SpotServe: Serving Generative Large Language Models on Preemptible Instances

Xupeng Miao*
Carnegie Mellon University
Pittsburgh, PA, USA
xupeng@cmu.edu

Chunan Shi*
Peking University
Beijing, China
spirited_away@pku.edu.cn

Jiangfei Duan
The Chinese University of Hong Kong
Hong Kong, China
dj021@ie.cuhk.edu.hk

Xiaoli Xi
Carnegie Mellon University
Pittsburgh, PA, USA
xiaolix@andrew.cmu.edu

Dahua Lin
The Chinese University of Hong Kong
Hong Kong, China
dhlin@ie.cuhk.edu.hk

Bin Cui
Peking University
Beijing, China
bin.cui@pku.edu.cn

Zhihao Jia
Carnegie Mellon University
Pittsburgh, PA, USA
zhihao@cmu.edu

Abstract

The high computational and memory requirements of generative large language models (LLMs) make it challenging to serve them cheaply. This paper aims to reduce the monetary cost for serving LLMs by leveraging preemptible GPU instances on modern clouds, which offer accesses to spare GPU resources at a much cheaper price than regular instances but may be preempted by the cloud provider at any time. Serving LLMs on preemptible instances requires addressing challenges induced by frequent instance preemptions and the necessity of migrating instances to handle these preemptions.

This paper presents SpotServe, the first distributed LLM serving system on preemptible instances. Several key techniques in SpotServe realize fast and reliable serving of generative LLMs on cheap preemptible instances. First, SpotServe dynamically adapts the LLM parallelization configuration for dynamic instance availability and fluctuating workload, while balancing the trade-off among the overall throughput, inference latency and monetary costs. Second, to minimize the cost of migrating instances for dynamic reparallelization, the task of migrating instances is formulated as a bipartite graph matching problem in SpotServe, which uses the Kuhn-Munkres algorithm to identify an optimal migration plan that minimizes communication cost. Finally, to take advantage of the grace period offered by modern cloud platforms, we introduce stateful inference recovery, a new inference mechanism that commits inference progress at a much finer granularity and allows SpotServe to cheaply resume inference upon preemption. We evaluate SpotServe on real spot instance preemption traces and various popular LLMs and show that SpotServe can reduce the P99 tail latency by 2.4 - 9.1× compared with the best existing LLM serving systems. We also show that SpotServe can leverage the price advantage of preemptive instances, saving 54% monetary cost compared with only using on-demand instances. The code is open-sourced at: https://github.com/Hsword/SpotServe.

ACM Reference Format:
Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. 2023. SpotServe: Serving Generative Large Language Models on Preemptible Instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS'24).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Generative large language models (LLMs), such as ChatGPT [13] and GPT-4 [31], have demonstrated remarkable capabilities of creating natural language texts across various application domains, including summarization, instruction following, and question answering [27, 51]. However, the high computational and memory requirements of LLMs make it challenging to efficiently serve them on modern hardware platforms. To address this challenge, recent work has introduced a variety of approaches to parallelizing LLM inference by partitioning the LLM into multiple sub-models, each of which is deployed on a dedicated GPU. For example, GPT-3 includes 175 billion parameters and requires more than 16 NVIDIA A100-40GB GPUs to store the model parameters in single-precision floating points, which costs more

*Equal contribution.

ASPLOS'24, April 27-May 1, San Diego, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

than $66 per hour to serve a single inference pipeline for GPT-3 on AWS [13]. As the size of LLMs progressively increases, serving them on regular cloud GPU instances becomes prohibitively expensive for most organizations, especially those with limited budgets.

Modern clouds offer a variety of *preemptible* GPU instances (e.g., AWS spot instances and Azure spot VMs [1, 3]), which provides a more affordable approach to serving LLMs. These instances run on spare capacity on modern clouds at a price up to 90% lower than on-demand instances [3]. However, different from on-demand instances, spot instances may be preempted at any time when the capacity is needed by other instances. When a spot instance is preempted, modern clouds provide a *grace period* (e.g., 30 seconds for AWS spot instances), which allows the instance to complete running tasks and gracefully stop.

Prior work has introduced several DNN serving systems that leverage spot instances to reduce the monetary cost of DNN inference. Most of these systems (e.g., MArk [50], Cocktail [19]) target small DNN models that can fit on a single spot instance with one or multiple GPUs [23, 49], and handle preemptions using request rerouting [50] or redundant computation [23, 41]. While these approaches can effectively serve small models using data parallelism, they cannot scale to LLMs, serving which requires combining data, tensor model, and pipeline model parallelism [28, 40, 43, 54]. Model parallelism enlarges the minimal inference granularity from a single GPU instance to a group of instances (i.e., an inference pipeline), which requires more efficient methods to handle preemptions than rerouting and redundant computation, since preemptions are no longer independent and each preemption affects all other instances in the same inference pipeline.

This paper presents SpotServe, the first distributed generative LLM serving system on spot instances. SpotServe parallelizes LLM inference across multiple spot GPU instances by combining data, tensor model, and pipeline model parallelism, and produces identical results as serving the LLM using on-demand instances. Serving LLMs on spot GPU instances requires addressing three main challenges: (1) dynamically reparallelizing LLM inference, (2) cheaply migrating instances, and (3) effectively leveraging grace period. We elaborate on these challenges and the key ideas SpotServe uses to overcome them.

Challenge #1: dynamic reparallelization. Serving LLMs requires parallelizing the model parameters and computations across multiple GPUs using a combination of intra-operator (e.g., data and tensor model [8, 22]) and inter-operator (e.g., pipeline model [29, 54]) parallelization strategies. The first challenge SpotServe must address is the frequently changing number of available spot instances due to instance preemptions and acquisitions, which requires dynamically adapting the parallelization configuration to achieve optimized LLM serving performance, a problem we called *dynamic reparallelization*.

To address this challenge, SpotServe's *parallelization controller* dynamically adapts the parallelization strategy for serving LLMs in response to changes in spot-instance availability. SpotServe considers both the inference latency of a parallelization strategy and its serving throughput, and uses a hybrid optimization algorithm to balance the trade-off between throughput and latency. Dynamically reparallelizing LLM inference allows SpotServe to quickly adapt to changes to spot instances' availability and requests' arrival rates.

Challenge #2: instance migration. A second challenge SpotServe must tackle is minimizing the cost of migrating GPU instances for reparallelization. In particular, when transitioning to a different parallelization strategy, SpotServe must reinitialize all spot instances to incorporate new model parameters and establish new communication groups. Prior work on serving small DNN models on spot instances presumed negligible overheads to reinitialize a spot instance [19, 50]. However, we have observed that this assumption is not valid for LLMs, since restarting LLM serving from scratch results in substantial overheads. For example, loading a GPT model with 120 billion parameters from persistent storage takes more than 2 minutes on AWS.

To minimize the migration cost for reparallelization, SpotServe opportunistically reuses the model parameters and intermediate results such as key/value cache of an inference request (see Section 2) to avoid unnecessary communication between instances. The task of mapping available spot instances to the device mesh of a parallelization strategy is formalized as a *bipartite graph matching* problem in SpotServe, which leverages the Kuhn-Munkres (KM) algorithm to identify an optimal device mapping that minimizes the cost of migrating spot instances for reparallelization. In addition, to decide in which order to migrate instances, SpotServe's *migration planner* leverages the sequential execution order of pipeline stages to overlap instance migration with inference computation.

Challenge #3: grace period. Leveraging the grace period provided by modern clouds presents another challenge as the inference time for LLMs may surpass the grace period, therefore leading to unfinished requests. In existing spot-instance serving systems, these unfinished requests are generally rerouted to other inference pipelines, where the inference computation of these requests is restarted from the beginning. This approach does not efficiently use grace period and results in redundant computations.

