AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving

Abstract

This paper introduces a novel model serving system, AlpaServe, leveraging model parallelism to enhance the efficiency and quality of multiple large-scale deep learning models. Model parallelism achieves statistical reuse by partitioning the model across multiple devices, reducing memory footprint, increasing service throughput, and addressing burst request loads. However, effective implementation of model parallelism necessitates consideration of diverse parallel strategies and configurations, as well as their intricate trade-offs with cluster resources, request arrival patterns, and service objectives.To tackle these challenges, AlpaServe proposes a set of algorithms and technologies, including an automated model parallel compiler for generating efficient parallel solutions tailored for inference. Additionally, a simulator-based greedy algorithm determines optimal model placement and scheduling strategies based on workload characteristics. An enumeration-based algorithm searches for the optimal grouping of clusters and model parallel configurations. AlpaServe's evaluation is conducted on a 64-GPU cluster using real production workloads. The results demonstrate that AlpaServe can achieve a 10x improvement in request processing rates while meeting latency constraints or a 6x enhancement in handling burst requests.

Keywords: Model parallelism , Serving system.

1 Introduction

The continuous growth in the scale and complexity of deep learning models poses significant challenges for online model serving. Particularly, efficiently utilizing limited computational and memory resources to simultaneously serve multiple large models while meeting user latency requirements and reducing service costs is a pressing issue. Existing model serving systems often adopt model replication, deploying a model on one or multiple devices to enhance throughput and fault tolerance. However, this approach fails to fully exploit statistical multiplexing between devices and cannot handle models exceeding the memory limits of a single device.

Model parallelism, a technique that partitions a model into multiple segments and executes them in parallel on distributed devices, presents a solution to these challenges. However, it introduces additional overheads such as communication and load imbalance. Therefore, flexibly employing model parallelism in model serving, selecting optimal partitioning and deployment strategies based on varying model characteristics, devices, and load features, is a valuable research topic.

This paper addresses the model parallelism issue in model serving by proposing a novel service system, AlpaServe [1]. AlpaServe automatically explores different model parallel strategies and dynamically adjusts model partitioning and deployment based on real-time workload conditions to maximize service quality. Through a detailed analysis of the advantages and costs of model parallelism, the paper reveals its applicability conditions and influencing factors in model serving, providing valuable guidance for its design. Experimental validations on real clusters and workloads demonstrate the effectiveness and superiority of AlpaServe, highlighting the immense potential of model parallelism in model serving.

2 Related works

2.1 Traditional model serving system

Traditional model serving solutions allocate one or more dedicated devices for each model to meet memory and latency requirements. To address large-scale model serving requests, systems replicate models across devices to enhance throughput and fault tolerance. Upon request arrival, load balancing strategies distribute requests to different devices based on the model's workload.

2.2 Model service based on model variants

The system achieves model service through the generation of model variants and traverses a vast trade space of these variants. This approach allows developers to specify only the performance and accuracy requirements of the application without the need to specify particular model variants. By leveraging diverse variants and sharing hardware resources across models, the system attains higher throughput, fewer instances of violating latency targets, and lower costs compared to existing inference service systems.

3 Method

3.1 Overview

This paper proposes leveraging model parallelism to enhance the efficiency and quality of model serving. Specifically, the approach involves partitioning a model across multiple devices, thereby reducing the memory footprint and computational load on individual devices. Additionally, it capitalizes on the effects of statistical reuse to address burst requests. This method not only improves GPU utilization but also enhances the throughput of the model serving system.

The optimization of the model serving system through model parallelism is exemplified using Figure 2, which illustrates the case of two models and two GPUs. This figure serves to explain why model parallelism is advantageous for optimizing the model serving system.

