AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving

Zhuohan Li and Lianmin Zheng, *UC Berkeley;* Yinmin Zhong, *Peking University;*
Vincent Liu, *University of Pennsylvania;* Ying Sheng, *Stanford University;*
Xin Jin, *Peking University;* Yanping Huang and Zhifeng Chen, *Google;*
Hao Zhang, *UC San Diego;* Joseph E. Gonzalez and Ion Stoica, *UC Berkeley*

https://www.usenix.org/conference/osdi23/presentation/li-zhuohan

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Open access to the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving

Zhuohan Li[1,*] Lianmin Zheng[1,*] Yinmin Zhong[2,*] Vincent Liu[3] Ying Sheng[4]
Xin Jin[2] Yanping Huang[5] Zhifeng Chen[5] Hao Zhang[6] Joseph E. Gonzalez[1] Ion Stoica[1]

[1]*UC Berkeley* [2]*Peking University* [3]*University of Pennsylvania*
[4]*Stanford University* [5]*Google* [6]*UC San Diego*

Abstract

Model parallelism is conventionally viewed as a method to scale a single large deep learning model beyond the memory limits of a single device. In this paper, we demonstrate that model parallelism can be additionally used for the statistical multiplexing of multiple devices when serving multiple models, even when a single model can fit into a single device. Our work reveals a fundamental trade-off between the overhead introduced by model parallelism and the opportunity to exploit statistical multiplexing to reduce serving latency in the presence of bursty workloads. We explore the new trade-off space and present a novel serving system, AlpaServe, that determines an efficient strategy for placing and parallelizing collections of large deep learning models across a distributed cluster. Evaluation results on production workloads show that AlpaServe can process requests at up to 10× higher rates or 6× more burstiness while staying within latency constraints for more than 99% of requests.

1 Introduction

Advances in self-supervised learning have enabled exponential scaling in model sizes. For example, large pretrained models like BERT [14] and GPT-3 [5] have unlocked a plethora of new machine learning (ML) applications from Copilot [18] to copy.ai [7] and ChatGPT [35].

Serving these very large models is challenging because of their high computational and memory requirements. For example, GPT-3 requires 325 GB of memory to store its parameters as well as a requisite amount of computation to run inference. To serve this model, one would need at least 5 of Nvidia's newest Hopper 80 GB GPUs just to hold the weights and potentially many more to run in real-time. Worse yet, the explosive growth of model sizes continues unabated [6, 17]. Techniques like model compression and pruning are not sufficient in face of the exponential growth in model sizes and often come at the expense of reduced model quality [15].

*Equal contribution.

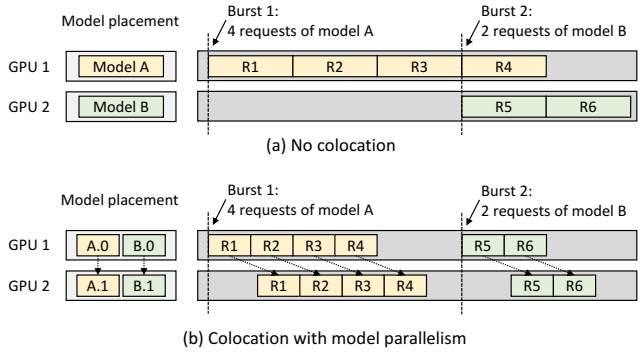


Figure 1: Two placement strategies for serving two models on two GPUs. In each subfigure, the left part shows the model placements and the right part shows the timeline for handling bursty requests. At the time of "Burst 1", 4 requests of model A come at the same time. Colocation with model parallelism can reduce the average completion time of bursty requests.

Provisioning sufficient resources to serve these models can be arduous as request rates are bursty. For example, using common workload traces, we observe frequent spikes in demand of up to 50× the average [54]. Meeting the service level objective (SLO) of latency usually means provisioning for these peak loads, which can be very expensive; additional devices allocated for this purpose would remain underutilized most of the time. Making matters worse, it is increasingly common to serve multiple models and multiple variations of the same large model in situations like A/B testing or serving fine-tuned models for specific domains (§2).

This paper studies how to efficiently serve multiple large models concurrently. Specifically, we explore the underappreciated benefits of model parallelism in online model serving, even for smaller models that can fit on a single device. Model parallelism refers to partitioning and executing a model on distributed devices (§2.1). The benefits of model parallelism have been well studied [23, 27, 31, 56] in the *throughput-oriented* training setting. However, its effects for model serving under *latency-sensitive* settings remains largely untapped.

We observe that there are fundamental transition points in

the model serving design space that challenge prior assumptions about serving, even for models that fit on a single device. For example, consider the scenario with two models and two GPUs, each of which has sufficient memory to hold one complete model. As shown in Fig. 1(a), the natural approach assumed by almost all existing serving systems [9, 33, 34] is to allocate one dedicated GPU for one model. This approach appears rational because partitioning the model across GPUs would incur communication overheads that would likely increase the prediction latency. However, we find that inducing additional model parallelism (to the point where per-example execution time actually *increases*) enables a wider range of placement strategies, e.g., model co-location, which can improve the statistical multiplexing of the system under bursty workloads. In Fig. 1(a), assuming the execution time of a model is y, the average end-to-end latency of request 1 through 4 is $(1y+2y+3y+4y)/4 = 2.5y$. In Fig. 1(b), assuming a 10% model-parallel overhead, the average latency of request 1 through 4 is reduced to $(1.1y+1.6y+2.1y+2.6y)/4 = 1.85y$. Co-location with model parallelism can utilize more devices to handle bursty requests and reduces the average completion time, despite its overheads (§3.1). Even if we batch the requests, the case still holds (§6.5).

Unfortunately, the decision of how to optimally split and place a collection of models is complex. Although leveraging model parallelism as above has its benefits, it still adds overheads that may negate those benefits for less bursty workloads. For example, we find that a particularly influential axis on the efficacy of model parallelism is per-GPU memory capacity (§3.2), although other factors (e.g., the arrival pattern, SLO) can also have a significant effect. Further, besides the inter-op model parallelism presented in Fig. 1, another kind of model parallelism, intra-op parallelism, presents its own distinct tradeoffs (§3.3). Ultimately, different styles of parallelism and their tradeoffs create a complex, multi-dimensional, and multi-objective design space that existing systems largely ignore and/or fail to navigate. However, not leveraging model parallelism in the serving setting is typically not an option for large models, and not addressing this trade-off space directly results in significant increases in cost and serving latency.

To that end, we present AlpaServe[2], a system that automatically and efficiently explores the tradeoffs among different parallelization and placement strategies for model serving. AlpaServe takes a cluster resource specification, a set of models, and a periodic workload profile; it then partitions and places the models and schedules the requests to optimize SLO attainment (i.e., the percentage of requests served within SLO). To assist the design of AlpaServe, we first introduce a taxonomy and quantify the tradeoffs between different parallelization strategies in model serving (§3). We then present key algorithms to navigate the tradeoff space (§4). We design an iterative simulator-guided model placement algorithm to optimize the colocation of models and a group partition algorithm to search for the best way to partition the cluster into disjoint model-parallel groups. In addition, we extend the existing auto-parallelization algorithms for training to make them more suitable for inference.

We evaluate AlpaServe with production workloads on a 64-GPU cluster (§6). Evaluation results show that, when optimizing one metric at a time, AlpaServe can choose to increase the request processing rate by 10×, achieve 2.5× lower latency deadlines, or tolerate 6× burstier traffic compared to previous state-of-the-art serving systems.

In summary, we make the following contributions:

- A detailed analysis of the tradeoff space of different model parallel strategies for efficient model serving.
- Novel model placement algorithms to incorporate model parallelism in a serving system.
- A comprehensive evaluation of AlpaServe with both synthetic and production workloads.

2 Background

Over the past few years, increasingly capable models have been developed for everything from recommendations to text generation. As a result, serving predictions from these models has become an essential workload in modern cloud systems. The structure of these workloads often follows a simple request-response paradigm. Developers upload a pre-trained model and its weights ahead of time; at runtime, clients (either users or other applications) submit requests for that model to a serving system, which will queue the requests, dispatch them to available GPUs/TPUs, and return the results.

The requirements of these model-serving systems can be stringent. To satisfy user demand, systems often must adhere to aggressive SLO on latency. At the same time, serving systems that must run continuously need to minimize their operational costs associated with expensive accelerators. Minimizing serving costs can be challenging because dynamically scaling compute resources would be too slow on the critical path of each prediction request: it can take multiple seconds just to swap a large model into accelerator memory [37]. Furthermore, there is significant and unpredictable burstiness in the arrival process of user requests. To meet tight SLO, contemporary serving systems are forced to over-provision compute resources, resulting in low cluster utilization [48].

Another pattern that emerges in serving large models is the use of multiple instances of the same or similar model architectures. This is commonly seen in the practice of pretraining on large unlabeled data and fine-tuning for various downstream tasks [14], which can significantly boost accuracy but results in multiple instances of the same model architecture. For example, Hugging Face serves more than 9,000 versions of fine-tuned BERT [24]. They either share a portion of the parameters or do not share any parameters at all for better accuracy. Prior works have [44, 57] exploited the property of shared parameters, but we do not consider the shared parame-

[2] https://github.com/alpa-projects/mms

ters in this paper because AlpaServe targets general settings and full-weight tuning is still a major use case.