To take advantage of grace period, SpotServe leverages the *autoregressive* nature of LLMs and introduces *stateful* inference recovery, which allows inference engines in SpotServe to commit their progress at the token level, rather than the request level as seen in prior work. SpotServe's inference

engine uses a *just-in-time* (JIT) arrangement to determine when to migrate the key/value cache of committed tokens to other available instances, which use the cached results to resume inference.

The above techniques allow SpotServe to significantly outperform existing approaches. We have evaluated SpotServe on real traces and a variety of LLMs and shown that SpotServe reduces the P99 tail latency by 2.4 - 9.1× compared with existing LLM serving systems. In addition, SpotServe can utilize spot instance to reduce the monetary cost for serving LLMs by up to 54% compared with the on-demand instance while preserving close average inference latency.

2 Background and Related Work

2.1 Generative LLM Inference

Generative LLMs usually stack several identical Transformer [44] layers and each layer is mainly made up of multi-head attention mechanisms and feed-forward networks (FFNs), as shown in Figure 1a. The generative LLM adopts the auto-regressive decoding mechanism, leading to an incremental inference process consisting of several iterations. We dive deeper into the iterative process to provide a better understanding of the generative LLM inference. Given a batch of input requests, the corresponding execution latency l_{exe} is divided into two components in E.q.(1):

$$l_{exe}(S_{out}|S_{in}) = t_{exe}(S_{in}) + \sum_{i=1}^{S_{out}} t_{exe}(S_{in}+i) \quad (1)$$

$$\approx t_{exe}(S_{in}) + S_{out} \times t_{exe}(1) \quad (2)$$

where t_{exe} indicates the LLM's execution time cost as a function of decoding sequence length, S_{in} is the sequence length of the input tokens provided the users, and S_{out} is the sequence length of output tokens the generated by the LLM. The first iteration is the *initial phase*, which takes all input tokens, processing them in parallel, and produces the first output token. After that, each *incremental decoding* iteration considers all input together with currently generated tokens and generates one output token. Figure 1a illustrates an example where the generative LLM takes "*ABCD*" as the input sequence (i.e., $S_{in} = 4$) and generates one output token in each iteration.

Existing generative inference systems (e.g., FasterTransformer [5], Orca [49], FairSeq [32], Megatron-LM [38]) use a key-value (KV) caching optimization that caches the keys and values of all Transformer layers in GPU device memory. The KV cache helps avoid recomputing preceding tokens during attention calculation, resulting in an almost constant per-iteration overhead (i.e., $t_{exe}(1)$ in E.q.(2) and Figure 1a). However, as the output sequence grows longer, the memory space of KV cache keeps expanding, which can be huge in real workloads (i.e., 1.7 GB per-sequence in LLaMA-13B [7], or even terabytes in OPT-175B [37]).

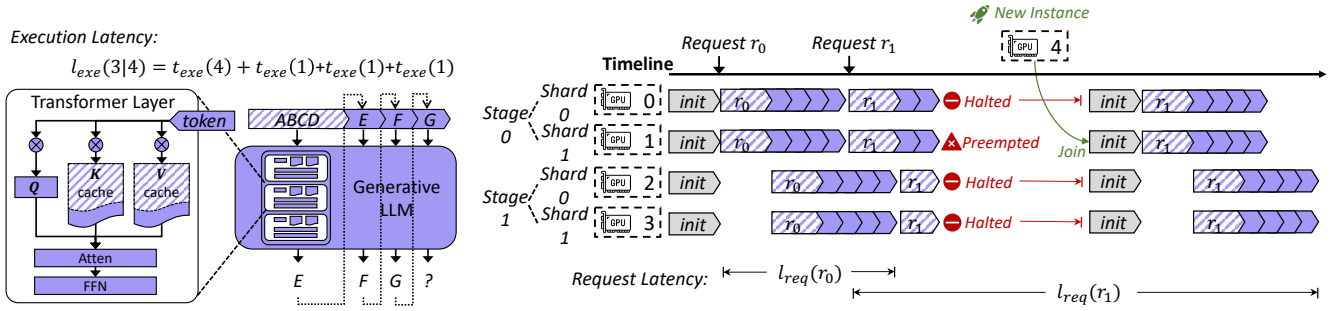
2.2 Distributed Inference of DNNs

Existing distributed DNN serving systems such as NVIDIA Triton [2] generally maintain multiple concurrent inference pipelines, each of which independently serves an inference engine such as FasterTransformer [5] on several GPU devices. An inference server receives input requests, partitions them into small batches, and dispatches them to these inference pipelines. All GPUs of each inference pipeline work collaboratively to perform DNN inference and send the output back to the inference server. For each inference request, its end-to-end inference latency l_{req} is divided into two parts: the scheduling overhead l_{sch} and the execution latency l_{exe}. The former is determined by the arrival rate of input requests and the peak serving rate of the inference system. If the arrival rate exceeds the peak serving rate, input requests cannot be processed in time, resulting in an increase of scheduling overhead. In this case, the inference system must improve the serving capability by improving the overall throughput. When the arrival rate is lower than the peak serving rate, l_{sch} still exists because the requests' arrival intervals can be non-uniform, in which case burst requests introduce scheduling overheads. All GPUs within an inference pipeline parallelize inference computation by combining two categories of parallel paradigms, as illustrated in Figure 2.

Inter-operator parallelism. Pipeline model parallelism [21] is the most representative inter-operator parallelism strategy, which groups operators into stages with data dependencies. Figure 2a shows an example of partitioning the model into two stages and each stage has half consecutive Tranformer layers. These stages can form a pipeline based on certain pipeline scheduling mechanisms [29, 30] that brings stage overlapping as well as cross-stage communications.

Intra-operator Parallelism. Tensor model parallelism [38] splits each DNN operator into several shards across the devices. As shown in Figure 2b, the corresponding tensors are also sharded based on certain distributed data layout. The participating devices compute in parallel and perform collective communications (i.e., All-Reduce) to transform the data layout if necessary.

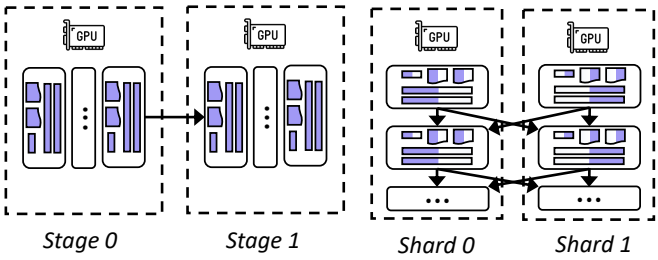
Note that both the data dependencies in pipeline model parallelism and the collective communications in tensor model parallelism do not naturally provide fault tolerance. The preemption of a single GPU instance can potentially hang all the other instances in the same inference pipeline. A preemption may also potentially break multiple inference pipelines if these pipelines are supported by different GPUs located on the same preempted instance. This chain crashing problem enlarges the affects of a single instance's preemption from itself to several pipelines. The affected instances are not physically terminated but stay idle until new instances are allocated to establish new inference pipelines.



(a) Incremental decoding in generative LLM

(b) Distributed LLM inference (P=2, M=2, batch size=1) on preemptible instances

Figure 1. Illustration of incremental decoding in generative LLM and distributed LLM inference on preemptible instances



(a) Pipeline model parallelism

(b) Tensor model parallelism

Figure 2. Illustration of different model parallelisms

2.3 Preemptible LLM Inference

Recent work has introduced a variety of techniques on how to handle instance preemptions when using cheap spot instances for DNN computation. For example, Varuna [12] maximizes training throughput by dynamically changing the hybrid data and pipeline parallel configuration after each instance preemption. Bamboo [41] uses a redundancy-based preemption recovery mechanism in pipeline parallel training by replicating each instance's computation on another spot instance. However, these techniques are designed for distributed DNN training and do not apply to generative LLM serving. Since distributed LLM inference is an important and timely research topic, accompanied by huge emerging demands in practice, it is obvious that serving LLM over spot instances could be a worthwhile attempt. Existing LLM serving systems such as FasterTransformer [5] do not provide any preemption handling capability for distributed LLM inference.

