth in both DL applications and DL-specific hardware, the pursuit of higher efficiency never ends. This dissertation takes a step towards deeper application/hardware-awareness in the infrastructure with the hope that our crude work may be a basis for further research to better scheduling algorithms and overall to better integrations among DL applications, hardware and the infrastructure in between.

BIBLIOGRAPHY

[1] Rishi Bommasani et al. *On the Opportunities and Risks of Foundation Models*. 2021. arXiv: 2108.07258 [cs.LG].

[2] Yizeng Han et al. *Dynamic Neural Networks: A Survey*. 2021. arXiv: 2102.04906 [cs.CV].

[3] Lukasz Wesolowski et al. "Datacenter-Scale Analysis and Optimization of GPU Machine Learning Workloads". In: *IEEE Micro* 41.5 (2021), pp. 101–112. DOI: 10.1109/MM.2021.3097287.

[4] Peifeng Yu, Jiachen Liu, and Mosharaf Chowdhury. "Fluid: Resource-aware Hyperparameter Tuning Engine". In: *Proceedings of Machine Learning and Systems 3*. MLSys. 2021.

[5] Daniel Crankshaw et al. "InferLine: latency-aware provisioning and scaling for prediction serving pipelines". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. SoCC. 2020, pp. 477–491.

[6] William J Dally, Yatish Turakhia, and Song Han. "Domain-specific hardware accelerators". In: *Communications of the ACM* 63.7 (2020), pp. 48–57.

[7] Arpan Gujarati et al. "Serving DNNs like clockwork: Performance predictability from the bottom up". In: *14th USENIX Symposium on Operating Systems Design and Implementation*. OSDI. 2020, pp. 443–462.

[8] Ting-Kuei Hu et al. *Triple Wins: Boosting Accuracy, Robustness and Efficiency Together by Enabling Input-Adaptive Inference*. 2020. arXiv: 2002.10025 [cs.CV].

[9] Kate Keahey et al. "Lessons Learned from the Chameleon Testbed". In: *USENIX Annual Technical Conference*. ATC. 2020.

[10] Tan N. Le et al. "AlloX: Compute Allocation in Hybrid Clusters". In: *ACM EuroSys*. 2020, 31:1–31:16.

[11] Richard Liaw et al. *HyperSched: Dynamic Resource Reallocation for Model Development on a Deadline*. 2020. arXiv: 2001.02338 [cs.DC].

[12] Peter Mattson et al. "MLPerf: An Industry Standard Benchmark Suite for Machine Learning Performance". In: *IEEE Micro* 40.2 (2020), pp. 8–16.

[13] Andrew Or, Haoyu Zhang, and Michael Freedman. "Resource Elasticity in Distributed Deep Learning". In: *Proceedings of Machine Learning and Systems 2*. MLSys. 2020.

[14] Stephen Roller et al. *Recipes for building an open-domain chatbot*. 2020. arXiv: 2004.13637 [cs.CL].

[15] Francisco Romero et al. *INFaaS: A Model-less and Managed Inference Serving System*. 2020. arXiv: 1905.13348 [cs.DC].

[16] Mohammad Shahrad et al. "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider". In: *USENIX Annual Technical Conference*. ATC. 2020, pp. 205–218.

[17] Yuqing Tang et al. *Multilingual Translation with Extensible Multilingual Pretraining and Finetuning*. 2020. arXiv: 2008.00401 [cs.CL].

[18] Peifeng Yu and Mosharaf Chowdhury. "Salus: Fine-Grained GPU Sharing Primitives for Deep Learning Applications". In: *Proceedings of Machine Learning and Systems 2*. MLSys. 2020.

[19] Ahsan Alvi et al. "Asynchronous Batch Bayesian Optimisation with Improved Local Penalisation". In: *Proceedings of the 36th International Conference on Machine Learning*. ICML. 2019, pp. 253–262.

[20] Anirban Bhattacharjee et al. "Barista: Efficient and scalable serverless serving system for deep learning prediction services". In: *IEEE International Conference on Cloud Engineering*. IC2E. IEEE. 2019, pp. 23–33.

[21] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. NAACL. 2019, pp. 4171–4186.

[22] Emily Dinan et al. *The Second Conversational Intelligence Challenge (ConvAI2)*. 2019. arXiv: 1902.00098 [cs.AI].

[23] Dmitry Duplyakin et al. "The Design and Operation of CloudLab". In: *2019 USENIX Annual Technical Conference*. USENIX ATC. 2019, pp. 1–14.

[24] Debo Dutta and Xinyuan Huang. "Consistent Multi-Cloud AI Lifecycle Management with Kubeflow". In: *OpML*. 2019.

[25] Juncheng Gu et al. "Tiresias: A GPU Cluster Manager for Distributed Deep Learning". In: *16th USENIX Symposium on Networked Systems Design and Implementation*. NSDI. 2019, pp. 485–500.

[26] John L Hennessy and David A Patterson. "A new golden age for computer architecture". In: *Communications of the ACM* 62.2 (2019), pp. 48–60.

[27] Myeongjae Jeon et al. "Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads". In: *2019 USENIX Annual Technical Conference*. USENIX ATC. 2019, pp. 947–960.

[28] Ram Srivatsa Kannan et al. "Grandslam: Guaranteeing slas for jobs in microservices execution frameworks". In: *Proceedings of the 14th European Conference on Computer Systems*. EuroSys. 2019, pp. 1–16.

[29] Mike Lewis et al. *BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension*. 2019. arXiv: 1910.13461 [cs.CL].

[30] Kshiteej Mahajan et al. *Themis: Fair and Efficient GPU Cluster Scheduling for Machine Learning Workloads*. 2019. arXiv: 1907.01484 [cs.DC].

[31] Deepak Narayanan et al. "PipeDream: Generalized Pipeline Parallelism for DNN Training". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP. 2019, pp. 1–15.

[32] Nathan Ng et al. *Facebook FAIR's WMT19 News Translation Task Submission*. 2019. arXiv: 1907.06616 [cs.CL].

[33] Fahrettin Özbey, Mehmet Güngör, and Yunus Bulut. "On distributions of order statistics for nonidentically distributed variables". In: *Appl. Math* 13.1 (2019), pp. 11–16.

[34] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019*. NeurIPS. 2019, pp. 8024–8035.

[35] Colin Raffel et al. *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. 2019. arXiv: 1910.10683 [cs.LG].

[36] Albert Reuther et al. "Survey and Benchmarking of Machine Learning Accelerators". In: *IEEE High Performance Extreme Computing Conference*. HPEC. 2019, pp. 1–9.