2.1 Model Parallelism in Model Serving

Distributed parallel model execution is necessary when attempting to satisfy the serving performance requirements or support large models that do not fit in the memory of a single device. At a high level, distributed execution of deep learning models can be classified into two categories: intra-operator parallelism and inter-operator parallelism [56].

Intra-operator parallelism. DL models are composed of a series of operators over multidimensional tensors, e.g., matrix multiplication over input and weight tensors. Intra-operator parallelism is when a single operator is partitioned across multiple devices, with each device executing a portion of the computation in parallel [43, 45, 50]. Depending on the specific partitioning strategy and its relationship to prior and subsequent operators in the model, partitioning can require communication among participating GPUs to split the input and then merge the output.

The benefit of intra-operator parallelism for single-request execution is twofold. First, it can expand the total amount of computation available to the target model, reducing its end-to-end latency. In a similar fashion, it can expand the total memory available to the model for storing its inputs, weights, and intermediate values. The cost is the aforementioned communication overhead.

Inter-operator parallelism. The other type of parallelism available to DL models is inter-operator parallelism, which assigns different operators of the model's execution graph to execute on distributed devices in a pipeline fashion (a.k.a. pipeline parallelism) [23, 28, 30]. Here, devices communicate only between pipeline stages, typically using point-to-point communication between device pairs.

Unlike intra-operator parallelism, pipeline parallelism does not reduce the execution time of a single request. In fact, it typically increases the execution time due to modest amounts of communication latency between pipeline stages, although the total amount of transferred data is often lower than it is in intra-operator parallelism. Instead, the primary use of inter-operator parallelism in traditional serving systems is to allow the model to exceed the memory limitation of a single GPU.

3 Motivation and Tradeoff Analysis

As mentioned, both types of model parallelism reduce per-device memory usage by partitioning a model on multiple devices. A key motivation for this work is that we can use this property to fit more models on one device, enabling better statistical multiplexing of the devices when handling bursty requests. We explore this idea through a series of empirical examinations and theoretical analysis, starting with an illustrative example (§3.1), followed by an empirical analysis of when model parallelism is beneficial (§3.2), the overhead of model parallelism (§3.3), and a queueing theory-based analysis (§3.4). All the experiments in this section are performed on an AWS EC2 p3.16xlarge instance with 8 NVIDIA 16GB V100 GPUs.

3.1 Case Study: A Two-model Example

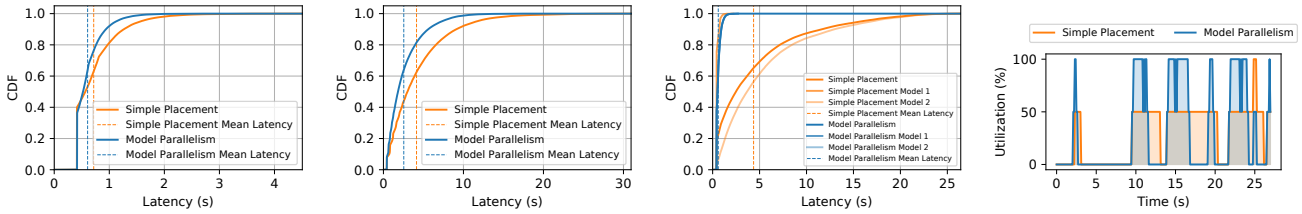
We start with an illustrative experiment to show how model parallelism can benefit the serving of multiple models. We use two GPUs to serve two Transformer models with 6.7 billion parameters each (13.4 GB to store its FP16 weights). Because each GPU has 16 GB of memory, it can fit one and only one model. A single request takes around 0.4 s to process on one GPU.

We compare the following model placements, corresponding to the strategies in Fig. 1. The first is *simple placement*, where we place one model on each GPU due to the memory constraint. The second is *model-parallel placement*, where we use inter-op parallelism to partition each model to a 2-stage pipeline and let each GPU execute half of each model.

We evaluate the two placements when the requests to each model follow an independent Poisson process with an arrival rate of 1.5 request/s. Fig. 2a shows the cumulative distribution function (CDF) and average of request latency (which includes the GPU execution time and queuing delay). Model-parallel placement reduces the average latency of the simple placement from 0.70s to 0.55s, a 1.3× speedup. The speedup comes from the better burst tolerance: when a burst arrives that exceeds the capability of a single GPU, simple placement must begin queuing requests. However, as long as the other model does not receive many requests, the model parallel placement can use both GPUs to serve the requests for the popular model via statistical multiplexing of the GPUs.

This effect becomes more pronounced with higher burstiness, which we can demonstrate using a Gamma request arrival process with the same average request rate as above but a higher coefficient of variance (CV) of 3. As shown in Fig. 2b, the speedup on mean latency is now increased to 1.9×. Fig. 2d shows a representative trace of the corresponding total cluster utilization over time. Note that for each request burst, model-parallel placement can use the whole cluster and only take half of the time to process, while simple placement can only use half of the cluster.

In addition, we also evaluate the case where one model receives more requests than another. In Fig. 2c, we use Poisson arrival but let 20% of the requests ask for model 1 and 80% ask for model 2. Although replication performs slightly better for model 1 requests, it is drastically worse on model 2 requests compared to the model-parallel placement. For model-parallel placement, because both GPUs are shared across two models, the requests to both models follow the same latency distribution. Overall, model-parallel placement reduces the mean latency by 6.6×.



(a) Poisson arrival. (b) High CV Gamma arrival. (c) Different rates. (d) Cluster utilization.

Figure 2: Latency CDF and cluster utilization in the 2-model example.

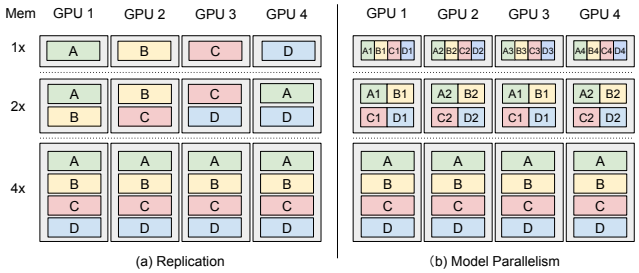


Figure 3: Replication and model parallel placement illustration with different memory budgets, where the memory budgets are set to be multiples of a single model's size.

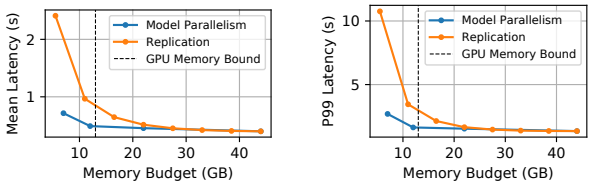


Figure 4: Serving performance with changing per-GPU memory budgets. Model parallelism is beneficial for limited memory budget. The dashed vertical line is the real per-GPU memory bound of a 16GB V100. The value is around 13GB due to the need to store activations and other runtime context.

3.2 When is Model Parallelism Beneficial

To further explore the nuances of model parallelism in serving, we increase the size of the deployment to 8 GPUs and 8 Transformer models with 2.6B parameters each. As a base setting, we set the requests to each model as a Gamma process with an average rate of 20 request/s and CV of 3; we then vary a range of factors to see their effects. Note that some of the settings we evaluate are impossible on real hardware (e.g., exceeding the memory capacity of a single device) so we leverage the simulator introduced in §5. The fidelity of the simulator is very high as verified in §6.1.

The model in this case is smaller (5.2GB), so one GPU can also store multiple models without model parallelism. We compare two placement methods: (1) *Replication.* In this setting, we replicate the models to different devices until each device cannot hold any extra models. Because all the models receive equal amounts of loads on average, we replicate each model the same number of times (Fig. 3a). (2) *Model Parallelism.* Here we use inter-operator parallelism and uniformly assign the Transformer layers to different GPUs.

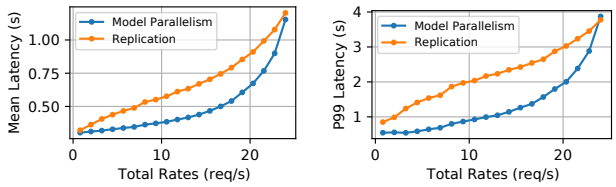


Figure 5: Serving performance with changing arrival rates. Model parallelism is beneficial for smaller rates.

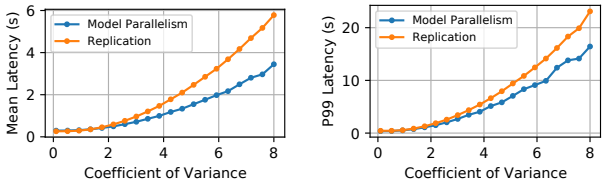
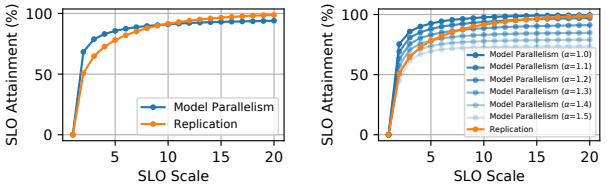


Figure 6: Serving performance with changing CVs. Model parallelism is beneficial for larger CVs.