Figure 1b illustrates the obstacle of existing systems when serving LLMs on preemptible instances. We show an example of one distributed LLM inference pipeline deployed over 4 instances (one GPU per instance) through the combination of 2-way pipeline model parallelism and 2-way tensor model parallelism. The inference process starts after system initialization and it goes well for request r_0. But during the incremental decoding process of request r_1, GPU 1 unfortunately gets preempted at a certain timestamp and the other three GPUs has to be halted in the meanwhile. Due to the preemption, the inference state of r_1 (i.e., KV cache) is lost. When a new GPU instance is launched, it can join the inference pipeline and these 4 GPUs reinitialize and restart the inference process of r_1. As a result, the request latency can be significantly increased due to instance preemption handling.

3 SpotServe Design

The increased request inference latency caused by instance preemption is mainly manifested in three aspects. Firstly, once a preemption happens, the entire inference pipeline comes to a halt, which may result in request waiting overhead and/or additional request scheduling overhead (i.e., rerouting to another inference pipeline). Secondly, after a new instance joins, there are necessary system initialization costs, such as launching the distributed inference engine and loading model parameters. Finally, throughout this process, the overall reduction in system throughput can potentially lead to an accumulation of subsequent incoming requests, thereby amplifying their inference latency.

We develop SpotServe to mitigate the impacts of these issues on the end-to-end inference latency. First, to alleviate the waiting time caused by the integration of new instances, SpotServe facilitates the integration of on-demand instances to ensure swift instance acquisition. Second, to reduce the runtime overhead of system re-initialization, SpotServe introduces an efficient context management mechanism that leverages inter-instance network links to preserve inference progress (in the form of KV cache) and obviate the need for expensive model parameter reloading. Third, to strike a better balance among serving throughput, latency, and monetary cost during node availability fluctuations, SpotServe incorporates a workload-aware adaptive configuration optimization algorithm, which dynamically selects an optimal parallel configuration, enabling real-time dynamic context migration and seamless configuration transitions.

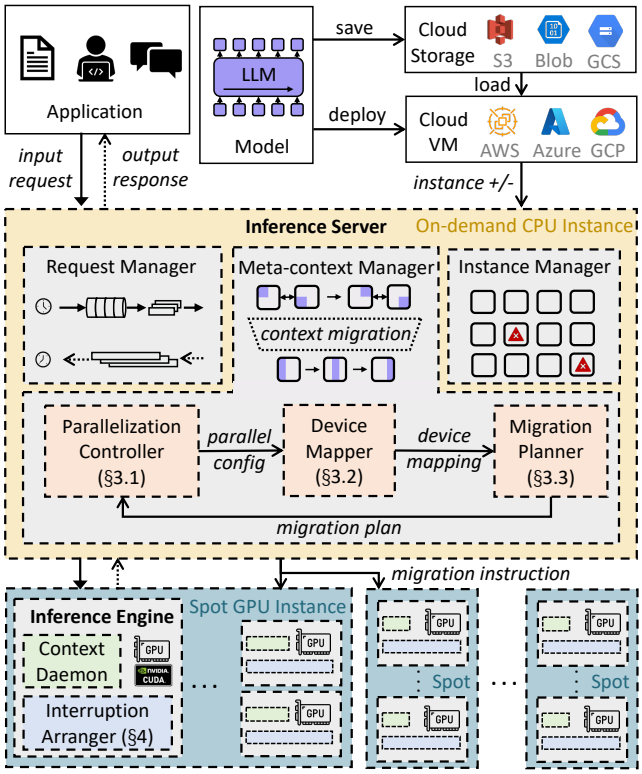


Figure 3. An overview of SpotServe.

3.1 System Overview

Figure 3 illustrates an overview of SpotServe. The inference server is deployed on a dedicated on-demand CPU instance and hosts a request manager, a meta-context manager, and an instance manager. The *request manager* is responsible for receiving input requests, dynamically partitioning them into batches, assigning these batches to inference instances running on spot GPU instances, and ultimately collecting generated outputs from the inference instances, sending the results back to users. The *instance manager* interacts with the cloud and receives instance preemption/acquisition notifications.

SpotServe's inference engine is deployed on each spot or on-demand GPU instance to serve LLM inference. Each inference engine includes a *context daemon* that manages the model parameters (i.e., model context) and intermediate activations (i.e., cache context) for different requests inside a certain GPU. The inference engine can access these context information through the proxy provided by the context daemon. If the inference engine has to be interrupted due to the preemption of dependent instance, the context daemon process is still alive and avoids to reload the context into GPU when restarting inference.

When the system's serving capability becomes incompatible with the workload or is about to, the *meta-context manager* manages the adjustment of the parallel configuration by sending instructions for context migration to all GPU instances. The new configurations are proposed by the *parallelization controller* and materialized by the *device mapper* and *migration planner*. Each inference engine also launches an *interruption arranger* to support stateful inference recovery for lower inference latency.

Algorithm 1 Adaptive configuration optimizer.

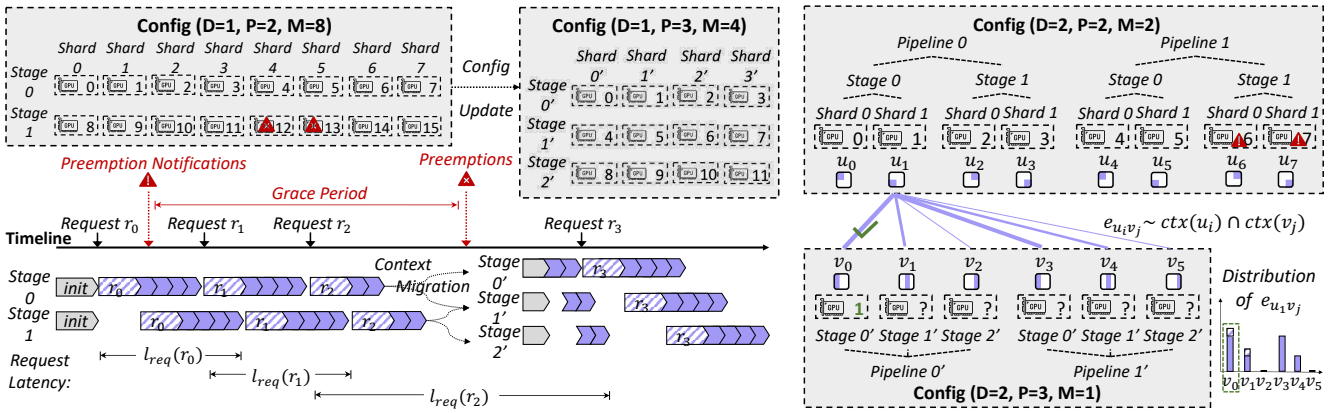
function ConfigOptimizer(N_t, C_t, α_t)
  if $\exists C.\phi(C) \geq \alpha_t$ **and** cloud has enough instances for C **then**
    $C_{t+1} \leftarrow \arg\min_{C|\phi(C)\geq\alpha_t} l_{req}(C)$
  else
    $C_{t+1} \leftarrow \arg\max_{C|N_t} \phi(C)$
  $\Delta \leftarrow$ `#Instances`$(C_{t+1}) - N_t$
  if $\Delta > 0$ **then**
    `InstanceManager.alloc`$(\Delta, ondemand\_and\_spot)$
  else
    `InstanceManager.free`$(-\Delta, ondemand\_first)$
  `ConfigUpdate`(C_t, C_{t+1})

For the rest of this paper, we first introduce the SpotServe's design, including parallelization controller in §3.2, device mapper in §3.3, migration planner in §3.4, and interruption arranger in §4. Finally, we introduce SpotServe's implementation in §5 and evaluate its performance in §6.