[37] Amit Samanta et al. *No DNN Left Behind: Improving Inference in the Cloud with Multi-Tenancy*. 2019. arXiv: 1901.06887 [cs.DC].

[38] Haichen Shen et al. "Nexus: a GPU cluster engine for accelerating DNN-based video analysis". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP. 2019, pp. 322–337.

[39] Carole-Jean Wu et al. "Machine Learning at Facebook: Understanding Inference at the Edge". In: *IEEE International Symposium on High Performance Computer Architecture*. HPCA. 2019, pp. 331–344.

[40] Xin Xu et al. "Characterization and Prediction of Performance Interference on Mediated Passthrough GPUs for Interference-aware Scheduler". In: *11th USENIX Workshop on Hot Topics in Cloud Computing*. HotCloud. 2019.

[41] Chengliang Zhang et al. "MArk: Exploiting Cloud Services for Cost-Effective,SLO-Aware Machine Learning Inference Serving". In: *USENIX Annual Technical Conference*. ATC. 2019, pp. 1049–1062.

[42] Yizhe Zhang et al. *DialoGPT: Large-Scale Generative Pre-training for Conversational Response Generation*. 2019. arXiv: 1911.00536 [cs.CL].

[43] Ond rej Bojar et al. "Findings of the 2018 conference on machine translation (wmt18)". In: *Proceedings of the Third Conference on Machine Translation*. WMT. Vol. 2. 2018, pp. 272–307.

[44] Stefan Falkner, Aaron Klein, and Frank Hutter. *BOHB: Robust and Efficient Hyperparameter Optimization at Scale*. 2018. arXiv: 1807.01774 [cs.LG].

[45] Kim Hazelwood et al. "Applied machine learning at Facebook: A datacenter infrastructure perspective". In: *IEEE International Symposium on High Performance Computer Architecture*. HPCA. 2018, pp. 620–629.

[46] Zhe Jia et al. *Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking*. 2018. arXiv: 1804.06826 [cs.DC].

[47] Liam Li et al. *Massively Parallel Hyperparameter Tuning*. 2018. arXiv: 1810.05934 [cs.LG].

[48] Richard Liaw et al. *Tune: A Research Platform for Distributed Model Selection and Training*. 2018. arXiv: 1807.05118 [cs.LG].

[49] Jongsoo Park et al. *Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications*. 2018. arXiv: 1811.09886 [cs.LG].

[50] Jun Woo Park et al. "3Sigma: Distribution-Based Cluster Scheduling for Runtime Uncertainty". In: *Proceedings of the 13th European Conference on Computer Systems*. EuroSys. 2018, pp. 1–17.

[51] Yanghua Peng et al. "Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters". In: *Proceedings of the 13th European Conference on Computer Systems*. EuroSys. 2018, 3:1–3:14.

[52] Esteban Real et al. *Regularized Evolution for Image Classifier Architecture Search*. 2018. arXiv: 1802.01548 [cs.NE].

[53] Ziv Scully, Mor Harchol-Balter, and Alan Scheller-Wolf. "SOAP: One Clean Analysis of All Age-Based Scheduling Policies". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2.1 (2018), 16:1–16:30.

[54] Xin Wang et al. "SkipNet: Learning Dynamic Routing in Convolutional Networks". In: *Proceedings of the European Conference on Computer Vision*. ECCV. 2018, pp. 409–424.

[55] Wencong Xiao et al. "Gandiva: Introspective Cluster Scheduling for Deep Learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation*. OSDI. 2018, pp. 595–610.

[56] Yang You et al. "Imagenet training in minutes". In: *Proceedings of the 47th International Conference on Parallel Processing*. ICPP. 2018, pp. 1–10.

[57] Kai Zhang et al. "G-NET: Effective GPU Sharing in NFV Systems". In: *15th USENIX Symposium on Networked Systems Design and Implementation*. NSDI. 2018, pp. 187–200.

[58] Daniel Crankshaw et al. "Clipper: A low-latency online prediction serving system". In: *14th USENIX Symposium on Networked Systems Design and Implementation*. NSDI. 2017, pp. 613–627.

[59] Jeff Dean. "Machine learning for systems and systems for machine learning". In: *Presentation at 2017 Conference on Neural Information Processing Systems*. 2017.

[60] Max Jaderberg et al. *Population Based Training of Neural Networks*. 2017. arXiv: 1711.09846 [cs.LG].

[61] Norman P Jouppi et al. "In-datacenter performance analysis of a tensor processing unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA. 2017, pp. 1–12.

[62] Gábor Melis, Chris Dyer, and Phil Blunsom. *On the State of the Art of Evaluation in Neural Language Models*. 2017. arXiv: 1707.05589 [cs.CL].

[63] Szymon Migacz. "8-bit inference with tensorrt". In: *GTC*. 2017.

[64] Philipp Moritz et al. *Ray: A Distributed Framework for Emerging AI Applications*. 2017. arXiv: 1712.05889 [cs.DC].

[65] Linh Nguyen, Peifeng Yu, and Mosharaf Chowdhury. "No! Not Another Deep Learning Framework". In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. HotOS. 2017, pp. 88–93.

[66] Christopher Olston et al. *Tensorflow-Serving: Flexible, high-performance ml serving*. 2017. arXiv: 1712.06139 [cs.DC].

[67] Hang Qi, Evan R. Sparks, and Ameet Talwalkar. "Paleo: A Performance Model for Deep Neural Networks". In: *5th International Conference on Learning Representations*. ICLR. 2017.

[68] Jeff Rasley et al. "Hyperdrive: Exploring hyperparameters with POP scheduling". In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. Middleware. 2017, pp. 1–13.

[69] Abigail See, Peter J. Liu, and Christopher D. Manning. *Get To The Point: Summarization with Pointer-Generator Networks*. 2017. arXiv: 1704.04368 [cs.CL].

[70] Shaohuai Shi and Xiaowen Chu. *Performance Modeling and Evaluation of Distributed Deep Learning Frameworks on GPUs*. 2017. arXiv: 1711.05979 [cs.DC].

[71] Thomas N Theis and H-S Philip Wong. "The end of moore's law: A new beginning for information technology". In: *Computing in Science & Engineering* 19.2 (2017), pp. 41–50.

[72] Ashish Vaswani et al. "Attention is all you need". In: *Advances in Neural Information Processing Systems 30*. NIPS. 2017, pp. 5998–6008.

[73] Tsung Tai Yeh et al. "Pagoda: Fine-Grained GPU Resource Virtualization for Narrow Tasks". In: *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP. 2017, pp. 221–234.

[74] Haoyu Zhang et al. "Live Video Analytics at Scale with Approximation and Delay-Tolerance". In: *14th USENIX Symposium on Networked Systems Design and Implementation*. NSDI. 2017, pp. 377–392.