Device memory. We evaluate the mean and the tail latency of the two placement methods under different device memory capacities. For replication, more GPU memory can fit more models onto a single GPU. For model parallelism, more GPU memory can also reduce the number of pipeline stages and reduce the overhead as in Fig. 3b. The resulting mean and P99 latency is shown in Fig. 4. With more memory, more models can fit into a single GPU, so the benefit of statistical multiplexing diminishes because replication can also effectively use multiple devices to serve the bursty requests to a single model. When the GPU memory capacity is large enough to hold all models, there is no gain from model parallelism.

Request arrival. We vary the parameters of the arrival process and compare the replication placement with the model-parallel placement with 8-stage pipeline parallelism. The mean and P99 latency results of changing arrival rate are shown in Fig. 5. When the arrival rate is low, model parallelism can greatly reduce the serving latency. However, when the arrival rate approaches the peak serving rate of the cluster, the benefit of model-parallel placement starts to diminish. Eventually, it starts to perform worse than replication. This is because when all models are equally saturated, the replication



(a) Real model latency. (b) Changing overhead.

Figure 7: SLO attainment with changing SLOs. Model parallelism is beneficial for smaller SLOs.

placement is able to achieve efficient cluster utilization and there is no benefit to the statistical multiplexing afforded by model parallelism. Instead, the overhead of model parallelism (§3.3) starts to become a significant factor.

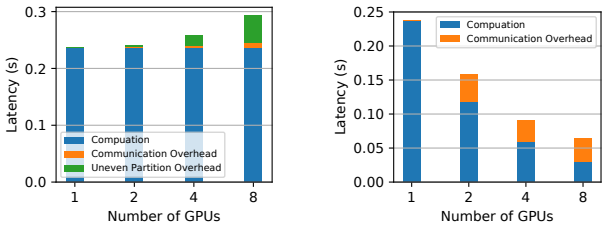
The mean and P99 latency results of changing CV are in Fig. 6. With a higher CV, the requests become more bursty, and the benefit of model parallelism becomes more significant. As shown in the results, with a higher CV, model parallelism can greatly outperform the performance of replication.

Service level objectives. In prediction serving settings, it is common to have tight latency SLO and predictions made after these deadlines are often discarded [19]. For example, advertising systems may choose not to show an ad rather than delay rendering user content. In this case, the goal of the serving system is to optimize the percentage of requests that can be finished within the deadline, i.e., *SLO attainment*.

In this experiment, we measure how SLOs affect the performance of the placement methods. We compare the replication and the model-parallel placement with 8-stage pipeline parallelism. During execution, we drop the requests that will exceed the deadline even if we schedule it immediately. We scale the SLO to different multiplies of the single device execution latency (*SLO Scale* in Fig. 7a) and compare the SLO attainment of the two methods.

As in Fig. 7a, when SLO is tight ($< 10\times$ model latency), model parallelism can greatly improve SLO attainment. However, when the SLO becomes looser, its SLO attainment plateaus but that of the replication placement keeps growing. This result shares the same core logic as previous experiments: When SLO becomes looser, more requests can stay in the waiting queue, and thus the effective burstiness of the requests decreases. When many requests are queued, the system is bounded by its total processing capability, which might be affected by the model parallelism overhead. In the real world, the SLO requirement is often less than $5\times$ of the model execution latency [19], where model parallelism can improve SLO attainment.

Summary: Model parallelism benefits model serving through statistical multiplexing when the device memory is limited, the request rate is low, the request CV is high, or the SLO is tight.



(a) Inter-op parallelism. (b) Intra-op parallelism.

Figure 8: The overhead decomposition. The overhead of inter-op parallelism mainly comes from uneven partition while the overhead of intra-op parallelism comes from communication.

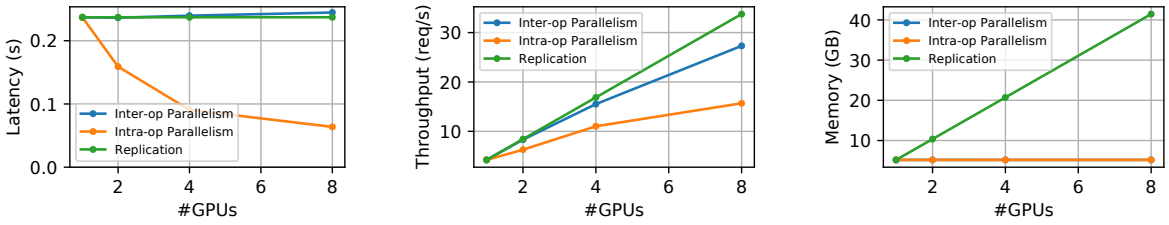
3.3 Overhead of Model Parallelism

In this section, we further investigate the overheads of different model parallel strategies and how they affect serving performance. Similar to the setup in Fig. 7a, we manually modify the overhead of model parallelism. Specifically, let the latency of a single model executing on the GPU be L and the number of pipeline stages be n. We set the total latency of pipeline execution to be αL and the latency of each pipeline stage to be $\alpha L/n$, where α is a parameter that controls the overhead. When $\alpha = 1$, model parallelism does not have any overhead and larger α means higher overhead.

We show the results in Fig. 7b. If model parallelism does not have any overhead ($\alpha = 1$), it can always outperform replication due to its ability to multiplex the devices. When the overhead becomes larger and the SLO is low, model parallelism still outperforms replication. However, with a larger SLO, the effective burstiness is reduced and the performance is dominated by the overhead.

Given that the overhead can greatly affect serving performance, we perform a detailed study of the multiple sources of model-parallel overhead in Fig. 8. For inter-op parallelism, when partitioning a single model into multiple stages, different stages need to communicate the intermediate tensors, and we denote this overhead as the *communication overhead*. In addition, the pipeline execution will be bottlenecked by the execution time of the slowest stage, making the effective latency to be the number of pipeline stages times the latency of the slowest stage [23]. We denote this as the *uneven partition overhead*. As in Fig. 8a, for inter-op parallelism, most overhead comes from the latency imbalance among different pipeline stages, instead of the communication between stages. While our previous discussion mainly focuses on inter-op parallelism, the other type of model parallelism, intra-op parallelism, has very different performance characteristics. Its overhead is merely brought by the collective communication across multiple devices [31], which cannot be overlapped with the neural network computation due to data dependency. From Fig. 8b, we can see that the communication overhead of intra-op parallelism is much higher than inter-op parallelism.

Finally, we compare the latency, throughput, and memory consumption of different model-parallel placements and the



(a) Single input latency. (b) Throughput. (c) Total memory usage.

Figure 9: The latency, throughput and memory usage vs. #GPUs for inter-op parallelism, intra-op parallelism, and replication. In subfigure (c), the lines for inter-op and intra-op parallelism overlap.

replication placement in Fig. 9. Because of the sequential dependence between the different stages, inter-op parallelism cannot reduce the execution latency of a single input data. Instead, the latency is slightly higher due to the communication between the stages. On the other hand, intra-op parallelism can largely reduce the latency via the parallel execution of different GPUs (Fig. 9a). However, because inter-op parallelism can pipeline the execution of different stages and only communicate a relatively small amount of data, it attains higher throughput compared to intra-op parallelism (Fig. 9b). Because both parallel methods split the model weight tensors across different GPUs, the total memory usage stays constant with increasing numbers of GPUs (Fig. 9c). This makes the statistical multiplexing of different GPUs across multiple models possible.

In the end, the tradeoff between parallelization strategies and their interplay with cluster resources, arrival patterns, and serving objectives forms an intricate design space.

3.4 Queueing Theory Analysis

In this section, we use queuing theory to mathematically verify the conclusions in §3.2 and §3.3. Specifically, we analyze the case where the inputs follow the Poisson arrival process. Since the execution time of a deep learning inference task is highly predictable [19], we assume the request serving time is deterministic. For the single device case, suppose the request rate to a model is λ_0 and the single device latency is D with the utilization $\lambda_0 D < 1$, then the average number of requests L_Q and the average latency W in this M/D/1 queue [46] are:

$$L_Q = \frac{\lambda_0 D}{2(1-\lambda_0 D)}, \quad W = D + L_Q D = D + \frac{\lambda_0 D^2}{2(1-\lambda_0 D)}.$$

Now consider the example in §3.1. Let $p\lambda, (1-p)\lambda$ be the average request rates for the two models respectively, where $p \in [0,1]$ controls the percentage of requests for both models. Then for the simple placement, the average latency can be derived as the average latency of two independent queues:

$$W_{simple} = D + \frac{p^2\lambda D^2}{2(1-p\lambda D)} + \frac{(1-p)^2\lambda D^2}{2(1-(1-p)\lambda D)}.$$

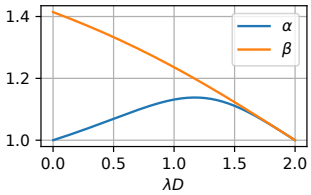


Figure 10: Maximal communication overhead α and uneven partition overhead β satisfy $W_{pipeline} \le W_{simple}$ as a function of total utilization λD.