3.2 Parallelization Controller

SpotServe uses parallel configurations to identify a strategy to parallelize LLM serving across multiple GPU instances. A *parallel configuration* is represented as a tuple $C = (D, P, M, B)$, where D, P, and M indicate the data, pipeline-model and tensor-model parallelism degrees, and B is the maximum mini-batch size. A key difference between SpotServe and existing spot-instance serving systems is that SpotServe can *proactively* adjust its parallel configuration by leveraging the ahead-of-time notifications provided by the cloud to handle instance preemptions and acquisitions. For each preemption and acquisition notification, SpotServe's parallelization controller opportunistically adjusts the parallelization configuration to improve LLM serving performance. Such reparallelization mechanism is also adaptive for fluctuating inference workload, which has been extensively studied in prior approach [50].

Grace period of spot instance. Modern clouds generally offer a grace period (e.g., 30 seconds on Azure [3]) to allow a spot instance to complete running tasks before preempting the instance. Allocating new instance doesn't have a grace period, but initializing inference engine also takes a short period of time (e.g., 2 minutes for launching and initializing in our evaluations), which can be measured in advance and treated as the acquisition grace period in SpotServe.



(a) Illustration of configuration update and context migration in SpotServe.

(b) Illustration of device mapping in SpotServe.

Figure 4. Figure 4a shows an example of SpotServe changes the parallel configuration from (1,2,8) to (1,3,4) through context migration within the grace period and continues previous decoding progress of request r_3. Figure 4b shows an example bipartite graph between six available instances (i.e., $u_0 \sim u_5$) and topology positions in the new configuration (2,3,1). Here we only draw the weighted edges starting from u_1.

Adaptive configuration optimizer. SpotServe uses an adaptive optimization algorithm to balance the trade-off among throughput, latency, and cost. We use two time-varying variables C_t and N_t to denote the parallel configuration and the number of available instances at time step t. Note that N_t considers instances in the grace period, which includes newly allocated instances and excludes instances to be preempted. Let $\phi(C)$ to be the serving throughput with the parallel configuration C and α_t be the request arrival rate at time step t[1]. Algorithm 1 shows the workflow of the optimizer, which mainly works when the current serving capability is not compatible with α_t due to changes in instances' availiability or serving workload.

Overall, the optimizer minimizes the end-to-end inference latency $l_{req}(C)$ while maintaining a throughput higher than α_t (line 3). Specially, if there are multiple configurations that can achieve similar minimum inference latency, SpotServe selects the configuration with lower monetary cost (i.e., using fewer instances). Note that, in addition to minimizing $l_{req}(C)$, other targets are also feasible, such as meeting the requirements of pre-defined SLO (i.e., $l_{req}(C) \leq l_{SLO}$). When SpotServe's peak serving throughput can not exceed the request arrival rate α_t (i.e., $\nexists C.\phi(C) \geq \alpha_t$), SpotServe updates its parallel configuration to maximize the overall serving throughput (line 5). The suggested configuration C_{t+1} may require more or less instances than before (line 6). Since the allocation of spot instance might not always success, SpotServe supports to optionally allocate on-demand instances to further improve serving throughput. Specifically, the instance manager allocates on-demand and spot instances at the same time (line 8) to avoid the waiting overhead when spot-instance allocation fails. The instance manager is also in charge of releasing the allocated instances (line 10) to alleviate over-provision, where on-demand instances have higher priority due to their costs. To alleviate the impacts of frequent disturbance of instance availability, SpotServe often keeps few addition instances (e.g., two in the experiments of §6) as a candidate pool for smoother instance substitution. Finally, SpotServe updates the parallel configuration (line 11), and the interruption arranger (§4) decides when to complete reparallelization, especially for the cases triggered by instance availability changes. This step is still necessary even when $C_{t+1} = C_t$, since instance preemptions and acquisitions update instances' memberships.

The optimizer runs online and has negligible overhead (i.e., less than 1s) since the latency estimation of different configurations is done offline in advance. SpotServe's configuration exploration space is much larger than prior approach like Varuna [12] which only considers data and pipeline parallelism. It is also possible to extend SpotServe to more complicated model parallelisms [43, 54], which we leave as future work.

3.3 Device Mapper

Given the target configuration C_{t+1}, a straightforward approach to migrating instances is to restart the inference engines on current instances and reinitialize all available GPU instances from scratch. However, this approach does not leverage the opportunity to reuse the model parameters and KV cache available on existing GPU instances, resulting in unnecessary migration cost and inference delay. As shown in Figure 4a, instead of destroying and rebuilding

[1]Since the request arrival rate might change randomly, we estimate α_t by observing the request arrivals within a short past duration (e.g., 30s).

these context, SpotServe adopts a more lightweight context migration mechanism and can resume interrupted requests' inference. As migrating these context information among GPU instances may also increase latency, a key challenge SpotServe must address is mapping the available GPU instances to the logical device mesh identified by the new parallel configuration to opportunistically reuse previous context. We ignore batch size and use (D, P, M) to denote a parallel configuration, where D, P, and M indicate the data, pipeline model, and tensor model parallel degrees. SpotServe binds each GPU instance with a pipeline-stage-shard topology position (d, p, m), which represents the m-th shard $(1 \leq m \leq M)$ of the p-th stage $(1 \leq p \leq P)$ in the d-th pipeline $(1 \leq d \leq D)$.

To switch between different parallel configurations, SpotServe formalizes device mapping as a *bipartite graph matching* problem, and uses the Kuhn-Munkres (KM) algorithm to find an optimal device mapping that minimizes total data transmission during context migration.

Bipartite graph matching. SpotServe uses a bipartite graph $\mathcal{G} = (\mathcal{V}_a, \mathcal{V}_t, \mathcal{E})$ to describe device mapping, where each node $u \in \mathcal{V}_a$ is a GPU device, each node $v \in \mathcal{V}_t$ represents a pipeline-stage-shard position of the parallel configuration, and a weighted edge e_{uv} $(u \in \mathcal{V}_a, v \in \mathcal{V}_t)$ indicates the amount of *reusable* model parameters and key/value cache when mapping GPU u to position v of the parallel configuration. As shown in Figure 4b, given the current state of each GPU's context daemon (i.e., organized as $(D = 2, P = 2, M = 2)$) and a target parallel configuration $(D = 2, P = 3, M = 1)$, SpotServe builds a complete bipartite graph and computes the edge weight between every (u, v) pair using the size of their intersection contexts. For example, u_1 holds half sharded context of the first stage in the first pipeline, and overlaps the most model context with v_0 and v_3 since they are in charge of the first stage of the new pipeline. Suppose the new pipeline $0'$ inherits the interrupted inference requests from pipeline 0, we may prefer to match u_1 with v_0 as it has more cache context to reuse. SpotServe transforms the optimal device mapping problem to a bipartite graph matching task and uses the KM algorithm to find a maximum-weight match, which maximally reuses the model parameters and KV cache on available GPU instances and minimizes the total data transmission.

SpotServe also considers the cases when each instance has multiple GPUs with higher inter-GPU bandwidth (e.g., NVLink). We facilitate the hierarchical architecture by conducting a two-step matching (i.e., intra-instance and inter-instance) to discover an optimal solution. More details can be found in the supplemental material.

When the new parallel configuration C_{t+1} handles less concurrent inference requests than the original configuration C_t (i.e., $D_t \times B_t \geq D_{t+1} \times B_{t+1}$)[2], SpotServe discards part of the cached results to avoid exceeding the memory capacity of the new parallel configuration. To minimize the recomputation cost, SpotServe keeps the batches of requests with more decoding progresses (i.e., iterations).

Algorithm 2 Workflow of the SpotServe migration planner.