[75] Barret Zoph et al. *Learning Transferable Architectures for Scalable Image Recognition*. 2017. arXiv: 1707.07012 [cs.CV].

[76] Martín Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation*. OSDI. 2016, pp. 265–283.

[77] Atul Adya et al. "Slicer: Auto-Sharding for Datacenter Applications". In: *12th USENIX Symposium on Operating Systems Design and Implementation*. OSDI. 2016, pp. 739–753.

[78] Lisha Li et al. *Efficient Hyperparameter Optimization and Infinitely Many Armed Bandits*. 2016. arXiv: 1603.06560 [cs.LG].

[79] B. Shahriari et al. “Taking the Human Out of the Loop: A Review of Bayesian Optimization”. In: *Proceedings of the IEEE* 104.1 (2016), pp. 148–175.

[80] Wenzhe Shi et al. “Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition*. CVPR. 2016, pp. 1874–1883.

[81] Yusuke Suzuki et al. “Towards Multi-tenant GPGPU: Event-driven Programming Model for System-wide Scheduling on Shared GPUs”. In: *MaRS*. 2016.

[82] Maohua Zhu et al. *CNNLab: a Novel Parallel Framework for Neural Networks using GPU and FPGA-a Practical Study with Trade-off Analysis*. 2016. arXiv: 1606.06234 [cs.LG].

[83] Tianqi Chen et al. *MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems*. 2015. arXiv: 1512.01274 [cs.DC].

[84] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. “Speeding Up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curves”. In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*. IJCAI. 2015, pp. 3460–3468.

[85] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[86] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep Learning”. In: *Nature* 521.7553 (2015), pp. 436–444.

[87] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. “Chimera: Collaborative Preemption for Multitasking on a Shared GPU”. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS. 2015, pp. 593–606.

[88] Christian Szegedy et al. *Rethinking the Inception Architecture for Computer Vision*. 2015. arXiv: 1512.00567 [cs.CV].

[89] Chen Zhang et al. “Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks”. In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA. 2015, pp. 161–170.

[90] D. Bernstein. “Containers and Cloud: From LXC to Docker to Kubernetes”. In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84.

[91] Trishul Chilimbi et al. “Project Adam: Building an Efficient and Scalable Deep Learning Training System”. In: *11th USENIX Symposium on Operating Systems Design and Implementation*. OSDI. 2014, pp. 571–582.

[92] Awni Hannun et al. *Deep speech: Scaling up end-to-end speech recognition*. 2014. arXiv: 1412.5567 [cs.CL].

[93] Mu Li et al. “Scaling Distributed Machine Learning with the Parameter Server”. In: *11th USENIX Symposium on Operating Systems Design and Implementation*. OSDI. 2014, pp. 583–598.

[94] Olga Russakovsky et al. *ImageNet Large Scale Visual Recognition Challenge*. 2014. arXiv: 1409.0575 [cs.CV].

[95] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. *Sequence to Sequence Learning with Neural Networks*. 2014. arXiv: 1409.3215 [cs.CL].

[96] James Bergstra, Dan Yamins, and David D Cox. "Hyperopt: A Python Library for Optimizing the Hyperparameters of Machine Learning Algorithms". In: *SCIPY* (2013).

[97] Yun Chi et al. "Distribution-Based Query Scheduling". In: *Proceedings of the VLDB Endowment* 6.9 (2013), pp. 673–684.

[98] Zohar Karnin, Tomer Koren, and Oren Somekh. "Almost Optimal Exploration in Multi-Armed Bandits". In: *Proceedings of the 30th International Conference on Machine Learning*. ICML. 2013, pp. 1238–1246.

[99] Diederik P Kingma and Max Welling. *Auto-encoding variational bayes*. 2013. arXiv: 1312.6114 [stat.ML].

[100] Sreepathi Pai, Matthew J Thazhuthaveetil, and Ramaswamy Govindarajan. "Improving GPGPU concurrency with elastic kernels". In: *Architectural Support for Programming Languages and Operating Systems*. ASPLOS. 2013, pp. 407–418.

[101] Ilya Sutskever et al. "On the importance of initialization and momentum in deep learning". In: *Proceedings of the 30th International Conference on Machine Learning*. ICML. 2013, pp. 1139–1147.

[102] Vinod Kumar Vavilapalli et al. "Apache Hadoop YARN: yet another resource negotiator". In: *ACM Symposium on Cloud Computing*. SoCC. 2013, 5:1–5:16.

[103] James Bergstra and Yoshua Bengio. "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13.10 (2012), pp. 281–305.

[104] Jeffrey Dean et al. "Large Scale Distributed Deep Networks". In: *Advances in Neural Information Processing Systems 25*. NIPS. 2012, pp. 1223–1231.

[105] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems 25*. NIPS. 2012.

[106] James Bergstra et al. "Algorithms for Hyper-Parameter Optimization". In: *Proceedings of the 24th International Conference on Neural Information Processing Systems*. NIPS. 2011, pp. 2546–2554.

[107] James Bergstra et al. "Theano: Deep learning on GPUs with Python". In: *BigLearn, NIPS Workshop*. 2011.

[108] Yun Chi, Hyun Jin Moon, and Hakan Hacigümüş. "iCBS: incremental cost-based scheduling under piecewise linear SLAs". In: *Proceedings of the VLDB Endowment* 4.9 (2011), pp. 563–574.

[109] Cristian Danescu-Niculescu-Mizil and Lillian Lee. *Chameleons in imagined conversations: A new approach to understanding coordination of linguistic style in dialogs*. 2011. arXiv: 1106.3077 [cs.CL].

[110] Benjamin Hindman et al. “Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center”. In: *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*. NSDI. 2011.

[111] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. “Sequential Model-Based Optimization for General Algorithm Configuration”. In: *Learning and Intelligent Optimization - 5th International Conference*. LION. 2011, pp. 507–523.

[112] Vignesh T Ravi et al. “Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework”. In: *Proceedings of the 20th ACM International Symposium on High Performance Distributed Computing*. HPDC. 2011, pp. 217–228.

[113] Alex Krizhevsky, Geoffrey Hinton, et al. “Learning multiple layers of features from tiny images”. In: (2009).

[114] Kevin Irick et al. “A Hardware Efficient Support Vector Machine Architecture for FPGA”. In: *16th IEEE International Symposium on Field-Programmable Custom Computing Machines*. FCCM. 2008, pp. 304–305.

[115] Robert Sedgewick. “Left-leaning red-black trees”. In: *Dagstuhl Workshop on Data Structures*. Vol. 17. 2008.

