Multi-tenant Machine Learning Model Serving Systems on GPU Clusters

Lequn Chen

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2024

Reading Committee:
Arvind Krishnamurthy, Chair
Ratul Mahajan
Luis Ceze

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Multi-tenant Machine Learning Model Serving Systems on GPU Clusters

Lequn Chen

Chair of the Supervisory Committee:
Arvind Krishnamurthy
Computer Science & Engineering

In an era where GPUs are both costly and scarce, efficiently serving machine learning models has become a critical challenge. Assuming that serving one model requires k GPUs, serving n models would seemingly require kn GPUs. In the multi-tenant setting, we can pool the whole cluster's GPUs to serve the n models collectively, thus requiring far fewer GPUs. This talk addresses how to optimize cluster-wide GPU utilization in a multi-tenant setting. Key challenges addressed include: (1) batching efficiency under latency constraints, (2) bursty requests and GPU consolidation, (3) GPU cluster auto-scaling.

This dissertation discusses two projects that address the above research problems. The first project, Symphony, focuses on serving DNN models. With a novel Deferred Batch Scheduling algorithm and a system design supporting it, Symphony makes high-quality batching decisions and enables robust auto-scaling. Symphony achieves 6x goodput given the same number of GPUs, saves 60% GPUs when serving the same request rate, and is capable to handle 15 million requests per second. The second project, Punica, creates a new paradigm of serving multiple LoRA fine-tuned large language models at the cost of one. Punica improves throughput by 12x without latency sacrifice.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Table Number Page

ACKNOWLEDGMENTS

This dissertation would not have been possible without the collective support and encouragement of these individuals, and for that, I am truly grateful.

I want to give a huge thanks to my advisor, Arvind. I never have to worry about funding. When other research labs struggle to get even a single consumer-grade GPU, Arvind managed to get me dozens of flagship datacenter cards. I am super grateful that Arvind always allocates time for me despite that his calendar is always filled with meetings. He is willing to dive into the deepest technical details with me while giving farsighted directions. I learned a lot from him about how to formulate research problems and how to find research directions. I sincerely enjoyed working with Arvind over the past six years.

I also want to thank the committee members and mentors. Every time I chat with Ratul, I get enlightened by his soul-searching questions. Chatting with Luis has broadened my understanding of technology's intersection with business and product development. Danyang is a great mentor to me, both in research and in life, and I am grateful for the optimism he has instilled in me.

I am grateful to Weixin for helping with Symphony and to Yongji for helping with Punica. I would like to thank my closest collaborator, Zihao. I learned many important aspects of research from Zihao, including tracking the latest breakthroughs, networking with researchers, and collaboration.

I would like to thank my cohort and lab mates for support and discussion. Thank you, Kevin, Tianyi, Tapan, Liangyu, Xieyang, Jialin, Katie, Samantha, Henry, Pratyush, Chien-Yu, Zhen Zhang, Zihou, Keisuke, Priyal, Chenxingyu, Xiangfeng, Kan, Yile, Yilong, Yifei, Yixin.

I also want to thank my earlier cohort for research advice. Thank you, Haichen, Yuchen, Jialin, Nacho, Shumo.

Lastly, I want to thank my parents for taking care of themselves so that I don't need to worry. I want to thank my friends for companionship and moral support. I want to thank my girlfriend, Hui, for pulling me out of stubbornness and overly rational worldview.

Chapter 1

INTRODUCTION

1.1 Overview

Machine learning (ML) models have shown astonishing improvements over the past decade. Artificial intelligence applications, such as image classification, object tracking, text understanding, and chatbots, are backed by ML models. The demand for model inference serving grows rapidly year by year. [2, 3, 21]

ML models require massive computation and fast memory storage. GPUs are the most common type of hardware accelerator on which models are run. Although powerful, GPUs are expensive and scarce [2, 3, 17], no matter in the public cloud or by purchasing.

A straightforward way to serve models is to allocate dedicated servers and GPUs to each model, as if they were regular web applications. Earlier research [9, 52, 88] adopts this approach. Assuming that each model needs k GPUs, serving n models would seemingly require kn GPUs. Later research [65, 18, 98] began to explore options of multi-tenant model serving. The kn GPUs can thus be pooled together to serve the n models.

There are, however, many key problems to be addressed to enable highly efficient model serving in these settings. These include:

Batching efficiency under latency constraints. GPU consists of many parallel computing units. Batching is an essential technique to improve GPU efficiency, given this parallelism. However, batching increases latency as well. In contrast to model training or batch inference [66], which are both naturally batched, model serving is considered a real-time task. Each model serving job is associated with a latency constraint, typically ranging from 5 to 50 ms. Prior systems either downplay batching [9, 52, 18], produce suboptimal

batches [65], or neglect latency constraints [66].

Handling bursty requests. Single-tenant serving systems provision GPUs based on request load. Prior multi-tenant serving systems periodically calculate a static partition of models to GPUs. Due to the slow response time and lack of work stealing, both have to overprovision by a significant amount to handle bursty requests.

Resource consolidation and autoscaling. Given a fixed number of GPUs, a good model serving system should pack the workload on the smallest possible subset of GPUs instead of evenly distributing loads to all available GPUs. Cluster-wide GPU utilization should indicate the system's overall load. Existing systems fail to meet this trait. Spread loads among GPUs, although giving the illusion of high cluster-wide GPU utilization, implies that GPUs run with suboptimal batch sizes. This mismatch between GPU utilization and system throughput capacity creates difficulties in cooperating with cluster orchestration tools [79, 75, 62, 35] to scale the size of the cluster automatically.

Computation and storage amortization across tenants. Transfer learning and fine-tuning are popular methods to adapt a pre-trained model to a new domain or a new task. The delta required to adapt the model can sometimes be factorized as matrices of much smaller dimensions [30], adding small overhead to computation and storage. Different models in a multi-tenant model serving system might share the same pre-trained model, giving the system a good opportunity to amortize the computation and storage cost across tenants. This is especially relevant in the era of large language models (LLMs), where there are only a few pre-trained models.

State affinity across iterations. Text generation using LLM is an iterative process. Each iteration consumes and updates states that need to persist across iterations. As opposed to deep neural network (DNN) applications like image classification, where each request is

stateless, requests in LLM are stateful. The state introduces an affinity for GPU execution. It also creates pressure on the GPU memory. Model serving systems designed for LLM are therefore required to handle states and manage cluster-wide GPU memory allocation.

1.2 Thesis Contributions

This thesis tackles the following two concrete scenarios of model serving.

Multi-tenant DNN model serving. Multi-tenant DNN model serving involves two main difficulties: First, GPU efficiency must be ensured in light of the batching characteristics of different models while adhering to latency constraints. Second, adjusting to shifts in the computation workload, both in immediate demand spikes and in longer-term resource allocation. In response to these issues, we propose Symphony [6], an optimized DNN serving system with a novel *Deferred Batch Scheduling* algorithm, capable of achieving high batch efficiency while also enabling robust autoscaling. Through extensive experiments, we demonstrate that Symphony outperforms prior systems by up to 6x higher goodput given the same number of GPUs and saves 60% GPUs when serving the same request rate.

Multi-tenant LoRA fine-tuned LLM serving. Low-rank adaptation (LoRA) [30] is an increasingly popular technique for adapting pre-trained models to specific domains. The pretrained model itself requires a huge amount of computation and takes up a large storage and memory space, while the LoRA delta only adds around 30% latency and 1% storage. We present Punica [7], a system that serves multiple LoRA models in a shared GPU cluster. Our system enables a GPU to hold only a single copy of the pre-trained model while serving multiple different LoRA models, significantly reducing GPU memory usage and markedly improving computation efficiency. Our evaluations show that Punica delivers 12x throughput when serving multiple LoRA models compared to state-of-the-art LLM serving systems, with a minimal latency increase of 2 ms per token.

1.3 Published Materials

- Lequn Chen, Weixin Deng, Anirudh Canumalla, Yu Xin, Danyang Zhuo, Matthai Philipose, Arvind Krishnamurthy. **Symphony: Optimized DNN Model Serving using Deferred Batch Scheduling.** *ArXiv abs/2308.07470 (2023). Under review.*

- Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, Arvind Krishnamurthy. **Punica: Multi-Tenant LoRA Serving.** *In Proceedings of the 7th Conference on Machine Learning and Systems (MLSys'24).*

1.4 Thesis Outline

The thesis is organized as follows. Chapter 2 covers the background of DNN models and LLMs and discusses the current state of machine learning serving systems. Chapter 3 studies DNN model serving, presenting Symphony. Chapter 4 studies LoRA fine-tuned LLM serving, presenting Punica. Chapter 5 concludes the report and proposes directions for future work.

Chapter 2

BACKGROUND

A multi-tenant model serving system manages a cluster of GPUs to host multiple models. The system accepts requests for different models, runs inference, and returns the results back to clients. In essence, the system provides a *Model-as-a-Service* API:

```
run(model, input) -> output
```

For example, image classification with ResNet[22]: `run(ResNet, image) -> "cat"`, text generation with GPT[4]: `run(GPT, "The University of Washington is a public") -> "research university in Seattle."`

Model serving is a real-time job. Each request is associated with a latency constraint, usually in the range of 5 milliseconds to 50 milliseconds. The latency constraint is determined by the complexity of the model and the nature of application needs. Unlike throughput-oriented jobs like model training or batch inference, model serving systems need to optimize cluster *goodput*, defined as throughput that satisfies the latency constraint.

Major cloud providers and established startups provide similar APIs as commercial solutions, such as Amazon SageMaker, Google Cloud Vertex AI, Azure Machine Learning, OpenAI API, and Anthropic API. The total market size is estimated to be $7.8B in 2022 [57].

Although the API is simple, the underlying technology is challenging. This section briefly discusses previous research efforts to make model serving efficient.

2.1 Deep neural network model serving

2.1.1 The Batching Effect in DNN Serving

We consider scheduling DNN inferences on accelerators, such as GPUs. DNNs are networks of linear algebra operations called *layers* or *kernels*, where the networks are typically referred to as *models*. Batching groups input matrices into higher-dimensional ones before applying custom "batched" implementations of the GPU kernels. It is one of the most, if not the most, important techniques to improve GPU utilization. This is because accelerators, like GPUs, are designed for highly parallel operations. Batching uses GPU resources (including both processing cores and GPU memory) more effectively and amortizes GPU memory I/O and GPU kernel launch time for individual inference requests. Previous work has found that DNN model execution is highly predictable [18] and the latency profile can be fit with a linear function with high fidelity [65, 92]:

$$\ell(b) = \alpha b + \beta$$

where b is batch size, β is the fixed cost of invoking a model on a batch of requests, and α is the cost of each additional task in the batch. Large batches amortize the fixed cost β.

However, leveraging this batching effect is not simple. Requests must be satisfied within latency bounds set by the application. In the context of datacenter applications (e.g., serving ads or real-time detection tasks), the latency SLOs are a few tens of milliseconds and constrain the extent to which requests can be batched. To quantify a model serving system's performance under latency SLOs, we define its *goodput* as the highest aggregate throughput over all models such that the p99 tail latency of each model is less than their respective latency SLO.

2.1.2 DNN serving systems

Clipper [9] and TensorFlow Serving [52] belong to the first generation of deep neural network (DNN) model serving systems focused on a single model or a few models associated with a single application. These systems start to tackle batching efficiency. However, these systems do not provide benefits for multi-tenant deployment.

Nexus [65] represents the second generation of serving systems, explicitly designed to support multiple models by different applications. Nexus extends efficient batching to multiple models by a bin-packing algorithm. However, Nexus statically partitions models across GPUs, which makes it less responsive to changing workloads.

Clockwork [18] is another multi-model serving system that uses a centralized scheduler design to achieve high responsiveness to workload changes. Clockwork's primary goal is to ensure a predictable execution of model inference, thus it lacks considerations for batching efficiency.

Several other papers explore related topics. Salus [89] emphasizes loading multiple models into the GPU memory, but it mainly focuses on training and only discusses inference on a single-machine setup. INFaaS [59] focuses on model selection, trading off between the inference latency and the DNN model accuracy. Perseus [38] is a good motivational paper for multi-tenant model serving, but they didn't dive deep into scheduling algorithms. Pretzel [37]'s most important technique is sharing layers across models. MArk [91] and Morphling [80] focus more on autoscaling and machine provisioning. Llama [60] and Scrooge [31] focus more on complex query pipelines, whereas we focus on the batching efficiency of individual models. MArk, Morphling, and Llama try to cut down costs on cloud infrastructures by choosing the right VMs/accelerators.

2.2 Large lanuage model serving

Large language models (LLMs) are the dominant models for natural language processing tasks, including text generation, text understanding, classification, and embedding. LLM

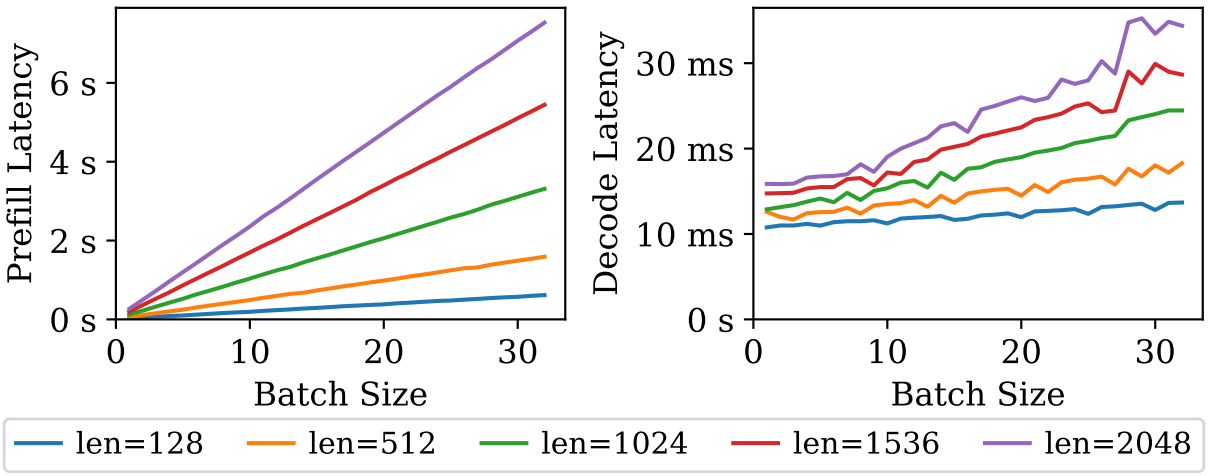


Figure 2.1: Batching effects in Prefill stage and in Decode stage

serving requires system designs different from DNN serving systems because they differ substantially.

2.2.1 Transformer and Text Generation

Transformer-based LLMs operate on a sequence of tokens. A token is roughly ¾ of an English word. An LLM's operation consists of two stages. The *prefill* stage accepts a user prompt and generates a subsequent token and a Key-Value cache (KvCache). The *decode* stage accepts a token and the KvCache, and it then generates one more token and appends a column in the KvCache. The decode stage is an iterative process. The generated token then becomes the input for the next step. This process ends when the end-of-sequence token is generated.

A transformer block contains a self-attention layer and a multilayer perceptron (MLP). Let us assume that the length of the prompt is s and the attention head dimension is d. For the prefill stage, the computation of the self-attention layer is $(s, d) \times (d, s) \times (s, d)$, and the MLP computation is $(s, h) \times (h, h)$. For a decode step, assuming s represents the past sequence length, the computation of the self-attention layer is $(1, d) \times (d, s + 1) \times (s + 1, d)$

and the MLP computation is $(1, h) \times (h, h)$. The decode stage has low GPU utilization because the input is a single vector.

Figure 2.1 shows the latency for the prefill stage and the decode stage for different batch sizes. The GPU's computation capability is fully utilized during the prefill stage. Prefill latency is proportional to batch size. However, this is not the case for the decode stage. Increasing the batch size from 1 to 32, the decode step latency increases from 11ms to 13ms for short sequences and from 17ms to 34ms for longer sequences. This means that batching can improve GPU utilization significantly for the decode stage. Orca [88] leveraged this opportunity to build an efficient LLM serving system. This type of batching is especially important because the decode stage predominately determines the serving latency for long output length responses.

2.2.2 Difference between DNN and LLM

Model size As the name suggests, LLMs are large. DNN model sizes are typically in the range of 10 MB to 500 MB, whereas LLMs range from 10GB to 200GB. The large size of the model imposes many challenges in training and inferencing LLMs on GPUs. Pipeline parallelism [49, 33] and tensor parallelism [50, 39] are widely used paradigms to split a large model onto multiple GPUs. In addition, Quantization [16, 11, 84, 42, 14] and sparsification [24, 87] are techniques to reduce the size of the model while minimizing the loss of accuracy of the model.

Model variants DNN models have many architectures [70, 22, 67, 27], and each architecture has many variants and pre-trained models. Even within a DNN model, DNN layers usually have different hyperparameters and architectures. On the other hand, LLM architectures are dominated by Transformer [78]. Newer models [55, 4, 94, 76, 77, 34] only vary slightly. Transformer layers are all homogeneous. Since training an LLM is prohibitively expensive, there are only a few pre-trained LLMs.

Research method Because DNNs have countless variants, DNN serving systems tend to treat DNN models as a *black box*, only relying on metadata such as latency profile. On the other hand, LLM architectures are simple and standardized, allowing researchers to treat LLM as a *white box*. Hand-optimizing LLM operators and systems opens a bigger and deeper optimization space.

Statefulness DNN models are stateless. Temporary states associated with DNN requests are released as soon as the DNN model finishes execution, which is at the scale of 5 ms – 50 ms. However, LLM execution is usually an iterative process. An LLM request requires multiple invocations of the model. The total elapsed time is usually a few to tens of seconds. In addition, efficient LLM execution requires preserving intermediate computation results of previous iterations. The state is called *KvCache*. The size of KvCache can take hundreds of megabytes to tens of gigabytes.

2.2.3 LLM serving systems

A series of recent work has focused on optimizing LLM inference. Orca [88] proposes batching transformer-based text generation by splitting concatenated batch input at the self-attention operation. vLLM [36] further reduces the memory fragmentation of KvCache by borrowing the idea of virtual pages in operating systems. FlashAttention [10] provides an optimized implementation of self-attention operation by reducing data movement via block-wise computation. FlexGen [66] designed an efficient swapping schedule to maximize throughput on a single GPU while sacrificing latency. AlpaServe [41] enables statistical multiplexing for LLMs. DeepSpeed Inference [1] accelerates cluster-wide LLM serving by packing multiple techniques, including efficient CUDA kernel, pipeline parallelism, hierarchical storage, and improved collective communication. Speculative decoding [40, 48, 5] increases the operational intensity of auto-regressive models by using a lightweight "draft model" to propose candidates for the next k tokens and verifying these k tokens in parallel with large models.

Chapter 3

SYMPHONY: OPTIMIZED DNN SERVING USING DEFERRED BATCH SCHEDULING

We consider the setting of a cloud-scale inference service that supports the scalable execution of a high-volume inference workload involving many deep neural network (DNN) models on a cluster of accelerators (e.g., GPUs). Such an inference service is increasingly needed to consolidate the ML needs of cloud services within a datacenter. An effective inference service has to provide not only high throughput for a diverse set of models but also results within tight latency bounds that are appropriate for user-facing cloud services.

In many aspects, a cloud-scale inference service does not differ much from the traditional cloud service model; it would have to balance incoming requests across backends, adapt to workload changes (i.e., autoscale), and achieve high efficiency without compromising on the desired latency service-level objectives (SLOs). However, scheduling DNN inference requests presents a unique challenge, distinct from traditional request scheduling, primarily due to the *batching effect*. DNNs use linear algebra operators, and batched execution of these operators improves the utilization efficiency of the accelerators. While existing serving systems [9, 52, 65, 18, 92] batch requests, they struggle to achieve optimal batch sizes. The root cause is that the existing serving systems always try to eagerly dispatch a batch of requests whenever there is an idle accelerator to minimize device idle time. This leads to smaller batches and reduced batch efficiency and system throughput. In addition, eager dispatching also creates challenges for cluster horizontal autoscaling because all GPUs are always busy regardless of the workload.

In this chapter, we explore a new approach for request scheduling, called *deferred batch scheduling*. Deferring requests enables the system to: (1) accumulate a larger number of

requests, thereby increasing the batch size and throughput, and (2) consolidate the usage of GPUs so that the number of GPUs used is proportional to the load.

Designing a DNN inference system using this idea raises two key challenges. First, *how long can we afford to defer a batch?* Inference requests have tight SLOs, typically around 20–50 ms. Adding an excessive delay may lead to requests violating their SLOs. The second challenge lies in *facilitating scalable, low-latency coordination among accelerators*. This requires the scheduler to efficiently track the status of each accelerator and dispatch requests in a manner that optimizes system performance and efficiency.

To this end, we build Symphony, a DNN inference system with two key design aspects. First, Symphony defines a *schedulable window* for a request. Symphony dispatches requests only in this window. This window is computed to enhance the batch size and, at the same time, avoid SLO violations. The size of the window depends on the batching effect (i.e., the GPU throughput vs. batch size curve). For example, when the batching effect is strong, Symphony prefers smaller and later schedulable windows to allow the batch to accumulate more requests before being scheduled.

The second aspect is a series of system optimizations to enable scalable, low-latency coordination so that the scheduler can always dispatch requests to the GPU in the schedulable window. These optimizations include (1) separation of request batching and model-GPU matchmaking, (2) low time-complexity and multicore scalable scheduling algorithm, (3) low-latency and predictable control and data planes.