▷ Progressive Migration
function MigrationPlanner(context `ctx`, `plan` = [ ])
  `plan`.append(<migrate, `ctx.cache`>)
  $\mathbf{O} \leftarrow$ Layer migration order from MemOptMigPlanner
  for layer index i in `range`(0, #layers) **do**
    `plan`.append(<migrate, `ctx.weight`[$\mathbf{O}_i$]>)
    $p \leftarrow$ Get pipeline stage index of layer $\mathbf{O}_i$
    if stage p completes all context migration **then**
      `plan`.append(<start, instances of stage p>)
▷ Memory Optimized Migration
function MemOptMigPlanner(maximum buffer size U_{max})
  $\mathbf{O} \leftarrow$ [ ], $\mathbf{X} \leftarrow$ { }
  Instance buffer memory usage $\mathbf{U} \in \{0\}^N$
  for layer index i in `range`(0, #layers) **do**
    if (migrate, `ctx.weight`[i]) doesn't exceed U_{max} **then**
      Update buffer memory usage $\mathbf{U}$
      $\mathbf{O}$.append(i)
    else
      $\mathbf{X}$.add(i)
  while X is not empty **do**
    $x_{opt} \leftarrow$
      $\arg\min_{x \in \mathbf{X}} \max_{0 \leq i \leq N-1} \{\mathbf{U}_i \mid (\text{migrate}, \texttt{ctx.weight}[x])\}$
    $\mathbf{O}$.append(r_{opt})
    $\mathbf{X}$.remove(x_{opt})

3.4 Migration Planner

After mapping the available devices into the logical parallel positions, the next challenge is to determine the exact migration plan to finish the configuration adjustment. A naive approach is to make all instances follow a default tensor transmission order and wait until all instances' context are successfully transferred. This solution mainly has two drawbacks. One problem is that sending all context might be time-consuming especially for large models. To alleviate the context migration overheads, we propose a *progressive migration* schedule that utilizes the pipeline structure and prioritize the migration of front model layers' context. Then the front pipeline stages' instances can start serving, which can be potentially overlapped with the following stages' migration. Ideally, the context migration overheads could be reduced into the cost of a single stage's context transferring. Note that, we also prioritize the transfer of all layers' cache context considering the interruption fault-tolerance.

[2]Recall that in a parallel configuration $C = (D, P, M, B)$, D and B indicate the number of inference pipelines and the batch size of each pipeline, respectively. Therefore, $D \times B$ is the total number of concurrent requests.

Although it does not achieve the maximum overlapping, it can minimize the possibility of decoding progress lost.

Another problem is the memory consumption of the buffer space for context communication. The migration of every context tensor changes the runtime memory usage. Specifically, the sender's memory can be released while the receivers' memory consumption will increase. An improper migration plan may significantly increase the peak memory usage and leads to sub-optimal inference configurations (e.g., splitting the model into more stages) with higher latency. To provide a memory efficient migration plan, we propose to consider the memory usage during the progressive migration process. As shown in Algorithm 2, we start from a naive plan in sequential order of the layer index (line 12), applies the context migration of each layer (line 13), and tracks the buffer memory usage of each instance (line 14). The algorithm requires a default hyper-parameter U_{max} to represent the maximum threshold of buffer memory consumption for every instance. We first skip those layers whose context migration might exceed the buffer memory upper bound (line 17). After that, it generates the order of the rest layers by solving a min-max problem (line 19). In particular, it prefers to selects the layer whose context migration can minimize the maximum instance buffer memory usage. In this way, the combined layer context migration order has lower memory consumption and can be used to generate the final migration plan (line 3).

4 Stateful Inference Recovery

This section introduces *stateful inference recovery*, a new inference mechanism that allows SpotServe to recover interrupted inference request without recomputation. In addition, we discuss SpotServe's mechanism to handle frequent interruptions.

4.1 Just-in-time Arrangement

When instance preemption or acquisition notifications trigger reparallelization, SpotServe must decide when to terminate the inference engine and start the context migration for each GPU instance. A conservative approach is to immediately suspend the inference engine to preserve enough time for context migration. However, this approach would interrupt all active requests on the instance. These unfinished requests must be rerouted and restarted on other inference pipelines, resulting in high end-to-end inference latency. An aggressive alternative is to finish all active requests first, which might prevent the instance from finishing migration before the preemption.

To avoid these problems, SpotServe leverages the grace period offered by the cloud to opportunistically commit inference progress at the token level, which allows an inference request to be interrupted at any incremental decoding iteration. Since SpotServe's context daemon maintains the state (i.e., cache context) of an inference request, the request can be rerouted to another inference pipeline, which can directly continue its inference using the cached state without recomputing previously generated output tokens.

To determine how many iterations to run during a grace period, SpotServe adopts *just-in-time (JIT) arrangement* and let the inference engine decide when to stop decoding. Specifically, each spot GPU instance includes an *interruption arranger* that receives a notification when a grace period starts. Based on this notification, the interruption arranger checks the remaining time before feeding a new batch of requests into the inference engine. Suppose a batch of input requests are ready to serve at time t, SpotServe determines the number of the decoding iterations S_t differently based on the interruption type. For instance preemption, we have $S_t = \arg\max_{0\le S\le S_{out}}\{l_{exe}(S \mid C_t) < T^- - T_{mig}\}$, where $l_{exe}(S \mid C_t)$ is the execution latency for generating S tokens with C_t, T^- is the remaining grace period for the preemption, and T_{mig} is the cost of migrating instances for reparallelization. For instance acquisition, we also have $S_t = \arg\min_{0\le S\le S_{out}}\{l_{exe}(S \mid C_t) \ge T^+\}$, where T^+ is the remaining grace period for the acquisition (i.e., initialization). A key difference between these two arrangements is that we maximize the arranged iterations before preemption and minimize that before acquisition. The reason is that, unlike in-advance preemption handling, the context migration occurs after instance acquisition. Besides, both cases should also guarantee that the arrangement will not increase the request's latency (i.e., $T_{mig} < l_{exe}(S_t \mid C_t)$). For example, if the left time is only able to generate few tokens, simply rerouting might be better as it doesn't add context migration overheads, especially when the arrival requests are spare.

4.2 Interruption Fault-tolerance

One problem of the recovery approach is that previous arrangements only consider single interruption cases. For multiple consecutive and compact interruptions, their grace periods might overlap with each other and be insufficient to finish the arranged iterations or migrate the context. Another problem is that if we underestimate the migration costs due to unforeseen reasons (e.g., network vibration), the remaining time might also not be enough for the instance to follow the arrangements.

To build a reliable serving system, SpotServe has several fault-tolerance mechanisms to handle the failures. First, SpotServe manages to delay the acquired instance joining and make the arrangements for prior interruptions feasible. Second, if one instance indeed gets preempted before expected, SpotServe has to give up the cache context and only migrates the model context with the rest instances. Specially, when all replicas of the same piece of model context are lost due to unexpected failures, the migration can not work and SpotServe has to restart by loading weights locally (e.g., disk)

Table 1. Overview of LLMs evaluated.

Model	Size	min #GPUs	(P, M)	$l_{exe}(B=1)$
OPT-6.7B	25.0 GB	4	(1,4)	5.447s
GPT-20B	74.5 GB	12	(3,4)	14.373s
LLaMA-30B	111.8 GB	16	(2,8)	17.540s

or from remote cloud storage (e.g., S3) to fetch the required model parameters.

5 Implementation

We implement the inference server of SpotServe in 5.6K LoC in C++ and 2.2K LoC in Python, including three resident processes responsible for request manger, instance manager and meta-context manager respectively. The generated migration plan is organized in a JSON format and sent to running instances with a TCP connection. We build SpotServe's inference engine over FasterTransformer [5], a highly optimized Transformer inference framework built on top of CUDA, cuBLAS [16], and C++. We implement our context daemon and interruption arranger inside the inference engine. Specifically, the memory allocation of model context and cache context in FasterTransformer has been replaced by acquiring the corresponding GPU tensors from the context daemon. The context migration operations are implemented by the batched asynchronous NCCL send/recv primitives [6]. The context migration requires additional communication buffer space in GPU memory, which is dynamically allocated and released based on the migration plan. Since the context daemon and FasterTransformer belong to two different process, we involve CUDA Inter-Process Communications (IPC)[4] to share the context pointers. To support overlapping in progressive migration, we add a mutex lock to each context tensor to block the inference before its migration is finished. We also design a cost model and implement a offline profiler over SpotServe to estimate the required inference latency, system throughput and the context migration overheads in advance. To make the estimation more accurate, we carefully considers the resource under-utilization affects (i.e., GPU, network, PCIe) due to several practical factors (e.g., rarely small batch size, single input token, over-sharded intra-op parallelism, GPU memory accessing, and too small communication data volume) during cost profiling and modeling.

