Benefits of multiplexing machine learning models while serving

Aahan Tyagi
tyagi055@umn.edu

Hemanth Kumar Tirupati
tirup007@umn.edu

Vishal Kancharla
kanch042@umn.edu

Abstract

In the AlpaServe paper [7], the authors present a framework for serving multiple deep learning models. Typically, model parallelism is utilized only when a single model is too large to fit on one device. However, the authors assert that using model parallelism is beneficial even when serving several smaller models. They argue that the performance gains outweigh the communication overhead introduced via model parallelism when handling bursty workloads. AlpaServe works by identifying an efficient strategy for placing and parallelizing deep learning models across nodes. The paper claimed that evaluating AlpaServe on production workloads showed they can process requests at up to 10× higher rates or 6× more burstiness while staying within latency constraints for more than 99% of requests. We successfully reproduced and verified those performance claims.

1 Introduction

Many recent breakthroughs in machine learning, such as the large language models (LLMs), have been achieved in part due to the availability of huge training data and compute resources to train these large models. For example, GPT-3 needs 300 billion training tokens and can have a comparable number of parameters [3]. Hence it is no longer viable to train these models on a single machine. Implementations such as Alpa [15] enabled automatic data and model parallelism to aid in training such large models. However, we can leverage model parallelism when serving machine learning models too, and achieve efficiency.

Unfortunately, it is not easy to optimally split a set of models and place them on the cluster available. Different kinds of model parallelism such as inter-operator parallelism and intra-operator parallelism exist and each has its downside. In inter-operator parallelism, different devices execute different operators of a model. This allows the model to exceed the memory limitation of a single GPU. However inter-operator parallelism will typically increase the execution time of a single request. Inter-operator communication does not add much communication overhead as there is only point-to-point communication between devices. On the other hand in intra-operator parallelism, a single operator is partitioned across multiple devices, with each device executing a portion of the computation in parallel. This will too increase the amount of computation and memory available to the model and also decrease the latency of requests, however, it adds communication overhead since collective communication is required when we split and merge the operator.

The authors also emphasize on bursty workloads, as provisioning for peak workload is expensive. Even when serving just one model type, it is common to have different versions of the same model, especially in situations like testing or using models that are fine-tuned versions of the same base model. Hence, the authors assert that handling both bursty workloads and multiple small models is not uncommon.

AlpaServe automatically and efficiently explores the tradeoffs among different parallelization and placement strategies for model serving. It takes cluster resource specification, a set of models, and a periodic workload profile as inputs and then partitions and places the models It optimizes for the attainment of Service Level Objectives (SLOs).

We start our experiments by determining when model parallelism is beneficial. We assess if there are any benefits to using model parallelism even when the models we are serving are small and can fit in one device. We also evaluate how device memory, request rate, coefficient of variance (CV) in the request rate to different models, and SLO affect model parallelism. Finally, we evaluate AlpaServe on baseline models and analyze the results, specifically evaluating the performance on SLO attainment, defined in terms of latency.

2 Motivation

We begin by verifying whether model parallelism outperforms simple replication strategy. In replication, the model is not divided and we fit only one model per device if the device memory is limited. If the device memory allows we fit multiple full models. In the model parallelism strategy, we use inter-operator parallelism and divide the models into pipeline stages, and uniformly assign these stages to the devices. So every device will have at least one pipeline stage of every model. The simulator proposed in the paper was used. The simulator makes use of profiling data, and for this experiment, we used 8 bert-2.6B models. In this section, we discuss how model parallelism performs when we change the memory budget of our simulated GPUs, request CV, overall request rate, and SLO.

Device memory: Figure 1 shows mean latency compared to device memory budget. With low memory, as discussed, replication fits only one model per device, whereas model parallelism has pipeline stages of all models. Thus if requests target just a few models, devices not having those models will be idle in replication, while model parallelism keeps all devices busy. This explains the lower latency of model parallelism. As memory increases, multiple whole models can fit in one device, Hence there will be no need to divide the model into pipeline stages. Thus with ample memory, model parallelism and replication converge to the same strategy, as reflected in Figure 1. Figure 2 depicts 99th percentile latency exhibiting the same trend as mean latency.

Request CV: High request CV indicates burstiness, with some models receiving many more requests than others. In this scenario with replication, devices serving such popular models become throttled while others remain idle. In the model parallelism strategy, all devices will be working, hence as shown in Figures 3 and 4, as CV increases model parallelism will have better latency compared to replication.