Note that W_{simple} reaches minimum when $p = 1/2$. Intuitively, when p is not exactly half, one model receives more requests than the other. This larger portion of requests have a longer queueing delay, which leads to the higher overall mean latency.

For the model-parallel case, the requests to both models merged to a single Poisson Process with rate λ. For pipeline parallelism, suppose the latency for a single input to be D_s and the maximum stage latency to be D_m, then the average latency would be

$$W_{pipeline} = D_s + \frac{\lambda D_m^2}{2(1-\lambda D_m)}.$$

Suppose there is no model-parallel overhead, then $D_s = 2D_m = D$. Let's first consider the case where $p = 1/2$ (Fig. 2a). We have

$$W_{simple} = D + \frac{\lambda D^2}{4-2\lambda D}, \quad W_{pipeline} = D + \frac{\lambda D^2}{8-4\lambda D}.$$

In this case, the waiting time for model-parallel execution is half of the simple placement waiting time, as shown in the vertical lines in Fig. 2a. When the p is not $1/2$, W_{simple} will increase while $W_{pipeline}$ will stay the same, so the gap between W_{simple} and $W_{pipeline}$ will be even larger, as in Fig. 2c.

Next, we consider the case where model parallelism incurs overhead. We measure the two types of overheads in §3.3 separately: With the overhead from communication, $D_s = 2D_m = \alpha D$, where $\alpha \ge 1$ is the overhead factor. With the overhead from uneven stages, we suppose $D_s = D$ still holds, but $D_m = \beta D/2$ where $\beta \ge 1$ is the overhead factor. To keep $W_{pipeline} \le W_{simple}$, we can get the maximal α and

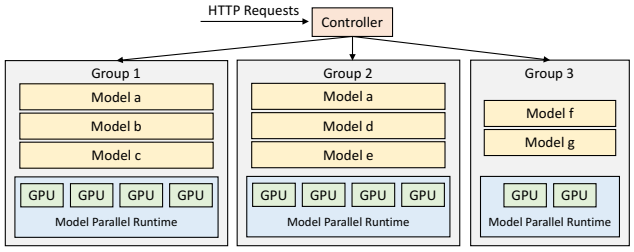


Figure 11: AlpaServe Runtime System Architecture

β as a function of the total utilization λD separately and we visualize the function in Fig. 10. When the utilization is high, the benefit of statistical multiplexing diminishes, and thus the overhead needs to be low, as in §3.2. On the other hand, when the utilization is very low, most requests will not be queued, and thus the communication overhead α needs to be low to keep the processing latency to be low. Note that the maximal overhead here is based on a uniform Poisson arrival distribution. A more bursty or more non-uniform arrival distribution will make the simple placement performs worse and make the model-parallelism placement outperforms the simple replication placement with even higher overhead.

4 Method

From §3, we can see that there are several key challenges to effectively utilize model parallelism for deep learning serving:

- Derive efficient model parallel strategies for inference to reduce the overhead of model parallelism. Specifically, find a partitioning strategy that minimizes the stage imbalance for inter-operator parallelism.
- Determine model-parallel placements according to the arrival pattern to maximize SLO attainment.

We built *AlpaServe* to specifically tackle these challenges. The runtime architecture of AlpaServe is shown in Fig. 11. AlpaServe utilizes a centralized controller to dispatch requests to different groups.[3] Each group hosts several model replicas on a shared model-parallel runtime. This section describes the architecture of AlpaServe and the key algorithms for efficiently leveraging model parallelism in a model serving system.

4.1 Automatic Parallelization for Inference

Since different parallelization configurations have different latency and throughput trade-offs, we need to enumerate multiple possible configurations for every single model and let the placement algorithm choose the best combination for all models in the whole cluster. Therefore, given a model, AlpaServe first runs an auto-parallelization compiler with various constraints to generate a list of possible configurations. We build several extensions on top of an existing auto parallelization training system, Alpa [56], to make it suitable for generating serving parallelization strategies. Alpa includes two passes for generating efficient model parallel partitions: inter-op pass and intra-op pass. The inter-op pass uses a dynamic programming (DP) algorithm to figure out the optimal inter-op parallel plan, and it calls the intra-op pass for each potential pipeline stage, which is formulated as an integer linear programming (ILP) problem, to profile its latency with the optimal intra-op parallel plan. In AlpaServe, we keep the two compilation passes, but extends both passes for serving.

The inter-op pass in Alpa optimizes the overall pipeline execution latency, which includes the time of forward and backward propagation and weight synchronization. However, in serving workloads, only forward propagation is being executed and there is no need for weight synchronization. Therefore, we reformulate the dynamic programming in AlpaServe to merely focus on minimizing the maximal stage latency. Specifically, denote $F(s,k)$ to be the maximum latency when slicing layers 1 to k into s stages. We can derive F as

$$F(s,k) = \min_{1 \le i \le k} \{\max\{F(s-1,i-1), latency(i,k)\}\},$$

where $latency(i,k)$ denotes the latency of a stage composes of layer i to k. In Alpa, the *latency* function of all possible $O(K^2)$ combinations is being profiled by the intra-op pass because of the complicated dependency between forward and backward passes. In AlpaServe, because the pipeline stages only perform forward propagation and only communicate intermediate results once between layer boundaries, we can accelerate the profiling by only profiling K layers and letting $latency(i,k)$ to be the sum of the latencies for layer i to k. This acceleration enables us to efficiently enumerate different inter- and intra-op device partition setups and generate a list of parallel strategies for the placement algorithm in §4.2.

For the intra-op pass, we extend the ILP in Alpa to drop all configurations that use data parallelism. For serving workloads, because there is no need for weight synchronization, data parallelism can be achieved by the replication placement. We leave the decision of whether to replicate a model to the placement algorithm in §4.2.

4.2 Placement Algorithm

Given a set of models and a fixed cluster, AlpaServe partitions the cluster into several groups of devices. Each group of devices selects a subset of models to serve using a shared model-parallel configuration. Different groups can hold the same model as replicas. The requests for a model are dispatched to the groups with the requested model replica. We call a specific cluster group partition, model selection, and parallel configuration as a *placement*. Our goal is to find a placement that maximizes the SLO attainment.

However, finding the optimal placement is a difficult combinatorial optimization problem. The overall placement configuration space grows exponentially with the number of devices and the number of models. To make things worse, the objective "SLO attainment" has no simple analytical formula for

[3] For a larger service, AlpaServe can be extended as a hierarchical deployment with each controller only managing a subset of devices as in [52].

Algorithm 1 Simulator-Guided Greedy Model Selection.

Input: Model list M, device group list G, group parallel configurations P, workload W, beam size k (default = 1).
Output: The model selection *best_sel*.

```
best_sel ← ∅
beam_sels ← {∅}
while true do
    new_sels ← ∅
    for sel ∈ beam_sels do
        for (m, (g, p)) ∈ M × (G, P) do
            // Parallelize the model as in §4.1.
            m_parallelized ← parallelize(m, g, p)
            sel' ← sel.add_model_to_group(m_parallelized, g)
            if sel' is in memory constraint then
                sel'.slo_attainment ← simulate(sel', W)
                new_sels.append(sel')
    if new_sels = ∅ then
        break
    beam_sels ← top-k_SLO_attainment(new_sels)
    sel* ← pick_highest_SLO_attainment(beam_sels)
    if sel*.slo_att > best_sel.slo_att then
        best_sel ← sel*
return best_sel
```

Algorithm 2 Enumeration-Based Group Partition and Model-Parallel Configuration Selection.

Input: Model list M, cluster C, workload W.
Output: The placement *best_plm*.

```
best_plm ← ∅
B ← get_potential_model_buckets(M)
for (B_1, B_2, ..., B_k) ∈ B do
    H ← get_potential_device_buckets(C, B, k)
    for (H_1, H_2, ..., H_k) ∈ H do
        // Get the placement for each bucket individually.
        for i from 1 to k do
            plm_i* ← ∅
            G ← get_potential_group_partitions(H_i)
            for G ∈ G do
                P ← get_potential_parallel_configs(G)
                for P ∈ P do
                    plm ← greedy_selection(B_i, G, P, W)
                    if plm.slo_att > plm_i*.slo_att then
                        plm_i* ← plm
        plm* ← concat(plm_1*, ..., plm_k*)
        if plm*.slo_att > best_plm.slo_att then
            best_plm ← plm*
return best_plm
```

an arbitrary arrival distribution. Existing tools and approximations from queueing theory can only analyze simple cases in §3.4 and cannot model more complex situations [46]. Therefore, we resort to a simulator-guided greedy algorithm that calls a simulator to compute SLO attainment.

To compute the SLO attainment with a given set of requests and placement, in AlpaServe, we assume we know the arrival process in advance. Although short-term burstiness is impossible to predict, the arrival pattern over longer timescales (e.g., hours or days) is often predictable [48]. Given this predictability, AlpaServe either directly uses the history request traces or fits a distribution from the trace and resamples new traces from the distribution as the input workload to the simulator to compute the SLO attainment.