[116] Frauke Friedrichs and Christian Igel. “Evolutionary tuning of multiple SVM parameters”. In: *Neurocomputing* 64 (2005), pp. 107–117.

[117] Florentina I Popovici and John Wilkes. “Profitable services in an uncertain world”. In: *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. SC. 2005, pp. 36–36.

[118] David E Irwin, Laura E Grit, and Jeffrey S Chase. “Balancing Risk and Reward in a Market-Based Task Service”. In: *13th International Symposium on High-Performance Distributed Computing (HPDC-13 2004)*. HPDC. 2004, pp. 160–169.

[119] Li Zhang and Danilo Ardagna. “SLA based profit optimization in autonomic computing systems”. In: *Service-Oriented Computing - ICSOC 2004*. ICSOC. 2004, pp. 173–182.

[120] Zhen Liu, Mark S Squillante, and Joel L Wolf. “On maximizing service-level-agreement profits”. In: *Proceedings of the 3rd ACM conference on Electronic Commerce*. 2001, pp. 213–223.

[121] David Saad. *On-line learning in neural networks*. Cambridge University Press, 1999.

[122] D. Michie, D.J. Spiegelhalter, and C.C. Taylor. *Machine Learning, Neural and Statistical Classification*. Ellis Horwood, 1994. ISBN: 9780131063600.

[123] Jayant R Haritsa, Michael J Canrey, and Miron Livny. “Value-based scheduling in real-time database systems”. In: *The VLDB Journal* 2.2 (1993), pp. 117–152.

[124] D Hong, Theodore Johnson, and Sharma Chakravarthy. “Real-time transaction scheduling: A cost conscious approach”. In: *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. SIGMOD. Vol. 22. 2. 1993, pp. 197–206.

[125] Jon Michael Peha. “Scheduling and Dropping Algorithms to Support Integrated Services in Packet-Switched Networks”. PhD thesis. Stanford University, 1991.

[126] Dorit S. Hochbaum and Wolfgang Maass. "Approximation Schemes for Covering and Packing Problems in Image Processing and VLSI". In: *Journal of the ACM* 32.1 (1985), pp. 130–136.

[127] Mark H Overmars and Jan Van Leeuwen. "Maintenance of Configurations in the Plane". In: *Journal of Computer and System Sciences* 23.2 (1981), pp. 166–204.

[128] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. ISBN: 0-201-00029-6.

[129] Amazon. *Amazon Elastic Graphics*. URL: https://web.archive.org/web/20220321204629/https://aws.amazon.com/ec2/elastic-graphics/ (visited on 2022-04-30).

[130] Amazon. *Amazon SageMaker*. URL: https://web.archive.org/web/20220418132217/https://aws.amazon.com/sagemaker/ (visited on 2022-04-18).

[131] Chris Duckett. *Baidu creates Kunlun silicon for AI*. URL: https://web.archive.org/web/20210610053157/https://www.zdnet.com/article/baidu-creates-kunlun-silicon-for-ai/ (visited on 2022-04-30).

[132] Cloud Evangelist. *Deep dive into Amazon Inferentia: A custom-built chip to enhance ML and AI*. URL: https://web.archive.org/web/20210226014942/https://www.cloudmanagementinsider.com/amazon-inferentia-for-machine-learning-and-artificial-intelligence/ (visited on 2022-04-30).

[133] GitHub. *GitHub Copilot*. URL: https://web.archive.org/web/20220314145357/https://copilot.github.com/ (visited on 2022-03-14).

[134] Google. *Google Document AI*. URL: https://web.archive.org/web/20220310182856/cloud.google.com/document-ai (visited on 2022-03-14).

[135] Google. *Vertex AI*. URL: https://web.archive.org/web/20220404233921/https://cloud.google.com/vertex-ai (visited on 2022-04-19).

[136] Emmett Kilgariff et al. *NVIDIA Turing Architecture In-Depth*. URL: https://web.archive.org/web/20220414165730/https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/ (visited on 2022-04-14).

[137] Ronny Krashinsky et al. *NVIDIA Ampere Architecture In-Depth*. URL: https://web.archive.org/web/20220402160741/https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/ (visited on 2022-04-02).

[138] George Leopold. *AWS to Offer Nvidia's T4 GPUs for AI Inferencing*. URL: https://web.archive.org/web/20220309000921/https://www.hpcwire.com/2019/03/19/aws-upgrades-its-gpu-backed-ai-inference-platform/ (visited on 2022-04-19).

[139] Microsoft. *Azure Machine Learning*. URL: https://web.archive.org/web/20220418202232/https://azure.microsoft.com/en-us/services/machine-learning/ (visited on 2022-04-19).

[140] NVIDIA. *CUDA Multi-Process Service*. URL: https://web.archive.org/web/20200228183056/https://docs.nvidia.com/deploy/mps/index.html (visited on 2020-02-28).

[141] NVIDIA. *cuDNN*. URL: https://web.archive.org/web/20220518165538/https://developer.nvidia.com/cudnn (visited on 2022-05-18).

[142] NVIDIA. *NVIDIA Multi-Instance GPU*. URL: https://web.archive.org/web/20201004004526/https://www.nvidia.com/en-us/technologies/multi-instance-gpu/ (visited on 2020-10-04).

[143] NVIDIA. *Programming Guide :: CUDA Toolkit Documentation*. URL: https://web.archive.org/web/20200218210646/https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (visited on 2020-02-28).

[144] NVIDIA. *Tnesor Cores*. URL: https://web.archive.org/web/20220514114924/https://www.nvidia.com/en-us/data-center/tensor-cores/ (visited on 2022-05-14).

[145] Tony Peng. *Alibaba's New AI Chip Can Process Nearly 80K Images Per Second*. URL: https://web.archive.org/web/20220427071754/https://medium.com/syncedreview/alibabas-new-ai-chip-can-process-nearly-80k-images-per-second-63412dec22a3 (visited on 2022-04-30).

[146] Paul Teich. *Tearing apart Google's TPU 3.0 AI coprocessor*. URL: https://web.archive.org/web/20220127084628/https://www.nextplatform.com/2018/05/10/tearing-apart-googles-tpu-3-0-ai-coprocessor/ (visited on 2022-04-30).

[147] TensorFlow. *TensorFlow Benchmarks*. URL: https://web.archive.org/web/20200228184228/https://github.com/tensorflow/benchmarks/tree/cnn_tf_v1.5_compatible (visited on 2020-02-28).

APPENDIX A

SALUS

A.1 Workloads

Table A.1 is the full list of workloads and their batch sizes we used in our evaluation.

Figure A.1 is the same peak and average GPU memory usage measurement done in PyTorch, except `overfeat`, which we could not find a working implementation.