Overall request rate: When the overall request rate is low and only a few models receive most of the requests model parallelism outperforms replication. However, when all models receive a large number of requests at once, all devices will become saturated in both model parallelism and replication strategy. In fact, at a high overall request rate, model parallelism can work worse than replication due to the communication overhead we have in model parallelism. Figures 5 and 6 show the plots we were able to reproduce by varying request rates against latency.

Service Level Objective: In Figure 7 the SLO scale ranges from 5x latency to 20x latency. When SLO is tight (less than 10x latency) we have more burstiness hence model parallelism has more SLO attainment than replication. As SLO becomes looses, we see less burstiness hence observe similar SLO attainment in both replication and model parallelism.

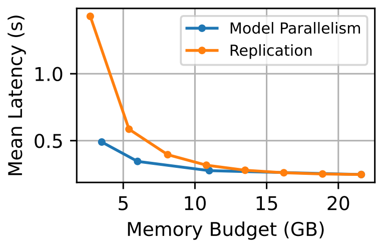

Figure 1: mean latency vs memory budget

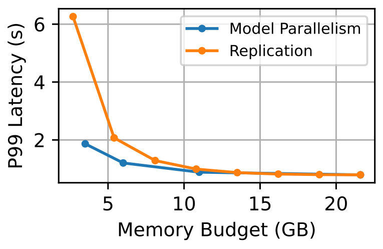

Figure 2: 99th percentile latency vs memory budget

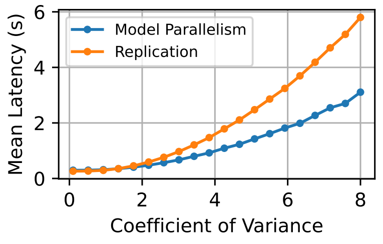

Figure 3: mean latency vs request CV

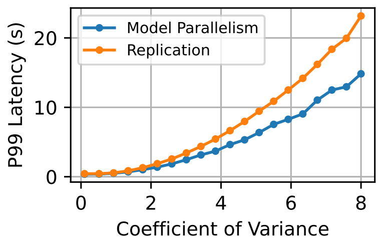

Figure 4: 99th percentile latency vs request CV

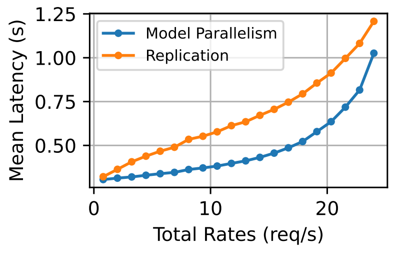

Figure 5: mean latency vs overall request rate

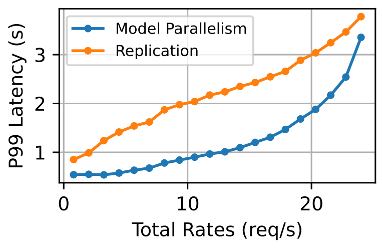

Figure 6: 99th percentile latency vs overall request rate

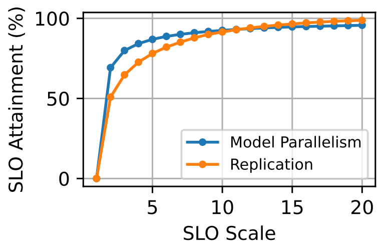

Figure 7: SLO attainment as we change SLO scale

3 Experimental Setup

Table 1 shows the configuration of the machines we used to run our experiments. The runtime environment is critical for this project, as many components of AlpaServe are compatible with certain underlying software stack versions. Hence, we have experimented with three versions of the software environment, The compatibility we found is shown in Table 2.

3.1 Simulator

The simulator used for the experiments is a continuous-time discrete event simulator. We model the inference tasks with discrete events modeled as per the latencies shown in Table 3. For instance, we model the inference for Bert-1.3B with a latency of 151 ms. The authors verify the accuracy of the simulator by experiments and find that it has less than 2% error rate when compared to actual systems.