Our evaluation includes extensive comparisons of Symphony with other serving systems and shows that Symphony can improve serving system performance across a wide range of workloads. Symphony further differentiates itself from other systems by exhibiting ideal *load-proportional* autoscaling properties that enable effective adaptation to workload changes. This is because Symphony can keep batch sizes consistently large, while previous systems reduced batch sizes when load is reduced, thus wasting GPU resources. We demonstrate that Symphony can handle 12 million requests per second, coordinate thousands of GPUs, and adapt to workload changes. Compared to state-of-the-art model

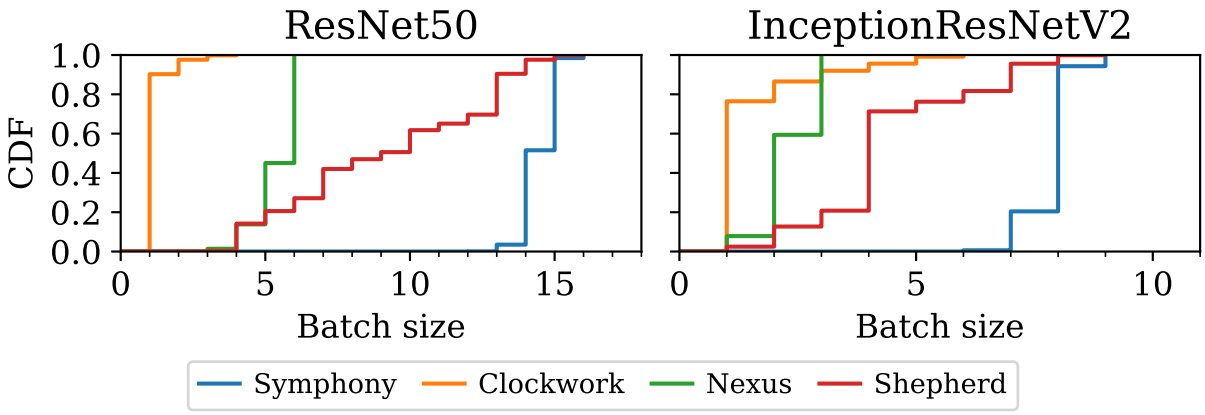


Figure 3.1: Batch size distribution

serving systems, Symphony provides up to 5x gain in goodput when given the same number of GPUs and up to 60% cut in the number of GPUs when serving the same workload.

3.1 Background

3.1.1 Batch Scheduling in Existing Serving Systems

How do existing DNN serving systems make decisions on batching? There are currently three options. The first option is to schedule a batch of requests whenever a GPU is idle, i.e., *eager batching*. The batch size is thus the number of requests accumulated before any GPU becomes idle. The rationale behind eager batching is to keep GPUs busy. Nexus [65] and Clockwork [18] use eager scheduling, and so does Shepherd [92] as a default policy. However, one key problem is that minimizing GPU idle time may lead to low GPU arithmetic intensities and, consequently, lower throughput.

The three systems have different ways of implementing eager scheduling. Scheduling in Nexus happens in three components: the scheduler, frontends, and GPU backends. Every 10 seconds, the Nexus scheduler determines the models and their corresponding expected batch sizes for each GPU. Each Nexus frontend routes a DNN request to one of the GPU backends that run the requested model. A Nexus backend runs scheduler-assigned models

using round-robin. When the actual batch size differs from the scheduler-assigned value, the Nexus backend either runs with the actual smaller batch size or drops excess requests. Clockwork and Shepherd both use a centralized scheduler that monitors all incoming requests and dispatches batches to GPUs. For an incoming request, Clockwork creates a batch candidate for every batch size and maintains these candidates for each GPU. When a GPU becomes free, Clockwork dispatches the batch candidate whose latest executable moment is the earliest and invalidates related candidates for other GPUs. The latest executable moment, as defined by Clockwork, is the latest point at which initiating a batch's execution does not violate its deadline. Shepherd only maintains one outstanding batch candidate for each model and dispatches the candidate with the biggest batch size when a GPU becomes free.

The second approach is *eager batching with preemption*. Preemption is the idea of canceling the dispatched batch by terminating its execution to make room for a different batch. If the new batch has a larger batch size, this can improve system throughput. This is an enhancement to eager batching to avoid small batch sizes. Sherpherd [92] allows a running batch to be preempted by another if the new one is at least 3x the size. When the latency constraint is tight, and the batch size is small (e.g., smaller than 16), such preemption might occur less often. However, preemption can lead to wasted work because the in-flight batch is canceled. Canceling also has its overheads, so this is only worthwhile when the performance benefits can significantly outweigh the drawbacks. This results in systems still issuing small-size batches.

The third approach is *time-out based batch scheduling*. The serving system dispatches the batch to a GPU if the batch size has reached the maximum or the elapsed time since the first request arrival has exceeded the given timeout value. TensorFlow Serving [52] uses the time-out-based approach and uses a constant timeout value and a constant maximum batch size value. Tuning the two parameters introduces operation complexity, and they have to be changed when the workload changes.

To understand the batching quality of DNN model serving systems, we ran a single

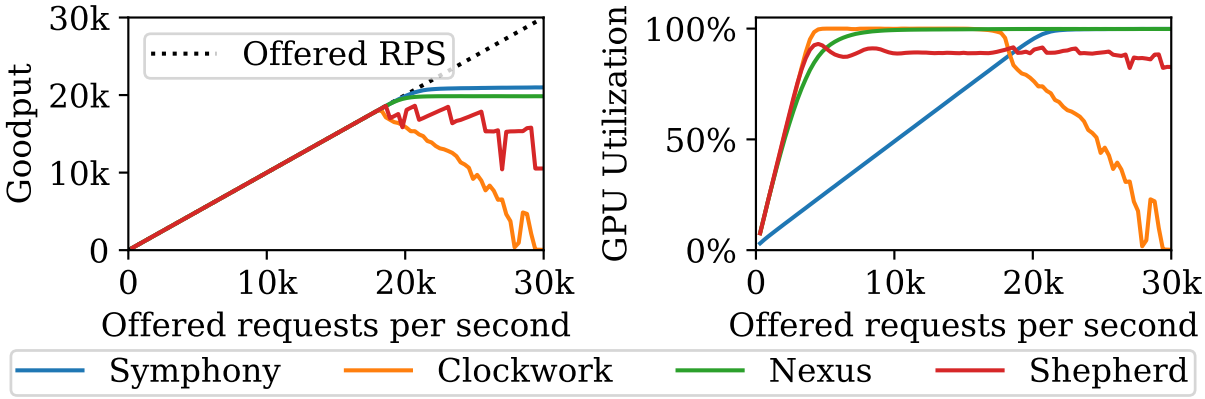


Figure 3.2: (Left) Goodput. (Right) GPU Cluster Utilization

copy of ResNet50 [22] (SLO 25 ms) and InceptionResNetV2 [69] (SLO 70 ms) separately on 8 GPUs using Clockwork, Nexus, Shepherd, and Symphony. Figure 3.1 shows that the median batch sizes for the four systems are 1, 6, 9, 14 on ResNet50, and 1, 2, 4, 8 on InceptionResNetV2. The batch sizes of prior systems are much smaller than those dispatched by Symphony.

3.1.2 Batching and Auto-scaling

Having small batch sizes can also affect the effectiveness of auto-scaling. When batch sizes are small, the GPUs are busy but inefficient. The autoscaling controller (e.g., Kubernetes) will mistakenly believe insufficient GPUs are allocated and ask for more GPUs. More GPUs means GPU idle events are triggered more frequently, subsequently making batch sizes even smaller. An ideal system should achieve a higher batch size and only ask for the required GPU resources it needs. Figure 3.2 shows the behavior of Clockwork, Nexus, and Shepherd when we scale the offered load from 0 to 30K requests per second. When the offered load is only 3K requests per second, all the GPUs are already fully utilized with small batch sizes in these existing model inference systems (Figure 3.2 (Right).) In comparison, our system, Symphony, only needs 20% of the GPUs to serve the exact same

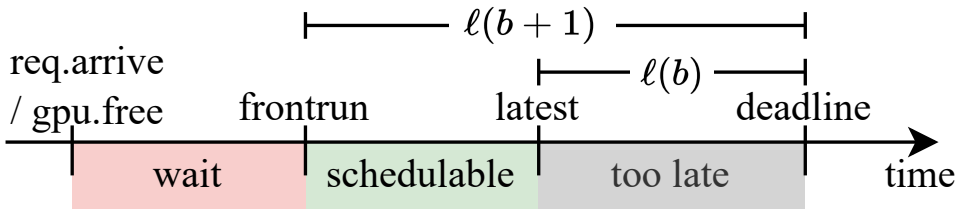


Figure 3.3: Schedulable window

workloads.

3.2 Deferred Batch Scheduling

The key idea of our design to improve the batching efficiency in DNN inference is Deferred Batch Scheduling, focusing on the optimal timing for dispatching a batch of requests to a GPU. In this section, we first discuss the notion of a *schedulable window* and justify why we would want to schedule request batches in this window. We then present the details of our scheduling algorithms.

3.2.1 Schedulable Window

Let's consider the timing of a batch of requests in Figure 3.3. Denote the batch size as b and the deadline as d. Now, we consider the time window for when such a batch can be dispatched to a GPU.

The latest moment is $d-\ell(b)$, because the execution of the batch takes $\ell(b)$ time. Such a late binding guarantees the maximum batching efficiency while still meeting the deadline. However, it might leave GPUs idle for too long.

We propose a new option to decide when a batch becomes schedulable. We define *frontrun* as $d-\ell(b+1)$, denoted as f. We start to allow a batch to be dispatched to a GPU at *frontrun*. Notice that if a new request arrived at time $t^\star$ between *frontrun* and *latest*, adding it to the batch would cause the batch to violate the deadline ($t^\star+\ell(b+1) > f+\ell(b+1)=d$). Hence, dispatching the batch at *frontrun* maintains the same batching

Algorithm 1 Scheduling Algorithm

procedure UPDATECANDIDATE(M) ▷ Model M
  $B \leftarrow$ GETBATCH(Q_M) ▷ Max batch that can fit in deadline
  if $|B| > 0$ **then**
    $d \leftarrow \min\{r.\textit{deadline} : r \in B\}$
    $\textit{exec} \leftarrow \max(\text{NOW}(), d - \ell_M(|B|+1))$
    $\textit{latest} \leftarrow d - \ell_M(|B|)$
    $c_M \leftarrow (B, \textit{exec}, \textit{latest})$
  else $c_M \leftarrow \varnothing$
procedure DISPATCH(M, G) ▷ Model M, GPU G
  UPDATECANDIDATE(M) ▷ Update exec
  Send c_M to G for execution
  $G.\textit{free} \leftarrow c_M.\textit{exec} + \ell_M(|c_M.B|)$
  Remove $c_M.B$ from Q_M
  UPDATECANDIDATE(M) ▷ Prepare next batch
procedure ONNEWREQUEST(r) ▷ Request r for Model M
  Enqueue r to Q_M
  UPDATECANDIDATE(M)
procedure ONMODELTIMER(M) ▷ Trigger at $c_M.\textit{exec}$
  $G^\star \leftarrow \text{argmin}_G\{G.\textit{id} : G.\textit{free} < c_M.\textit{exec}\}$
  if $G^\star \neq \varnothing$ **then** DISPATCH(M, $G^\star$)
procedure ONGPUTIMER(G) ▷ Trigger at $G.\textit{free}$
  $M^\star \leftarrow \text{argmin}_M\{c_M.\textit{latest} : c_M.\textit{exec} < G.\textit{free} < c_M.\textit{latest}\}$
  if $M^\star \neq \varnothing$ **then** DISPATCH($M^\star$,G)

efficiency as dispatching at *latest*. In addition, *frontrun* reduces GPU idle time compared to using *latest*. We explicitly disallow dispatching a batch prior to *frontrun* because we want to accumulate a larger batch. Note that compared to the timeout-based approach, we do not need to specify two parameters. As the batch size grows, the *frontrun* moment is pushed to an earlier time. When the current time is between *frontrun* and *latest*, the batch is dispatched to an idle GPU.

3.2.2 Scheduling Algorithm

We present our scheduling algorithm, as listed in Algorithm 1. We assume that the scheduler has global knowledge of incoming requests and GPU execution states. Section 3.3 will

cover how to achieve it by a centralized scheduler design.

For each model M, the scheduler maintains two states: a request queue Q_M and a candidate batch c_M. For each GPU G, the scheduler keeps track of when it will be free. The candidate batch is updated when a new request arrives or when a previous batch is dispatched to a GPU. The candidate batch contains three components: a set of requests B, the desired time to run (*exec*), and the *latest* time that the candidate remains valid. The desired time to run is either the *frontrun* moment or the current time, whichever is later.

Subroutine GETBATCH returns a maximum set of requests that can finish within the deadline. Typically, the batch-gathering algorithm starts from the head of the request queue and then repeatedly adds the next request to the set if it can still meet the deadline [9, 92]. Alternatively, the batch-gathering algorithm can prematurely drop the head of the queue in order to maintain a larger target batch size [65]. Our algorithm works well with both batch-gathering algorithms.

The Model-GPU matchmaking happens in the following two events. Each model has a timer that triggers at c_M.*exec*. When the candidate is updated, the old timer event is canceled, and the timer is reset to the new expiry. When the model timer triggers, the scheduler tries to find a free GPU (G.*free* $< c_M$.*exec*) and then dispatches the finalized batch to the GPU. If there are multiple GPUs available, the scheduler could pick an arbitrary one. For example, we can pick the one with the smallest identifier (e.g., UUID). This choice allows GPUs with bigger identifiers to remain completely idle when the system load is low. If these GPUs remain idle for an extended period of time, cluster auto-scaling tools can release these GPUs. On the other hand, if there is no free GPU at the moment, the candidate becomes schedulable and might be matched by a GPU later.

Each GPU has a timer that triggers when it finishes execution. When the GPU timer triggers, it considers candidates that are schedulable (c_M.*exec* $< G$.*free*) and still valid (G.*free* $< c_M$.*latest*). When multiple such candidates are present, the algorithm picks the one with the *latest* moment that is closest, which prioritizes the urgency of a batch.

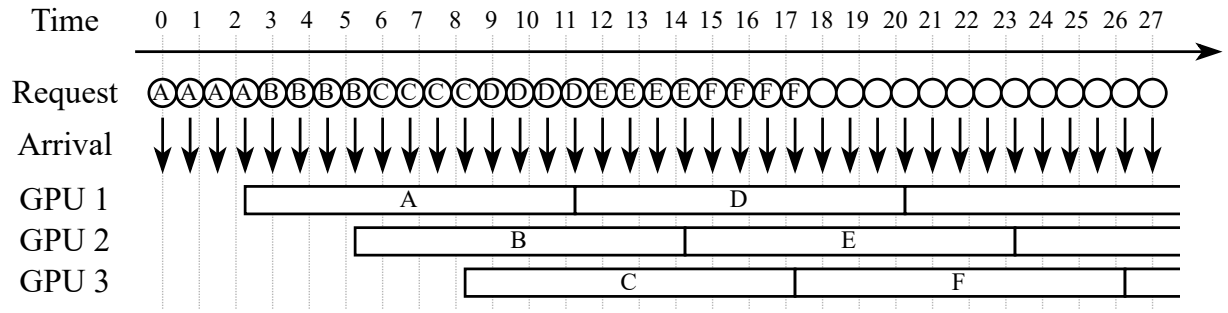
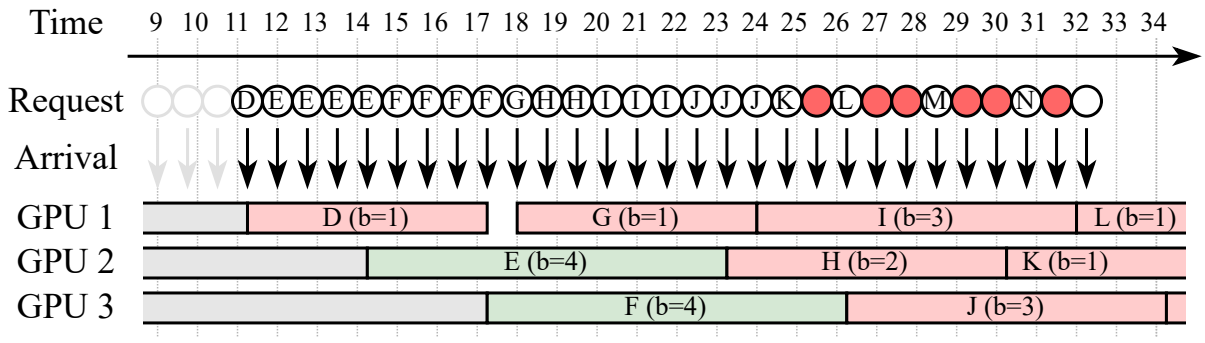


Figure 3.4: Deferred scheduling forms a staggered execution. Circle: request; Letter: batch identifier.

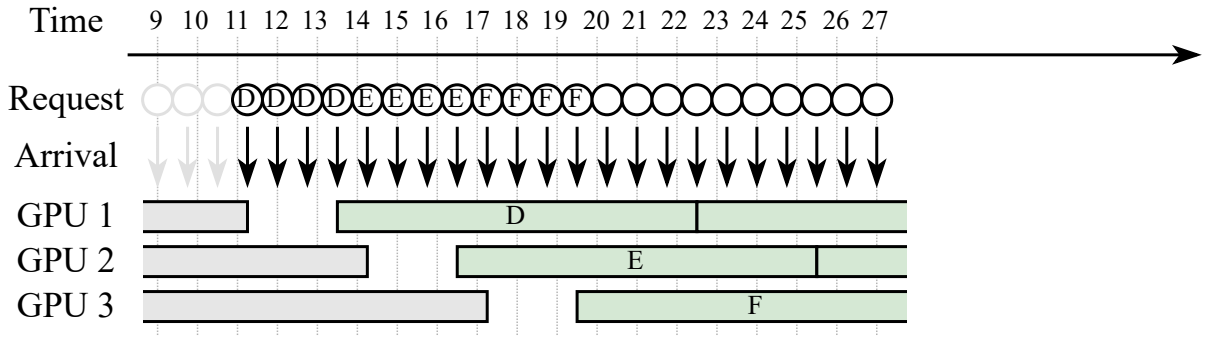
3.2.3 Examples of batch scheduling

We now use two examples to help understand the behavior of deferred batch scheduling and how it could achieve higher goodput and react to request rate fluctuation. We consider a cluster of 3 GPUs and 1 model. The latency profile is $l(b) = b + 5$ time units (i.e., $\alpha = 1$ and $\beta = 5$), and the SLO is 12. Consider a uniform arrival process with request R_i arriving at $t = 0.75 \cdot (i - 1)$. Assume that all GPUs are free at the beginning. At $t = 2.25$, R_4 arrives. The deadline of R_1 is at $t = 12$. The frontrun moment is $t = 12 - \ell(5) = 2$ and the latest is $t = 12 - \ell(4) = 3$. Therefore, the first batch, including the first four requests, is dispatched to a GPU. The rest of the scheduling can be reasoned about in a similar way. Figure 3.4 shows the execution trace of the deferred batch scheduling. The GPUs form a *staggered execution* pattern. In addition, the deferred batch scheduling is able to maintain this pattern.

Note that the staggered execution achieves the highest batch efficiency as it bounds the worst-case queueing delay while dispatching uniformly large batches. We can obtain an upper bound on model goodput using the following analysis. Denote the number of GPUs as N, batch size as b, request rate as λ. In the staggered execution pattern, the worst queuing delay for a request is $\ell(b)/N$. The latency constraint and the throughput



(a) Eager batch scheduling deteriorates



(b) Deferred batch scheduling runs normally

Figure 3.5: Reaction to three missing requests. Red circle: dropped request; Red/Green block: small/large batch.

requirement give the following two equations:

$$(1 + 1/N) \cdot \ell(b) \leq \text{SLO} \tag{3.1}$$

$$N \cdot b/\ell(b) \geq \lambda \tag{3.2}$$

Combining the two equations, we can solve for b and N, which stands for the optimal configuration. We use this to analyze the scheduling quality in Section 3.4.3.

Next, we make a small change to the request arrival process and examine how it affects the staggered execution pattern under different scheduling algorithms. We skip R_{13}, R_{14}, R_{15} while R_{16} and later requests remain unchanged. We study the reactions to the three

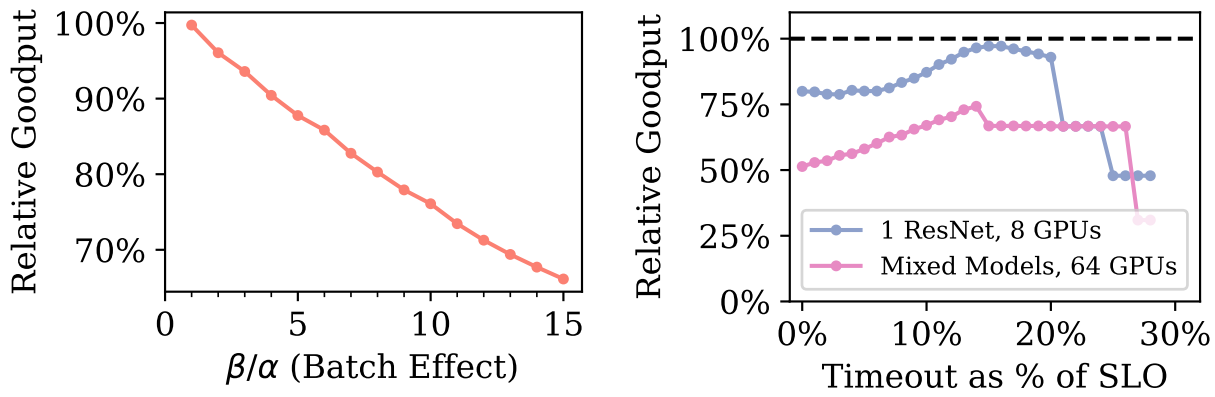
missing requests in deferred and eager scheduling. If the system uses eager scheduling starting from R_{16}, the throughput will deteriorate, as depicted in Figure 3.5a. At $t = 11.25$, R_{16} arrives and GPU 1 finishes the previous batch. Eager scheduling immediately dispatches R_{16} to GPU 1. Then, GPU 2 and GPU 3 each dispatch a batch of size 4 when they finish the last batch. At $t = 18$, R_{25} arrives, and GPU 1 runs with batch size 1 again. At $t = 23.25$, GPU 2 finishes, and the system has accumulated 7 requests (R_{26}–R_{32}). However, the system can only run with batch size 2 because the deadline of R_{26} is at $t = 30.75$. This decreasing batch size eventually causes the system to drop R_{35}, R_{37}, R_{38}, and so on. On the other hand, Figure 3.5b shows that deferred batch scheduling lets GPUs idle for a short period and regains the staggered execution pattern again.