We design a two-level placement algorithm: Given a cluster group partition and a shared model-parallel configuration for each group, Algorithm 1 uses a simulator-guided greedy algorithm to decide which models to select for each group. Then, Algorithm 2 enumerates various potential cluster partitions and parallel configurations and compares the SLO attainment from Algorithm 1 to determine the optimal placement.

Given a cluster group partition with a fixed model-parallel configuration for each group, Algorithm 1 selects model replicas iteratively as a beam search algorithm: At each iteration, it enumerates all possible (model, group) pairs, parallelizes the model on the device group with the algorithms in §4.1, and checks whether the model can be put on the group under the memory constraint. For all valid selections, it runs the simulator and computes SLO attainment. It then picks the top-k solutions and enters the next iteration. The algorithm terminates when no more replicas can be put into any groups.

The complexity of Algorithm 1 is $O(MGRSB)$, where M is the number of models, G is the number of groups, R is the number of replicas we can put according to the memory constraint, S is the number of requests in the workload (the simulation time is proportional to the number of the requests) and B is the beam size. It runs reasonably fast for our medium-scale cluster when the number of requests is small. When the number of requests is very large, we propose another heuristic to accelerate: Instead of using the simulator to evaluate all (model, group) pairs at each iteration, we can run the simulator only once and place a model with the most unserved requests in an available group with the lowest utilization. This reduces the time complexity to $O((M+G)RS)$. We find this heuristic gives solutions with SLO attainment higher than 98% of the SLO attainment get by the original algorithm in our benchmarks.

Algorithm 2 enumerates different group partitions and model-parallel configurations and picks the best one via multiple calls to Algorithm 1. When designing Algorithm 2, the first phenomenon we notice is that putting small and large models in the same group causes convoy effects, where the requests of small models have to wait for the requests of large models and miss the SLO. Therefore, in Algorithm 2, we first cluster models into model buckets. Each bucket contains a set of models with relatively similar sizes

and every model is assigned to one and only one bucket. Specifically, the function `get_potential_model_buckets` returns all the possible model bucket partitions that separate models whose latency difference is larger than a threshold into different disjoint buckets. We then enumerate all the potential ways to assign the devices to each bucket in `get_potential_device_buckets`.

Because different buckets include a disjoint set of models, we can then figure out the optimal placement for each bucket individually. For each bucket, we enumerate possible ways to partition the devices in the bucket into several groups in `get_potential_group_partitions` and enumerate the potential parallel configurations for each group with the method in `get_potential_parallel_configs`. We then call Algorithm 1 with `greedy_placement` to place models in the model bucket to the groups in the device bucket. We send the whole workload W to Algorithm 1, which ignores the requests that hit the models outside of the current bucket. Finally, a complete solution is got by concatenating the solutions for all buckets. The algorithm returns the best solution it finds during the enumerative search process.

Enumerating all possible choices can be slow, so we use the following heuristics to prune the search space. Intuitively, we want the different buckets to serve a similar number of requests per second. Therefore, we eliminate the bucket configurations with high discrepancies in the estimated number of requests it can serve per second for each bucket. Additionally, in `get_potential_group_partitions` and `get_potential_parallel_configs`, we assume all groups have the same size and the same parallel configurations except for the last group which is used when the number of devices is not divisible by the group size.

4.3 Runtime Scheduling

We use a simple policy to dispatch and schedule the requests at runtime. All requests are sent to a centralized controller. The controller dispatches each request to the group with the shortest queue length. Each group manages a first-come-first-serve queue. When a group receives a request, it checks whether it can serve the request under SLO and rejects the request if it cannot. This is possible because the execution time of a DNN model is very predictable and can be got in advance by profiling [19]. In most of our experiments, we do not include advanced runtime policies such as batching [19], swapping, and preemption [21]. These techniques are complementary to model parallelism. Nevertheless, we discuss how they fit into our system.

Batching. Batching multiple requests of the same model together can increase the GPU utilization and thus increase the throughput of a serving system. In our system, we do find batching is helpful, but the gain is limited. This is because we mainly target large models and a small batch size can already fully saturate the GPU, which is verified in §6.5. To isolate the benefits of model parallelism and make the results more explainable, we decide to disable any batching in this paper except for the experiments in §6.5.

Preemption. The optimal scheduling decision often depends on future arrivals, and leveraging preemption can help correct previous suboptimal decisions. The first-come-first-serve policy may result in convoy effects when models with significantly different execution times are placed in the same group. We anticipate a least-slack-time-first policy with preemption can alleviate the problems [12].

Swapping. The loading overheads from the CPU or Disk to GPU memory are significant for large models, which is the target of this paper, so we do not implement swapping in AlpaServe. We assume all models are placed on the GPUs. This is often required due to tight SLOs and high rates, especially for large models. The placement of models in AlpaServe can be updated in the periodic re-placement (e.g., every 24 hours).

Fault tolerance. While the current design of AlpaServe does not have fault tolerance as a focus, we acknowledge several potential new challenges for fault tolerance: With model parallelism, the failure of a single GPU could cause the entire group to malfunction. Additionally, the use of a centralized controller presents a single point of failure.

5 Implementation

We implement a real system and a simulator for AlpaServe with about 4k lines of code in Python. The real system is implemented on top of an existing model-parallel training system, Alpa [56]. We extend its auto-parallelization algorithms for inference settings to get the model-parallel strategies. We then launch an Alpa runtime for each group and dispatch requests to these groups via a centralized controller.

The simulator is a continuous-time, discrete-event simulator [39]. The simulator maintains a global clock and simulates all requests and model executions on the cluster. Because the simulator only models discrete events, it is orders of magnitude faster than the real experiments. In our experiment, it takes less than 1 hour for a 24-hour trace. The fidelity of the simulator is very high because of the predictability of DNN model execution, which is verified in §6.1.

6 Evaluation

In this section, we evaluate AlpaServe's serving ability under a variety of model and workload conditions. The evaluation is conducted on a range of model sizes, including those that do and do not fit into a single GPU, and we show that AlpaServe consistently outperforms strong baselines across all model sizes. In addition, we evaluate the robustness of AlpaServe against changing arrival patterns and do ablation studies of our proposed techniques. Evaluation results show that AlpaServe can greatly improve various performance metrics. Specifically, AlpaServe can choose to save up to 2.3× devices, handle 10× higher rates, 6× more burstiness, or 2.5× more stringent SLO, while meeting the latency SLOs for over 99% requests.

Name	Size	Latency (ms)	S1	S2	S3	S4
BERT-1.3B	2.4 GB	151	32	0	10	0
BERT-2.7B	5.4 GB	238	0	0	10	0
BERT-6.7B	13.4 GB	395	0	32	10	0
BERT-104B	208 GB	4600	0	0	0	4
MoE-1.3B	2.6 GB	150	0	0	10	0
MoE-2.4B	4.8 GB	171	0	0	10	0
MoE-5.3B	10.6 GB	234	0	0	10	0

Table 1: The first three columns list the sizes and inference latency of the models. The latency is measured for a single query with a sequence length of 2048 on a single GPU. BERT-104B's latency is reported using a minimal degree of inter-op parallelism. The latter columns list the number of instances for each model in different model sets named as S1-S4.

6.1 Experiment Setup

Cluster testbed. We deploy AlpaServe on a cluster with 8 nodes and 64 GPUs. Each node is an AWS EC2 p3.16xlarge instance with 8 NVIDIA Tesla V100 (16GB) GPUs.

Model setup. Since Transformer [47] is the default backbone for large models, we choose two representative large Transformer model families: BERT [14] and GShard MoE [27] for evaluation.[4] In ML practice, the large model weights are usually pretrained and then finetuned into different versions for different tasks. Hence, for each model family, we select several most commonly used model sizes [5], and then create multiple model instances at each size for experimentation. Also, we design some model sets to test the serving systems under different model conditions; details about model sizes, their inference latency on testbed GPUs, and the number of model instances in each model set are provided in Tab. 1.

Metrics. We use *SLO attainment* as the major evaluation metric. Under a specific SLO attainment goal (say, 99%), we concern with another four measures: (1) the minimal number of devices the system needs, (2) the maximum average request rate, (3) the maximum traffic burstiness the system can support, and (4) the minimal SLO the system can handle. We are particularly interested in a SLO attainment of 99% (indicated by vertical lines in all curve plots), but will also vary each variable in (1) - (4) and observe how the SLO attainment changes.

Simulator fidelity. We want to study the system behavior under extensive models, workload, and resource settings, but some settings are just beyond the capacity of our testbed. Also, it is cost- and time-prohibitive to perform all experiments on the testbed for the days-long real traces. To mitigate the problem, we use the simulator introduced in §5 for the majority of our experiments, noticing that DNN model execution [19] has high predictability, even under parallel settings [27, 56].

[4] In this paper, we focus on non-autoregressive large models which perform inference with one forward pass, but note that the techniques proposed in this paper can be extended to auto-regressive models like GPT-3.