Operating System	Ubuntu 22.04 LTS
CPU Name	AMD RYZEN 9 7950X
CPU Architecture	X86_64
CPU Cores	16x2 (Hyperthreading)
Memory	64GB
GPU Name	NVIDIA GeForce RTX 4090
GPU Memory	24GB
GPU Compute Capability	8.9

Table 1: Machine setup

Environment 1	**Environment 2**	**Environment 3**
Python 3.10 ✓	Python 3.9 ✓	Python 3.8 ✓
CUDA 12.3 ✓	CUDA 11.3 ✓	CUDA 11.2 ✓
cuDNN [4] 8.9.7 ✓	cuDNN 8.2.0 ✓	cuDNN 8.1.0 ✓
Ray 2.8.0 ✓	Ray 2.1.0 ✓	Ray 2.1.0 ✓
Jaxlib [2] ×	Jaxlib 0.3.22 ✓	Jaxlib 0.3.22 ✓
cupy-cuda123 ×	cupy-cuda113 ✓	cupy-cuda112 ✓
Ubuntu 22.04 ✓	Ubuntu 20.04 ×	Ubuntu 20.04 ×

Table 2: Software environment Setup

Further simulator is also used to guide the model placement. Since SLO attainment doesn't have a closed-form objective function to maximize, alpaserv uses a simulator-guided model placement algorithm. The algorithm chooses the model placement which gives maximum SLO for known request traces. These request traces have to be known beforehand and we have used publically available Azure function traces. Since request traces over longer periods tend to follow similar patterns this is a good approximation of real-world request patterns. The placement here refers to the specific models chosen for a partition, the specific partition of clusters, and the parallel configuration for models in each partition. We compute all these parameters based on the maximum attainment of SLO across various configurations based on the simulation.

4 End-to-end Results and Observations

Workloads: We leverage existing production traces as a substitute for an open-source ML inference trace. Specifically, we rely on two traces from Microsoft Azure function invocations: MAF1[9]

Name	Size	Latency (ms)	S1	S2	S3
Bert-1.3B	2.4GB	151	32	0	10
Bert-2.7B	5.4GB	238	0	0	10
Bert-6.7B	13.4GB	395	0	32	10

Table 3: The initial three columns display the dimensions and inference speed of the models. The subsequent columns provide information on the number of instances for each model in distinct model sets identified as S1-S3.

and MAF2 [14]. These traces were originally collected over two weeks and were initially intended for Azure serverless function research. However, they have been repurposed for machine learning serving research.

These traces showcase distinct traffic patterns. MAF1 demonstrates a consistent and dense flow of incoming requests with gradual rate changes for each function. On the other hand, MAF2 exhibits a bursty traffic pattern, distributed unevenly across models, with some models receiving significantly more requests than others.

Baselines: We explore two distinct model serving strategies: Selective Replication (SR) and Clockwork++. Selective Replication (SR) adopts AlpaServe's placement algorithm without utilizing model parallelism. Clockwork++ represents an enhanced version of the state-of-the-art model serving system Clockwork[5]. The original Clockwork swaps models in and out of GPUs continuously, which is beneficial for small models but introduces significant overhead for larger ones. In our simulation, Clockwork++ follows Clockwork's replacement strategy at the boundary of every two windows in the trace, using SR's algorithm. However, we assume zero swapping overheads for a fair comparison.

SLO attainment vs. cluster size: In conducting this experiment, our focus was on assessing the performance of AlpaServe in comparison to two baseline methods across varying cluster sizes while serving a specific (model set, trace) pair. The results presented in Figure 8 demonstrate the superiority of AlpaServe throughout, showcasing its ability to outperform the baselines and achieve 99% SLO attainment with better efficiency. What sets AlpaServe apart is its utilization of model parallelism, a key factor that allows it to deliver comparable throughput to replication-only methods while employing significantly fewer devices and only a single replica of memory. The strategic placement of model replicas onto multiple devices enables AlpaServe to navigate bursty traffic and maintain robust performance.

SLO attainment vs. workload rate: We delved into the relationship between SLO attainment and workload rate, as illustrated in Figure 9. The variation in workload rates was a pivotal aspect of our experimentation, to gain insights into how AlpaServe handles diverse scenarios. Notably, when confronted with a stable trace like MAF1, AlpaServe demonstrated capacity to manage significantly higher rates compared to the baseline methods. This highlights the system's robustness and efficiency in handling workloads with consistent traffic patterns too. On the other hand, when dealing with a skewed and dynamically shifting trace such as MAF2, where traffic is dominated by a few models undergoing rapid changes, replication-based methods faced challenges. These methods necessitated the allocation of a substantial number of GPUs to create numerous replicas for popular models, attempting to cope with their bursty traffic. However, this approach led to periods of GPU idleness between bursts, despite frequent re-placement strategies as seen in Clockwork++. In contrast, AlpaServe's approach involved each model requiring fewer replicas to effectively manage its peak traffic which shows how AlpaServe sets itself apart from other serving systems in terms of handling bursty traffic.