Table A.1: *DL models, their types, and the batch sizes we used. Note that the entire model must reside in GPU memory when it is running. This restricts the maximum batch size we can use.*

Model	Type	Batch Sizes
`alexnet`	Classification	25, 50, 100
`googlenet`	Classification	25, 50, 100
`inception3`	Classification	25, 50, 100
`inception4`	Classification	25, 50, 75
`overfeat`	Classification	25, 50, 100
`resnet50`	Classification	25, 50, 75
`resnet101`	Classification	25, 50, 75
`resnet152`	Classification	25, 50, 75
`vgg11`	Classification	25, 50, 100
`vgg16`	Classification	25, 50, 100
`vgg19`	Classification	25, 50, 100
`vae`	Auto Encoder	64, 128, 256
`superres`	Super Resolution	32, 64, 128
`speech`	NLP	25, 50, 75
`seq2seq`	NLP	Small, Medium, Large

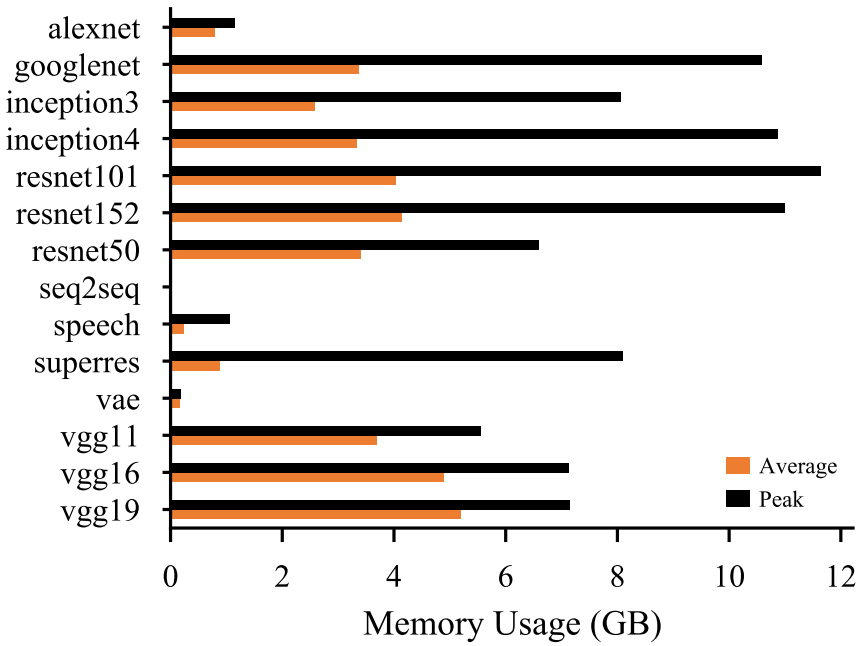


Figure A.1: *Average and peak GPU memory usage per workload, measured in PyTorch and running on NVIDIA P100 with 16 GB memory. The average and peak usage for vae is 156 MB, 185 MB, which are too small to show in the figure.*

APPENDIX B

FLUID

B.1 Analysis of Algorithms

B.1.1 Problem Formulation

Let the TrialGroup scheduling be represented as a strip packing problem $I = \{A, M\}$. Each rectangle a_i in $A = \{a_1, \ldots, a_k\}$ with width and height corresponds to a trial with allocated resources and remaining runtime. It is worth noting that in our problem setting, *each rectangle's width w_i determines its height $h_{i,w}$*. $h_{i,w}$ thus implies the relationship between different resource allocation and its corresponding runtime for this trial. Strips in $M = \{m_1, \ldots, m_n\}$ with identical width 1 and infinite height represents n available identical resources for current hyperparameter tuning jobs.

B.1.2 StaticFluid Algorithm

As shown in Algorithm 3.1, FLUID allocates resources w_i based on each trial's runtime $h_{i,1}$ ratio among trials in the TrialGroup. FLUID then schedules the trials in non-increasing order of resources onto the idle worker set.

$$
\begin{aligned}
w_i^* &= \frac{h_{i,1}}{\sum_j h_{j,1}} n \\
w_i &= \begin{cases} \lfloor w_i^* \rfloor & w_i^* \geq 1 \\ \frac{1}{c} & w_i^* < 1 \end{cases}
\end{aligned}
\tag{B.1}
$$

B.1.3 Theoretical Results

Theorem 1 *In the ideal situation, FluidStatic is a 2-approximation algorithm.*

Proof of Theorem 1.

Given an instance $I = \{A, M\}$ of the strip packing problem and let the optimal solution be $\mathrm{OPT}(I)$. In the ideal case, no overheads will occur, which means the job size (rectangular area) remains the same for one job.

We break down this proof into proofs of two disjoint sub-problems. First we show how this problem can be divided into two sub-problems, and then we prove the approximation factor for each sub-problem separately.

After calculating the resources w_i for each trial a_i by Eq. (B.1), we divide trials into a small set and a large set: one with resources $w_x < 1$ and the other with resources $w_y \geq 1$. The sub-problems are defined as scheduling small set of trials a_x on $n - \sum w_y$ denoted as I_s and scheduling large set of trials a_y on $\sum w_y$ denoted as I_l. We simplify $\mathrm{StaticFluid}(\cdot)$ as $\mathrm{Fluid}(\cdot)$ for convenience in the following proof.

Lemma 1. *By applying our heuristic method, the result makespan of original problem is no worse than the maximum of the result makespan of two sub-problems:*

$$\mathrm{Fluid}(I) \leq \max\{\mathrm{Fluid}_{small}(I_s), \mathrm{Fluid}_{large}(I_l)\}$$

Proof of Lemma 1.

Since the trials in large trial set are ensured with w_y resources, all of them can be scheduled at the beginning. However, small trial set only has $n - \sum w_y$ resources in total, where $n - \sum w_y \leq n - \frac{\sum t_y}{\sum t} n = \frac{\sum t_x}{\sum t} n \leq \sum w_x^* < X$. ($X$ is the size of small trial set). Thus, trials in small trial set may be scheduled when any resource becomes idle. Such idle resources can belong to either small set or large set. If the queued small trial is scheduled on resources belong to the large set, the makespan would be shorter than waiting on small set resources becoming idle. As a result, proving two sub-problems separately is sufficient to bound our original heuristic method.

Lemma 2. *In the ideal situation,*

$$\mathrm{Fluid}_{small}(I_s) < 2H_{small} + \max(h_1)$$

Proof of Lemma 2.