6 Evaluation

6.1 Experiment Setup

Baseline. To our knowledge, SpotServe is the first distributed LLM inference system for spot instances. Therefore, we build two baseline systems on top of FasterTransformer by generalizing two representative ideas of prior approach respectively. One approach is request *rerouting*, which dynamically reroutes interrupted requests to other available

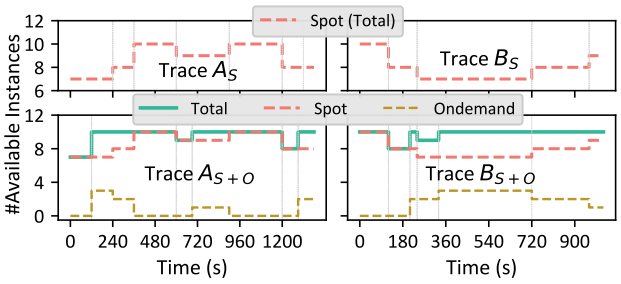


Figure 5. Trace A_S and B_S are extracted from real trace, while A_{S+O} and B_{S+O} are traces created by Algorithm 1 mixing on-demand instances based on A_S and B_S. Each instance has four GPUs.

pipelines when preemption happens. It takes a fixed predefined optimal model parallel configuration and drops/adds the inference pipeline adaptively. Another baseline is model *reparallelization*, which changes the parallel configuration like ours, but has to restart and reinitialize all instance without context migration. Both of them handle preemption in a reactive manner that has to interrupt the current requests' inference and recompute later. They are implemented with the same inference engine as SpotServe to avoid unfairness in the backbone system. Redundancy-based approaches, which serve several model replicas at the same time, are not included due to the huge cost of LLMs.

Models. We evaluate SpotServe on three LLMs with different scales, including OPT-6.7B [53], GPT-20B [33], and LLaMA-30B [42]. Table 1 summarizes the minimum number of GPUs to serve these models and the corresponding model parallel strategies and their single-request execution latency.

Setting. We collect a real 12-hour availability trace with AWS `g4dn` spot instance and extract two representative 20-minute segments (i.e., A_S and B_S in Figure 5) with different dynamic behaviors. For reproducible comparisons, we replay the traces on AWS `g4dn.12xlarge` instances (4 NVIDIA Tesla T4 GPUs per instance) in our evaluations. We include both stable and fluctuating inference request arrival workloads. For static workloads, considering that different models have different computational requirements, we set different request arrival rates for them (i.e., 1.5, 0.35 and 0.2 requests/s for OPT-6.7B, GPT-20B and LLaMA-30B by default respectively). To simulate the bursty in real workloads [26], we use Gamma request arrival process with a coefficient of variance (CV) of 6. Moreover, we separately studied the system performance under the condition of whether to allow mixing with on-demand instances. To achieve that, we generate another two traces (i.e., A_{S+O} and B_{S+O} in Figure 5) following Algorithm 1 to opportunistically allocate on-demand instances and mix up with spot instances. For dynamic workloads, we include a production trace MAF [35] publicly released by

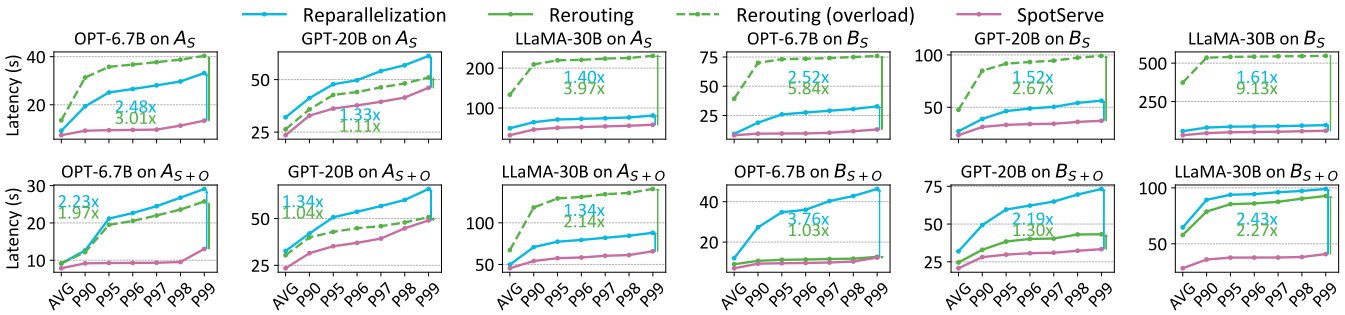


Figure 6. End-to-end serving performance comparison among SpotServe, Reparallelization, and Rerouting. The x-axis shows the average and various tail latencies achieved by different approaches, while the numbers report SpotServe's latency improvement compared to the baselines.

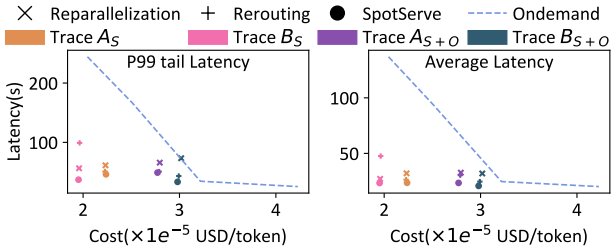


Figure 7. Monetary cost comparison on GPT-20B.

Microsoft and discuss in §6.3. During the optimization, the maximum batch size B is selected from {1,2,4,8}, S_{in} is 512 and S_{out} is 128.

6.2 Comparison on Stable Workload

End-to-end inference latency. Figure 6 shows the latency performance of all three models on stable workloads. SpotServe defeats both Rerouting and Reparallelization in terms of all latency metrics on four different traces. Taking the P99 latency as an example, SpotServe outperforms Reparallelization and Rerouting around 1.34-2.43× and 2.14-9.13× respectively on the largest LLaMA-30B model. The improvement mainly comes from three aspects: the dynamic re-paralleization, efficient proactive migration and the stateful inference recovery.

Compared with Reparallelization, the most advantage of SpotServe is its lightweight context migration mechanism. Prior approach like Varuna requires system restarting for each reparallelization and their context has to be lost. Reloading all model parameters and then recompute all interrupted requests will incur long tail latency.

Compared with Rerouting, SpotServe can support more fine-grained preemption handling, instead of dropping the entire inference pipeline. Many cases of Rerouting in Figure 6 are marked with dashed line, representing overload (i.e., the system serving capability becomes lower than the request arrival rate and request accumulation happens). Taking GPT-20B as an example, when the instance availability is high ($\geq$ 8 instances), Rerouting supports a configuration of $(D = 2, P = 2, M = 8)$ with minimum inference latency and sufficient system throughput (i.e., larger than 0.35 requests/s). Once an instance gets preempted, Rerouting has to drop one inference pipeline and degenerates to $(D = 1, P = 2, M = 8)$, which makes upcoming requests stacked and unable to be served in time. However, SpotServe will serve with $(D = 2, P = 3, M = 4)$ to avoid overload. SpotServe may occasionally propose the same configuration as Rerouting, but SpotServe should still have superior performance because of the KV-cache recovery. Another observation is that mixing on-demand instances helps alleviate the overload due to the faithful instances acquisitions.