SLO attainment vs. CV scale: This analysis (as shown in Figure 10) involved varying the burstiness of traffic, with higher CV values and increasing the risk of SLO violations. Traditionally, handling bursty traffic has often relied on over-provisioning, a resource-intensive strategy that tends to result in the inefficient use of resources. However, our findings with AlpaServe unveil a previously unrecognized opportunity for mitigating burstiness through the application of model parallelism. Unlike conventional approaches that resort to resource over-provisioning, AlpaServe leverages the inherent advantages of model parallelism to efficiently address bursty traffic without unnecessary resource wastage.

SLO attainment vs. SLO: In our experiment, Figure 11 explored the impact of different SLOs on system performance. Prior research often set SLOs to hundreds of milliseconds, even for small models with actual inference latencies well below

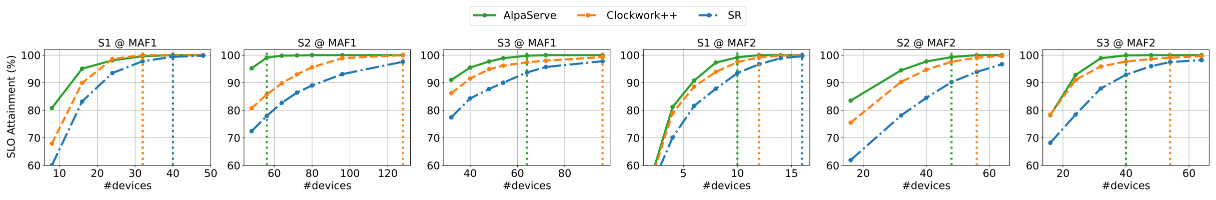

Figure 8: SLO attainment vs cluster size

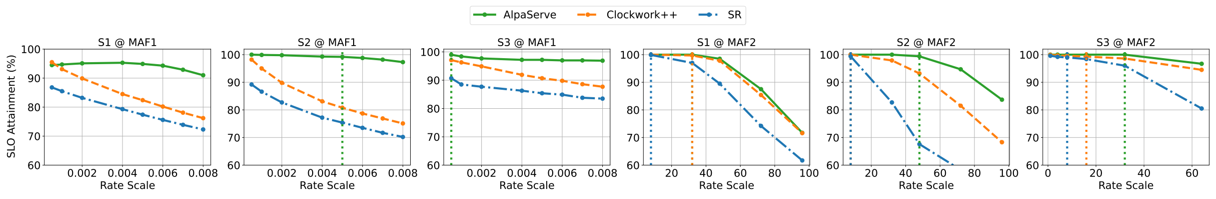

Figure 9: SLO attainment vs workload rate

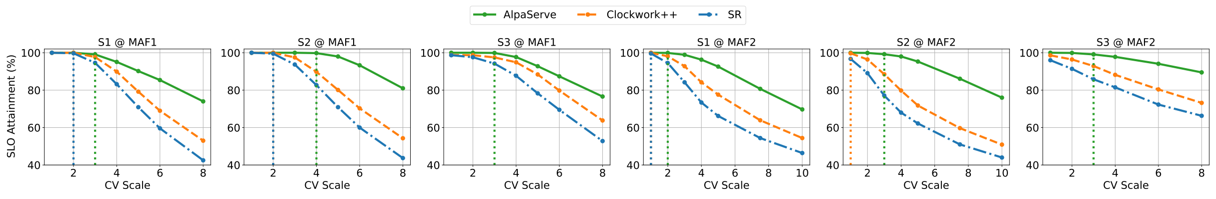

Figure 10: SLO attainment vs CV scale

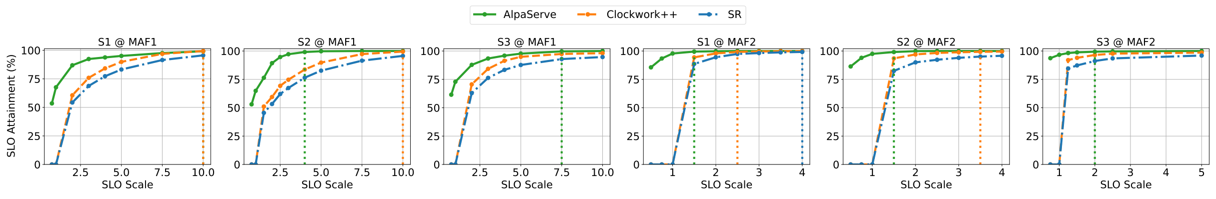

Figure 11: SLO attainment vs SLO

10 ms. AlpaServe's innovative approach, empowered by intra-operation parallelism, demonstrated its adaptability in maintaining robust performance across a spectrum of SLOs, particularly when serving large models with inference latencies exceeding 100 ms. Notably, under tight SLO constraints, where the target is even less than the model inference time, AlpaServe strategically leveraged intra-operation parallelism to reduce inference latency. This optimization, while introducing communication overhead and a subsequent reduction in peak throughput, enabled AlpaServe to fulfill more requests and meet stringent SLO requirements. As the SLO constraints became looser, AlpaServe dynamically transitioned to employ more inter-operation parallelism, effectively increasing throughput. This dynamic adjustment mechanism showcases AlpaServe's versatility in optimizing performance based on varying SLO requirements, making it a valuable and adaptive solution for diverse deployment scenarios with different latency constraints.

5 Learnings and Future Work

5.1 Learnings

In the paper, the authors present AlpaServe, a framework for serving multiple large deep learning models. Typically, model parallelism is only used when a single model is too large to fit on one accelerator. In those cases, we do not have any other option than using model parallelism. This paper asserts model parallelism can also benefit when serving multiple smaller models. The authors demonstrate situations where model parallelism improves performance, discussing associated tradeoffs. We particularly liked how the authors identified a problem, extended their previous work to solve this, and quantified the tradeoffs.

5.2 Future work on the project

While AlpaServe pioneered statistical multiplexing-based model parallelism for machine learning serving, its comparisons are currently limited to replication and Clockwork in the paper. As more serving frameworks emerge over time, it will be interesting to benchmark AlpaServe against them. AlpaServe also uses static provisioning and hence cannot autoscale. However, recent work like AsyFunc [8] shows promise for automatic scaling in serving frameworks. Some of the additional works are discussed in section 5.3. Overall, we believe serving of machine learning models is an exciting and rapidly developing space worth watching.

5.3 Related work