3.2.4 Empirically validating deferred batch scheduling

In this subsection, we present empirical benchmarks against eager batch scheduling and timeout-based batch scheduling to justify our design of deferred batch scheduling. Timeout-based batch scheduling can be implemented by changing Line 5 of Algorithm 1 to *exec* $\leftarrow \max(\text{NOW}(), a + k)$ where the earliest request arrival time $a = \min\{r.\textit{arrival} : r \in B\}$ and k is the constant timeout value. In particular, $k = 0$ is equivalent to eager scheduling. Goodput is found by a binary search over sending a fixed request rate. This subsection assumes that all models are equally popular.

Case Studies

First, we use case studies to deepen our understanding of deferred batch scheduling. In Figure 3.6a, we consider the impact of a model's batch characteristic and compare eager scheduling with deferred scheduling. The figure plots the goodput achieved by eager scheduling as a percentage of deferred scheduling's goodput. We use a synthetic latency profile where α is 1 ms and β varies from 1 ms to 15 ms, and we set latency SLO to $2\ell(8)$. The simulated cluster has 32 GPUs and 10 models of the same latency profile. The request



(a) Compared with Eager (b) Compared with Timeout

Figure 3.6: Case study of Deferred Batch Scheduling

arrival pattern follows Poisson distribution. The figure shows that when $\beta/\alpha = 1$, deferred batch scheduling and eager scheduling have nearly the same goodput. When the model's batching effect is stronger, deferred batch scheduling has a bigger advantage.

In Figure 3.6b, we compare deferred batch scheduling to different timeout settings of timeout-based batch scheduling. We consider two simulation setups: single ResNet50 with 50 ms latency SLO on an 8-GPU cluster and 37 mixed models with various SLO (Table A.2) on a 64-GPU cluster. We set the timeout value of each model to be a varying percentage of the latency SLO. Figure 3.6b shows the relative goodput compared to the deferred batch scheduling. As the timeout value increases, the timeout-based approach gets closer to the deferred one because a larger batch can be accumulated with a longer timeout. However, when the timeout value is too big, we can see that goodput of the timeout-based approach drops because it might let GPUs idle for too long. In the single-model case, the optimal timeout value can reach the same goodput as the deferred batch scheduling. However, in the presence of multiple models, the goodput of the timeout-based batch scheduling is much lower than that of the deferred batch scheduling. A potential way to improve multi-model performance with a timeout-based approach is to carefully tune the timeout value

Model	DenseNet121, InceptionV3, ResNet50V2, VGG16, Xception, Bert
#Models	8, 16, 24, 32, 48, 64
#GPUs:#Models	1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0
SLO (ms)	20, 25, 30, 40, 50
Burstiness	$\Gamma(.1)$, $\Gamma(.2)$, $\Gamma(.3)$, $\Gamma(.5)$, $\Gamma(.7)$, $\Gamma(1.)$

Table 3.1: Configurations for synthetic workloads

for each model, which will introduce significant operational overhead, especially when the set of models served by the system changes.

Synthetic Workload

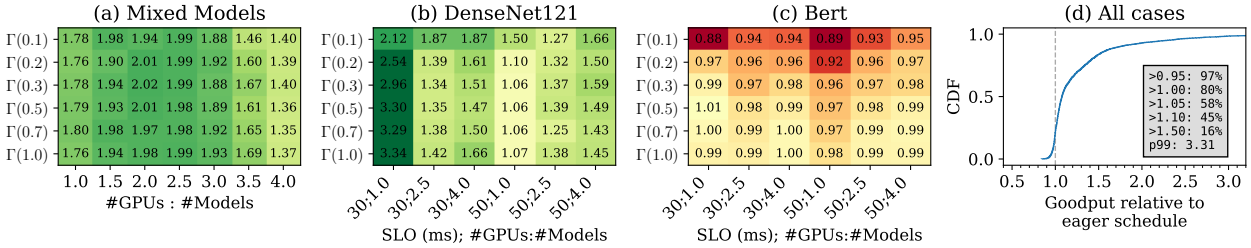


Figure 3.7: Goodput of the deferred batch scheduling relative to the goodput of the eager batch scheduling

We evaluate 5880 different synthetic workloads. We consider homogeneous-model setups where all models have the same latency profile and mixed model setups. Table 3.1 lists options for each configuration dimension. Models listed in Table 3.1 are ordered by descending batching effect (β/α ranging from 9.7 to 0.02). A mixed-model setup contains 35 different models in a single run. Details about these models are in the Appendix A.3. The arrival process follows the Gamma distribution, with the shape varying from 0.1 to 1.0. A smaller Gamma shape value represents a burstier request pattern. Notably, $\Gamma(1.0)$ is mathematically equivalent to the Exponential distribution, meaning the request rate follows the Poisson distribution.

Figure 3.7(a) shows that the deferred batch scheduling works very well when serving a mixture of models, gaining 35% to 102% higher goodput. Figure 3.7(b) examines a few cases with DenseNet121, which has a strong batching effect. Our deferred batch scheduling shows 34% to 234% goodput increase under tighter latency SLO constraint (30ms). For looser latency constraints (50ms), the advantage is less prominent. Figure 3.7(c) examines a few cases with Bert. Bert is a model with a weak batching effect, which should favor eager scheduling. Our deferred batch scheduling achieves a goodput similar to eager scheduling in these extreme cases. Figure 3.7(d) plots all cases. For almost all cases, the deferred batch scheduling is no worse than (>0.95) the eager scheduling. The deferred batch scheduling achieves 50% higher goodput for 16% cases and more than double in extreme cases.

3.2.5 Auto-scaling Support

Batching not only has an effect on efficiency but also on how the system can perform auto-scaling. We desire a certain form of schedule stability that will aid us in performing auto-scaling in response to workload properties. In particular, we want our scheduler to exhibit the following two properties:

- **Goodput Stability:** Assume that the peak goodput of a cluster for a given workload is p. If we were to issue an offered load o higher than p, we would like our scheduler to have a bad rate comparable to $(o - p)/o$. Otherwise, in overloaded conditions, the system exhibits a goodput lower than the peak goodput, indicating a suboptimal congestion response.

- **Load-Proportional GPU Usage:** Given a peak goodput p, if we were to issue an offered load o lower than p, then we would like our scheduling system to have an average GPU idle time fraction comparable to $(p - o)/p$. (GPU idle time fraction is defined as the fraction of time the GPU spends idling without executing any tasks.)

We refer to these two properties together as a desired *flat-top* behavior. The flat-top behavior is desirable because a sudden burst in the aggregate load should not cause an undue increase in bad rate. More importantly, the scheduling system can then monitor the system performance signals of bad rate and GPU idle time to determine whether it needs to allocate or deallocate resources from the cluster. For example,

Allocate GPUs: If the bad rate r is above a threshold, the autoscaler requests $N \cdot r/(1-r)$ additional GPUs, where N is the current number of cluster GPUs.

Deallocate GPUs: If the GPU idle time fraction is f, then it deallocates $N \cdot f$ GPUs.

It is easy to achieve flat-top behavior when serving tasks that do not exhibit batch amortization benefits. However, when there are varying degrees of batch amortization across different candidates with different batch sizes, it is hard to get a robust signal for the autoscaling capability. For example, if a scheduler were to use suboptimal batch sizes, then the GPU idle time would underestimate the true overprovisioning available in the system. Similarly, in overloaded conditions, the bad rate could provide an overestimate of additional resources required to achieve parity.

Figure 3.2 is a measurement of the same workload on the four systems. Symphony and Nexus achieve goodput stability. Shepherd and Clockwork exhibit a substantial loss in goodput as we increase the offered load beyond their capabilities. Symphony's GPU usage is load proportional due to deferred batch scheduling, whereas all three previous systems are eager to occupy all GPUs before reaching their peak goodputs, creating challenges for cluster auto-scaling. We will describe the design of Symphony in Section 3.3, and we will discuss this experiment in more detail in Section 3.4.4.

3.3 System Design and Implementation

In this section, we describe the system design of Symphony.

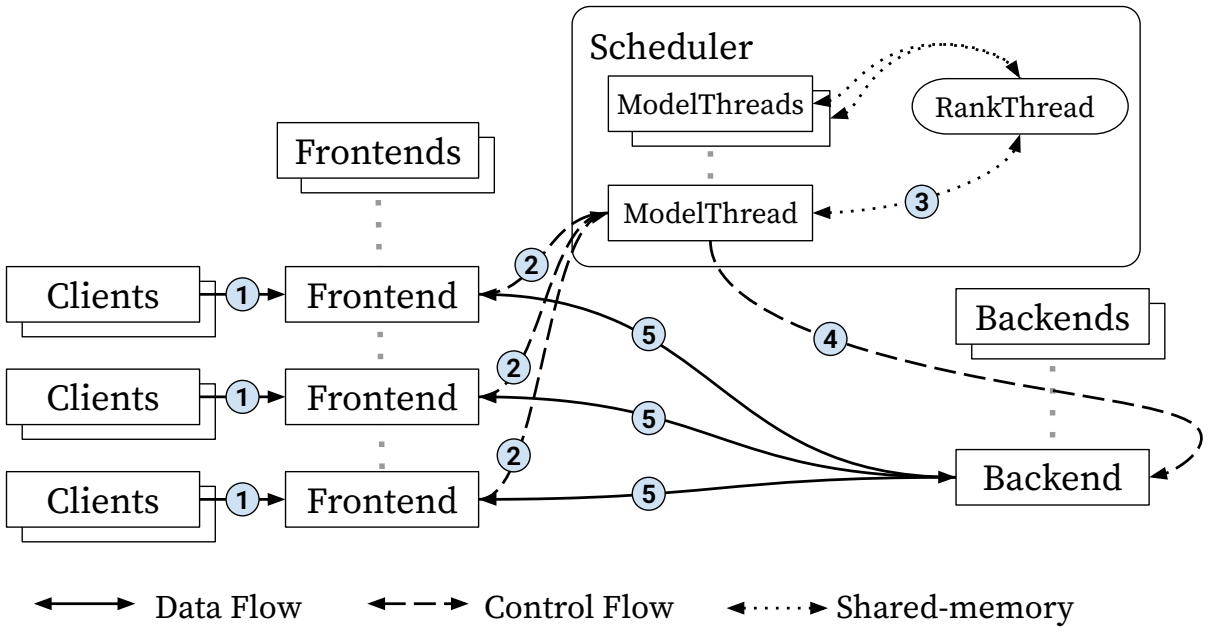


Figure 3.8: Symphony Architecture Design

3.3.1 System architecture

Now we outline the overall system architecture and how the different system components work together (depicted in Figure 3.8). As with other model-serving systems, frontends represent nodes that run application logic and invoke model inferences. Multiple frontends running the same or different applications can invoke inference tasks for the same model. The model inferences are performed in batches on a cluster of backends, each equipped with GPUs.

Frontends accept requests from clients ①, then communicate tasks and their deadlines to the scheduler ②. Tasks are concisely represented using unique task IDs. The scheduler determines the batching of inference tasks and identifies the backend that will execute a given batch ③. Once the scheduler has sent the metadata of the batch to the chosen backend ④, the backend directly pulls the input data for the inference tasks from the frontends ⑤. When the batch finishes, the backend pushes the outputs to the frontends.

Similar to previous work [92], backends are grouped into different sub-clusters, where GPUs in the same sub-cluster have the same set of models loaded at a given time. Periodically, over epochs of a few minutes, Symphony performs the following tasks: allocating and loading models on sub-clusters and auto-scaling to react to workload changes.

Our deferred batch scheduling algorithm requires the scheduler to have global knowledge of the cluster, including request information and GPU work states. Therefore, we use this centralized scheduler design. Centralized scheduling brings three benefits. First, the scheduler can dynamically schedule batches of inference tasks to any one of the available backends to obtain statistical multiplexing benefits, wherein a bursting model can use resources left unused by models operating with lower-than-expected loads; in other words, it can balance out short-term variations in loads across different models. Second, a centralized scheduler can evenly distribute loads across successive dispatches of a model's batches to various GPU backends. Third, a centralized scheduler can reduce the queueing delay for batches by accumulating just one active batch for each model type and funneling it to the next available GPU. This reduced queueing delay translates to being able to execute larger batches within the latency SLO and an increase in GPU efficiency.

3.3.2 Scalability of the centralized scheduler

The centralized scheduler is architected to process millions of requests per second and manage thousands of GPUs.

At the scheduler, when we update a batch candidate, we do not need to access other models' information. On the other hand, GPU availability is global information that all models need to access when trying to bind a candidate to a GPU. For multi-core scalability, we separate the scheduler into the following two different kinds of entities.

A MODELTHREAD accepts incoming requests to a particular model. It accesses only model-local information and updates the candidate. The candidate is then sent to RANK-THREAD.

The RANKTHREAD organizes the global information: GPU free time, each model's timer, and each GPU's timer. Model-GPU matchmaking is triggered by the timers and is handled in the RANKTHREAD. If matchmaking succeeds, RANKTHREAD sends a "GPU Granted" message to the matched MODELTHREAD and marks the GPU as unavailable.

When the corresponding MODELTHREAD receives the "GPU Granted" message, it updates the candidate and sends out the finalized batch to the GPU backend immediately. It also informs the RANKTHREAD about when the GPU will become available. Then it registers with the RANKTHREAD a new candidate, which contains tasks for the next batch.

With this design, the centralized scheduler can utilize multiple independently executing MODELTHREADs spread across multiple CPU cores. The RANKTHREAD, however, is shared by all MODELTHREADs and represents the bottleneck in the system. Note that although a MODELTHREAD needs to keep up with the request line rate, the RANKTHREAD only needs to be fast enough to keep up with the start and the finish events of GPU execution. Since the GPU execution is batched, usually with batch sizes larger than 10, the process rate for RANKTHREAD is an order of magnitude lower.

Besides, we have engineered the system to have a limited amount of ranking logic in the RANKTHREAD, and the single RANKTHREAD is able to support dozens of MODELTHREADs. With the help of advanced data structures [68], the algorithm time complexity on new requests and on batch completion are both $O(\log M + \log G)$ where M is the number of models and G is the number of GPUs.

Combining these techniques, Symphony's centralized scheduler can scale to millions of requests per second on a modern multi-core server. For reference, we put an extended pseudo-code of Algorithm 1 in Appendix A.4, which captures the scalability design.

3.3.3 Fast and predictable networking

Deferred batch scheduling and the centralized scheduler put pressure on networking. (1) Metadata for every request and every batch has to traverse through the scheduler ②④. (2)

The batching algorithm needs to account for the worst latency of the network delay. (3) A backend cannot fetch input data from frontends until it has received the batch metadata from the scheduler.

Therefore, it is critical to have a low-latency, high-bandwidth, and predictable connection among the scheduler, frontends, and backends. We use one-sided RDMA READs initiated from the backend to fetch inference inputs from the frontends. We use two-sided RDMA for control plane messages. Networking behavior is studied in detail in Section 3.4.6.

Besides reading inputs, preprocessing inputs (e.g., decoding images) also takes a significant amount of CPU time. In our design, we choose to do preprocessing at frontends for two reasons. One is that the cluster can easily add CPU computation power by adding more frontends. More importantly, Symphony can then overlap the preprocessing time with the request's queueing time instead of introducing an additional delay at the backends before GPU computations.

3.3.4 Sub-cluster Partitioning

Every a few minutes, we invoke a global partitioning algorithm for solving the following resource allocation problem. Given a cluster and a set of models to be served from the cluster, we subdivide the cluster into multiple sub-clusters and an associated set of models for each sub-cluster. We then ensure that each backend in a sub-cluster is pre-loaded with all models associated with the sub-cluster. This ensures that any of the backends in a sub-cluster can execute any associated models and increases the ability to consolidate the loads across models. Note that a single centralized scheduler is still responsible for scheduling all sub-clusters.

The primary constraint considered in partitioning a set of models across multiple sub-clusters is the total memory size of the models associated with a sub-cluster. This constraint arises because each backend in a sub-cluster needs to load all of the sub-cluster's associ-

ated models into bounded GPU memory. This lets the dispatcher send a model batch to any of the GPUs associated with the sub-cluster and simplifies the centralized scheduling problem. In addition to this primary constraint, we let operators impose an additional optional bound on the combined request rate that a sub-cluster can handle, as a configuration parameter that reflects network capacities and other resource constraints.

To address these constraints, we decide to distribute the workload at the granularity of models. In particular, we partition the set of all models in the system into disjoint sets of models, with each associated with a sub-cluster and managed by a corresponding dispatcher thread on the centralized scheduler. The benefit of such a design is that each dispatcher thread can function independently, so no communications between dispatcher threads are needed.

Because the set of models and their request rates can change over time, we also want to minimize the disruption of such changes on the partition because backends won't be able to process requests during the loading and unloading of models. In order to find a particular partition, we formulated the following mixed integer linear programming (MILP) model. Appendix A.1 provides details on the MILP and an evaluation of its effectiveness.

3.4 Evaluation

We implemented Symphony with 10k lines of C++ code. We use TensorFlow v2.5.0 as the engine to run DNNs. All models are profiled with all different batch sizes to obtain actual execution latency.

We ran most evaluations in a 9-machine cluster. Each machine has two sockets of Intel Xeon E5-2690 v4 CPU (28 physical cores in total) and 64GB memory. The cluster is interconnected by 56Gbps Infiniband with Mellanox ConnectX-3 network interface card. 8 of the machines are equipped with one NVIDIA GeForce 1080Ti GPU on each machine, and the other machine without a GPU is used as the scheduler.

The same cluster is also used to emulate bigger clusters of more GPUs and faster GPU cards. Since the execution time of DNNs on GPU is highly predictable [18], we emulate

the execution by simply introducing a delay at the backend. The introduced delay times are based on model profiles from 1080Ti and A100. We implemented the emulation mechanism for Symphony, Clockwork, Nexus, and Shepherd.

Clockwork uses TVM as the engine to run DNNs. We found that the TVM version shipped with Clockwork runs significantly slower than TensorFlow. To minimize the impact of different DNN execution engines, we always use emulated GPUs based on the profiling results on TensorFlow in Clockwork's end-to-end experiments. Due to the limitation of TVM, Clockwork only supports power-of-two batch sizes. To make comparisons fairer, we added support for arbitrary batch sizes to Clockwork. Clockwork routes all input data through their centralized controller, creating a bandwidth bottleneck. To make our comparisons fair with Clockwork, we omit transferring input data for Clockwork.

Shepherd is not open-sourced. We have communicated with Shepherd's authors to clarify the ambiguity in their pseudocode and implemented its Flex scheduling algorithm.

We used multiple workloads in our evaluations. Workloads differ in the following dimensions: (a) batching profiles of the models, (b) latency SLOs, (c) invocation popularity across models, (d) request arrival patterns, and (e) changes in the average request rate. The arrival of the request follows the Poisson distribution unless specified.

Our evaluation answers the following questions:

- How is the scheduling quality of Symphony under different kinds of workloads?
- How much GPUs cost can Symphony save?
- Where do performance benefits come from?
- Can Symphony work with cluster auto-scaling tools?
- Will the centralized scheduler become a bottleneck?
- What is the impact of RDMA?

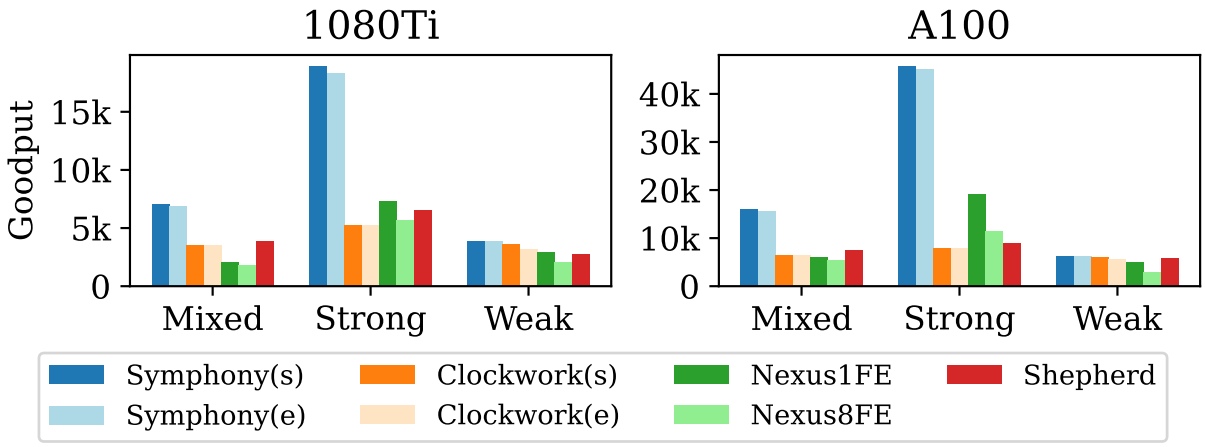


Figure 3.9: Goodput comparison in mixed-model settings

- Does Symphony respond to fast-changing workloads?

3.4.1 End-to-end Goodput

We collected a mixed model zoo consisting of 37 widely-used DNN models, including variants of DenseNet [32], EfficientNet [72, 73], Inception [71, 69], MobileNet [27, 63, 26], NASNet [99], ResNet [22, 23], VGG [67], Xception [8], SSDMobileNet [44], and Bert [15]. These models span a wide range of parameter sizes and execution speeds. Latency SLOs of models vary from 20 to 400 milliseconds. Batching characteristics of each model also differ. The full list of models and latency SLO settings is available in Appendix A.3.