For **small** trial set a_x with $n - \sum w_y$ resources, this sub-problem can be completely modeled as strip packing. According to our heuristic method, each small trial will be allocated with $\frac{1}{c}$ and longest trials are prioritized to be scheduled onto most idle worker. We simplify our method to next-fit decreasing height shelf-based algorithm, which means we schedule trials in non-increasing runtime to an available shelf and add a new shelf if previous shelves are full. Let's denote A_j as the area of j_{th} shelf, $A = \sum A_j$as the sum of shelves' area and H_j as the height of j_{th} shelf. Since the area of rectangle is decreasing with the increase of packing trials if the number of packing trials doesn't exceed the optimal number c, we have $A \leq \sum h_{x,1} < H_{small}(n - \sum w_y)$, where H_{small} is defined as the largest average height of small trial set. We have

$$A_j + A_{j+1} > H_{j+1} \times 1 + H_{j+1}\frac{1}{c} > H_{j+1} \tag{B.2}$$

$$\begin{aligned} \sum H_j &\leq H_1 + 2A = \max(h_{\frac{1}{c}}) + 2A \\ &\leq \max(h_{\frac{1}{c}}) + 2(n - \sum w_y)H_{small} \end{aligned} \tag{B.3}$$

Also, by taking one shelf as a whole trial with one unit resources, this sub-problem can be reduced to shelf-based job scheduling, which is to assign shelves to machines at particular times in order to minimize the makespan. And our heuristic becomes scheduling the shelf in non-increasing runtime order onto the strip with the smallest height. Denote m_k as the strip with the largest height and H_l as the height of the last shelf assigned to m_k. If m_k only has one trial, then this shelf has the

longest runtime which means it must be the optimal solution. If m_k have more than one shelf, we have

$$\begin{aligned}
\text{Fluid}_{small}(I_s) = h(m_k) &\leq \frac{1}{n - \sum w_y}(\sum H_j - H_l) + H_l \\
&= \frac{1}{n - \sum w_y} \sum H_j + (1 - \frac{1}{n - \sum w_y}) H_l \\
&\leq \frac{2H_{small}(n - \sum w_y) + \max(h_{\frac{1}{c}})}{n - \sum w_y} \\
&+ (1 - \frac{1}{n - \sum w_y}) \max(h_{\frac{1}{c}}) \\
&= 2H_{small} + \max(h_{\frac{1}{c}}) \\
&< 2H_{small} + \max(h_1)
\end{aligned} \tag{B.4}$$

Lemma 3. *In the ideal situation,* $\text{Fluid}_{large}(I_l) < 2H$

Proof of Lemma 3.

For **large** trial set a_y with $\sum w_y$ resources, this sub-problem can be completely modeled as strip packing. Based on our heuristic method, each trial a_y will be scheduled on w_y machines.

$$\begin{aligned}
h_{j,w} &= \frac{h_{j,1}}{w} \\
&= \frac{\sum t_{y,1}}{n} \frac{w^*}{w} \\
&= \frac{w^*}{\lfloor w^* \rfloor} \underbrace{\frac{\sum h_{i,1}}{n}}_{H} < 2H
\end{aligned} \tag{B.5}$$

Since $\sum w_y < n$, every trial can get its own resources at the beginning, which result in one-level packing.

$$\text{Fluid}_{large}(I_l) = \max(h_{j,w}) < 2H \tag{B.6}$$

Combine the result of two sub-problems and Lemma 1, we have

$$\begin{aligned}
\text{Fluid}(I) &\leq \max(Fluid_{small}(I_s), \text{Fluid}_{large}(I_l)) \\
&< \max(2H_{small} + \max(h_1), 2H) \\
&\leq \max(2\,\text{OPT}(I) + \max(h_1), 2\,\text{OPT}(I)) \\
&= 2\,\text{OPT}(I)
\end{aligned} \tag{B.7}$$

Theorem 2 *In the real situation,*

$$\text{StaticFluid}(I) < \max(2\,\text{OPT}(I) + \max(h_1)\alpha^{\frac{1}{\alpha-1}}, 2\,\text{OPT}(I)\beta^{\frac{1}{\beta-1}-1})$$

Proof of Theorem 2. In the real situation, we consider the impact of GPU sharing overheads on the problem set up: *a*) for rectangular with fractional width (trial using intra-GPU sharing),

the relationship between height and width is $h_i(w) = h_{i,1}\alpha_i^{\frac{1}{w}-1}, \quad w \in (0,1)$; *b*) for rectangular with integral width (trial using inter-GPU training), the relationship between height and width is $h_i(w) = \frac{h_{i,1}}{w}\beta_i^{w-1}, \quad w \in [1,d]$.

Lemma 4. $\alpha \in \left[1, \frac{c}{c-1}\right)$

Proof of Lemma 4.

Since FLUID ensures the performance of intra-GPU sharing increases by packing with more trials by limiting the number of packing trials under the maximum packing number c, we have $\frac{h_{\frac{1}{a}}}{a} < \frac{h_{\frac{1}{b}}}{b} \leq h_1$ if $1 \leq b < a \leq c$, where a and b are the number of packing trials.

$$\frac{a}{b} > \frac{h_{\frac{1}{a}}}{h_{\frac{1}{b}}} = \alpha^{a-b} \tag{B.8}$$

$$1 \leq \alpha < \frac{c}{c-1} \tag{B.9}$$

In addition, we have $c < \frac{\alpha}{\alpha-1}$.

Lemma 5. *In the real situation,*

$$\text{Fluid}_{small}(I_s) < 2H_{small} + \max(h_1)\alpha^{\frac{1}{\alpha-1}}$$

Proof of Lemma 5.

Combine the result of Eq. (B.4) and Lemma 4, we have

$$\begin{aligned} \text{Fluid}_{small}(I_s) &= 2H_{small} + \max(h_{\frac{1}{c}}) \\ &< 2H_{small} + \max(h_1)\alpha^{\frac{1}{\alpha-1}} \end{aligned} \tag{B.10}$$

Lemma 6. $\beta \in \left[1, \frac{d+1}{d}\right)$

Proof of Lemma 6.

Since FLUID ensures the performance of inter-GPU sharing increases with distributing on more workers by limiting $w < d$, we have $h_1 \leq h_{\frac{1}{a}} < h_{\frac{1}{b}}$ if $1 \leq b < a \leq d$.

$$\frac{h_1\beta^{a-1}}{a} < \frac{h_1\beta^{b-1}}{b} \tag{B.11}$$

$$1 \leq \beta < \frac{a}{b} \leq \frac{d+1}{d} \tag{B.12}$$

In addition, we have $w < \frac{1}{\beta-1}$.

Lemma 7. *In the real situation,*

$$\text{Fluid}_{large}(I_l) < 2H\beta^{\frac{1}{\beta-1}-1}$$

Proof of Lemma 7.