vLLM: vLLM is an open-source library built primarily for LLM inference based on efficient management of attention key and value memory using a technique called Paged Attention [6]. With the batching of requests and the dynamic nature of the size of the key-value cache for each request, there is a need for efficient cache memory management. vLLM not only achieves near-zero wastage of cache memory but also enables the sharing of the cache within and across requests.
DeepSpeed: Developed by Microsoft, DeepSpeed [1] is an inference system aiming to reduce latency and increase throughput for serving transformer models. This system proposes two solutions: A multi-GPU inference solution, for when the transformer models fit within the total GPU memory. A heterogeneous inference solution that utilizes CPU and NVMe memory in addition to GPU memory for situations when models exceed aggregate GPU memory.
Megatron-LM: Developed by Nvidia, unlike AlpaServe, DeepSpeed, and vLLM, which uses model parallelism for serving, Megatron-LLM [11] uses model parallelism for training. It uses a novel intra-layer model parallel approach to speed up the training of transformer models. This intra-layer technique can be combined with any inter-layer model parallelism methods.
FlexGen: Unlike other systems that focus on reducing inference latency, FlexGen [10] focuses on achieving high throughput in resource-constrained environments, such as a single GPU. FlexGen accomplishes this by compressing weights and attention cache to four bits, and by aggregating memory and compute resources from the GPU, CPU, and disk.
Orca: Orca [13] is one of the earlier distributed serving systems for Transformer-based generative models. Since transformers are auto-regressive models, Orca introduces a more granular per-iteration level scheduling instead of per-inference-request scheduling, along with more efficient selective batching. These two techniques improve both latency and throughput for LLM serving.
FastServe: FastServe [12] improves upon Orca by introducing a novel Skip-Join Multi-Level Feedback Queue Scheduler. It is a preemptive scheduler that reduces head-of-line blocking, thus minimizing overall job completion time. FastServe also presents an accompanying GPU memory management mechanism that automatically loads and evicts models between GPU memory and main system memory.

References

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

[2] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: Composable transformations of python+numpy programs. URL http://github.com/google/jax, 2018.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[4] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[5] 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.

[6] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. *arXiv preprint arXiv:2309.06180*, 2023.

[7] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, and et al. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. *arXiv preprint arXiv:2302.11665*, 2023.

[8] Qiangyu Pei, Yongjie Yuan, Haichuan Hu, Qiong Chen, and Fangming Liu. Asyfunc: A high-performance and resource-efficient serverless inference system via asymmetric functions. In *Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC '23)*, 2023.

[9] 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.

[10] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. *arXiv preprint arXiv:2303.06865*, 2023.

[11] 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:2104.08691*, 2019.

[12] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920,*, 2023.

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

[14] 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.

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