A System for Microserving of LLMs

Hongyi Jin [*1] **Ruihang Lai** [*1] **Charlie F. Ruan** [*1] **Yingcheng Wang** [*2] **Todd C. Mowry** [1] **Xupeng Miao** [3]
Zhihao Jia [1 4] **Tianqi Chen** [1 5]

Abstract

The recent advances in LLMs bring a strong demand for efficient system support to improve overall serving efficiency. As LLM inference scales towards multiple GPUs and even multiple compute nodes, various coordination patterns, such as prefill-decode disaggregation and context migration, arise in serving systems. Most inference services today expose a coarse-grained request-level API with a pre-configured coordination strategy, limiting the ability to customize and dynamically reconfigure the coordination. In this paper, we propose LLM microserving, a multi-level architecture for structuring and programming LLM inference services. We introduces simple yet effective microserving APIs to support fine-grained sub-request level actions. A programmable router transforms user requests into sub-request calls, enabling the dynamic reconfiguration of serving patterns. To support diverse execution patterns, we develop a unified KV cache interface that handles various KV compute, transfer, and reuse scenarios. Our evaluation shows that LLM microserving can be reconfigured to support multiple disaggregation orchestration strategies in a few lines of Python code while maintaining state-of-the-art performance for LLM inference tasks. Additionally, it allows us to explore new strategy variants that reduce up to 47% of job completion time compared to the existing strategies.

1 Introduction

Large language models (LLMs) have achieved remarkable capabilities, handling diverse tasks like text generation (Touvron et al., 2023), question answering, and code synthesis (Rozière et al., 2024). The recent advances in LLMs bring a strong demand for efficient system support for improving the overall serving efficiency. As LLM serving scales towards multiple GPUs and even multiple compute instances, many coordination and optimization patterns arise. For example, prefill-decode disaggregation (Zhong et al., 2024; Patel et al., 2024) separates the prefilling and decoding stages of LLM serving and assigns them to dedicated GPU instances. As we start to bring prefix caching and radix attention (Zheng et al., 2023) across serving instances, there is also a strong demand to enable effective migration of the cached prefix to reduce overall redundant computation across multiple requests in the system. There are also multiple ways to dispatch an input request across different LLM workers based on the ongoing traffic (Sun et al., 2024). The complex combination of the coordination opportunities creates a rich space for orchestrating LLM deployment on multiple devices.

[*]Equal contribution [1]Carnegie Mellon University [2]Tsinghua University [3]Purdue University [4]Amazon [5]NVIDIA. Correspondence to: Hongyi Jin <hongyij@cs.cmu.edu>.

Most of the LLM inference services today build on top of systems that implement a fixed set of the strategies and expose a request-level REST API for the end users. The configuration of the coordination strategies is hidden behind the request-level endpoint and managed by the underlying system. Changing the strategy usually involves reconfiguring the underlying system and restarting the service, causing disruptions to the production environment. The coarse-grained LLM serving architecture also limits our ability to customize and explore different coordination strategies and dynamically reconfigure them based on the incoming traffic.

This paper aims to address the above gap by introducing a multi-level architecture for structuring and programming LLM inference services. We propose LLM microserving, which exposes simple yet effective APIs for fine-grained sub-request level actions. At the global level, we introduce a programmable router that transforms an end-user-provided API request into corresponding calls to microserving endpoints using python async functions. Such transformation enables flexible program schedules that orchestrate the LLM engines on the fly, enabling dynamic reconfiguration according to the dynamic workloads. Finally, we build a unified KV cache abstraction to support the common attention transfer, reuse, and compute patterns, and use that to build an efficient microserving engine for LLM inference.

The proposed microserving architecture enables developers to easily specify multiple disaggregation and coordination

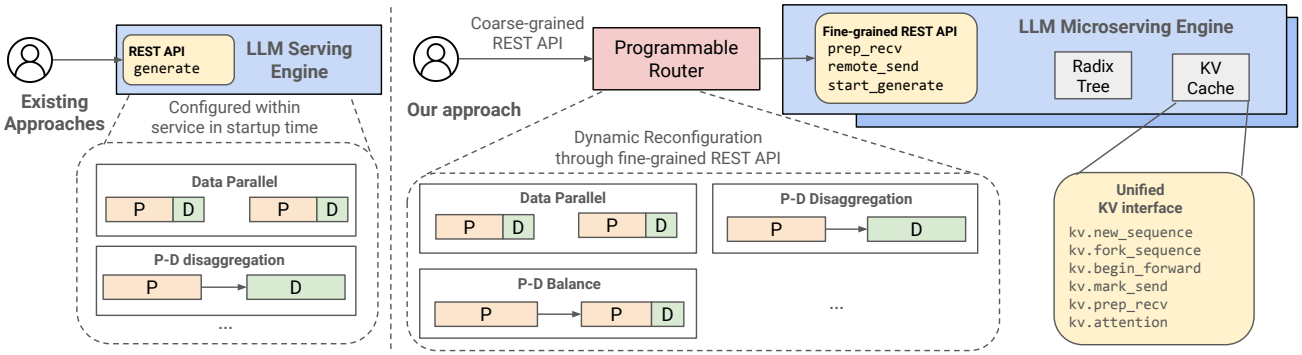


Figure 1. LLM microserving System Overview. Our architecture enables dynamic reconfiguration of different orchestration strategies with a programmable router through three fine-grained REST APIs. The LLM microserving engines implement the APIs with a unified KV cache interface.

patterns dynamically in the router through a simple yet effective asynchronous Python programming model. For example, LLM microserving can easily reproduce existing static scheduling patterns, such as data parallel and prefill-decode disaggregation. Dynamic scheduling patterns such as prefill-decode work balancing and distributed context cache management can also be supported by programming with the REST APIs.