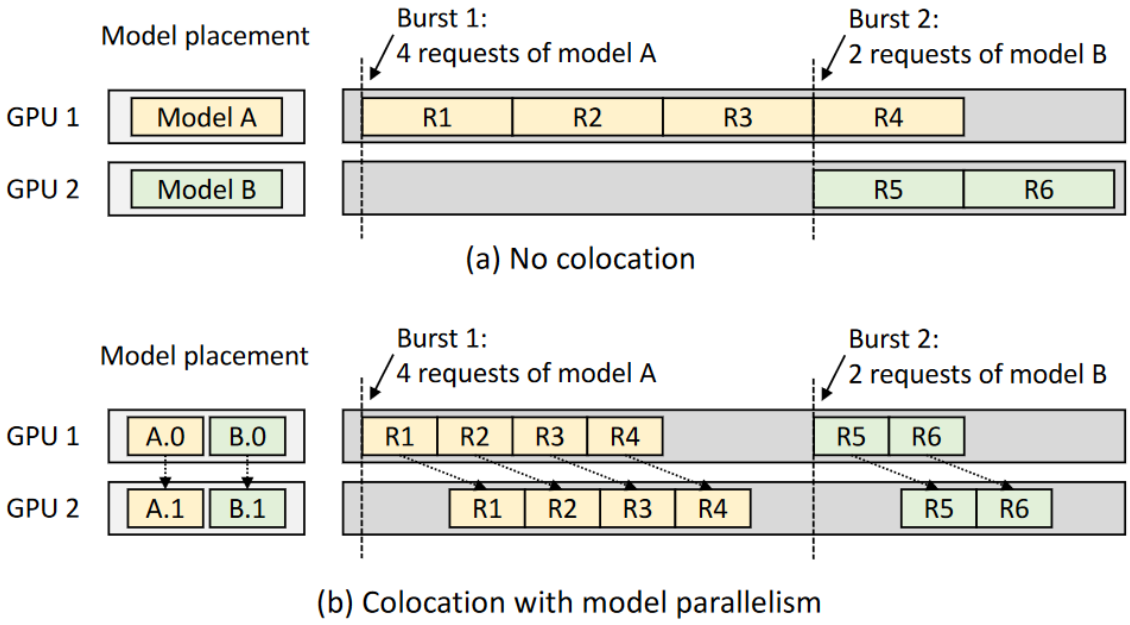

Figure 1. Overview of the method

3.2 Automatic parallelization compiler

This paper introduces an automatic parallelization compiler designed to generate various possible parallelization configurations based on different combinations of device groups and models. The compiler encompasses two types of parallelization: internal operation parallelization and external operation parallelization. Internal operation parallelization involves partitioning an operation for parallel execution across multiple devices, while external operation parallelization divides the execution graph of a model into multiple stages, executing them in a pipelined fashion on different devices.

3.3 Simulator-guided greedy algorithm

This paper introduces a simulator-guided greedy placement algorithm designed to find an optimal placement strategy based on a given device cluster, model set, and workload. The algorithm aims to determine the optimal division of devices into multiple groups, the selection of model replicas for each group, and the choice of parallelization configurations for each model. Utilizing a simulator, the algorithm evaluates different placement strategies concerning their achievement rates of Service Level Objectives (SLOs) and selects a strategy that maximizes the SLO achievement rate.

4 Implementation details

4.1 Comparing with the released source codes

In the replication process of this study, Alpa's open-source code [2] and AlpaServe's open-source code were referenced. Alpa's open-source code was utilized to generate various model parallel configuration files, documenting runtime, memory usage, and other parameters for different models under diverse parallel configurations. Subsequently, the AlpaServe open-source code was employed to enhance the current work presented

in this paper.

The open-source code is based on generating model placement strategies using a global workload, without adjusting these strategies according to the current and future workloads. This limitation results in underutilization of system resources, leading to inefficient resource allocation and increased service costs.

The improved code is designed to flexibly adjust model placement strategies based on local workloads. Under each local workload, model selection and parallel placement strategies are reconsidered to maximize model throughput. By dynamically adapting model placement strategies according to the current workload, the improved code aims to enhance system throughput further.

4.2 Experimental environment setup

Enter the following commands.
'git clone https://github.com/alpa-projects/mms.git'

Install alpa_ serve package by running
'pip install -e .'
in the root folder of mms project.

Launch the ray runtime
'ray start –head'
Now you are ready to reproduce all the main results in the paper.

To get and use Azure Function Trace Dataset, read this instruction.

Our algorithm relies on the profiling results, which is provided as profiling_ result.pkl in this issue. If you want reproduce the profiling results yourself, please follow this benchmarking script and this conversion script.

4.3 System design

The operational workflow of a model serving system typically involves several steps:

Client Request: The client sends a request to the service system, which includes the data requiring inference.

Load Balancing: The service system's load balancer receives the request and allocates it to the appropriate service node.

Model Loading: The selected service node loads the necessary deep learning model into memory.

Inference Execution: The service node performs model inference, processing the input data to generate prediction results.

Result Return: The service node returns the inference results to the client.