SLO Scale	Selective Replication		AlpaServe	
	Real System	Simulator	Real System	Simulator
0.5x	00.0%	00.0%	33.3%	33.3%
1x	00.0%	00.0%	53.5%	53.2%
1.5x	29.7%	30.2%	64.1%	64.7%
2x	36.9%	36.8%	79.0%	80.6%
3x	49.5%	48.5%	91.4%	92.1%
4x	58.6%	57.8%	96.4%	96.5%
5x	64.9%	64.0%	97.6%	97.9%
10x	83.1%	82.6%	100.0%	99.7%

Table 2: Comparison of the SLO attainment reported by the simulator and the real system under different SLO scales.

We study the fidelity of the simulator in Tab. 2. Given two model placement algorithms, we compare the SLO attainment reported by the simulator and by real runs on our testbed under different *SLO Scales*. The error is less than 2% in all cases, verifying the accuracy of our simulator. Additionally, we conduct experiments on cluster testbed for results in §6.3.

6.2 End-to-end Results with Real Workloads

In this section, we compare AlpaServe against baseline methods on publicly available real traces.

Workloads. There does not exist an open-source production ML inference trace to the best of our knowledge. Therefore, we use the following two production traces as a proxy: Microsoft Azure function trace 2019 (MAF1) [42] and 2021 (MAF2) [54]. They were originally collected from Azure serverless function invocations in two weeks, and have been repurposed for ML serving research [4, 25]. The two traces exhibit distinct traffic patterns. In MAF1, each function receives steady and dense incoming requests with gradually changing rates; in MAF2, the traffic is very *bursty* and is distributed across functions in a highly *skewed* way – some function receives orders of magnitude more requests than others. Note that most previous works [19] are evaluated on MAF1 only. Since there are more functions than models, following previous work [4, 25], given a model set from Tab. 1, we round-robin functions to models to generate traffic for each model.

Setup. SLO attainment depends on many factors. For each metric (1) - (4) mentioned in §6.1, we set a default value, e.g., the default SLO is set as tight as 5× inference latency (SLO Scale=5). This forms a *default setting*, given which, we then vary one variable (while fixing others) at a time and observe how it affects the resulting SLO attainment. To change the two variables (3) and (4), which characterize traffic patterns, we follow Clockwork [19] and Inferline [8] and slice the original traces into time windows, and fit the arrivals in each time window with a Gamma Process parameterized by rate and coefficient of variance (CV). By scaling the rate and CV and resampling from the processes, we can control the rate and burstiness, respectively.

Baselines. We compare AlpaServe to two baseline methods:

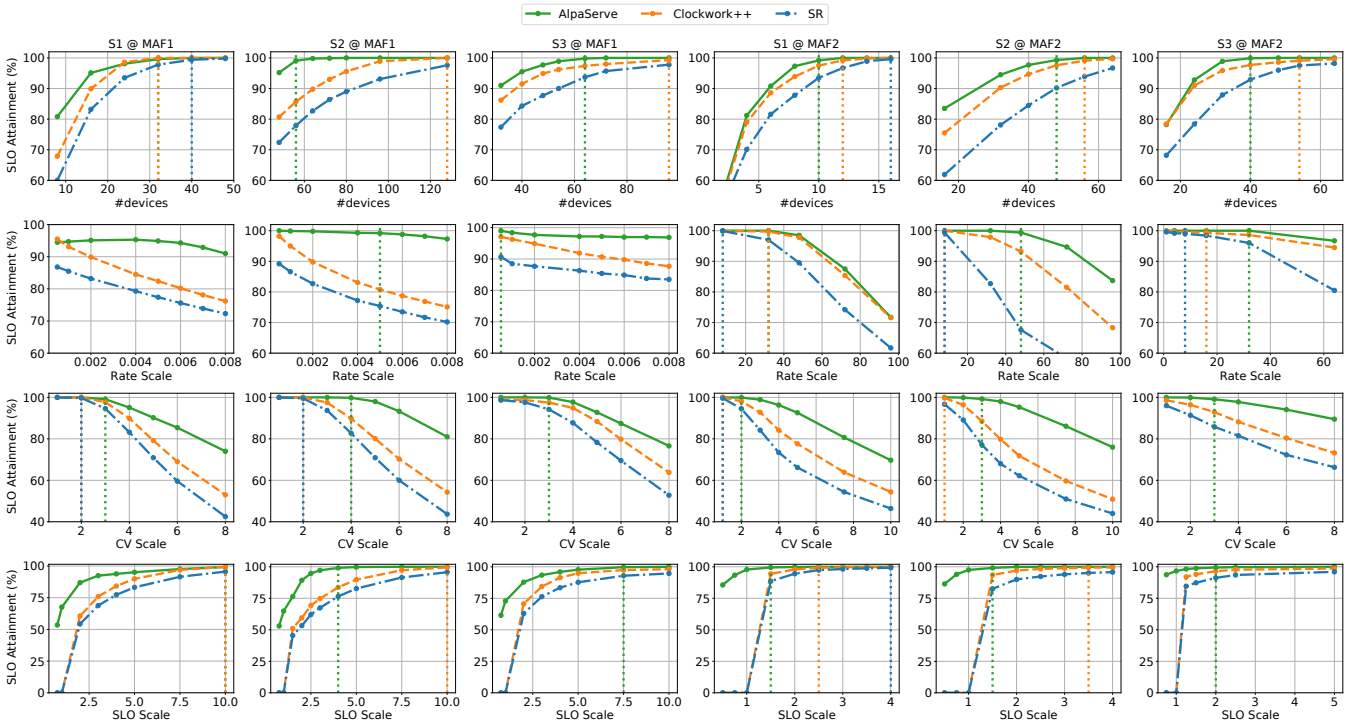


Figure 12: SLO attainment under various settings. In column S1@MAF1, we replay the MAF1 trace on the model set S1, and so on. In each row, we focus on one specific metric mentioned in §6.2 to see how its variation affects the performance of each serving system. If any, the dotted vertical line shows when the system can achieve 99% SLO attainment.

(1) *Selective Replication (SR)*: use AlpaServe's placement algorithm without model parallelism, which mimics the policy of a wide range of existing serving systems [9,44]; (2) *Clockwork++*: an improved version of the state-of-the-art model serving system Clockwork [19]. The original Clockwork continuously swaps models into and out of GPUs. This helps for very small models (e.g., w/ several million parameters) but incurs significant swapping overheads on larger models. For fair comparisons, we implement Clockwork++ in our simulator, which swaps models following Clockwork's replacement strategy at the boundary of every two windows[5] of the trace using SR's algorithm, but *assuming zero swapping overheads*. We believe it represents a hypothetical upper bound of Clockwork's performance. Since all the baselines can only support models that can fit into one GPU memory,[6] we use model set S1, S2 and S3 from Tab. 1 in this experiment.

SLO attainment vs. cluster size. Fig. 12's first row shows the SLO attainment with varying cluster sizes when serving a specific (model set, trace) pair. AlpaServe outperforms the two baselines at all times and uses far fewer devices to achieve 99% SLO attainment thanks to model parallelism. By splitting one model replica onto N devices, AlpaServe can achieve similar throughput as if N replica were created for replication-only methods; but note AlpaServe uses only one replica of memory. Surprisingly, although we let Clockwork++ adjust to the traffic dynamically with zero overhead, AlpaServe still wins with a static placement; this is because model-parallel placement is by nature more robust to bursty traffic.

It is worth noting that replication-only methods can at most place 2 replicas of BERT-2.6B on a V100 (13GB memory budget), resulting in a substantial memory fraction, while model parallelism can avoid such memory fractions and enable more flexible placement of models.

SLO attainment vs. rate. Fig. 12's 2nd row varies the rate of the workloads. For a stable trace like MAF1, AlpaServe can handle a much higher rate than baselines. While for a skewed and highly dynamic trace MAF2, whose traffic is dominated by a few models and changes rapidly, the replication-based methods have to allocate the majority of the GPUs to create many replicas for "hot" models to combat their bursty traffic; those GPUs, however, may go idle between bursts, even with frequent re-placement as in Clockwork++. In AlpaServe, each model needs fewer replicas to handle its peak traffic.

SLO attainment vs. CV. Fig. 12's 3rd row varies the CV of the workloads. The traffic becomes more bursty with a higher CV, which aggravates the queuing effect of the system and increases the possibility of SLO violation. The traditional

[5]For MAF1, we follow Clockwork to set the window size as 60 seconds. For MAF2, we set it as 5.4K seconds.

[6]In our cluster testbed, the per-GPU memory is 16GB, but the actual available space for model weights is around 13GB due to the need to store activations and other runtime context.

solution to handle burstiness is by over-provision, wasting a lot of resources. AlpaServe reveals a hidden opportunity to handle this by model parallelism.

SLO attainment vs. SLO. Fig. 12's 4th row shows the effect of different SLO. Previous work [19] which targets serving small models usually sets SLO to hundreds of milliseconds, even though the actual inference latency is less than 10 ms. Thanks to the intra-op parallelism, AlpaServe can maintain good performance under similar SLO when serving large models, whose inference latency can be over 100 ms. When SLO is tight, even less than the model inference time, AlpaServe favors intra-op parallelism to reduce the inference latency, which also reduces AlpaServe's peak throughput due to the communication overhead but can make more requests to meet their SLO. When SLO becomes looser, AlpaServe will automatically switch to use more inter-op parallelism to get higher throughput.