Monetary cost comparison. Besides inference latency, we also study their monetary cost to see whether it is cost-effective to serving LLM using preemptive instances. Figure 7 presents the per-token costs of all baseline systems and their latency on GPT-20B model. We also show the results (with the dashed line) of only using on-demand instance, which is more expensive than spot instance (i.e., 3.9 USD/h v.s. 1.9 USD/h). As the cost decreases, the latency of on-demand instances exhibits a significant increase since it is unable to meet the required serving capability with fewer on-demand instances. In contrast, serving with economical spot instances hit a balance between inference latency and monetary cost. SpotServe significantly saves the cost up to 54% while tolerating a relatively modest increase of less than 18% in average latency and 90% in P99 tail latency. Such cost advantage would be more significant on other types instances with higher on-demand/spot price ratio.

Ablation study. Figure 9 shows the P99 tail latency and average latency of GPT-20B on two traces with different SpotServe components. We start from SpotServe and gradually disables each system optimization one by one. By removing the parallelization controller, the tail latency improves 179%

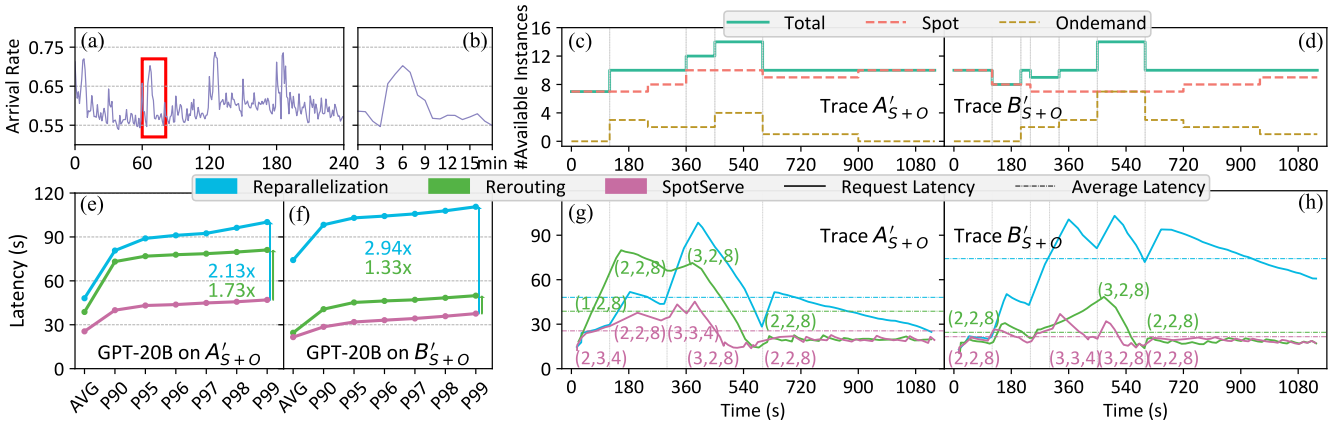


Figure 8. (a) Rescaled MAF trace. (b) The selected trace segment. (c)(d) Two traces based on the fluctuating workload trace. (e)(f) End-to-end serving performance. (g)(h) Per-request latency throughout the traces, and parallel configurations (D,P,M) after each re-parallelization. Note that the configuration of Rparallelization is always consistent with SpotServe.

on trace B_S. This is because the parallelization controller suggests switching to a new configuration with higher throughput to handle the stacked requests. If we further disables the migration planner, the tail latency improves to 1.4× and 3.1× on traces A_S and B_S respectively. Another important point is that the memory efficient migration planner also reduce the minimum number of GPUs to serve GPT-20B model from 16 to 12, which enlarges the former parallelization configuration exploration space. The interruption arranger also contributes to 29% tail latency reduction on trace B_S as it transfers the cache context during migration and avoids redundant computation for interrupted requests. Finally, after removing the device mapper, the system degrades to a plain approach only enables model context maintenance without any other optimizations. On the whole, all these optimizations helps SpotServe reduces the tail latency by 1.61× on trace A_S and 3.41× on trace B_S respectively.

6.3 Comparison on Fluctuating Workload

To study SpotServe's auto-scaling performance, we replay a piece of MAF [35] trace and rescale its arrival intensity like prior approach [10, 18, 52] to make it compatible with our experiment setup. Figures 8a and 8b show that the selected trace includes fluctuating and bursty workload, which is representative in real-world environments. We enable mixing with on-demand instances in this experiment and the generated instance availability traces (i.e., A'_{S+O} and B'_{S+O}) are listed in Figures 8c and 8d. The end-to-end inference latency statistics are shown in Figures 8e and 8f, and SpotServe reduces up to 2.94× and 1.73× P99 tail latency compared with Reparallelization and Rerouting, respectively.

Per-request latency study. Figures 8g and 8h show each arrival request's inference latency over time for both traces. SpotServe almost always performs the lowest latency during

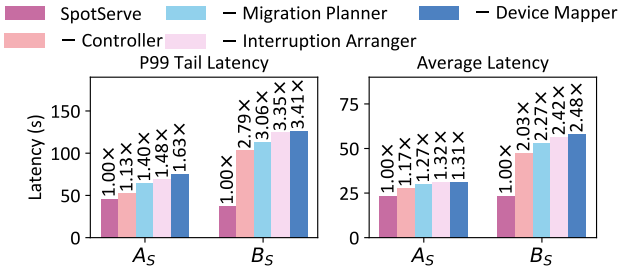


Figure 9. Ablation study of GPT-20B on traces A_S and B_S.

the whole trace due to the flexible parallel configuration optimization and the lightweight context migration. In the following, we take Figure 8h as an example for in-depth analysis. First, all approaches start with a feasible configuration of $(D = 2, P = 2, M = 8)$ as there are ten spot instances available at $t = 0$s. Preemption first occurs at $t = 120$s and $t = 240$s but the total available instances are still enough to support $(D = 2, P = 2, M = 8)$. From $t = 270$s, the increasing arrival rate overwhelms the system processing capacity. After 30s, such overload is detected and both SpotServe and Reparallelization change the configuration to $(D = 3, P = 3, M = 4)$. The instance acquisition completes at $t = 450$s, so they change to $(D = 3, P = 2, M = 8)$ for lower latency. But Reparallelization is suffering from expensive restarting overheads, resulting in the highest peak latency. Rerouting only changes the number of pipelines and incurs some request waiting overheads. After $t = 600$s, the arrival rate decreasing is detected and on-demand instances start to be released, then both SpotServe and Rerouting turn back to $(D = 2, P = 2, M = 8)$. As a result, SpotServe significantly outperforms the other two baselines for the fluctuating workloads.

7 Related Work

DNN inference system. The widespread DL applications bring great market values and lead to significant DL serving traffics. Some prior approaches (e.g., Clipper [15], Clockwork [18], Nexus [36], and so on) consider temporal multiplexing and increase the GPU utilization through batching and better scheduling. INFaaS [34] studies the model selection problem when considering multiple models with different inference efficiency or accuracy. Shepherd [52] considers both the resource utilization and the serving system effective throughput and improves the request scheduling. There are also some inference systems take customized GPU kernel optimization for Transformer models, like TurboTransformer [17] and LightSeq [45]. Some recent inference systems (e.g., FasterTransformer [5], Orca [49], FairSeq [32], DeepSpeed [11], AlpaServe [26]) support LLM inference by leveraging the parallelization techniques from distributed training approaches. Among them, AlpaServe is designed for resource multiplex scenarios and does not show performance superiority in our empirical study on single LLM inference (i.e., around 3× lower than FasterTransformer C++ version). Almost all of these prior work are designed for dedicated instances and can not tolerate instance preemptions.

ML Serving over Spot Instance. Previous approaches have also involved spot instances into ML inference systems for small ML models. Cocktail [19] leverages cheap spot instances to increase the number of ensembling models and instance preemptions can lead to certain intermittent loss in accuracy. MArk [50] studies the over-provisioning problem in previous auto-scaling systems (e.g., SageMaker) for ML serving and improves the cost-effectiveness by using a SLO-aware resource provision algorithm. It also considers involving spot instances for more cost savings but requires burstable CPU instances to handle the outstanding requests during instance interruptions. These approaches take a first step to use preemptible instances to serve ML models and motivate our approach on distributed inference of LLMs.