We evaluated Symphony, Clockwork, Nexus, and Shepherd on the mixed model zoo with 64 emulated GPUs in two separate clusters with NVIDIA 1080Ti and A100, respectively. Because both Symphony and Clockwork take the centralized approach, we ran both systems with both scheduler-only (s) and end-to-end (e) configurations. We also run Shepherd in the scheduler-only setting. Scheduler-only runs only the scheduler, and the load generator sends requests in the same process. The end-to-end configuration runs the scheduler, frontends, backends, and load generators on separate machines. Since Nexus

frontends take part in scheduling, we run Nexus with a single frontend and eight frontends, respectively, to observe the overhead of distributed scheduling.

Based on batching characteristics, we run the experiment in three settings: *Mixed* runs all models listed in the model zoo; *Strong* runs models whose $\beta/\alpha > 2$, i.e., models that have strong batching effect; *Weak* runs models whose $\beta/\alpha < 2$, i.e., models that do not benefit much from batching.

The goodput of each system is shown in Figure 3.9. Comparing the goodput between (s) and (e), we can see that both Symphony's and Clockwork's schedulers have good control over the cluster and are able to predict the end-to-end performance. Comparing Nexus1FE and Nexus8FE, we observe 11% to 45% goodput loss from distributed scheduling.

When running all models, Symphony shows 2.0-2.4x the goodput of the baselines. When running models with strong batch effects, Symphony shows 3.5x the goodput of the baseline systems for 1080Ti and 5.7x for A100, indicating that Symphony achieves better scheduling quality. It is worth pointing out that the advantage of Symphony is less prominent when running models with little batching effect. Symphony goodput is 23% and 10% higher in the 1080Ti cluster and the A100 cluster, respectively, in the *Weak* setup.

3.4.2 Savings in GPUs

We compare the four systems on the minimum number of GPUs required for serving 15k RPS as depicted in Figure 3.10. We examine the following two workloads: a single ResNet50 model with 25 ms latency SLO, and all 37 mixed models with various latency SLO in Table A.2. In this experiment, we use an emulated A100 cluster. In the single-model case, Symphony saves 2 to 6 GPUs compared with Shepherd and Nexus. Clockwork requires more than double the GPUs of the other three systems since it does not explicitly optimize for batching efficiency. Symphony's resource consolidation effect is larger for the mixed-model cases because it is more challenging for baseline systems to achieve

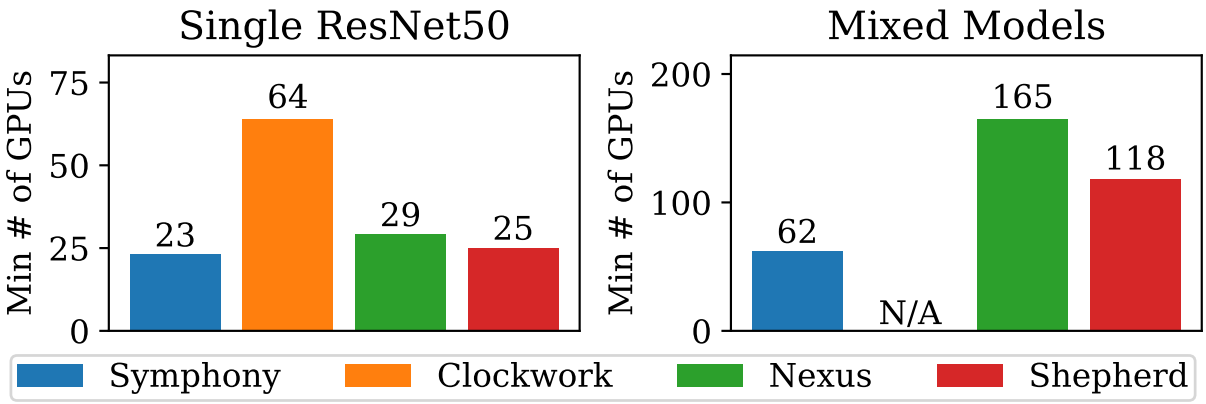


Figure 3.10: Minimum number of GPUs required for 15k RPS

large batch sizes when serving multiple models. To deliver the same goodput, Nexus and Shepherd need 166% and 90% more GPUs. Clockwork cannot deliver the desired goodput because its scheduling algorithm has a very high time complexity.

3.4.3 Scheduling Quality

We test Symphony's scheduling quality under various workload characteristics. Then, using two setups as examples, we take a closer examination to understand why Symphony achieves better scheduling quality.

Varying the workload characteristics. We conducted experiments with changes in three dimensions of workloads: latency SLO, model popularity, and request arrival. The experiments use 20 models whose batching profile is similar to ResNet50; this would represent specialized variants of the model for different applications [56, 28, 53]. All models are set to the same latency SLO, varying from 15ms to 100ms across data points. Popularity among models has two options: equally popular or Zipfian distribution with a shape of 0.9. The arrival process is either a Poisson or Gamma distribution with shape 0.05 (i.e., bursty request rate). 32 emulated GPUs are used in the experiments.

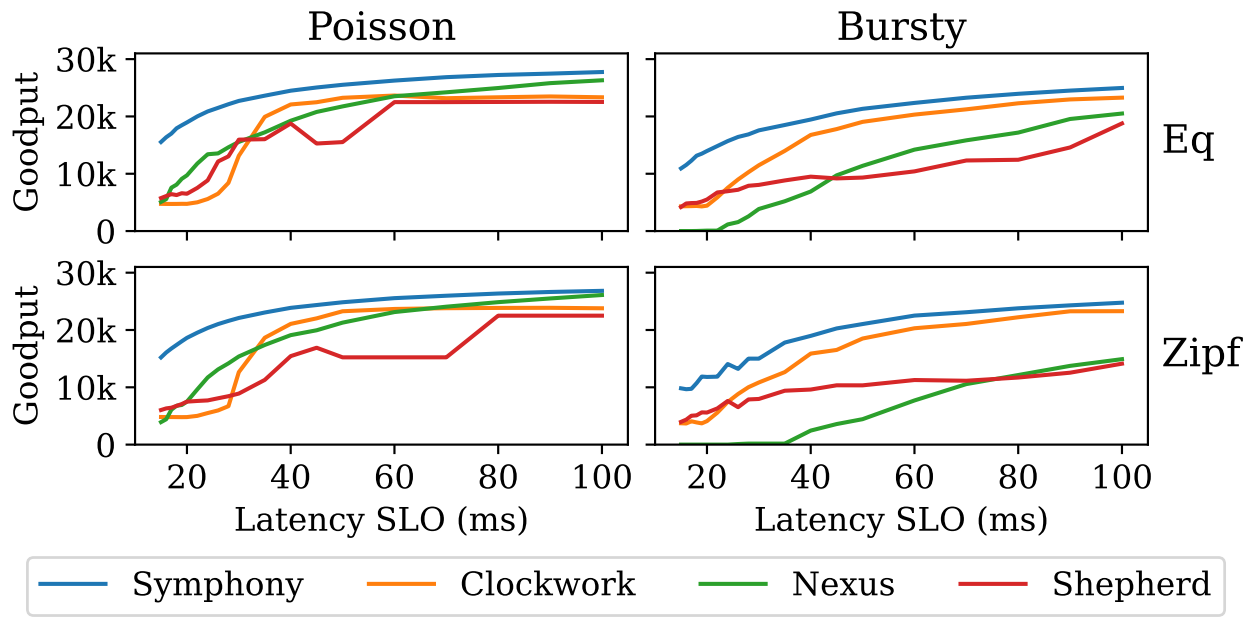


Figure 3.11: Effect of workload characteristics on goodput

GPUs	Model	α(ms)	β(ms)	SLO	No Coordination		Staggered		Measured Goodput (r/s)			
					BS	Tpt	BS	Tpt	Symphony	Clockwork	Nexus	Shepherd
8	[22]	1.053	5.072	25ms	7	4501 r/s	16	5839 r/s	5264	1358	4027	4445
8	[69]	5.090	18.368	70ms	3	713 r/s	8	1083 r/s	926	458	618	778

Table 3.2: Theoretical analysis of batching and empirical measurement of goodput. (BS: Batch Size, Tpt: Throughput)

Figure 3.11 shows the goodput of the different systems under the various combinations of the experimental setups. Our findings include: 1) Symphony provides significant benefits in the tight-SLO cases across all four setups, and this is because deferred batch scheduling optimizes batch efficiency. 2) Nexus does not work well with bursty workloads because of the static partitioning of models to backends; it does not enjoy the statistical multiplexing benefits of the other systems. 3) Loose-SLO cases are less challenging, with all systems producing relatively good schedules. This is because the batch sizes are large; therefore, the marginal goodput improvements from larger batch sizes are small.

Batch Size. Section 3.2.3 discussed the staggered execution pattern. The worst queuing delay in staggered execution is $\ell(b)/N$. For Nexus, due to its distributed design, the worst queuing delay for a request is $\ell(b)$. Assuming requests arrive at a constant gap, the analytical solution of batch size can be calculated as $\lfloor(\text{SLO}/2 - \beta)/\alpha\rfloor$ and $\lfloor(\text{SLO}/(1+1/N) - \beta)/\alpha\rfloor$, respectively. Given batch size b, the throughput of N GPUs can be calculated as $Nb/(\alpha b+\beta)$.

To understand how batch size affects goodput and to study how close is the batching behavior to the ideal staggered execution, we run a single copy of ResNet50 [22] and InceptionResNetV2 [69] separately with 8 GPUs on Symphony, Clockwork, and Nexus. Requests arrive in a Poisson distribution. The details of the models and the goodput on each system are listed in Table 3.2, and the distribution of batch size is shown in Figure 3.1.

Table 3.2 shows the analytical batch size and the throughputs for the different scheduling approaches. The data shows that the staggered execution can run twice the batch size and achieve $30\% \sim 50\%$ higher throughput compared to execution without coordination. The empirical measurement shows that the goodput of Symphony is close to the analytical throughput of staggered execution, and the goodput of Nexus is close to the analytical throughput of execution without coordination. The difference between the real system and analytical calculation comes from differences in request arrival patterns and delays in real systems.

Figure 3.1 shows the batch size distribution. When running ResNet50, the majority of requests run with batch sizes greater than 14 and 6 for Symphony and Nexus, respectively. When running InceptionResNetV2, most batch sizes are greater than 8 and 3, respectively. The observed batch sizes is close to the analytical solutions. Thus, Symphony's scheduler generates batches whose sizes are close to that of the optimal staggered execution and thereby achieves high goodput. Clockwork has an extremely low batch size since its scheduling algorithm does not consider batching efficiency. Shepherd runs larger batches than Nexus and achieves higher throughput thanks to its centralized scheduler and preemption. Compared to Symphony, Shepherd's batch size distribution is more spread out, indicating that Shepherd cannot form the ideal staggered execution due to its eager

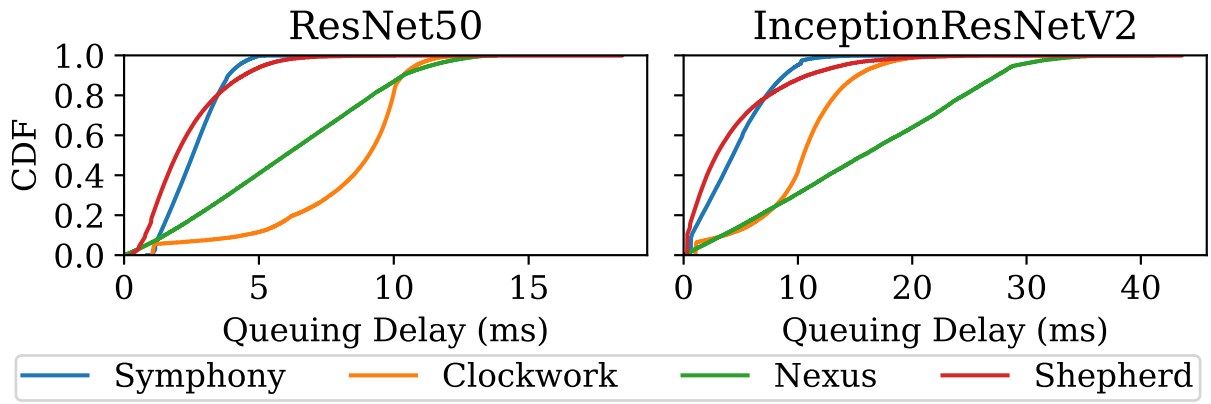


Figure 3.12: Queuing Delay

scheduling, hence we can see that Shepherd achieves a much lower goodput in Table 3.2.

Queuing Delay. Figure 3.12 shows the request queuing delay, which is defined as the duration starting from the system receiving the request to a GPU initiating a batch containing the given request. Symphony's queuing delay is 2x to 3x shorter than Nexus and Clockwork, allowing more of the SLO budget to be spent on execution. The longest queuing delay in Nexus is around half the latency SLO due to its lack of coordination. Although Clockwork uses a centralized scheduler, its longest queuing delay does not improve over Nexus in the ResNet50 case. Shepherd's queuing delay is comparable to Symphony, but it does not translate to larger batch sizes, as analyzed in the batch size evaluation.

3.4.4 Auto Scaling

We study whether our system and baseline systems can provide robust signals to cluster auto-scaling tools. We use 10 ResNet models with 100ms latency SLO with 24 emulated GPUs in the experiment.

Stability. Ideally, a model serving system should provide a stable goodput even when the offered request rate exceeds the cluster's maximum capacity. Figure 3.2 (Left) shows that

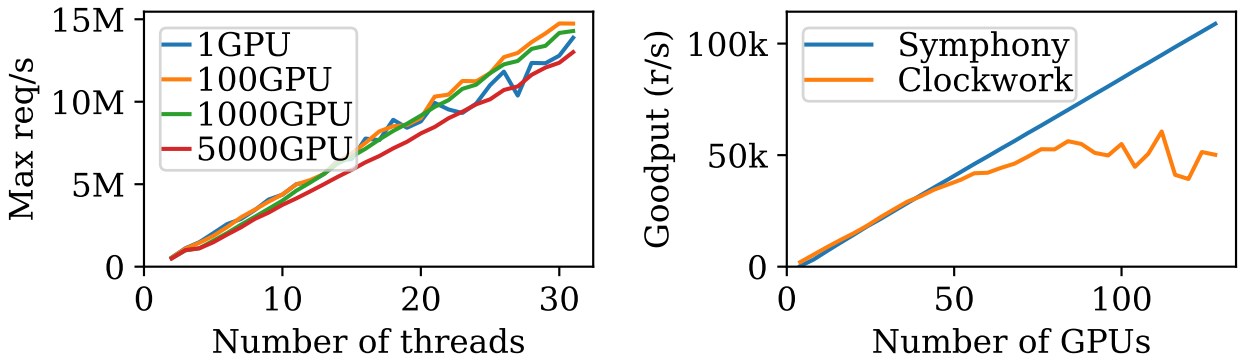


Figure 3.13: (Left) Symphony scheduler multicore scalabilty. (Right) Goodput varying the number of GPUs.

both Symphony and Nexus have flat-top behavior because both systems try to maintain a desirable batch size. Shepherd's goodput shows some fluctuation when overloaded. As shown in the graph, Clockwork's goodput starts to degrade as soon as the system is overloaded. This unstable goodput makes it hard to perform autoscaling.

GPU Cluster Utilization. Figure 3.2 (Right) measures cluster utilization as the average fraction of time that GPUs are busy. Clockwork, Nexus, and Shepherd reach full GPU busy levels before reaching their peak goodputs, indicating that both systems run with suboptimal batches. Symphony's GPU utilization gradually increases and reaches full GPU busy level at near-peak goodput, which means that Symphony can efficiently consolidate GPU usage. Hence, GPU utilization of Symphony is a good signal to cluster autoscaling tools.

3.4.5 Scalability

Since the centralized scheduler is involved in all requests, we want to ensure it can keep up with the request rates. Figure 3.13 (Left) measures the maximum request rate the Symphony scheduler can handle on a 32-core AMD EPYC 7502 CPU. We run this benchmark with the scheduler alone, without sending network messages or running GPUs. Requests

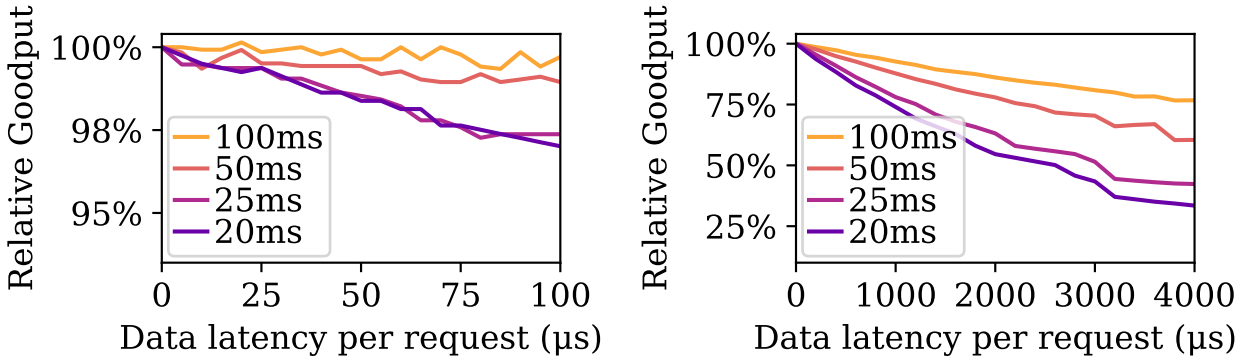


Figure 3.14: Effect of network latency on model serving goodput. (Left) within RDMA range. (Right) within TCP range.

and GPUs in these benchmarks are in-process objects used by the scheduler. Since there is no information exchange between ModelThreads, we observe that the throughput increases linearly with the number of threads. The plot also proves that the single-threaded RankThread is not a bottleneck. Although ModelThreads produces Candidates at line rate, RankThread only needs to pick up the latest candidate from ModelThreads. Therefore, a single RankThread can support dozens of ModelThreads. The plot also shows that adding more GPUs does not affect the scalability, as we use advanced data structures to manage states.

Figure 3.13 (Right) serves 20 equally popular ResNet-like models with 100ms SLO. Symphony's goodput increases linearly with the number of emulated GPUs. On the other hand, Clockwork was not designed for multicore scalability, and its use of locks on critical paths limits the overall throughput. We omit the comparison with Shepherd due to the lack of a canonically optimized implementation.

3.4.6 Impact of network

This subsection quantifies the benefits of using RDMA-based communications in Symphony. The network-level benchmarks on TCP and RDMA are in Appendix A.2. The lowest latency and 99.99-th tail in the RDMA network are close, at 24μs and 33μs respectively. The

median latency of TCP is 3034μs, and its 99.99-th tail is 12x the median. To study how network latency affects the model serving goodput, we run a workload consisting of 20 evenly popular models of similar batching profiles on an emulated 32-GPU cluster with four settings of latency SLO: 20ms, 25ms, 50ms, and 100ms. Figure 3.14 (Left) shows that the network latency within the RDMA range only marginally reduces goodput. Figure 3.14 (Right) shows that network latency within the TCP tail latency range significantly hurts goodput by up to 70%. We observe that the tighter the latency SLO is, the more severely the goodput is impacted by network latency. This is because a backend cannot fetch input data from frontends until the batch metadata is sent by the scheduler. The scheduler always uses the high percentile bound of network latency as the network delay estimation and would have to make earlier dispatch decisions. Therefore, unpredictable and high latencies lead to a significant decrease in the goodput.

3.4.7 Large cluster and changing workload

Lastly, we evaluate our system in a large cluster setup with a changing workload. The emulated cluster consists of 512 GPUs. The workload consists of 24 models with different batching characteristics and different SLOs. The request rate for each model is synthesized from 150 hours of videos.

Figure 3.15 shows the result. Color in the first subplot represents the goodput for each model. Grayscale in the first subplot shows the unsatisfied requests for each model. The stacked graph shows the overall request rate per second in total. The second subplot shows the number of GPUs used in the system. Symphony can consolidate GPU usage when the cluster is underloaded. The third subplot shows Symphony's advice to the cluster's autoscaling system regarding adding or removing GPUs. The fourth subplot shows the bad rate. Symphony can maintain a low bad rate when the cluster is not overloaded.