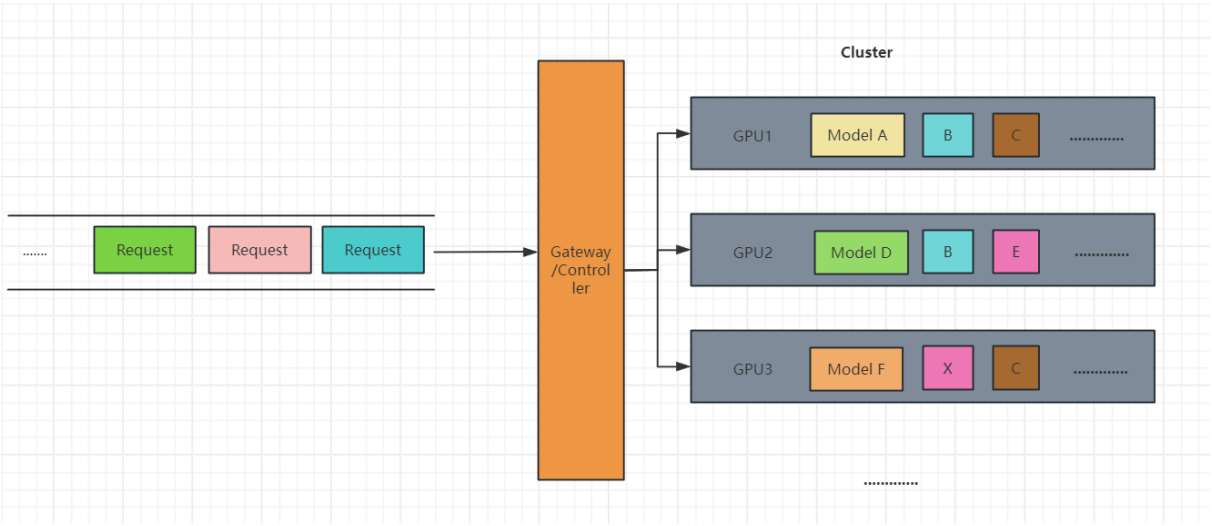

Figure 2. Model serving system

4.4 Main contributions

By periodically adjusting the model placement strategy, it is possible to enhance the system's throughput for request processing. The improved model serving system achieves a throughput approximately 1.2 times higher than that of the previous system.

5 Results and analysis

5.1 Experiment environment

Experiment Base: Ubuntu 20.04 devices: 4*3090 Python:3.9.15

Experiment parameters: Models , # devices , SLO Scale , Workload , Rate Scale , CV Scale

Model Sets:
S1(64 * 2.4 GB Bert Models)
S2(8 * 13.4 GB Bert Models)
S3(4 * [2.4 GB 5.4 GB 2.6 GB 4.8 GB]Bert Models and MoE)

Workload: Microsoft Azure function trace 2019 (MAF1) and 2021 (MAF2)

Experiment: In this experiment, six scenarios (S1@MAF1, S2@MAF1, S3@MAF1, S1@MAF2, S2@MAF2, S3@MAF2) were configured under a single GPU memory constraint of 20GB, with other parameters held constant as specified in the corresponding code. The experiment aimed to assess the system's maximum throughput under varying conditions, including changes in the number of GPUs, service level objectives, model request rates, and coefficient of variation. The testing evaluated the maximum achievable throughput under these conditions, denoted as "before" for the original throughput and "after" for the improved throughput.