6.3 Serving Very Large Models

Today's large models may possess hundreds of billions of parameters [5, 31, 53]. To serve large models at this scale, the common practice in production is to choose the model parallelism strategy manually and use dedicated GPUs for each model [51]. To show AlpaServe has improved capability in serving very large models, we deploy model set S4 on our testbed, each requiring at least 16 GPUs to serve in terms of memory usage. As baselines, for each model, we enumerate all combinations of inter- and intra-op parallelisms on 16 GPUs. In contrast, AlpaServe searches for the optimal GPU group allocation and model placement according to the arrival traffic and tries to achieve statistical multiplexing.

Offered load. In the default setting, the traffic is generated via a Gamma Process with an average rate of 8 requests/s and CV of 4. We then split the requests to each model following a power law distribution with an exponent of 0.5 to simulate the real-world skewness.[7] Similar to §6.2, we vary one of the rate, CV, or SLO in the default setting to see how each factor contributes to the resulting performance. It is worth noting that all results presented in this section are obtained via real execution on the testbed cluster.

SLO attainment. Fig. 13 shows the SLO attainment of each system under various settings. Although enumerating parallelism strategies and selecting the best can improve performance, it still remains a substantial gap compared to AlpaServe. This means that the traditional way of using dedicated GPUs to serve large models is not ideal. We check the solution of AlpaServe and find it slices the cluster evenly into two groups, each with the (4, 8) inter-/intra-op parallel configuration, and groups the models in a way that balances the requests between two groups. This further proves that our motivation in §3.1 still holds for extremely large models. By

[7] Uniform split yielded similar results.

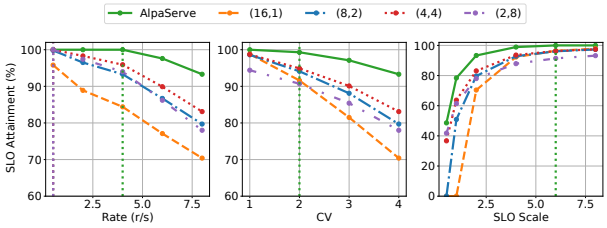


Figure 13: SLO attainment as we vary the rate, CV, and SLO scale. (8,2) means 8-way inter-op parallelism and in each pipeline stage using 2-way intra-op parallelism.

space-sharing the devices, AlpaServe can exploit new opportunities for statistical multiplexing, which is advantageous for bursty workloads but largely under-explored by prior work.

6.4 Robustness to Changing Traffic Patterns

Until now, AlpaServe's good performance is based on the assumption we make in its placement algorithm that we know the arrival process in advance. In practice, the arrival process can be approximated using historical traces but the unavoidable real-world variance may make the prediction inaccurate. In this experiment, we study how AlpaServe performs if the traffic patterns change.

We reuse the same setting for S2@MAF1 in §6.2, but this time for AlpaServe and SR, we randomly slice two one-hour traces from MAF1, one is what their algorithms are assumed, while the other one is used as the actual arrival process. While for Clockwork++, we still run its algorithm directly on the actual arrival process to respect its online nature. Similarly, we vary different factors and compute the SLO attainment for each system. We repeat the experiments three times and show the average results in Fig. 14.

Unsurprisingly, SR's performance drops significantly when traffic changes. By contrast, AlpaServe maintains good performance and still outperforms Clockwork++, an online adjustment algorithm, using a static placement generated from substantially different traffic patterns. This confirms that, in face of highly-dynamic traffic patterns, statistical multiplexing with model parallelism is a simple and better alternative than existing replication- or replacement-based algorithms.

6.5 Benefits of Dynamic Batching

Batching is a common optimization in other serving systems [19, 33, 34] and the choice of batch size is critical to the performance because it can increase GPU utilization and thus increase the system throughput. However, in large model scenarios, the benefit of batching is limited mainly due to two reasons. First, for large models, a small batch size will saturate the GPU, which means there is little gain to batching more requests. Second, the execution latency grows linearly with the batch size [44], so when the SLO is tight (say SLO Scale is less than 2), batching is simply not a choice.

To isolate the benefits of model parallelism and make the

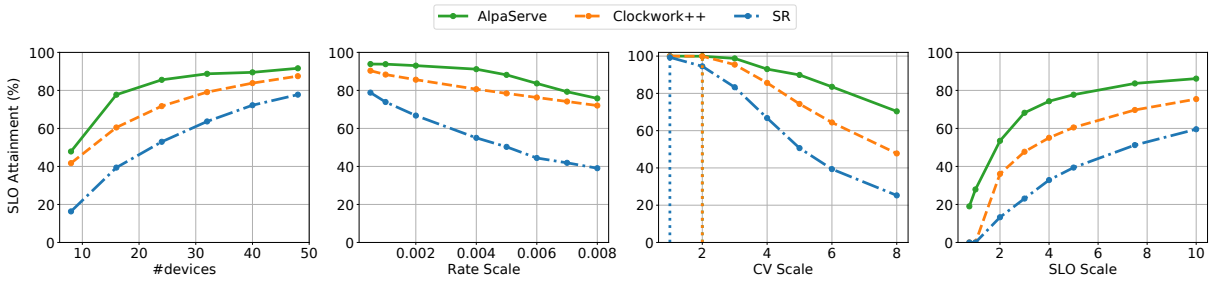


Figure 14: The actual arrival traffic for AlpaServe and SR is different from what their algorithms are assumed, while Clockwork++ runs directly on the actual traffic.

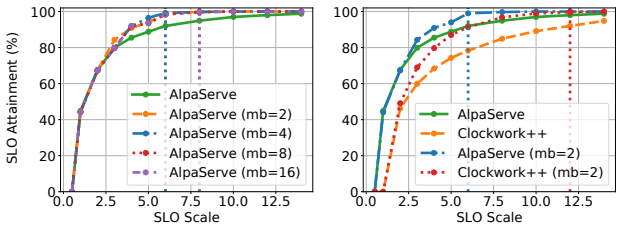
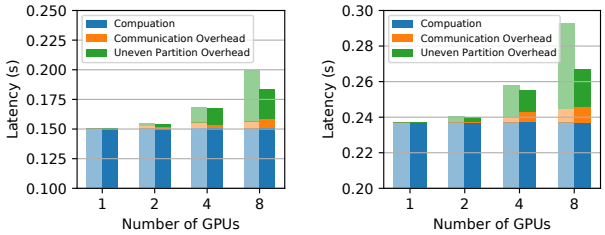


Figure 15: SLO Attainment when batching is enabled. mb=2 means the maximum batch size is 2.



(a) Transformer 1.3B. (b) Transformer 2.6B.

Figure 16: Comparison of the model parallel overhead between manual partition (lighter color) and the partition found by the automatic algorithm (darker color).

results more explainable, we decide to disable any batching in other experiments but prove that the batching strategy is purely orthogonal to the scope of this paper in this subsection. To prove this, we implement a standard batching algorithm in AlpaServe and evaluate its performance.

Batching strategy. When a request arrives, it will get executed immediately if any device group is available. Otherwise, it will be put into a per-model requests queue for batching. When a device group becomes idle, it will choose a model which has a replica on it and batch as many requests as possible from the requests queue of the model while satisfying the SLO requirements.

Setup. As the model size increases, the potential benefit of batching decreases. Therefore, we choose to evaluate model set S1. We generate a synthetic Gamma Process traffic with an average rate of 4 requests/s and a CV of 4 for each model.

SLO attainment. Fig. 15 (left) shows the SLO attainment achieved by AlpaServe with different maximum batch size settings under various SLO scales. When the SLO requirement is tight, any batching will violate the SLO so there is no gain with batching enabled. Also, although we choose to serve the smallest model in Tab. 1, a small batch size like 2 combined with a long sequence length of 2048 already saturates the GPU, so a larger maximum batch size brings no performance improvement. Fig. 15 (right) compares the improvement for AlpaServe and Clockwork++ with our batching algorithm enabled.[8] When the SLO requirement becomes loose, both AlpaServe and Clockwork++ have better SLO attainment to some extent. Since AlpaServe's performance is good even without batching and batched requests with different batch sizes will incur stage imbalance and pipeline bubble in inter-op parallel, the absolute improvement of Clockwork++ is slightly better.

[8] SR is left out to make the figure clearer as it is worse than Clockwork++.

6.6 Ablation Study

In this section, we study the effectiveness of our proposed auto-parallelization (§4.1) and placement algorithms (§4.2).

Benefits of auto-parallelization. We show that the auto-parallelization ability allows AlpaServe to not only generalize to arbitrary model architectures but even also reduce parallelism overheads – hence improved serving performance (see §3.3 for more discussion). To see that, typical manual model-parallel parallelization strategies offered in *de facto* systems [1, 31, 32] is to assign an equal number of (transformer) layers to each pipeline stage. These strategies often fail to create balanced workloads across distributed GPUs because contemporary large models have heterogeneous layers, such as embedding operations. The extensions introduced in §4.1 automatically partition the models at the computational graph level and generate nearly-balanced stages. Empirically, as shown in Fig. 16, for 8 pipeline stages, auto-parallelization reduces the total overhead by 32.9% and 46.7% for Transformer 1.3B and 2.6B respectively, which is necessary for achieving good serving performance when model parallelism is used for serving.