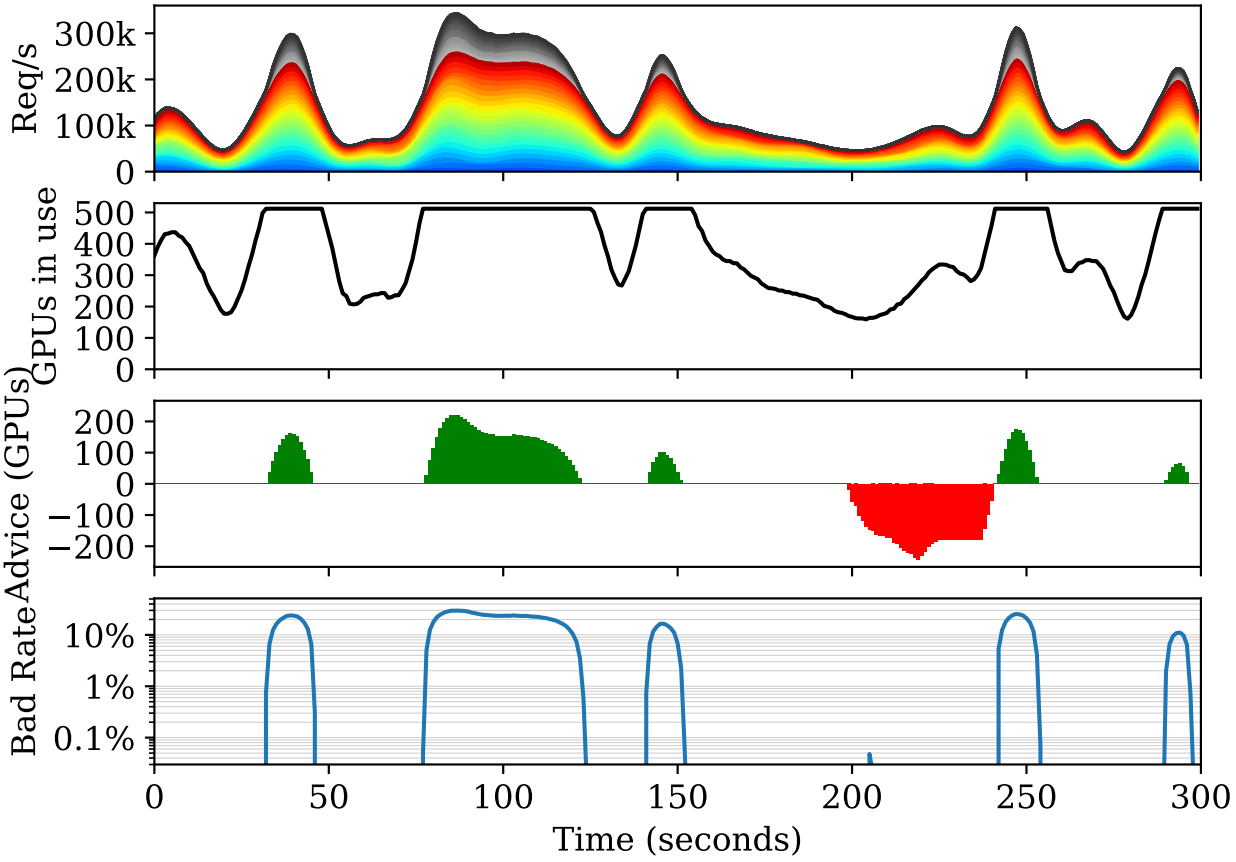


Figure 3.15: A changing workload on a 512-GPU cluster

3.5 Discussion

Failure Recovery The centralized scheduler is a single point of failure. We provide a millisecond-scale fast recovery design. We set aside a few hot-spare schedulers in the cluster whose only task is to have frequent heartbeat exchanges with the primary. With the help of RDMA, a failure can be detected in a few hundred microseconds. One of the hot spares can quickly switch to the primary. The new scheduler can query backends to populate GPU work states and start to accept new requests immediately. Unfinished requests during failure are inevitably lost and can be retried by clients.

Generalization to Large Language Models Our system is optimized for DNN inference with stringent latency SLOs. For instance, the machine learning models we focus on, such as ResNet, are widely used in computer vision and video analytics. In these domains, ML inference usually requires tight SLOs for timely object detection and real-time vehicle tracking purposes. Large language models (LLMs) [78] are an emerging type of DNN with a very different performance requirement. For improving LLM inference, batching is also important [88, 36] but must be considered with other factors, including how to improve kernel operator efficiency [10, 42, 16, 84, 12], how to partition model weights to accelerators through model parallelism [97, 1, 50], and how to maintain the KV cache across multiple model invocations. The performance requirements of LLMs are also quite different. An LLM's per-token latency is usually around dozens of ms, and the token decoding latency has other factors including the window length. How to extend our deferred batch scheduling idea to LLMs is an interesting future work.

3.6 Summary

We present the design of Symphony, an optimized DNN model serving system for GPU clusters. Symphony uses a novel *deferred batch scheduling* algorithm that improves batch sizes while meeting latency SLO. The number of GPUs that Symphony uses is *proportional to the load*, enabling easy autoscaling. To support the algorithm, we craft a low-latency, multicore-scalable centralized scheduler capable of handling millions of requests per second and managing thousands of GPUs. Compared to prior systems, Symphony achieves 5x higher goodput when given the same number of GPUs and saves 60% GPUs when serving the same workload. Symphony's source code is available at https://github.com/abcdabcd987/nexuslb.

Chapter 4

PUNICA: MULTI-TENANT LORA SERVING

Low-rank adaptation (LoRA) [30] is becoming increasingly popular in specializing pre-trained large language models (LLMs) to domain-specific tasks with minimal training data. LoRA retains the weights of the pre-trained model and introduces trainable rank decomposition matrices to each layer of the Transformer architecture, significantly reducing the number of trainable parameters and allowing tenants to train different LoRA models at a low cost. LoRA has been integrated into many popular fine-tuning frameworks [47]. Consequently, ML providers have to serve a large number of specialized LoRA models simultaneously for their tenants' needs.

Simply serving LoRA models as if they were independently trained from scratch wastes GPU resources. Assuming we need k GPUs to serve each LoRA model, serving n different LoRA models would seemingly require $k \times n$ GPUs. This straightforward approach overlooks the potential weight correlations among these LoRA models, given they originate from the same pre-trained models.

We believe an efficient system to serve multiple, different LoRA models needs to follow three design guidelines. (G1) GPUs are expensive and scarce resources, so we need to consolidate multi-tenant LoRA serving workloads to a small number of GPUs, increasing overall GPU utilization. (G2) As prior works have already noticed [88], batching is one of the, if not the most, effective approaches to consolidate ML workloads to improve performance and GPU utilization. However, batching only works when requests are for the exact same model. We thus need to enable batching for different LoRA models. (G3) The decode stage is the predominant factor in the cost of model serving. We thus only need to focus on the decode stage performance. Other aspects of the model serving are less important, and

we can apply straightforward techniques, e.g., on-demand loading of LoRA model weights.

Based on these three guidelines, we design and implement Punica, a multi-tenant serving framework for LoRA models on a shared GPU cluster. One key novelty is the design of a new CUDA kernel, **Segmented Gather Matrix-Vector Multiplication** (SGMV). SGMV allows batching GPU operations for the concurrent execution of multiple, different LoRA models. With SGMV, a GPU only needs to store a single copy of the pre-trained model in memory, significantly improving GPU efficiency in terms of both memory and computation. We pair this new CUDA kernel with a series of state-of-the-art system optimization techniques.

SGMV allows batching requests from different LoRA models, and *surprisingly*, we observe negligible performance differences between batching the same LoRA models and batching different LoRA models. At the same time, the on-demand loading of LoRA models has only millisecond-level latency. This gives Punica the flexibility to consolidate user requests to a small set of GPUs without being constrained by what LoRA models are already running on the GPUs.

Punica thus schedules multi-tenant workloads in the following two ways. For a new request, Punica routes the request to a small set of active GPUs, ensuring that they reach their full capacity. Only when the existing GPUs are fully utilized, Punica will allocate additional GPU resources. For existing requests, Punica periodically migrates them for consolidation. This allows freeing up GPU resources that are allocated to Punica.

We evaluate LoRA models that are adapted from Llama2 7B, 13B, and 70B models [77] on NVIDIA A100 GPU clusters. Given the same amount of GPU resources, Punica achieves 12x higher throughput compared to state-of-the-art LLM serving systems while only adding 2ms latency per token.

This chapter makes the following contributions:

- We identify the opportunity of batch processing requests of multiple, different LoRA models.

- We design and implement an efficient CUDA kernel for running multiple LoRA models concurrently.

- We develop new scheduling mechanisms to consolidate multi-tenant LoRA workloads.

4.1 Background

4.1.1 Low-Rank Adaptation (LoRA)

Fine-tuning allows a pre-trained model to adapt to a new domain or a new task or be improved with new training data. However, because LLMs are large, fine-tuning all the model parameters is resource-intensive.

Low-Rank Adaptation (LoRA) [30] significantly reduces the number of parameters needed to be trained during fine-tuning. The key observation is that the weight difference between the pre-trained model and the model after fine-tuning has a low rank. This weight difference can thus be represented as the product of two small and dense matrices. LoRA fine-tuning then becomes similar to training a small, dense neural network. Formally, let's consider the weights of the pre-trained model to be $W \in \mathbb{R}^{h_1 \times h_2}$. LoRA fine-tuning trains two matrices $A \in \mathbb{R}^{h_1 \times r}$ and $B \in \mathbb{R}^{r \times h_2}$, where r is the LoRA Rank. $W + AB$ is the new weight for the fine-tuned model. LoRA rank is usually much smaller than the original dimension (e.g., 16 instead of 4096). In addition to fast fine-tuning, LoRA has very low storage and memory overheads. Each fine-tuned model only adds 0.1% to 1% of the model weight. LoRA is usually applied to all dense projections in the transformer layer [13], including the Query-Key-Value-Output projections in the attention mechanism and the MLP. Note that the self-attention operation itself does not contain any weight.

How to serve multi-tenant LoRA models efficiently on a shared GPU cluster? LoRA provides an efficient algorithm to fine-tune LLMs. Now the question is: how to serve those LoRA models efficiently? One approach is to regard each LoRA model as an independent

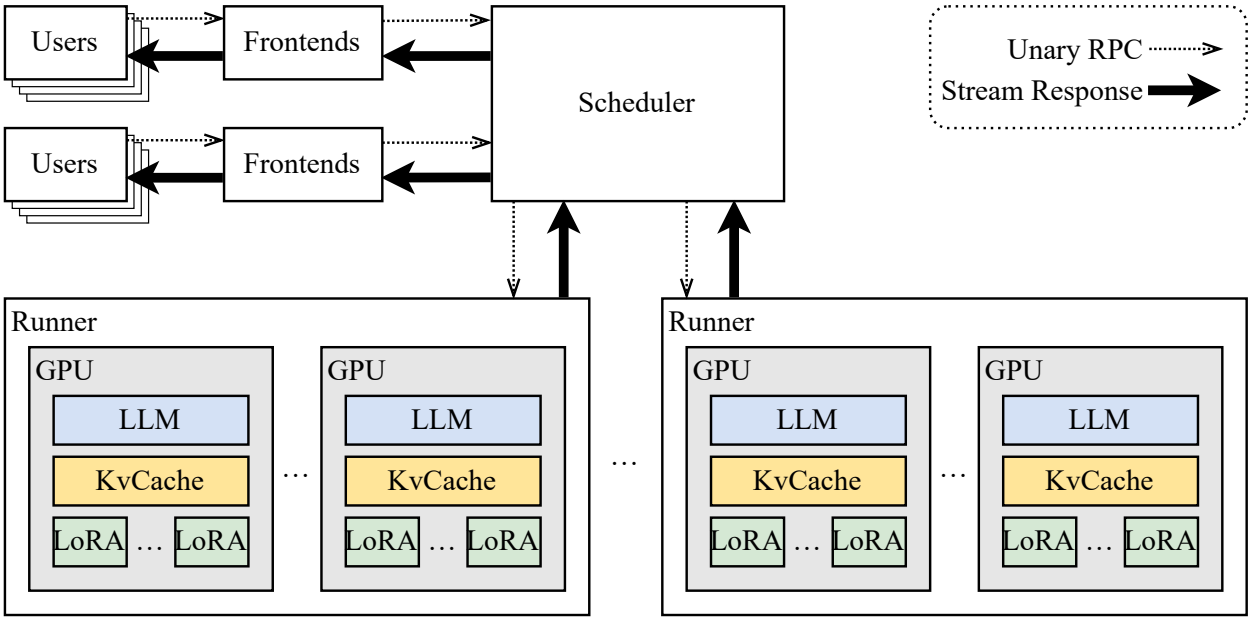


Figure 4.1: The system architecture of Punica.

model and use traditional LLM serving systems (e.g., vLLM). However, this neglects the weight sharing among different LoRA models that can be used to significantly improve GPU efficiency. Further, if we treat each LoRA model as an independent one, model loading time can be a substantial performance bottleneck when bootstrapping model serving. Even if we share the backbone model across the LoRA models, it remains a question as to how to batch compute the LoRA addon efficiently.

4.2 Punica Overview

We design Punica as a multi-tenant system that manages a cluster of GPUs to serve multiple LoRA models with shared pre-trained backbone models. Figure 4.1 shows the system architecture of Punica. Like other model serving systems, Punica has frontend servers that expose RESTful API to end-users and forward users' serving requests to the Punica scheduler. A user request contains the identifier of the LoRA model and a prompt. The scheduler dispatches requests to the GPUs. Each GPU server starts a runner, which communicates

with the scheduler and controls the execution of all the GPUs. As GPUs generate new tokens, new tokens are streamed from the runners to the scheduler, to the frontends, and finally to the end-users.

In Punica, each GPU loads the backbone pre-trained large language model. A large fraction of GPU memory is reserved for KvCache. Only the LoRA components of models are swapped in from remote storage when needed. Note that this design allows fast cold-start for model serving. Because the pre-trained model is already loaded into the GPU memory, Punica only needs to load matrices A and B for a new LoRA model.

Punica needs to address two key research challenges. The first challenge is how to run multiple LoRA models efficiently on a GPU. Because requests have to be served by different LoRA models, each request has to go through a different GPU computation. We use the existing matrix multiplication for the backbone computation. And we present a new CUDA kernel for adding the LoRA addons to the backbone computation in a batched manner. We call this kernel *Segmented Gather Matrix-Vector Multiplication* (SGMV). SGMV parallelizes the feature-weight multiplication of different requests in the batch and groups requests corresponding to the same LoRA model to increase the operational intensity of the kernel and use GPU Tensor Cores units for acceleration.

The second challenge is how to design an efficient system on top of SGMV for multi-tenant LoRA model serving. Our goal here is to consolidate multi-tenant workloads to the smallest set of GPUs possible, occupying the least amount of GPU resources. Punica schedules user requests to active GPUs, which already serve or train LoRA models. This is feasible in Punica, because with SGMV adding the batch size, even if for different LoRA models, improves the GPU utilization. For old requests, Punica migrates them periodically to consolidate the workloads, thereby freeing up GPU resources.

Next, we describe the details of Punica's CUDA kernel and other design details of Punica.

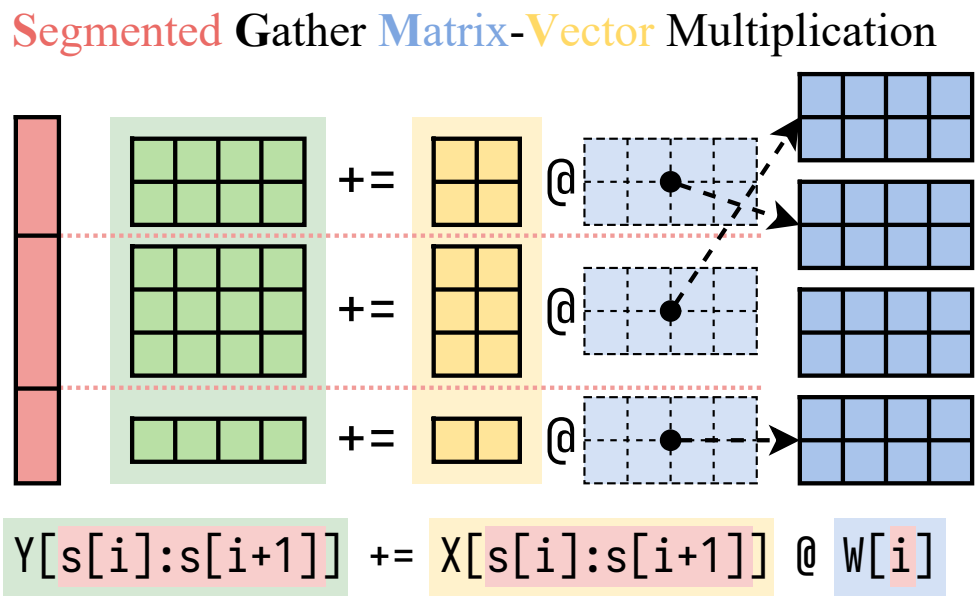


Figure 4.2: Semantics of SGMV.

4.3 Segmented Gather Matrix-Vector Multiplication

When a LoRA model has multiple inputs in the batch, we can further batch them together. We group inputs to the same LoRA model consecutively. Denote n as the number of LoRA models in a batch. Denote sequence s_i as the last element index for i-th model within the batch. In particular, $s_0 = 0$ and s_n is the batch size. Input $\{\vec{x_i} \mid i \in [1, s_n]\}$ is then partitioned as $\{\{\vec{x_j} \mid j \in (s_{n-1}, s_n]\} \mid i \in [1, n]\}$. The dense projection output can then be

written as:

$$\begin{pmatrix} \vdots \\ \overrightarrow{y_{s_1}} \\ \vdots \\ \overrightarrow{y_{s_{n-1}+1}} \\ \vdots \\ \overrightarrow{y_{s_n}} \end{pmatrix} := \begin{pmatrix} \vdots \\ \overrightarrow{x_{s_1}} \\ \vdots \\ \overrightarrow{x_{s_{n-1}+1}} \\ \vdots \\ \overrightarrow{x_{s_n}} \end{pmatrix} W + \begin{pmatrix} \vdots \\ \overrightarrow{x_{s_1}} \\ \vdots \\ \overrightarrow{x_{s_{n-1}+1}} \\ \vdots \\ \overrightarrow{x_{s_n}} \end{pmatrix} \begin{matrix} A_1 B_1 \\ \\ \\ A_n B_n \end{matrix}$$

The left-hand-side multiplication is the computation for the backbone model, which is batched through regular GEMM. We need a fast kernel to compute the right-hand-side LoRA addon. Note that operator $\vec{y} \mathrel{+}= \vec{x}AB$ can be separated as two launches of the same kernel: Initialize $\vec{v} := \vec{0}$. Then we run $\vec{v} \mathrel{+}= \vec{x}A$ and follow by $\vec{y} \mathrel{+}= \vec{v}B$.

We name this operator SGMV, Segmented Gather Matrix-Vector multiplication. Figure 4.2 illustrates the semantics of SGMV.

CUDA Kernel Schedule We classify SGMV operator into two categories, SGMV-shrink and SGMV-expand, based on their input and output feature dimensions. The first operator in the LoRA module: $\vec{v} = \vec{x}A$ is SGMV-shrink because it shrinks a high-dimensional input feature to low-rank output. The second operator $\vec{y} = \vec{v}B$ is SGMV-expand as it expands the low-rank input feature to a high-dimensional output feature.

Figure 4.3 shows how we schedule the SGMV kernel in these two cases: For both kernels, we bind the LoRA index to BLOCKIDX.Y in CUDA. Then, the computation on each BLOCKIDX.Y is a matrix multiplication between features and a specific LoRA weight. We designed different schedules for matrix multiplication under expand and shrink settings: for the expand kernel, we split A on the output feature dimension $A = \begin{bmatrix} A^{(1)} & \dots & A^{(n)} \end{bmatrix}$ and dispatch different $\vec{v}^{(i)} = \vec{x}A^{(i)}$ to different threadblocks in GPU, and the concatenation $\vec{v}^{(i)}$ on different threadblocks forms the final result $\vec{v} = \begin{bmatrix} \vec{v}^{(1)} & \dots & \vec{v}^{(n)} \end{bmatrix}$; for the shrink

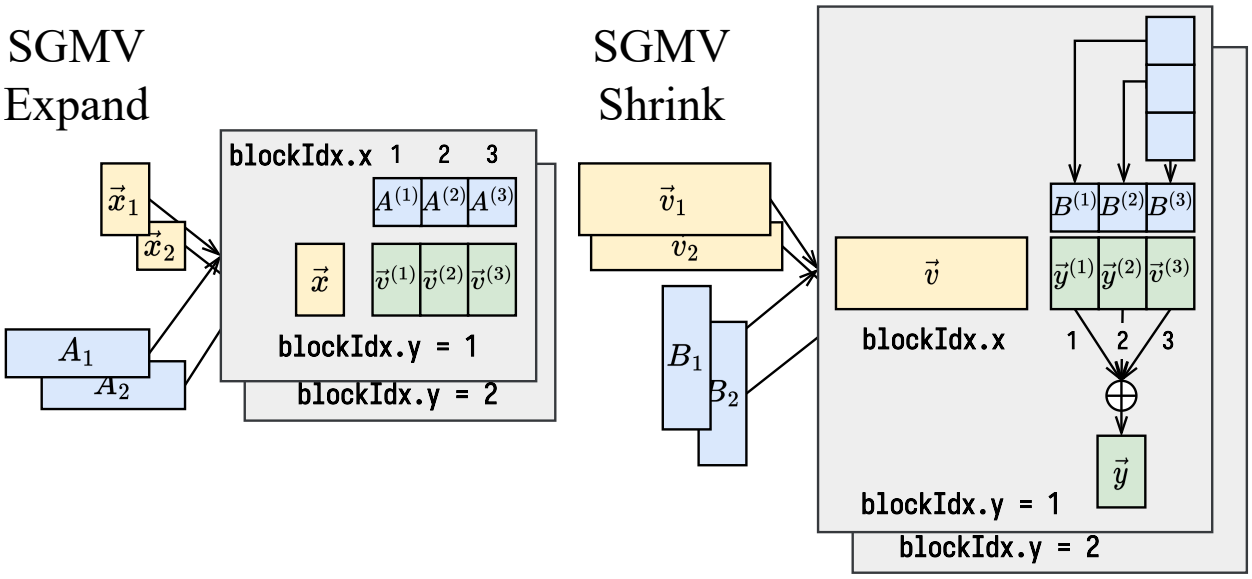


Figure 4.3: Scheduling of SGMV expand/shrink kernels

kernel, the output dimension is too thin and we adopt the Split-K strategy [74] to increase parallelism: we split B on the input feature dimension $B = \begin{bmatrix} B^{(1)} \\ \cdots \\ B^{(k)} \end{bmatrix}$. We dispatch different $\vec{y}^{(i)} = \vec{v}B^{(i)}$ to different threadblocks in GPU, after the partial sum $\{\vec{y}^{(i)}\}$ computation on all threadblocks are finished, we perform a grid synchronization followed by a cross threadblock reduction $\vec{y} = \sum_{i=1}^{k} \vec{y}^{(i)}$ to aggregate the partial results. We use GPU Tensor Cores to accelerate matrix multiplication for both kernels.

In the case that each request has a distinct LoRA index, the computation corresponding to each LoRA index degrades to matrix-vector multiplication, which is totally IO-bound. We designed a specific schedule for this case that maximizes memory bandwidth utilization and does not use Tensor Cores because of the low operational intensity of the operator.

4.4 Punica in Detail

Punica schedules new user requests at a per-request level and migrates old requests between GPUs at a per-iteration level. The scheduler adds requests to a GPU or cancels a working request from a GPU. Each GPU batches all requests in its working set for LLM invocation. GPU runs the Prefill steps and Decode steps continuously. When a request reaches the stopping condition (end-of-sequence token or length limit), the GPU removes the request from the batch and notifies the scheduler about the stopping.