Similar to Eq. (B.5), we consider extra term β on the height of rectangular:

$$
\begin{aligned}
h_{j,w} &= \frac{h_{j,1}\beta_j^{w-1}}{w} \\
&= \frac{\sum t_{y,1}}{n}\frac{w^*}{w}\beta_j^{w-1} \\
&= \frac{w^*}{\lfloor w^* \rfloor}\beta_j^{w-1}\underbrace{\frac{\sum h_{i,1}}{n}}_{H} \\
&< 2H\beta_j^{w-1}
\end{aligned}
\tag{B.13}
$$

Since $\sum w_y < n$, every trial can get its own resources at the beginning, which result in one-level packing.

$$
\begin{aligned}
\mathrm{Fluid}_{large}(I_l) &= \max(h_{j,w}) < 2H\max(\beta_j^{w-1}) \\
&< 2H\beta^{\frac{1}{\beta-1}-1}
\end{aligned}
\tag{B.14}
$$

Combine the result of two sub-problems and Lemma 1, we have

$$
\begin{aligned}
\mathrm{Fluid}(I) &\le \max(\mathrm{Fluid}_{small}(I_s), \mathrm{Fluid}_{large}(I_l)) \\
&< \max(2H_{small} + \max(h_{\frac{1}{c}}), 2H\beta^{\frac{1}{\beta-1}-1})
\end{aligned}
\tag{B.15}
$$

Based on Eq. (B.1), we can derive the relationship between the average height of the large trial set H_{large} and the average height of the whole trial set H.

$$
\frac{S_{large}}{S_{small}} = \frac{\sum w_y^*}{n - w_y^*} \ge \frac{\sum w_y}{n - w_y} \tag{B.16}
$$

$$
H_{large} = \frac{S_{large}}{\sum w_y} \ge \frac{S_{small}}{n - \sum w_y} = H_{small} \tag{B.17}
$$

$$
H_{large} > H > H_{small} \tag{B.18}
$$

where S denotes the total trial size $w \times h$ of a trial set. Since $\mathrm{OPT}(I) \ge H$, we conclude

$$
\mathrm{Fluid}(I) < \max(2\,\mathrm{OPT}(I) + \max(h_1)\alpha^{\frac{1}{\alpha-1}}, 2\,\mathrm{OPT}(I)\beta^{\frac{1}{\beta-1}-1}) \tag{B.19}
$$

APPENDIX C

ORLOJ

C.1 Additional Experiment Results

We report more results here. Table C.1 contains results for experiments where the input request execution time distribution is bimodal. Case ID shows the standard deviation of each modal's normal distribution. In Table C.2, we vary the modality of request execution time distribution from 1 to 8. Table C.3 gives results on static models. Finally, for image classification (Table C.4), chatbot (Table C.5), summarization (Table C.6), and translation (Table C.7) tasks, the first part of the case ID is the model name and second part the dataset name.

C.2 Generalization to Piece-wise Step Cost Functions

We can extend our calculation for Eq. (4.2) from single-step SLO cost function to multiple-step cost functions. For example, consider a multiple-step cost function with three deadlines d_1, d_2 and d_3, and corresponding costs c_1, c_2, c_3. Such a cost function is actually decomposable into the sum of three single-step cost functions: deadline d_1 with cost $c1$, deadline d_2 with cost $c_2 - c_1$, and deadline d_3 with cost $c_3 - c_2$. Therefore, we can compute the priority score for each of the single-step cost function and sum up the results to get the priority score for the multiple-step cost function.

Table C.1: *Evaluation results for cases where request execution time distribution is bimodal.*

Case ID	SLO (× P99)	Finish Rate Clipper	Nexus	Clockwork	ORLOJ
std-0.5	1.5	0.01	0.01	0.44	**0.57**
std-0.5	2	0.01	0.02	0.53	**0.76**
std-0.5	3	0.04	0.05	0.62	**0.96**
std-0.5	4	0.11	0.13	0.61	**0.99**
std-0.5	5	0.20	0.25	0.57	**1.00**
std-1	1.5	0.02	0.02	0.46	**0.60**
std-1	2	0.03	0.03	0.53	**0.76**
std-1	3	0.10	0.10	0.55	**0.97**
std-1	4	0.21	0.19	0.53	**0.99**
std-1	5	0.33	0.34	0.56	**1.00**
std-2	1.5	0.04	0.03	0.43	**0.59**
std-2	2	0.07	0.05	0.51	**0.73**
std-2	3	0.19	0.13	0.55	**0.97**
std-2	4	0.34	0.30	0.50	**1.00**
std-2	5	0.49	0.54	0.59	**1.00**
std-2/0.5	1.5	0.16	0.18	0.52	**0.75**
std-2/0.5	2	0.28	0.32	0.59	**0.83**
std-2/0.5	3	0.61	0.68	0.60	**0.98**
std-2/0.5	4	0.89	0.93	0.59	**0.96**
std-2/0.5	5	0.97	1.00	0.63	**1.00**
std-0.5/2	1.5	0.01	0.01	0.38	**0.40**
std-0.5/2	2	0.01	0.02	0.50	**0.61**
std-0.5/2	3	0.04	0.04	0.57	**0.90**
std-0.5/2	4	0.08	0.09	0.66	**0.97**
std-0.5/2	5	0.15	0.17	0.52	**0.99**

Table C.2: *Evaluation results for cases where we vary the modality of request execution time distribution.*