Effectiveness of the placement algorithm. We now test the

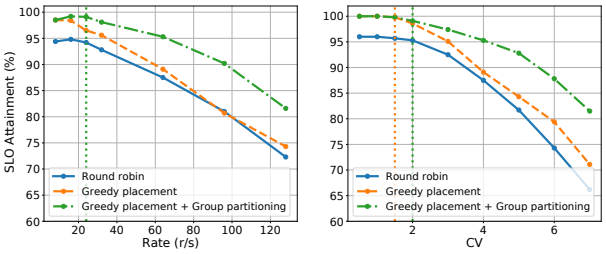


Figure 17: Ablation study of placement algorithms.

effectiveness of our placement algorithm on a synthetic workload. We serve the most challenging model set S3 (Tab. 1) on our testbed. The rate distribution of the models follows a power law distribution. The arrival pattern of each model is a Gamma process. Three variants of the placement algorithms are evaluated. *Round robin* means placing models in a round-robin fashion and using 4-stage pipelines for all groups. *Greedy placement* uses our greedy placement and 4-stage pipeline for all groups. *Greedy placement + Group partitioning* performs greedy placement plus group partitioning search. As shown in Fig. 17, both placement and group partitioning are necessary to achieve good SLO attainment. In the left subfigure, the group partitioning increases the rate by 1.5× compared to greedy placement without group partitioning over 99% SLO attainment, while round robin can never reach 99% SLO attainment. In the right subfigure, the group partitioning increases the traffic burstiness that can be handled to meet 99% SLO attainment by 1.3×.

7 Related Work

Model serving systems. There has been a proliferation of model serving systems recently. These range from general-purpose production-grade systems like TensorFlow Serving [34] and NVIDIA Triton [33], which are widely used but do not provide any support for automatic placement or latency constraints. They also include systems that are optimized for single-model serving [51] or serving of specific classes of models (e.g., transformers) [16, 51, 57]. AlpaServe targets a broader set of models and features than these systems.

For SLO-aware, distributed serving, most serving systems ignore placement-level interactions between models. Clockwork [19], for instance, primarily focuses on predictability; when scheduling, it greedily loads and executes models on available GPUs. Shepherd [52] utilizes preemption to correct sub-optimal scheduling decisions. For large models, loading model weights and preemption can easily overwhelm practical SLOs. Other systems like Clipper [9], Infaas [40], and DVABatch [10] also do not reason about the latencies of co-located models.

Nexus [44] is very related to our work in that it examines the placement of models; however, Nexus is an example of a system that takes the traditional replication approach described in §3 and, thus, misses a broad class of potential parallelization strategies that we explore in this paper.

Inference optimizations for large models. AlpaServe is complementary to another large body of work on optimizations for inference over large models. These include techniques like quantization [13], distillation [41], offloading [1], better operator parallelism [36], and CUDA kernel optimization [11, 26]. Some of these optimizations are intended to stem the tide of increasing model sizes; however, all of these gains are partial— the challenge of serving large models has continued to escalate rapidly despite these efforts.

Model parallelism for training. AlpaServe is largely orthogonal to the large body of work on model parallelism in training [23, 28, 31, 37, 56]. As described in §3, serving presents a unique set of constraints and opportunities not found in training workloads. Where these systems do intersect with AlpaServe, however, is in their methods for implementing model parallelism along various dimensions. In particular, AlpaServe builds on some of the parallelization techniques introduced in [56].

Resource allocation and multiplexing. The problem of how to multiplex limited resources to the incoming requests is one of the oldest topics in computer science and has been studied in different application domains [3, 29, 38]. Recent work on DL scheduling uses swapping [2], preemption [20], interleaving [55], and space-sharing [49] to realize fine-grained resource sharing. Rather, the contribution of this paper is a deep empirical analysis of the applications of these ideas to an emerging space: the serving of multiple large models.

8 Conclusion and Future Work

In this paper, we presented AlpaServe, a system for prediction servings of multiple large deep-learning models. The key innovation of AlpaServe is integrating model parallelism into multi-model serving. Because of the inherent overheads of model parallelism, such parallelism is traditionally applied conservatively—reserved for cases where models simply do not fit within a single GPU or execute within the required SLO. AlpaServe demonstrates that model parallelism is useful for many other scenarios, quantifies the tradeoffs, and presents techniques to automatically navigate that tradeoff space.

In the future, we will extend AlpaServe to more complicated scenarios, including serving multiple parameter-efficient adapted models (e.g., LoRA [22]), models with dependencies, and autoregressive models [5].

9 Acknowledgement

We thank the OSDI reviewers and our shepherd, Heming Cui, for their valuable feedback. This work is in part supported by NSF CISE Expeditions Award CCF1730628, NSFC under the grant number 62172008, and gifts from Astronomer, Google, IBM, Intel, Lacework, Microsoft, Nexla, Samsung SDS, Uber, and VMware. Yinmin Zhong and Xin Jin are also with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

References

[1] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, et al. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale. *arXiv preprint arXiv:2207.00032*, 2022.

[2] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. {PipeSwitch}: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 499–514, 2020.

[3] Nirvik Baruah, Peter Kraft, Fiodar Kazhamiaka, Peter Bailis, and Matei Zaharia. Parallelism-optimizing data placement for faster data-parallel computations. *Proceedings of the VLDB Endowment*, 16(4):760–771, 2022.

[4] Anirban Bhattacharjee, Ajay Dev Chhokra, Zhuangwei Kang, Hongyang Sun, Aniruddha Gokhale, and Gabor Karsai. Barista: Efficient and scalable serverless serving system for deep learning prediction services. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 23–33. IEEE, 2019.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[6] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[7] Copy.ai. Copy.ai: Write better marketing copy and content with ai. https://www.copy.ai/.

[8] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 477–491, 2020.

[9] Daniel Crankshaw, Xin Wang, Guilio 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 17)*, pages 613–627, 2017.

[10] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. Dvabatch: Diversity-aware multi-entry multi-exit batching for efficient processing of dnn services on gpus. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 183–198, 2022.

[11] Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 2022.

[12] Robert I Davis, Ken W Tindell, and Alan Burns. Scheduling slack time in fixed priority pre-emptive systems. In *1993 Proceedings Real-Time Systems Symposium*, pages 222–231. IEEE, 1993.

[13] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems*, 2022.

[14] 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.

[15] Mengnan Du, Subhabrata Mukherjee, Yu Cheng, Milad Shokouhi, Xia Hu, and Ahmed Hassan Awadallah. What do compressed large language models forget? robustness challenges in model compression. *arXiv preprint arXiv:2110.08419*, 2021.

[16] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 389–402, 2021.

[17] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[18] Github. Github copilot: Your ai pair programmer. https://github.com/features/copilot.

[19] 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, 2020.

[20] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, Carlsbad, CA, July 2022. USENIX Association.

[21] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, 2022.

[22] 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, 2021.

[23] 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.

[24] Huggingface. Models - huggingface. https://huggingface.co/models.

[25] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serving deep learning models in a serverless platform. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 257–262. IEEE, 2018.

[26] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems*, 3:711–732, 2021.

[27] 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 *International Conference on Learning Representations*, 2020.

[28] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. In *International Conference on Machine Learning*, pages 6543–6552. PMLR, 2021.

[29] Xiaoqiao Meng, Canturk Isci, Jeffrey Kephart, Li Zhang, Eric Bouillet, and Dimitrios Pendarakis. Efficient resource provisioning in compute clouds via vm multiplexing. In *Proceedings of the 7th international conference on Autonomic computing*, pages 11–20, 2010.

[30] 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*, pages 1–15, 2019.

[31] 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.

[32] NVIDIA. Fastertransformer. https://github.com/NVIDIA/FasterTransformer.

[33] NVIDIA. Triton inference server. https://developer.nvidia.com/nvidia-triton-inference-server.

[34] 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. *arXiv preprint arXiv:1712.06139*, 2017.

[35] OpenAI. Chatgpt. https://chat.openai.com/chat.

[36] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *arXiv preprint arXiv:2211.05102*, 2022.

[37] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[38] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 401–417, 2016.

[39] Stewart Robinson. *Simulation: the practice of model development and use*. Bloomsbury Publishing, 2014.

[40] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411. USENIX Association, July 2021.

[41] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[42] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218, 2020.

[43] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-TensorFlow: Deep learning for supercomputers. In *Neural Information Processing Systems*, 2018.

[44] 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*, pages 322–337, 2019.

[45] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[46] John F Shortle, James M Thompson, Donald Gross, and Carl M Harris. *Fundamentals of queueing theory*, volume 399. John Wiley & Sons, 2018.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[48] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. Mlaas in the wild: Workload analysis and scheduling in large-scale heterogeneous gpu clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, 2022.

[49] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. Transparent GPU sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 69–85, Boston, MA, April 2023. USENIX Association.

[50] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. Gspmd: general and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

[51] 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, 2022.

[52] 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, 2023.

[53] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[54] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 724–739, 2021.

[55] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. Multi-resource interleaving for deep learning training. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 428–440, 2022.

[56] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E Gonzalez, 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)*, 2022.

[57] 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, 2022.