We run batch requests of prefill and decode stages in a single model invocation. To minimize latency penalty, we limit the prefill batch size to 1 for each batch. The single prefill and the batch of decodes invoke two separate CUDA kernels for the self-attention operation. All other operations, including dense projection and LoRA addon, treat all tokens in the prefill stage and decode stage as a single batch input. In this way, we increase the batch efficiency of dense projection and LoRA addon.

4.4.1 Scheduling new requests

Punica scheduler has a global view of the state of all the GPUs. In particular, for each GPU, Punica maintains the working set of requests, which is the batch input of LLM invocation. As new requests are added to the working set and as the decode steps unfold, the KvCache consumes more and more GPU memory. Therefore, Punica also continuously tracks the memory space available for KvCache on each GPU.

Punica schedules a new request to the GPU that currently has the largest working set of requests (i.e., the LLM invocation batch size) while satisfying the following constraints: (1) It has not yet reached the max batch size limit. (2) It has enough memory for the new request's KvCache. When there are multiple candidates, the one that has the highest GPU UUID gets the new request. When all GPUs are fully occupied (i.e., have reached the maximum batch size or have insufficient memory), the request is queued. When some GPUs become available in the future, queued requests are scheduled in a first-come-first-

serve (FCFS) manner.

The max batch size limit balances the cluster throughput and the per-token latency. Oversized batches greatly slow down latency while providing marginal throughput gains. We profile A100 GPUs and decide to set the maximum batch size to 32.

The GPU selection logic emphasizes cluster throughput within the latency sweet spot. Our scheduling approach has the following attributes: a busy GPU is likely to stay busy as more requests will be assigned to it, a lightly loaded GPU is likely to lower its load as requests terminate, and an idle GPU is likely to stay idle. As a consequence, our scheduler maintains peak throughput and consolidates GPU usage based on the current overall system load. This allows easier decisions to scale up/down the GPU cluster. In a cloud setting where Punica can allocate and deallocate GPU servers, we perform the following cluster allocations: (1) If no lightly loaded GPU exists in the cluster, Punica should request more GPUs. (2) Punica can return the GPU resources for GPU servers with no load.

4.4.2 On-demand model loading

Weight sharing between the LoRA model and the underlying pre-trained model makes model loading fast. The size of the LoRA model (which is matrices A and B in subsection 4.1.1) is only 1% of the underlying pre-trained model.

Loading a LoRA model from the main memory to the GPU memory is merely an asynchronous host-to-device memory copy. The latency is bounded by the PCIe bandwidth. On PCIe Gen4 x16, it takes around 50μs to load a layer and 2ms to load the entire model. Since the memory copy and the GPU computation can overlap, it is feasible to implement sophisticated layer-by-layer or even matrix-by-matrix loading to minimize the model loading delay.

However, notice that each decode step takes around 30ms to complete, and each request might need thousands of decode steps. We opt to use a simpler yet equivalently efficient method. When a request is newly added to a GPU, if its LoRA model is not already

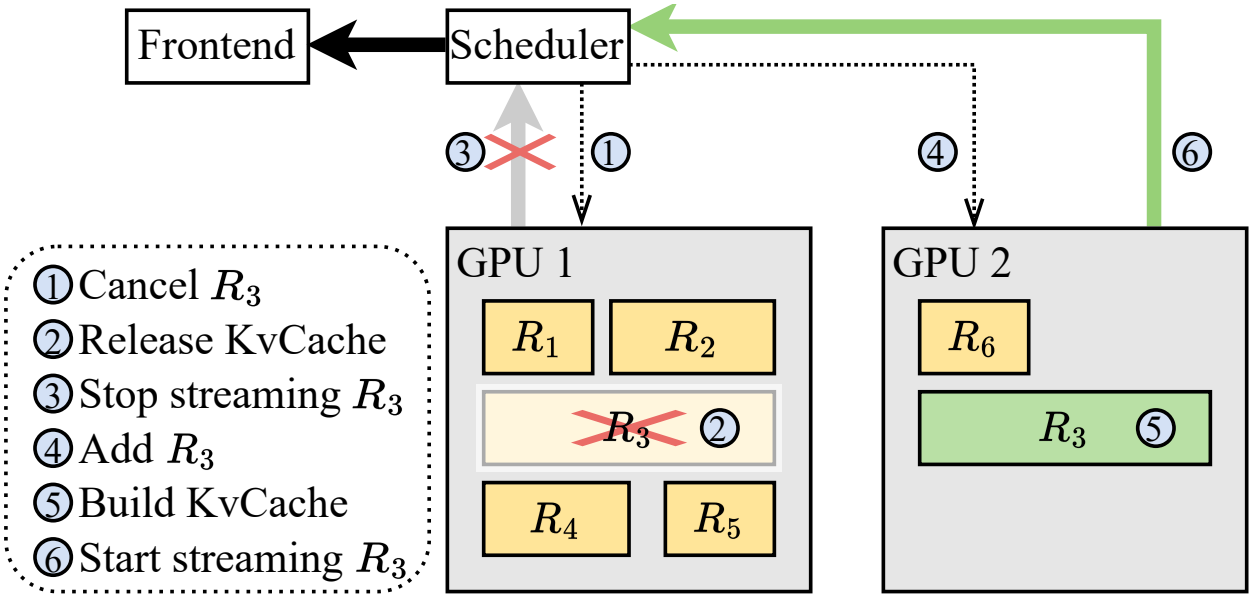


Figure 4.4: Request migration procedure for Request R_3.

loaded, we issue an asynchronous memory copy to load the LoRA weight, and let the GPU continue running other inputs in the batch. By the end of the model execution, the weight already finished loading. Then, the new request is able to join the batch naturally.

4.4.3 Request migration

As each request generates more tokens, their KvCache occupies more GPU memory. When a GPU runs out of space for KvCache, it migrates some requests to other GPUs. The request migration consists of two steps — evict and add. The scheduler evicts the newest request from the GPU. This preserves the FCFS semantics. The scheduling for the evicted request is the same as adding a new request.

Punica scheduler supports canceling requests. Cancellation is straightforward: remove the request from both the GPU and the scheduler states. A typical scenario for cancellation is user disconnection. More importantly, request cancellation as a scheduling primitive enables request migration.

Figure 4.4 shows the workflow to migrate a request, R_3, from GPU 1 to GPU 2. The scheduler first sends the cancellation of the request to GPU 1. After GPU 1 finishes running the previous batch, it picks up the cancelation and releases KvCache. GPU 1 also omits the R_3's new token generated in the previous batch. Immediately after sending the cancellation to GPU 1, the scheduler adds R_3 to GPU 2. GPU 2 runs a prefill step on R_3's original prompt plus all previously generated tokens. This reinstablishes the R_3's KvCache on GPU 2. GPU 2 then starts to stream R_3's new tokens to the scheduler.

We opt for recomputation instead of moving the KvCache for its simplicity. PagedAttention [36] has shown that recomputation's latency is equal to or better than moving the KvCache in most cases, which agrees with our observation.

4.4.4 Memory layout for KvCache

Punica uses a separable KvCache layout, which is important for text generation batching throughput. The HuggingFace Transformers library's KvCache layout consists of complicated nested lists of tensors, which can conceptually be viewed as the following shape:

$$[L, 2, B, N, S, D]$$

where L is the number of layers, 2 is for Key and Value projection, B is batch size, N is the number of heads, S is the sequence length, and D is the head dimension. In each decode step, HuggingFace Transformers concatenates one tensor along the sequence length dimension. The concatenation is inefficient as it needs to read the whole KvCache and write a new copy, whereas the new tensor is only $1/S$ of the KvCache.

The bigger problem with the HuggingFace Transformer's approach is that the batching dimension is not the outermost dimension, which means that requests in the batch are hard to separate in the KvCache. Under this restriction, requests that enter the batch together need to remain together during all decode steps *until all requests meet their own stopping condition*.

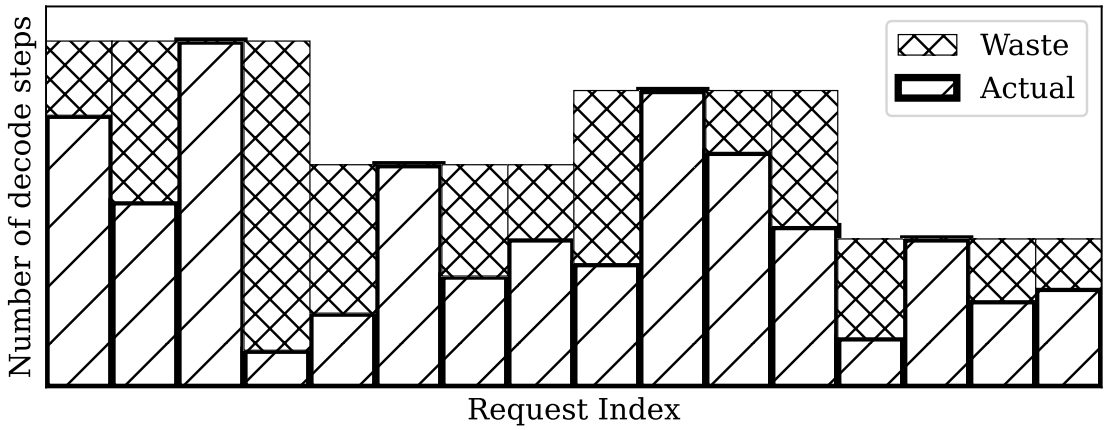


Figure 4.5: Inseparable KvCache adds wasted decode steps.

Figure 4.5 is an illustrative figure that explains the problem. In the figure, consecutive 4 requests are batched together. The striped bar represents the number of decode steps that each request actually needs to reach its stopping condition. Due to inseparable KvCache, shorter requests in the batch run additional decode steps, which is essentially wasted computation. FasterTransformer [29] and DeepSpeed [1] also suffer from similar problems.

Instead, our KvCache layout is

$$[\textstyle\sum_i \lceil {}^{S_i}/_{P} \rceil, L, 2, N, P, D]$$

where S_i is the length of the sequence i and P is the page size. We use paged KvCache [36] to minimize memory fragmentation. We put the batching dimension on the outmost to enable continuous batching.

4.5 Implementation

Punica implementations consist of two parts: a Python library on top of PyTorch that runs large language models on a single GPU and other system components to support model

serving across a GPU cluster.

Python Library We expose our CUDA kernels as a PyTorch Extension using PyBind11. Llama model implementation is adapted from the HuggingFace Transformers library. We use FlashInfer [86] open source project for fast and memory-efficient computation of self-attention. Besides fusing the computation of $\mathrm{softmax}(QK^T)V$ like FlashAttention [10] does, FlashInfer further supports batch decoding without padding. Similar to PagedAttention [36], FlashInfer supports paged KvCache to minimize GPU memory fragmentation due to KvCache. We also fuse LayerNorm, which reduces latency from 110μs to 4μs.

We mix new requests in the Prefill stage and existing requests in the Decode stage in one batch together. This way, dense projections and LoRA can benefit from a bigger batch size. For batching, we concatenate all inputs along the sequence length dimension. We always put Prefill requests at the beginning and Decode requests at the latter part. We then pass a `BatchLen` struct to distinguish different requests. `BatchLen` contains a list of indices indicating the starting index of each Prefill request. It also contains a number indicating the number of Decode requests. In the self-attention layer, we pass the indices and leading input states to the BatchPrefill kernel, and we pass the trailing input states to the BatchDecode kernel. Within a batch, we further organize the batch input order such that requests that share the same LoRA model are consecutive. The tail of Prefill requests and the head of Decode requests can share a LoRA model if possible. We then generate the segment indices for the SGMV kernel. Before each batched model invocation, we concatenate batch inputs and construct `BatchLen` and SGMV segment indices. Both `BatchLen` and SGMV segment indices remain constant for the entire model invocation. This design avoids recomputation (L times for `BatchLen` and $7L$ times for SGMV segment indices, where L is the number of layers).

Other system components We write our scheduler, frontend, and runner in Rust. Unary RPC and streaming text chunks are both implemented via web socket. I/Os are handled

asynchronously. Runner spawns a Python subprocess for each GPU. The subprocess is a thin warper around our PyTorch library. The Runner main process communicates with the subprocesses using pipes.

4.6 Evaluation

We evaluate Punica on two testbeds. Testbed #1 is a single server with one NVIDIA A100 80GB GPU. Testbed #2 consists of two NVIDIA HGX A100 40GB servers with 8 GPUs on each server. Testbed #1 contains a GPU with large GPU memory, which allows us to study the LoRA batching effect. Testbed #2 is equipped with fast NvSwitch technology for us to study tensor parallelism and evaluate cluster deployment. We use the Llama-2 [77] model with 7B, 13B, and 70B parameters. For all experiments, we use 16 as the LoRA rank. LoRA is applied to all dense projections. We use random weights for LoRA models as the weight does not affect latency performance.

Workloads Prompt lengths and response lengths are key workload characteristics for LLM serving. We use the prompt and response length distributed from ShareGPT [64], which consists of user-bot conversations from Internet users. We consider four types of request distribution among LoRA models. (1) **Distinct**: each request is for a distinct LoRA model. (2) **Uniform**: all LoRA models are equally popular. Given n requests, we use $\lceil\sqrt{n}\rceil$ models. (3) **Skewed**: model popularity follows Zipf-α distribution. The number of requests to the i-th most popular model is α times that of the $i+1$-th's. In our experiments, we choose α to be 1.5. (4) **Identical:** all requests are for the same LoRA model.

Baselines Since there is no well-known multi-LoRA serving system, we compare Punica against a variety of popular LLM backbone serving systems. We allow various degrees of relaxation that are in favor of baseline systems. We use HuggingFace PEFT [47] library to add LoRA weights to HuggingFace Transformers [82] library and DeepSpeed [1]. We run

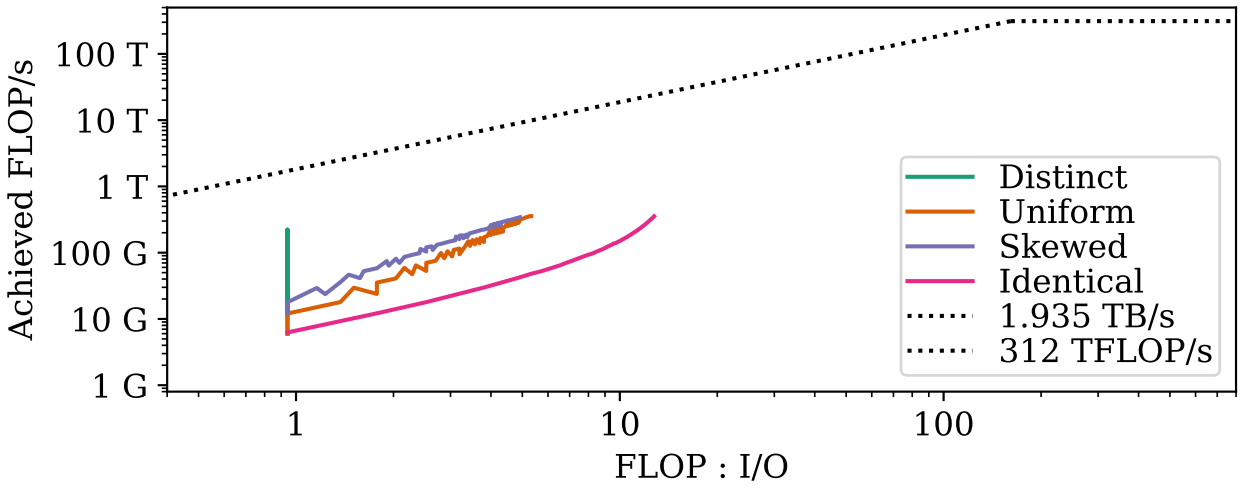


Figure 4.6: Roofline plot of the SGMV kernel.

backbone-only for FasterTransformer [29] and vLLM[1] [36] since these two systems do not support LoRA models. We omit the model switching costs for baseline systems.

4.6.1 Microbenchmarks

We use analysis and testbed evaluations to benchmark SGMV and distill the -performance implications for the LoRA operator and a single transformer layer.

Roofline analysis for SGMV First, we use the roofline model [81] to understand the performance of our SGMV kernel. The number of floating point operations (FLOP) and the number of memory I/O bytes of SGMV are calculated as:

$$\text{FLOP} = s_n \times h_i \times h_o \times 2$$

$$\text{I/O} = [s_n \times (h_i + h_o) + n \times h_1 \times h_2] \times 2$$

[1]With commit 928de46

where n is the number of LoRA models, s is the segment indices, s_n is the total number of inputs, and h_i and h_o are the input and output dimensions of the SGMV weight matrix. The factor 2 in FLOP comes from multiply-add operations for matrix multiplication. The factor 2 in I/O comes from the byte size of 16-bit floating point data type. We use $h_i = 16$, $h_o = 4096$ for this case study. We measure the latency of batch size 1 to 64 under the four different popularity distributions on Testbed #1.

Figure 4.6 shows the roofline model plot. The x-axis in the roofline model is arithmetic intensity, which is defined as the ratio of FLOP to I/O. The y-axis is the achieved throughput in terms of FLOP per second, calculated using measured latency. The diagonal dotted line and the top dotted line represent the memory bandwidth and peak FP16 performance of the NVIDIA A100 GPU, respectively.

In the **Distinct** case, the arithmetic intensity does not change because FLOP and I/O grow at the same rate. Since each input only utilizes a small amount of GPU compute units, increasing the batch size increases performance. In the **Identical** case, the line goes up diagonally following the slope of memory bandwidth, which means that SGMV is bounded by memory bandwidth. The **Uniform** case and the **Skewed** case sit in between, as a combination of both effects, the increasing degree of parallelism and the increasing arithmetic intensity.

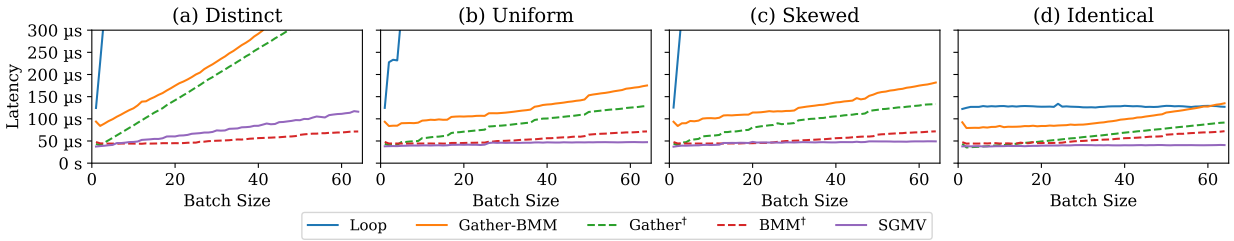


Figure 4.7: Microbenchmark for LoRA operator implementations. †Gather and BMM are measured separately for reference.

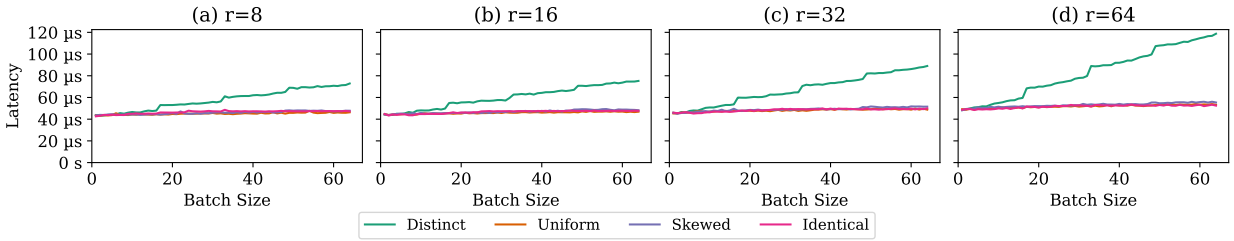


Figure 4.8: Microbenchmark for LoRA operator on various LoRA rank.

LoRA operator microbenchmark We implement the batched LoRA operator as two SGMV kernel launches. We compare our SGMV-based implementation against two PyTorch-based baseline implementations. One is a for-loop over each LoRA model. Another is Gather-BMM. In the gather step, we stack the weight matrices that each input needs into a single matrix. Then, we use `torch.bmm()` to perform a batched matrix multiplication on the input and the stacked matrix. Similar to SGMV, Gather-BMM launches Gather twice and BMM twice. Note that Gather-BMM uses much more I/O than SGMV. Gather reads in $n \times h_i \times h_o$ elements and writes to $s_n \times h_i \times h_o$. Then, BMM needs to read in $s_n \times h_i \times h_o$ weight elements that Gather just wrote. In combination, Gather-BMM incurs $s_n \times h_i \times h_o \times 2$ more elements memory I/O than SGMV.

Figure 4.7 shows the latency comparison of the three implementations across four workloads on Testbed #1. Gather and BMM are also measured separately for reference. Since BMM is data-independent, its latency is consistent across four workloads.

Our benchmark results match our analysis very well. In the **Distinct** case, Loop behaves terribly because it runs multiple rounds of batch size 1. Gather-BMM latency increases fast due to the slowdown of Gather. SGMV latency increases gradually as well, from 37μs to 116μs, because batching does not change arithmetic intensity. The **Uniform** case and the **Skewed** case are similar to the **Distinct** case. Gather-BMM performs slightly better than the **Distinct** case since there are fewer matrices to read. SGMV latency increases only marginally, from 37μs to 46μs, as a combination of both effects: increasing degree of parallelism and increasing arithmetic intensity. In the **Identical** case, all implementations

have the same semantics: BMM. We can, therefore, infer that SGMV implements BMM more efficiently than `torch.bmm()` in the case of LoRA. SGMV latency remains almost constant, from 37μs to 40μs.

Overall, SGMV significantly outperforms baseline implementations regardless of workloads.