5.2 Experiment result

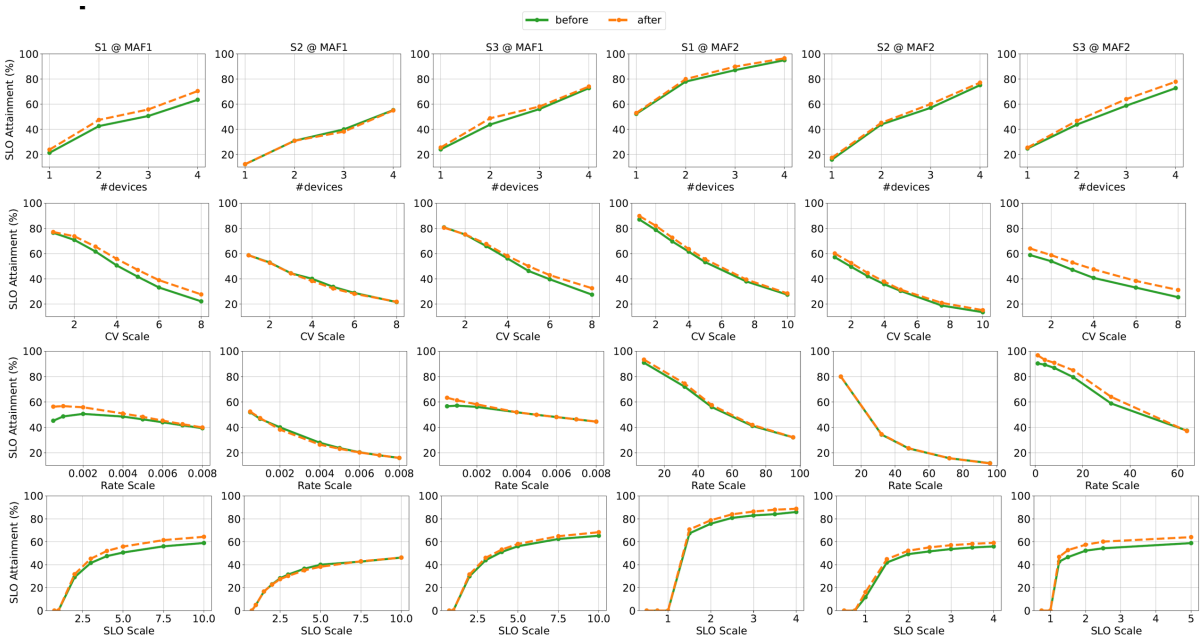

Figure 3. Experimental results

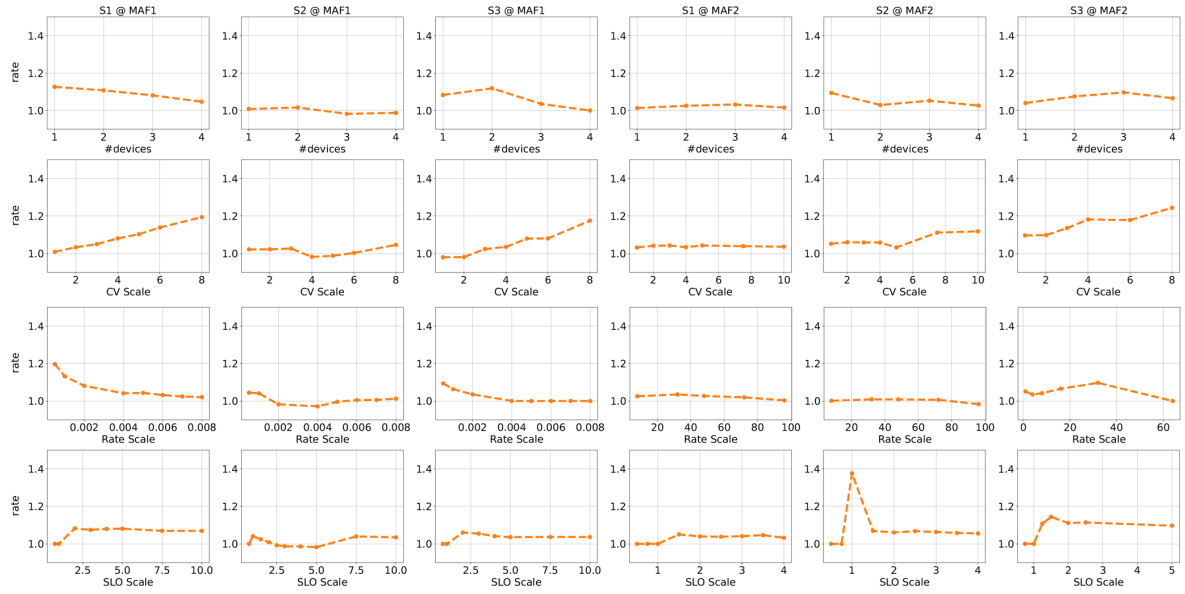

Figure 4. Result Rate

5.3 Experiment analysis

From the figure, it is evident that when the model size is relatively small and a single GPU can accommodate numerous models, periodically switching model placement strategies by unloading models with anticipated lower future throughput and loading higher throughput models into the system enhances overall system throughput. Despite incurring time delay costs due to model switching, the increase in throughput far outweighs the reduction in latency, resulting in a net improvement in overall system throughput.

However, when the model size becomes excessively large, causing each GPU to accommodate only a minimal number of models, employing a periodic model scheduling strategy may not be advantageous. This is primarily because scheduling large models can introduce significant switching time delay costs, during which numerous model requests may go unprocessed. The benefits of model scheduling might be overshadowed or even counter productive during this period.

6 Conclusion and future work

Scheduled modifications to model placement strategies ensure a balanced workload distribution in the model serving system. By dynamically adjusting the deployment locations based on real-time load conditions, instances of overloading and underutilization can be mitigated, thereby improving the overall system performance.

The current improvement has not fully unlocked the potential of adjusting model placement strategies. Future enhancements could involve flexibly and non-periodically adjusting model placement strategies based on workload predictions. Identifying optimal times for modifying model placement strategies according to workload forecasts can further boost the throughput of the model serving system. Additionally, incorporating request preemption strategies into the model serving system, which adjusts the order of service requests based on preemption policies, represents another avenue for enhancing system throughput.

References

[1] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {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, 2023.

[2] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. 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, 2022.