Case ID	SLO	Finish Rate			
	(× P99)	Clipper	Nexus	Clockwork	ORLOJ
one-modal	1.5	0.03	0.04	**0.48**	0.46
one-modal	2	0.10	0.13	0.55	**0.74**
one-modal	3	0.45	0.54	0.61	**0.98**
one-modal	4	0.73	0.80	0.58	**1.00**
one-modal	5	0.82	0.91	0.60	**1.00**
two-modal	1.5	0.01	0.03	0.45	**0.60**
two-modal	2	0.04	0.03	0.58	**0.75**
two-modal	3	0.13	0.14	0.51	**0.97**
two-modal	4	0.27	0.30	0.56	**1.00**
two-modal	5	0.37	0.44	0.55	**1.00**
three-modal	1.5	0.03	0.04	0.45	**0.59**
three-modal	2	0.05	0.05	0.55	**0.68**
three-modal	3	0.10	0.09	0.61	**0.97**
three-modal	4	0.18	0.23	0.60	**0.99**
three-modal	5	0.35	0.42	0.61	**0.99**
four-modal	1.5	0.03	0.05	0.46	**0.59**
four-modal	2	0.05	0.07	0.56	**0.73**
four-modal	3	0.08	0.12	0.59	**0.94**
four-modal	4	0.10	0.16	0.60	**0.98**
four-modal	5	0.14	0.19	0.60	**1.00**
five-modal	1.5	0.02	0.03	0.47	**0.60**
five-modal	2	0.02	0.03	0.55	**0.73**
five-modal	3	0.03	0.07	0.59	**0.94**
five-modal	4	0.04	0.08	0.57	**0.97**
five-modal	5	0.07	0.08	0.56	**0.99**
six-modal	1.5	0.04	0.04	0.46	**0.58**
six-modal	2	0.03	0.05	0.53	**0.71**
six-modal	3	0.06	0.08	0.53	**0.92**
six-modal	4	0.07	0.09	0.51	**0.97**
six-modal	5	0.08	0.11	0.54	**0.99**
seven-modal	1.5	0.03	0.05	0.43	**0.59**
seven-modal	2	0.04	0.05	0.51	**0.73**
seven-modal	3	0.08	0.09	0.54	**0.93**
seven-modal	4	0.10	0.12	0.52	**0.98**
seven-modal	5	0.12	0.15	0.54	**0.99**
eight-modal	1.5	0.03	0.04	0.33	**0.60**
eight-modal	2	0.05	0.06	0.49	**0.74**
eight-modal	3	0.08	0.09	0.43	**0.93**
eight-modal	4	0.09	0.13	0.52	**0.97**
eight-modal	5	0.11	0.14	0.50	**0.99**

Table C.3: *Evaluation results for static models.*

Case ID	SLO	Finish Rate			
	(× P99)	Clipper	Nexus	Clockwork	ORLOJ
`inception-imagenet`	1.5	0.01	0.01	0.16	**0.41**
`inception-imagenet`	2	0.31	0.30	**0.58**	0.48
`inception-imagenet`	3	0.86	**0.88**	0.84	0.84
`inception-imagenet`	4	0.96	0.97	0.82	**0.99**
`inception-imagenet`	5	0.98	0.98	0.83	**0.99**
`resnet-imagenet`	1.5	0.01	0.00	0.30	**0.42**
`resnet-imagenet`	2	0.15	0.15	**0.59**	0.48
`resnet-imagenet`	3	0.62	0.64	0.77	**0.85**
`resnet-imagenet`	4	0.89	0.91	0.88	**0.98**
`resnet-imagenet`	5	0.95	0.96	0.87	**0.99**

Table C.4: *Evaluation results for image classification tasks.*

Case ID	SLO	Finish Rate			
	(× P99)	Clipper	Nexus	Clockwork	ORLOJ
`rdinet-cifar`	1.5	**0.61**	0.61	0.48	0.49
`rdinet-cifar`	2	0.98	**0.99**	0.47	0.84
`rdinet-cifar`	3	0.99	1.00	0.48	**1.00**
`rdinet-cifar`	4	0.99	1.00	0.48	**1.00**
`rdinet-cifar`	5	0.99	1.00	0.48	**1.00**
`skipnet-imagenet`	1.5	0.00	0.00	0.05	**0.24**
`skipnet-imagenet`	2	0.00	0.03	0.14	**0.62**
`skipnet-imagenet`	3	0.00	0.12	0.21	**0.92**
`skipnet-imagenet`	4	0.00	0.31	0.12	**0.95**
`skipnet-imagenet`	5	0.00	0.48	0.16	**0.89**

Table C.5: *Evaluation results for chatbot tasks.*

Case ID	SLO (× P99)	Finish Rate Clipper	Nexus	Clockwork	ORLOJ
blenderbot-convAI	1.5	0.05	0.06	0.43	**0.45**
blenderbot-convAI	2	0.18	0.21	0.61	**0.65**
blenderbot-convAI	3	0.56	0.63	0.64	**0.74**
blenderbot-convAI	4	0.71	0.81	0.65	**0.81**
blenderbot-convAI	5	0.75	**0.87**	0.65	0.81
blenderbot-cornell	1.5	0.06	0.06	0.43	**0.44**
blenderbot-cornell	2	0.20	0.20	0.60	**0.68**
blenderbot-cornell	3	0.54	0.57	0.62	**0.75**
blenderbot-cornell	4	0.71	0.74	0.64	**0.81**
blenderbot-cornell	5	0.74	0.77	0.64	**0.82**
gpt-convAI	1.5	**0.39**	0.38	0.39	0.36
gpt-convAI	2	0.79	**0.83**	0.57	0.64
gpt-convAI	3	0.91	**0.99**	0.61	0.86
gpt-convAI	4	0.92	**1.00**	0.63	0.97
gpt-convAI	5	0.92	**1.00**	0.61	0.98
gpt-cornell	1.5	0.45	0.44	**0.46**	0.44
gpt-cornell	2	0.84	**0.86**	0.65	0.68
gpt-cornell	3	0.94	**1.00**	0.73	0.97
gpt-cornell	4	0.94	**1.00**	0.73	0.99
gpt-cornell	5	0.95	1.00	0.74	**1.00**

Table C.6: *Evaluation results for summarization tasks.*

Case ID	SLO (× P99)	Finish Rate Clipper	Nexus	Clockwork	ORLOJ
bart-cnn	1.5	0.12	0.11	0.44	**0.46**
bart-cnn	2	0.36	0.36	0.46	**0.71**
bart-cnn	3	0.73	0.73	0.46	**0.97**
bart-cnn	4	0.78	0.79	0.44	**0.99**
bart-cnn	5	0.80	0.81	0.44	**1.00**
t5-cnn	1.5	0.48	0.46	0.47	**0.49**
t5-cnn	2	**0.86**	0.84	0.50	0.74
t5-cnn	3	0.99	1.00	0.50	**1.00**
t5-cnn	4	0.99	1.00	0.52	**1.00**
t5-cnn	5	0.99	1.00	0.51	**1.00**

Table C.7: *Evaluation results for translation tasks.*

Case ID	SLO	Finish Rate			
	(× P99)	Clipper	Nexus	Clockwork	ORLOJ
`fsmt-wmt`	1.5	0.04	0.04	0.45	**0.45**
`fsmt-wmt`	2	0.21	0.23	0.50	**0.59**
`fsmt-wmt`	3	0.63	0.65	0.51	**0.88**
`fsmt-wmt`	4	0.73	0.76	0.53	**0.93**
`fsmt-wmt`	5	0.75	0.79	0.54	**0.95**
`mbart-wmt`	1.5	0.07	0.10	0.38	**0.47**
`mbart-wmt`	2	0.28	0.30	0.36	**0.59**
`mbart-wmt`	3	0.71	0.73	0.35	**0.91**
`mbart-wmt`	4	0.76	0.78	0.36	**0.96**
`mbart-wmt`	5	0.78	0.80	0.35	**0.98**