We also run the microbenchmark of different LoRA ranks on Testbed #1. Figure 4.8 shows the latency for LoRA rank 8, 16, 32, and 64. In the **Distinct** case, the latency gradually increases. The latency of a single request batch is around 42μs for all four ranks, while batch size 64 goes up to 72μs, 75μs, 89μs, and 118μs, respectively. When the workload exists weight sharing (**Uniform**, **Skewed**, and **Identical**), the latency remains almost the same across batch size 1 to 64, at around 42μs to 45μs.

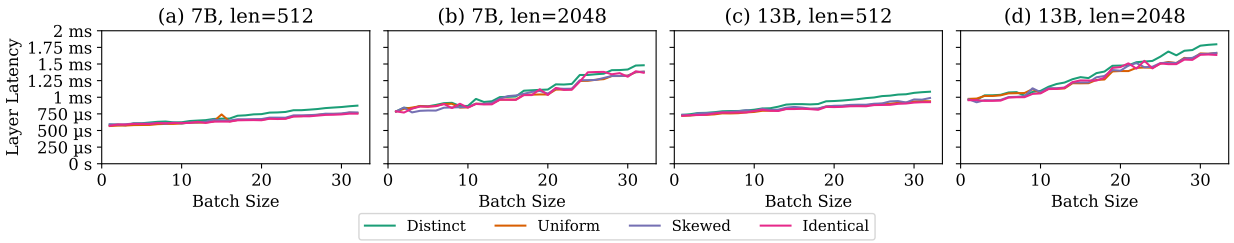


Figure 4.9: Transformer Layer Benchmark.

Transformer layer benchmark Next, we evaluate the transformer layer performance after incorporating the LoRA operator. Since the LLM is roughly a stack of transformer layers, the layer performance determines the overall model performance. We run the layer benchmark on Testbed #1 based on the 7B and 13B model configurations and sequence lengths of 512 and 2048. Figure 4.9 plots the layer latency. When the sequence length is shorter, the batching effect is stronger. The latency only increases by 72% when batch size increases from 1 to 32 when the sequence length is 512. When the sequence is longer, self-attention takes longer time, which reduces the layer-wise batching effect.

In contrast to the kernel microbenchmark, notice that the layer latency is roughly the

same across different workloads. This is because the computation time for the LoRA add-on is small compared to the backbone dense projection and the self-attention. *This LoRA-model-agnostic performance property enables us to schedule different LoRA models as if one model.* Our scheduling algorithm can then focus on the overall throughput instead of individual LoRA model placement, which is exactly how we design Punica.

4.6.2 Text generation

Next, we study the text generation performance of Punica and baseline systems.

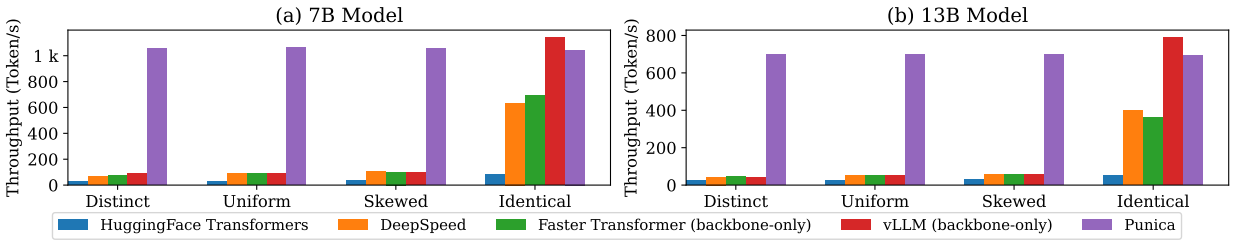


Figure 4.10: Single GPU text generation comparison

Serving 7B and 13B models on a single GPU We evaluate text generation using Punica and baseline systems on a single GPU on Testbed #1. The single-GPU performance serves as the base case for cluster-wide deployment. We generate 1000 requests (generating around 101k tokens) and restrict each system to batch in a first-come-first-serve manner. The max batch size is set to 32 for all systems. Punica can batch across different LoRA models, and baseline systems can only batch requests for the same LoRA models.

Figure 4.10 (a) and (b) show the results on the 7B model and the 13B model, respectively. Punica consistently delivers high throughput regardless of workloads. Punica achieves 1044 tok/s and 693 tok/s on the 7B and the 13B models, respectively. Although most baselines can achieve relatively high throughput in the **Identical** case, their performance deteriorates when there are multiple LoRA models.

In the **Distinct** case, all baseline systems run with a batch size of 1, and thus, the throughput is low. In the **Uniform** and the **Skewed** cases, most batches for the baseline systems have extremely small batch sizes (1–3), which explains the low performance. Punica is able to batch different LoRA models in one batch and, therefore, can run with a batch size of 32 consistently.

With only one LoRA model, all systems can run with a batch size of 32. Thus, all except for the HuggingFace Transformer can deliver high throughput. HuggingFace Transformer's low performance is due to its lack of critical CUDA kernel optimizations, including FlashAttention [10]. In the **Identical** case, both vLLM and Punica outperform other systems because the two systems' KvCache layout allows continuous batching. In contrast, other systems have to wait for the longest sequence in the batch to finish. vLLM's throughput is slightly higher than Punica (at 1140 tok/s and 789 tok/s, respectively) because we run vLLM backbone-only.

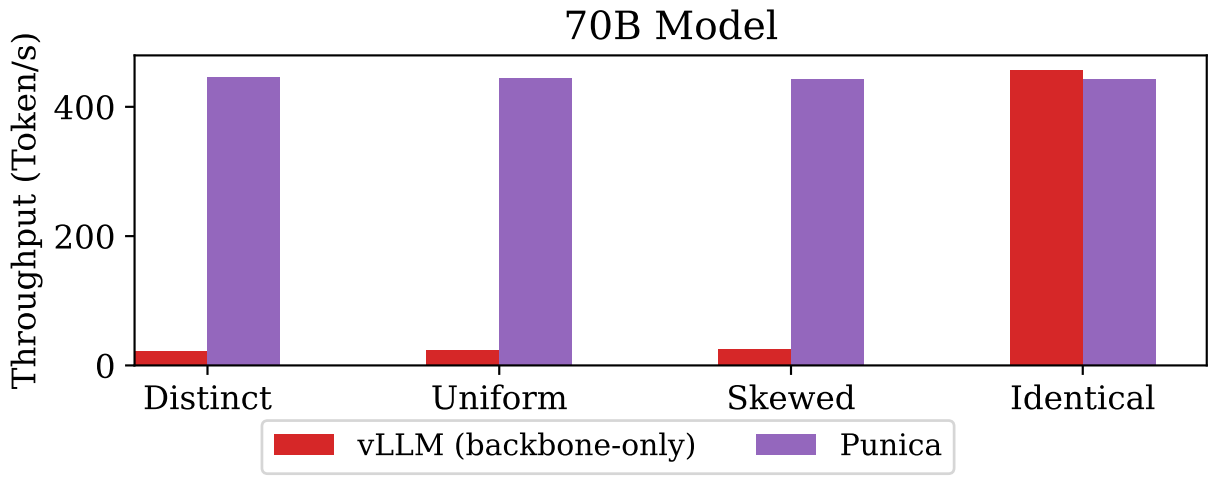


Figure 4.11: 70B model text generation comparison.

Serving 70B models with tensor parallelism We run the 70B model on Punica with Megatron's tensor parallel scheme [50, 51] on 8 GPUs in Testbed #2. We compare Punica and vLLM. vLLM also uses the same Megatron's tensor parallel scheme.

Figure 4.11 shows similar trends as our results in serving 7B and 13B models. In the presence of multiple LoRA models, vLLM's throughput is around 21 to 25 tok/s, whereas when serving the backbone, vLLM can achieve 457 tok/s due to the large batch size. For the **Identical** case, Punica and vLLM achieve the same performance because their parallel schemes are the same. However, Punica can consistently deliver 441 to 446 tok/s throughput regardless of LoRA popularity distribution, significantly outperforming vLLM for serving multiple LoRA models.

4.6.3 Cluster deployment

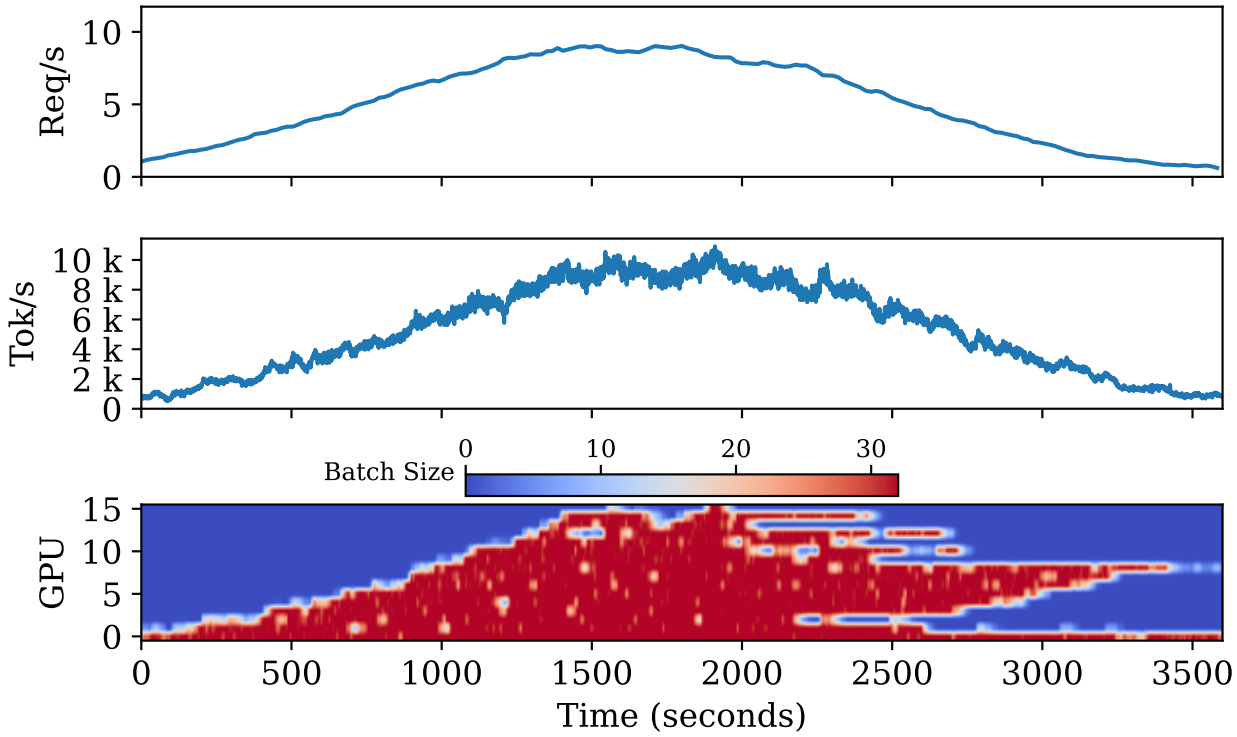


Figure 4.12: Cluster deployment.

We evaluate Punica on 16 GPUs in Testbed #2. The load varies as follows: In the macro view, the request rate of the workload gradually increases and then gradually decreases. In the micro view, gaps between request arrival time follow an exponential distribution, and

the arrival process follows a Poisson distribution. LoRA model popularity follows Zipf-1.5 distribution (same as our **Skewed** workload). The duration of the experiment is one hour. The model size is 7B in this experiment.

Punica is able to consolidate GPU usage while delivering high throughput. Figure 4.12's upper panel shows the request rate over time. The middle panel shows text generation throughput in terms of tokens per second. The lower panel shows the batch size of each GPU across time. GPUs usually run with the maximum batch size when they are not idle because our schedule algorithm prioritizes large batch sizes. Occasionally, a GPU runs with a smaller batch size because it runs out of KvCache space and migrates out a few requests to other GPUs. When a GPU becomes idle (batch size = 0), it is likely that it stays idle, which can then be released to the cloud provider if necessary.

4.7 Related Work

Multi-model inference serving A substantial body of work has been proposed for serving ML models on a GPU cluster. Clipper [9] is one of the earliest systems to optimize both throughput and latency in a GPU cluster. It is followed up by a series of systems [18]. However, they are mainly designed to serve smaller CNN models. One key difference is that serving CNN models is stateless whereas LLM serving needs to persist the KvCache. The state introduces an affinity that asks for a different system design. For example, Symphony [6] uses a deferred batch scheduling algorithm but Punica runs batches on a GPU back-to-back due to the KvCache affinity. Although Nexus [65] supports prefix sharing of different models, they offer limited support and optimization faced with LLMs and fine-grained sharing patterns as we witnessed in LoRA.

PetS [98] batches requests to different adapters (e.g., Adapters [25], MaskBert [96], Diff-Pruning [20], Bitfit [90]) of a LLM on a single GPU. It allows GPU memory sharing of the pre-trained model for different downstream tasks, however, it does not enable multiple different models to run concurrently.

Model and KvCache quantization/compression A substantial amount of work has been proposed to reduce the memory footprint of model weights, activations and KvCache by quantization [16, 84, 19, 42, 66]. Model quantization saves more headroom for KvCache, hence enabling Punica to serve requests of longer sequences without migration. In addition, KvCache quantization [66] and compression [46, 45, 95, 85] further reduces the memory I/O of the KvCache, through which inference latency can be reduced, as self-attention is bounded by GPU memory bandwidth [10]. QLoRA [13] proposes to fine-tune LoRA by storing LoRA weights/gradients in high-precision formats such as fp16 while keeping the original weight in quantized formats to save memory footprint during fine-tuning. Quantization reduces self-attention latency, which makes the high efficiency of Punica's LoRA kernel even more important.

4.8 Summary

Low-rank adaptation (LoRA) has become an important fine-tuning method to adapt pre-trained models to specific domains. We present Punica, a system to serve multiple LoRA models in a shared GPU cluster. Punica's design is centered around a new CUDA kernel design that allows batching of GPU operations for different LoRA models. For each GPU, Punica only requires a single copy of the underlying pre-trained model for the GPU to serve multiple, different LoRA models, significantly improving GPU efficiency in terms of both memory and computation. Additionally, Punica's scheduler consolidates multi-tenant LoRA serving workloads in a shared GPU cluster. With a fixed-sized GPU cluster, our evaluations show that Punica achieves 12x higher LoRA model serving throughput compared to state-of-the-art LLM serving systems while only adding 2ms latency per token. Symphony's source code is available at https://github.com/punica-ai/punica.

Chapter 5

CONCLUSION

Multi-tenant model serving systems play a critical role in powering machine learning applications. This report explores how to design systems that make efficient use of a GPU cluster by addressing the following challenges: (1) batching efficiency under latency constraints, (2) bursty requests and GPU consolidation, (3) GPU cluster auto-scaling.

We have introduced two systems, Symphony and Punica, each tackles a category of machine learning models. Symphony optimizes deep neural network model serving, achieving 6x goodput given the same number of GPUs compared to state-of-the-art systems, saving 60% GPUs when serving the same request rate, and enabling robust cluster auto-scaling behavior. Punica serves any number of LoRA fine-tuned large language models at the cost of one, improving throughput by 12x.

5.1 Future Work

5.1.1 Multi-LoRA serving for diffusion models

Recent latent diffusion models [58, 54] exhibit an eye-opening capability of text-to-image synthesis. Diffusion models rely heavily on fine-tuning to mimic different styles and personalize for different subjects [61]. In addition to text generation, LoRA has also been widely used for fine-tuning diffusion models. We propose to extend Punica's multi-LoRA serving capability to diffusion models.

One aspect in which diffusion models differ from large language models is that diffusion models can use more than one LoRA adapater in one image generation. This usage pattern signifies the multi-LoRA serving optimization. Another difference is that the linear layer shapes are heterogeneous in diffusion models, whereas the shape is the same across all

layers in large language models. It requires a different memory layout and API changes to the SGMV kernel to support different shapes in a concise way.

5.1.2 Flexible ranks

Currently, the SGMV kernel requires all LoRA adapters to use the same rank. Some applications have adapters of different ranks. Although the limitation can have a simple workaround, i.e., padding to the biggest rank, it would be best to support different ranks natively.

5.1.3 Other types of parameter-efficient fine-tuning adpaters

In addition to LoRA, researchers have proposed several other types of parameter-efficient fine-tuning adapters. DoRA [43] is similar to LoRA except that it separates magnitudes from vector directions. AdaLoRA [93] uses different LoRA ranks on different layers. MoLE [83] introduces the mixture-of-experts-style adapters by adding multiple LoRA adapters gated by a gating function. All these new methods are extensions of LoRA. Punica can add support for these types of parameter-efficient fine-tuning adapters.

5.1.4 Quantized adapters

Quantization has proven to be a great trade-off between model accuracy and speed. Newer generations of GPUs provide native support for quantized data types, including FP8, INT8, and INT4. Tensor cores of these quantized data types run way faster than the 16-bit version. Although the multi-LoRA computation only adds about 10% latency to the end-to-end model inference in 16-bit precision, it will become a bottleneck if the other parts of the model are quantized. Hence, the SGMV kernel also needs to have a new implementation to support quantization.

5.1.5 Better user experience as a open-source project

Out-of-the-box usability, user-friendly APIs, detailed documentation, and fast customer support are important for the adoption of an open-source project. There are many tasks to do to provide a better user experience in the Punica project.

First, one needs to provide a multi-LoRA layer abstraction and automatic replacement mechanism. Since LoRA is only applied to Linear layers, which are only a standard part of the bigger neural network architecture, it makes sense to have a way to replace the Linear layers with the multi-LoRA layer automatically in a constructed PyTorch model. In this way, the Punica project will not need to provide definitions for all variants of large language model architectures.

Second, supporting `sm75` (Turing) and `sm90` (Hopper) architectures. The current version of the SGMV kernel is developed for Ampere (`sm80`, `sm86`, `sm87`) and Ada Lovelace (`sm89`) architecture. Turing architecture lacks several important performance features, including hardware-accelerated async-copy, warp-level support for reduction ops, and L2 cache residency management. Nonetheless, Turing GPUs still occupy a large market share due to the expensive costs of newer-generation GPUs. We could provide an implementation for `sm75` by removing the usage of advanced GPU architecture features. On the other hand, making the current code run on Hopper architecture requires trivial efforts. However, it remains a challenge of how to make use of the performance features of Hopper, including distributed shared memory, thread block cluster, and tensor memory accelerator unit.

BIBLIOGRAPHY

[1] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. Deepspeed- inference: Enabling efficient inference of transformer models at unprecedented scale. In Felix Wolf, Sameer Shende, Candace Culhane, Sadaf R. Alam, and Heike Jagode, editors, *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, November 13-18, 2022*, pages 46:1–46:15. IEEE, 2022.

[2] Guido Appenzeller, Matt Bornstein, and Martin Casado. Navigating the high cost of ai compute. https://a16z.com/navigating-the-high-cost-of-ai-compute/, 2023. Accessed: 2023-11-03.

[3] Nathan Benaich and the Air Street Capital team. State of ai report 2023 edition. https://www.stateof.ai/, 2023. Accessed: 2023-11-03.

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[5] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, and Tri Dao. Medusa, September 2023.

[6] Lequn Chen, Weixin Deng, Anirudh Canumalla, Yu Xin, Danyang Zhuo, Matthai Philipose, and Arvind Krishnamurthy. Symphony: Optimized dnn model serving using deferred batch scheduling, 2023.

[7] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant lora serving, 2023.

[8] François Chollet. Xception: Deep learning with depthwise separable convolutions. *CoRR*, abs/1610.02357, 2016.

[9] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 613–627. USENIX Association, 2017.

[10] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[11] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale. *CoRR*, abs/2208.07339, 2022.

[12] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale. *CoRR*, abs/2208.07339, 2022.

[13] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023.

[14] Tim Dettmers and Luke Zettlemoyer. The case for 4-bit precision: k-bit inference scaling laws. In *International Conference on Machine Learning*, pages 7750–7774. PMLR, 2023.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[16] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: accurate post-training quantization for generative pre-trained transformers. *CoRR*, abs/2210.17323, 2022.

[17] Erin Griffith. The desperate hunt for the a.i. boom's most indispensable prize. https://www.nytimes.com/2023/08/16/technology/ai-gpu-chips-shortage.html, 2023. Accessed: 2023-11-03.

[18] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, November 2020.

[19] Cong Guo, Jiaming Tang, Weiming Hu, Jingwen Leng, Chen Zhang, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. Olive: Accelerating large language models via hardware-friendly outlier-victim pair quantization. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–15, 2023.

[20] Demi Guo, Alexander M. Rush, and Yoon Kim. Parameter-efficient transfer learning with diff pruning. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pages 4884–4896. Association for Computational Linguistics, 2021.

[21] Derrick Harris, Matt Bornstein, and Guido Appenzeller. Ai canon. https://a16z.com/ai-canon/, 2023. Accessed: 2023-11-03.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. *CoRR*, abs/1603.05027, 2016.

[24] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *J. Mach. Learn. Res.*, 22:241:1–241:124, 2021.

[25] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 2790–2799. PMLR, 2019.

[26] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3, 2019.

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

[28] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. In *Annual Meeting of the Association for Computational Linguistics*, 2018.

[29] Bo Yang Hsueh. FasterTransformer. https://github.com/NVIDIA/FasterTransformer, 2021.

[30] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

[31] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. Scrooge: A cost-effective deep learning inference system. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 624–638, New York, NY, USA, 2021. Association for Computing Machinery.

[32] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.

[33] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[34] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023.

[35] Kubernetes. Horizontal pod autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, 2023. Modified: 2023-05-18.

[36] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[37] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. Pretzel: Opening the black box of machine

learning prediction serving systems. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 611–626. USENIX Association, 2018.

[38] Matthew LeMay, Shijian Li, and Tian Guo. Perseus: Characterizing performance and cost of multi-tenant serving for cnn models. In *2020 IEEE International Conference on Cloud Engineering (IC2E)*, pages 66–72, 2020.

[39] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[40] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 19274–19286. PMLR, 2023.

[41] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, Boston, MA, July 2023. USENIX Association.

[42] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978*, 2023.