Serverless Computing and ML Serving. There are some recent approaches [9, 25, 39] applying serverless computing to support ML inference workloads for better cost-effectiveness. However, severless functions are designed to be lightweight with limited computational power, memory and storage, and hard be provisioned with GPUs [14]. And serverless functions cannot directly communicate with each other, which is also necessary to support distributed inference of LLMs. As a result, it works well for small models but can not easily serve LLMs due to the hardware constraints.

8 Limitations and Future Work

We introduce the limitations of our approach and outline avenues of future research in SpotServe. First, the key idea of SpotServe is to proactively handle instance availability changes, which strongly relies on the grace period. Although all cloud providers offer this functionality at present, it is still worth exploring more visionary solutions to improve the system performance, such as the combination with inference workload prediction [52] or instance availability prediction [47]. Second, our approach mainly focuses on single-type GPU instances. It is also possible to integrate heterogeneous spot instances [14] or even instances from different clouds (e.g., SkyPilot [48]) for monetary advantages. These scenarios also bring new challenges to context migration in SpotServe. Last, our approach currently takes inference latency minimization as the optimization target. As we mentioned in §3.2, it is still meaningful to explore other targets (e.g., strict SLO [20], high throughput [37]) to meet the needs of different inference scenarios. Besides, the exploration space of parallelization configurations can be enlarged to support emerging variants of large models (e.g., mixutre-of-experts [24]) in the future. While SpotServe focuses on spot instances, our techniques can easily generalize to other preemptible resources, e.g., resource scheduler may preempt resources for urgent jobs with switching overheads [46]. We believe that our approach inspires a new paradigm for distributed inference on preemptible instances, and the insights gleaned from SpotServe's design can motivate a variety of following-up research along this direction.

9 Conclusion

This paper presents SpotServe, the first distributed LLM serving system on preemptible instances. Several key techniques in SpotServe enable fast and reliable serving of generative LLMs on preemptible instances. First, SpotServe dynamically adapts the parallelization configuration to make the system serving capability compatible with the workload. The configuration optimization considers the trade-offs among throughput, latency and monetary cost. Second, to minimize the reparallelization overheads, we design the device mapping algorithm and the migration planning mechanism to achieve efficient context migration. Finally, to take advantage of the grace period offered by the cloud provider, we introduce stateful inference recovery, which allows SpotServe to commit inference progress at a much finer granularity. We evaluate SpotServe on real traces and various scales of popular LLMs and show that SpotServe can save 54% monetary cost compared with on-demand instance and reduce the P99 tail latency by 2.4 - 9.1× compared with existing approaches.

Acknowledgement

We thank the anonymous reviewers and our shepherd, Todd Mytkowicz, for their comments and helpful feedback. This material is based upon work supported by NSF awards CNS-2147909, CNS-2211882, and CNS-2239351, and research awards from Amazon, Cisco, Google, Meta, Oracle, Qualcomm, and Samsung.

References

[1] Amazon ec2 spot instances. https://aws.amazon.com/ec2/spot/.

[2] Nvidia triton inference server. https://developer.nvidia.com/nvidia-triton-inference-server.

[3] Use azure spot virtual machines. https://learn.microsoft.com/en-us/azure/virtual-machines/spot-vms.

[4] Cuda ipc. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html, 2021.

[5] Nvidia fastertransformer. https://github.com/NVIDIA/FasterTransformer, 2021.

[6] Nvidia nccl. https://developer.nvidia.com/nccl, 2021.

[7] vllm: Easy, fast, and cheap llm serving with pagedattention. https://vllm.ai, 2023.

[8] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2016.

[9] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020.

[10] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Optimizing inference serving on serverless platforms. *Proceedings of the VLDB Endowment*, 15(10), 2022.

[11] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. Deepspeed- inference: Enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2022.

[12] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 472–487, 2022.

[13] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[14] Junguk Cho, Diman Zad Tootaghaj, Lianjie Cao, and Puneet Sharma. Sla-driven ml inference framework for clouds with heterogeneous accelerators. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 20–32, 2022.

[15] 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, Boston, MA, March 2017. USENIX Association.

[16] Dense Linear Algebra on GPUs. https://developer.nvidia.com/cublas, 2016.

[17] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient GPU serving system for transformer models. In Jaejin Lee and Erez Petrank, editors, *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, pages 389–402. ACM, 2021.

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

[19] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. Cocktail: A multidimensional optimization for model serving in cloud. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1041–1057, Renton, WA, April 2022. USENIX Association.

[20] Yitian Hao, Wenqing Wu, Ziyi Zhang, Yuyang Huang, Chen Wang, Jun Duan, and Junchen Jiang. Deft: Slo-driven preemptive scheduling for containerized dnn serving. In *Symposium on Networked Systems Design and Implementation*, 2023.

[21] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 103–112, 2019.

[22] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *Proceedings of the 2nd Conference on Systems and Machine Learning*, SysML'19, 2019.

[23] Jack Kosaian, KV Rashmi, and Shivaram Venkataraman. Parity models: erasure-coded resilience for prediction serving systems. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 30–46, 2019.

[24] 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. *CoRR*, abs/2006.16668, 2020.

[25] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. Tetris: Memory-efficient serverless inference through tensor sharing. In Jiri Schindler and Noa Zilberman, editors, *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*. USENIX Association, 2022.

[26] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. *CoRR*, abs/2302.11665, 2023.

[27] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for gpt-3? *arXiv preprint arXiv:2101.06804*, 2021.

[28] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. *Proc. VLDB Endow.*, 16(3):470–479, 2023.

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

[30] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel DNN training. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 7937–7947. PMLR, 2021.

[31] OpenAI. Gpt-4 technical report, 2023.

[32] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. In Waleed Ammar, Annie Louis, and Nasrin Mostafazadeh, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Demonstrations*, pages 48–53. Association for Computational Linguistics, 2019.

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

[34] 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.

[35] Mohammad Shahrad, Rodrigo Fonseca, Inigo 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. USENIX Association, July 2020.

[36] 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 *SOSP '19: Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.

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

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

[39] Vikram Sreekanti, Harikaran Subbaraj, Chenggang Wu, Joseph E. Gonzalez, and Joseph M. Hellerstein. Optimizing prediction serving on low-latency serverless dataflow. *CoRR*, abs/2007.05832, 2020.

[40] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. Piper: Multidimensional planner for dnn parallelization. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 24829–24840. Curran Associates, Inc., 2021.

[41] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large dnns. *CoRR*, abs/2204.12013, 2022.

[42] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[43] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Patrick S. McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 267–284. USENIX Association, 2022.

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[45] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. Lightseq: A high performance inference library for transformers. In Young-bum Kim, Yunyao Li, and Owen Rambow, editors, *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 113–120. Association for Computational Linguistics, 2021.

[46] Xiaofeng Wu, Jia Rao, Wei Chen, Hang Huang, Chris Ding, and Heng Huang. Switchflow: preemptive multitasking for deep learning. In *Proceedings of the 22nd International Middleware Conference*, pages 146–158, 2021.

[47] Fangkai Yang, Lu Wang, Zhenyu Xu, Jue Zhang, Liqun Li, Bo Qiao, Camille Couturier, Chetan Bansal, Soumya Ram, Si Qin, et al. Snape: Reliable and low-cost computing with mixture of spot and on-demand vms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 631–643, 2023.

[48] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, and Ion Stoica. SkyPilot: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 437–455, Boston, MA, April 2023. USENIX Association.

[49] 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.

[50] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *USENIX Annual Technical Conference*, pages 1049–1062, 2019.

[51] Haoyu Zhang, Jianjun Xu, and Ji Wang. Pretraining-based natural language generation for text summarization. *arXiv preprint arXiv:1902.09243*, 2019.

[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, Boston, MA, April 2023. USENIX Association.

[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] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 559–578. USENIX Association, 2022.