[43] Shih-Yang Liu, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Kwang-Ting Cheng, and Min-Hung Chen. Dora: Weight-decomposed low-rank adaptation, 2024.

[44] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[45] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for LLM KV cache compression at test time. *CoRR*, abs/2305.17118, 2023.

[46] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Ré, and Beidi Chen. Deja vu: Contextual sparsity for efficient llms at inference time. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 22137–22176. PMLR, 2023.

[47] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. Peft: State-of-the-art parameter-efficient fine-tuning methods. https://github.com/huggingface/peft, 2022.

[48] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating generative LLM serving with speculative inference and token tree verification. *CoRR*, abs/2305.09781, 2023.

[49] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[50] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters using megatron-lm. In Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, page 58. ACM, 2021.

[51] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[52] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ML serving. *CoRR*, abs/1712.06139, 2017.

[53] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22:1345–1359, 2010.

[54] Dustin Podell, Zion English, Kyle Lacey, Andreas Blattmann, Tim Dockhorn, Jonas Müller, Joe Penna, and Robin Rombach. Sdxl: Improving latent diffusion models for high-resolution image synthesis, 2023.

[55] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[56] Colin Raffel, Noam M. Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *ArXiv*, abs/1910.10683, 2019.

[57] Grand View Research. Artificial intelligence as a service market size, share and trends analysis report. https://www.grandviewresearch.com/industry-analysis/artificial-intelligence-as-a-service-market-report, 2023. Accessed: 2023-12-03.

[58] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022*, pages 10674–10685. IEEE, 2022.

[59] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. Infaas: Managed & model-less inference serving. *CoRR*, abs/1905.13348, 2019.

[60] Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, and Christos Kozyrakis. Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 1–17, New York, NY, USA, 2021. Association for Computing Machinery.

[61] Nataniel Ruiz, Yuanzhen Li, Varun Jampani, Yael Pritch, Michael Rubinstein, and Kfir Aberman. Dreambooth: Fine tuning text-to-image diffusion models for subject-driven generation. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Vancouver, BC, Canada, June 17-24, 2023*, pages 22500–22510. IEEE, 2023.

[62] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: Workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[63] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018.

[64] ShareGPT. ShareGPT: Share your wildest ChatGPT conversations with one click., 2023.

[65] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 322–337, New York, NY, USA, 2019. Association for Computing Machinery.

[66] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

[67] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[68] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, jul 1985.

[69] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016.

[70] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 1–9. IEEE Computer Society, 2015.

[71] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.

[72] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *CoRR*, abs/1905.11946, 2019.

[73] Mingxing Tan and Quoc V. Le. Efficientnetv2: Smaller models and faster training. *CoRR*, abs/2104.00298, 2021.

[74] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. Cutlass, 1 2023.

[75] Muhammad Tirmazi, Adam Barker, Nan Deng, Md Ehtesam Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *EuroSys'20*, Heraklion, Crete, 2020.

[76] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[77] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[78] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[79] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In

Proceedings of the European Conference on Computer Systems (EuroSys), Bordeaux, France, 2015.

[80] Luping Wang, Lingyun Yang, Yinghao Yu, Wei Wang, Bo Li, Xianchao Sun, Jian He, and Liping Zhang. Morphling: Fast, near-optimal auto-configuration for cloud-native model serving. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 639–653, New York, NY, USA, 2021. Association for Computing Machinery.

[81] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, apr 2009.

[82] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[83] Xun Wu, Shaohan Huang, and Furu Wei. MoLE: Mixture of loRA experts. In *The Twelfth International Conference on Learning Representations*, 2024.

[84] Guangxuan Xiao, Ji Lin, Mickaël Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 38087–38099. PMLR, 2023.

[85] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *CoRR*, abs/2309.17453, 2023.

[86] Zihao Ye. FlashInfer: Kernel Library for LLM Serving. https://github.com/flashinfer-ai/flashinfer, 2023.

[87] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. Sparsetir: Composable abstractions for sparse compilation in deep learning. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating*

Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023, pages 660–678. ACM, 2023.

[88] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.

[89] Peifeng Yu and Mosharaf Chowdhury. Fine-grained GPU sharing primitives for deep learning applications. In Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze, editors, *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020.

[90] Elad Ben Zaken, Yoav Goldberg, and Shauli Ravfogel. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 1–9. Association for Computational Linguistics, 2022.

[91] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for Cost-Effective, SLO-Aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, Renton, WA, July 2019. USENIX Association.

[92] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 787–808, Boston, MA, April 2023. USENIX Association.

[93] Qingru Zhang, Minshuo Chen, Alexander Bukharin, Pengcheng He, Yu Cheng, Weizhu Chen, and Tuo Zhao. Adaptive budget allocation for parameter-efficient fine-tuning. In *The Eleventh International Conference on Learning Representations*, 2023.

[94] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models, 2022.

[95] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark W. Barrett, Zhangyang Wang, and

Beidi Chen. H_2o: Heavy-hitter oracle for efficient generative inference of large language models. *CoRR*, abs/2306.14048, 2023.

[96] Mengjie Zhao, Tao Lin, Fei Mi, Martin Jaggi, and Hinrich Schütze. Masking as an efficient alternative to finetuning for pretrained language models. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 2226–2241. Association for Computational Linguistics, 2020.

[97] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, Carlsbad, CA, July 2022. USENIX Association.

[98] Zhe Zhou, Xuechao Wei, Jiejing Zhang, and Guangyu Sun. PetS: A unified framework for Parameter-Efficient transformers serving. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 489–504, Carlsbad, CA, July 2022. USENIX Association.

[99] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. *CoRR*, abs/1707.07012, 2017.

Appendix A

APPENDIX FOR SYMPHONY

A.1 Global Partitioning Algorithm

We first present details on the MILP optimization formulation that we use to partition the models across sub-clusters and clusters. We then provide an evaluation of the quality of approximated solutions to the MILP in the context of our system.

A.1.1 Model formulation

We want to partition m models into l sub-clusters. For each sub-cluster, the maximum dispatcher capability of request rate is R_{max}, and the maximum memory size of backends is S_{max}. Let x_{ij} represents whether model i is assigned to sub-cluster j. The request rate, static memory size, dynamic (runtime) memory size of model i are r_i, s_i, d_i respectively. The average request rate per sub-cluster is $\bar{R} = \sum_i r_i/l$. Similarly, the average static memory size per sub-cluster is $\bar{S} = \sum_i s_i/l$. The partitioning problem can be formulated

as follows.

$$\begin{aligned}
&\text{minimize} && \Delta R + w\Delta S && && \text{(A.1)}\\
&\text{subject to} && \textstyle\sum_i r_i x_{ij} \le R_{\max} && \forall j && \text{(A.2)}\\
& && \textstyle\sum_i s_i x_{ij} + \max_i d_i x_{ij} \le S_{\max} && \forall j && \text{(A.3)}\\
& && \left|\textstyle\sum_i r_i x_{ij} - \bar{R}\right| \le \Delta R && \forall j && \text{(A.4)}\\
& && \left|\textstyle\sum_i s_i x_{ij} - \bar{S}\right| \le \Delta S && \forall j && \text{(A.5)}\\
& && \textstyle\sum_j x_{ij} = 1 && \forall i && \text{(A.6)}\\
& && x_{ij} \in \{0,1\} && \forall i,j && \text{(A.7)}
\end{aligned}$$

(A.6) and (A.7) represent that each model is assigned to exactly one sub-cluster. (A.2) requires that the request rate of each sub-cluster is less than the maximum dispatcher capability. (A.3) requires that the static memory size plus the maximum dynamic (runtime) memory size is less than the maximum memory size of backends. (A.4) and (A.5) denote that each sub-cluster's request rate and static memory size should be close to the averages. Therefore, our objective (A.1) is to minimize the differences to the averages as much as possible. The parameter w is a coefficient that balances the request rate and model memory size.

Disruption Minimization The above model can generate an initial assignment when the system starts. When the system is running, we would like to generate the next assignment considering the current assignment, so the disruption is minimized.

Given the current assignment matrix X' where x'_{ij} denotes whether model i is currently put into sub-cluster j. Let c_{ij} denote the cost for loading/unloading model i to/from sub-cluster j (assuming the cost is symmetric between loading and unloading). The maximum allowed cost is $C_{\max}$. Let y_{ij} represent whether there is a change for model i and sub-cluster

j as $y_{ij} = |x_{ij} - x'_{ij}|$. To bound the total cost of change within C_{max}, we have

$$\sum_i \sum_j c_{ij} y_{ij} \leq C_{\text{max}} \,. \tag{A.8}$$

Practical Implementation As solving general MILP problems is NP-hard, an optimal solution of the above model can not be obtained within a reasonable amount of time. Therefore, we need to use an approximated solution obtained within a time bound. Based on our testing with CPLEX solver and a running time of 10 seconds for $m = 400$ and $l = 4$, an approximate solution is significantly better than any randomly generated assignments.

A.1.2 Effectiveness of Cluster Partitioning

We next evaluate the effectiveness of the MILP partitioning model for partitioning the models and the workload across sub-clusters or clusters. We created a random solver as a baseline. The random solver will simply generate a random partition, check it against all the constraints, evaluate using the MILP objective, and repeat. It will output the partition with the least objective that satisfies all the constraints within a time constraint. Similarly, we use a solver to tackle the MILP problem and impose the same time limit for the solver (which is 10 seconds in our current system) to get the best solution so far.

We generate multiple configurations to represent different scenarios. Each configuration contains a random selection of models from our model zoo, where we generate multiple variants as specialized instantiations of a given model. We consider large-scale partitioning problems that partitions 800 models across 20 partitions. The request rate for each model is assumed to be independent and draw from an exponential distribution. We measure the goodness of a partition based on the imbalance factors for memory and load. The imbalance factor is defined as $\frac{\max - \min}{\text{avg}}$ across the different partitions generated by the algorithm. We compute the imbalance factor for both static memory and request rate across all dispatchers / subclusters. The smaller the imbalance metric, the better is the partition. Figure A.1 depicts the CDF of the imbalance factors for memory and load

across the different experiments. We observe that our MILP-based algorithm provides load-balanced partitions within our time window of 10 seconds, which makes it appropriate for epoch-level reconfigurations that happen every few minutes.

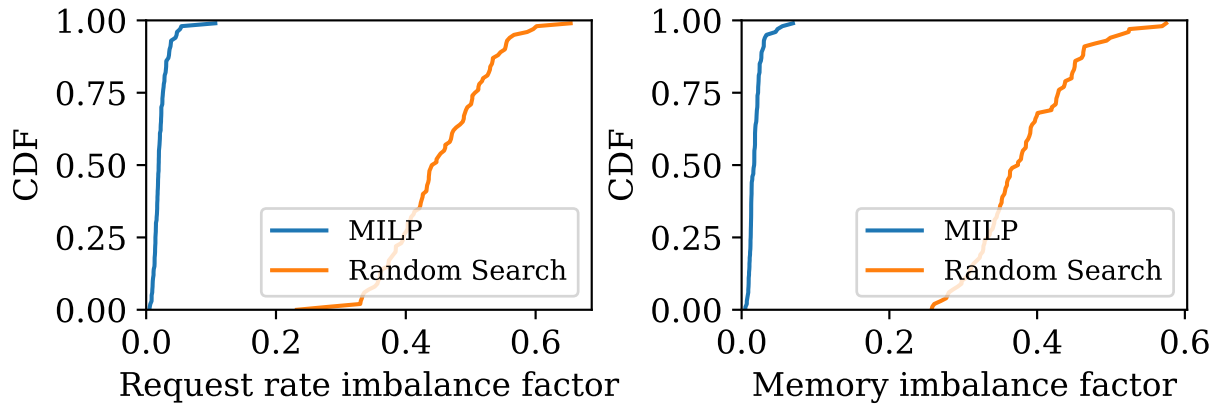


Figure A.1: Evaluating the effectiveness of MILP search for the partitioning problem

A.2 Networking Performance of Our Testbed

We study the performance difference between RDMA and TCP. We ran a network incast benchmark among eight servers. The incast behavior is a good representation of sending inputs of a batch from frontends to the executing backend. The benchmark concurrently reads eight objects of 150KB from each server. RDMA tail latency was measured on 56Gbps Infiniband. TCP tail latency was measured on 40Gbps Ethernet.

Figure A.2 shows the incast benchmark result. The lowest RDMA latency is within 24 μs, which is very close to the theoretical lower bound of 21.5 μs. The 99.99-th percentile latency in the RDMA network is within 33 μs. Thus RDMA network is both low-latency and highly predictable. TCP, on the other hand, is not only slower but also has a very long tail. The 99.99-th percentile latency is 12x the median.

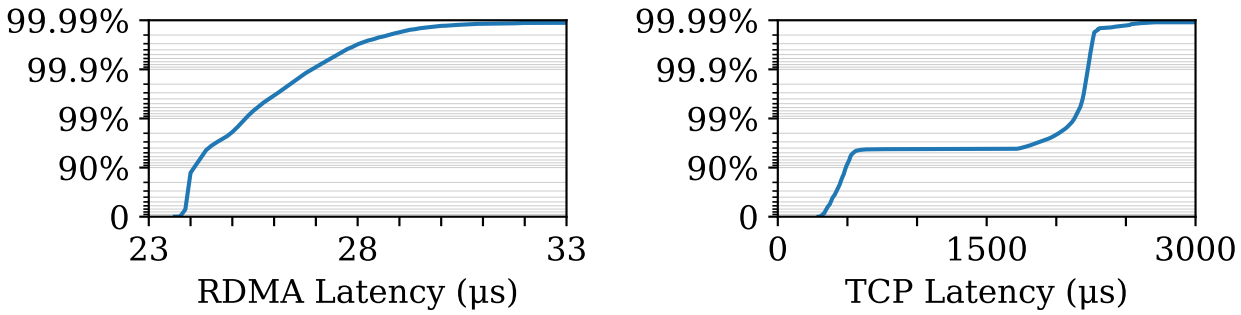


Figure A.2: RDMA and TCP tail latency

A.3 Model Zoo Details

Table A.1 and Table A.2 show the profiles of models used in the evaluation section for 1080Ti and A100 respectively. Latency SLO associated with each model ensures that each model can run with batch size greater than or equal to 4.

A.4 Scheduling Algorithm Pseudocode

Figure A.3 shows an extended version of Algorithm 1. The main differences are: (1) This version accounts for network latency. (2) This version includes the multicore scalability design.

Name	α (ms)	β (ms)	β/α	SLO
NASNetMobile	0.570	14.348	25.172	33ms
MobileNetV3Small	0.335	5.350	15.970	20ms
DenseNet169	1.271	13.618	10.714	37ms
DenseNet121	1.061	10.312	9.719	29ms
DenseNet201	1.733	15.687	9.052	45ms
EfficientNetV2B0	1.006	7.493	7.448	23ms
MobileNetV3Large	0.820	5.256	6.410	20ms
InceptionV3	1.964	8.771	4.466	33ms
EfficientNetV2B1	1.661	7.247	4.363	27ms
ResNet50V2	1.409	5.947	4.221	23ms
ResNet152V2	3.471	13.049	3.759	53ms
ResNet101V2	2.438	9.095	3.731	37ms
InceptionResNetV2	5.090	18.368	3.609	77ms
EfficientNetB0	1.569	5.586	3.560	23ms
MobileNetV2	1.180	3.483	2.952	20ms
ResNet101	3.164	9.065	2.865	43ms
EfficientNetB1	2.489	6.674	2.681	33ms
ResNet50	2.050	5.378	2.623	27ms
EfficientNetV2B2	2.254	5.896	2.616	29ms
VGG19	3.059	7.857	2.568	40ms
ResNet152	4.599	11.212	2.438	59ms
MobileNet	1.009	2.390	2.369	20ms
VGG16	2.734	5.786	2.116	33ms
EfficientNetB2	3.446	5.333	1.548	38ms
EfficientNetV2B3	4.072	5.981	1.469	44ms
NASNetLarge	17.656	18.952	1.073	179ms
EfficientNetV2S	8.463	8.862	1.047	85ms
EfficientNetB3	5.924	4.849	0.819	57ms
EfficientNetV2L	40.313	28.208	0.700	378ms
EfficientNetV2M	22.619	14.786	0.654	210ms
EfficientNetB5	23.435	10.301	0.440	208ms
Xception	4.751	2.046	0.431	42ms
SSDMobilenet	23.778	9.729	0.409	209ms
EfficientNetB4	12.088	4.412	0.365	105ms
BERT	7.008	0.159	0.023	56ms

Table A.1: Model profiles on an NVIDIA 1080Ti.

Name	α (ms)	β (ms)	β/α	SLO
DenseNet121	0.054	10.546	195.296	21ms
DenseNet201	0.304	14.345	47.188	31ms
DenseNet169	0.289	13.365	46.246	29ms
ResNet50V2	0.135	5.560	41.185	20ms
EfficientNetB0	0.115	4.326	37.617	20ms
ResNet101	0.284	8.266	29.106	20ms
ResNet152	0.390	10.449	26.792	24ms
ResNet101V2	0.391	8.219	21.020	20ms
MobileNetV3Large	0.196	4.072	20.776	20ms
EfficientNetB1	0.291	5.797	19.921	20ms
ResNet50	0.268	5.172	19.299	20ms
ResNet152V2	0.589	10.054	17.070	24ms
MobileNetV2	0.190	2.892	15.221	20ms
EfficientNetV2B3	0.543	7.596	13.989	20ms
InceptionResNetV2	1.112	15.27	13.732	39ms
EfficientNetV2B1	0.443	5.929	13.384	20ms
NASNetMobile	0.536	6.860	12.799	20ms
EfficientNetV2B0	0.377	4.272	11.332	20ms
EfficientNetB2	0.520	5.333	10.256	20ms
MobileNetV3Small	0.315	3.211	10.194	20ms
InceptionV3	0.913	6.732	7.373	20ms
MobileNet	0.285	1.901	6.670	20ms
EfficientNetV2S	1.454	7.378	5.074	26ms
EfficientNetV2B2	0.901	4.532	5.030	20ms
VGG16	0.660	2.252	3.412	20ms
EfficientNetB3	1.239	4.205	3.394	20ms
Xception	0.801	2.638	3.293	20ms
VGG19	0.893	2.181	2.442	20ms
NASNetLarge	3.464	7.154	2.065	42ms
EfficientNetV2M	4.479	6.861	1.532	49ms
EfficientNetB4	2.881	4.103	1.424	31ms
EfficientNetV2L	7.520	6.675	0.888	73ms
EfficientNetB5	6.121	2.283	0.373	53ms
SSDMobilenet	19.448	4.442	0.228	164ms
EfficientNetB6	9.754	1.984	0.203	82ms
EfficientNetB7	16.339	2.751	0.168	136ms
BERT	7.353	0.222	0.030	59ms

Table A.2: Model profiles on an NVIDIA A100.

```
struct Candidate: bs, exec_at, latest
def delay(bs): return d_ctrl + d_data * bs
class ModelThread:
  model: ModelID
  batch: BatchPolicy
  drop_timer: Timer
  c: Optional[Candidate]

  # Frontend -> ModelThread
  def schedule(req):
    batch.enqueue(req)
    update_candidate(-inf)
    rank_thread.inform_candidate(model, c)

  # RankThread -> ModelThread
  def granted_gpu(gpu, gpu_free_at):
    update_candidate(gpu_free_at)
    if c is None:
      free_at = max(now(), gpu_free_at)
    else:
      inputs = batch.pop_inputs()
      send(gpu, ExecutionMsg(inputs, c.exec_at))
      free_at = c.exec_at + exec_elapse(len(inputs))
      update_candidate(-inf)
    rank_thread.inform_gpu(gpu, free_at)
    rank_thread.inform_candidate(model, c)

  def update_candidate(gpu_free_at):
    batch.update_batch(gpu_free_at)
    bs = len(batch.inputs)
    if bs > 0:
      dl = batch.inputs[0].deadline
      drop_timer.cancel()
      drop_timer.set(dl, lambda: update_candidate(-inf))
      earliest = max(now()+delay(bs), gpu_free_at)
      frontrun_at = dl - exec_elapse(model, bs+1)
      exec_at = max(earliest, frontrun_at)
      latest = dl - exec_elapse(model, bs)
      c = Candidate(bs, exec_at, latest)
    else:
      c = None
    for req in batch.pop_dropped():
      send(req.frontend, QueryDroppedMsg(req))
```

```
class RankThread:
  gpu_free_at: Map[GpuID, TimePoint]
  gpu_timer: Map[GpuID, Timer]
  model_timer: Map[ModelID, Timer]
  mc: Map[ModelID, Candidate]

  # ModelThread -> RankThread
  def inform_candidate(m: ModelID, c: Optional[Candidate]):
    del mc[m]
    model_timer[m].cancel()
    if c is not None:
      model_timer[m].set(c.exec_at-delay(c.bs),
          lambda: on_model_timer(m, c))

  # ModelThread -> RankThread
  def inform_gpu(gpu: GpuID, free_at: TimePoint):
    gpu_free_at[gpu] = free_at
    set_gpu_timer()

  def on_model_timer(m: ModelID, c: Candidate):
    gpu, free_at = gpu_free_at.get_earliest()
    if free_at <= c.exec_at:
      gpu_free_at[gpu] = +inf
      model_thread[m].granted_gpu(gpu, free_at)
    else:
      mc[m] = c
      set_gpu_timer()

  def set_gpu_timer():
    gpu, free_at = gpu_free_at.get_earliest()
    gpu_timer[gpu].cancel()
    if len(mc) > 0:
      m, c = mc.get_by_max_delay()
      gpu_timer[gpu].set(free_at-delay(c.bs),
          lambda: on_gpu_timer(gpu))

  def on_gpu_timer(gpu: GpuID):
    free_at = gpu_free_at[gpu]
    Remove (m,c) from mc where free_at > c.latest
    if len(mc) > 0:
      m, c = mc.get_by_min_latest()
      model_thread[m].granted_gpu(gpu, free_at)
    set_gpu_timer()
```

Figure A.3: Pseudo-code of the scheduling algorithm.