We implement the proposed abstractions in an end-to-end LLM inference engine. Our evaluation demonstrates that LLM microserving enables a flexible programming model to support multiple disaggregation and orchestration strategies while maintaining state-of-the-art performance for LLM inference tasks. The flexible programming model also enables us to quickly explore a new disaggregation strategy that pushes part of prefill workloads to decode for load balancing. Our evaluation shows that the balanced prefill-decode disaggregation strategy can reduce up to 47% job completion time compared to existing strategies.

This paper makes the following contributions:

- We propose a microserving architecture for LLM inference service with simple yet effective composable fine-grained APIs.
- We introduce a programmable router to enable dynamic reconfiguration of multiple LLM inference orchestration patterns.
- We build a unified KV interface that handles model computation under different KV transfer, reuse, and compute patterns for microserving.

2 Overview

LLM microserving is an architecture that enables dynamic reconfiguration of different disaggregation and coordination patterns. It allows systems to smoothly adapt to the traffic during production, and offers framework users fine-grained control over scheduling their systems.

As depicted in Figure 1, existing frameworks implement a fixed set of strategies encapsulated in the underlying LLM serving engine and expose a coarse-grained request-level API to developers. As a result, reconfiguration may require system restarts and disrupt the production environment, and framework users have limited abilities to schedule their system dynamically according to the traffic. LLM microserving architecture addresses this gap with three key components.

First of all, LLM microserving architecture defines three simple fine-grained APIs that allow framework users to express various orchestration patterns when composed together (§3.1).

In the control layer, the programmable router transforms the request-level API into the fine-grained APIs dispatched to the engines. The router's simple asynchronous Python programming model allows framework users to program their own router to dynamically specify different disaggregation and coordination patterns, including data parallel, prefill-decode disaggregation, context cache migration, and more (§3.2). It also enables exploration of new strategy variants that may reduce up to 47% of job completion time compared to existing strategies (§3.3).

In the execution layer, to support the diverse semantics of the fine-grained APIs, the same LLM microserving engines need to organically execute a combination of different KV transfer, reuse, and compute patterns. To achieve this, we propose a unified KV interface that abstracts out such execution-level patterns (§3.4). To minimize KV transfer overhead, the engines implement KV transfer with async and one-sided GPU communication (§3.6).

With these components, LLM microserving enables a flexible programming model to orchestrate the underlying sys-

Table 1. Three fine-grained REST APIs

REST API	Parameters	Explanation
`prep_recv`	`prompt, end`	Match `prompt[:end]` in context cache to get the matched prefix length `matched_len`. Allocate KV entry to prepare receiving `prompt[matched_len:end]` Return KV address and `matched_len`.
`remote_send`	`prompt, kv_addr_info, recv_rank, begin, end`	Generate KV of `prompt[begin:end]` by matching in context cache and prefilling. Transfer the KV to address encoded in `kv_addr_info` on engine `recv_rank`. Return when all KV transfers are finished.
`start_generate`	`prompt, begin`	Prefill `prompt[begin:]` and start decode. Return generated chunks.

tem dynamically, while maintaining state-of-the-art performance for LLM inference tasks (§4).

3 Our Approach

3.1 Microserving REST API

Existing LLM serving systems typically treat token generation for a request as an atomic operation, requiring developers to implement different code inside the underlying system for each scheduling pattern. This approach limits framework users' abilities to schedule their system dynamically, and it often necessitates system restarts when reconfiguring the scheduling pattern, causing disruptions in production environments. Our insight is that common scheduling patterns can be expressed using two fundamental actions:

- Transferring some KV cache from one LLM engine to another. The transferred KV may originate from either the existing context cache or the new prefill computation.
- Initiating token generation with full or partial KV cache of the prompt.

Since transferring KV requires coordination between sender and receiver, we introduce three fine-grained REST APIs (Figure 1) microserving the actions. By invoking these APIs from the router, developers can compose various scheduling patterns and easily reconfigure the system from one pattern to another.

`prep_recv` checks for the existence of `prompt[:end]` in the KV cache. It allocates KV cache entries for the missing subsequence and returns their address in a compressed form.

`remote_send` sends KV of `prompt[begin:end]` to engine `recv_rank`. If part of the subsequence already exists in the KV cache, it is directly transferred to the receiver. For other parts, new KVs are first materialized through prefill computation and then sent to the receiver.

`start_generate` informs the engine that all KVs of `prompt[:begin]` already exist in its KV cache and it's ready to prefill `prompt[begin:]` and start decoding.

By calling the APIs sequentially, we can create a workflow like the following. The router wants to perform partial prefill and decode on engine A. It first determines which part of KV is missing on engine A (with `prep_recv`), and then asks engine B to transfer those missing parts to engine A (with `remote_send`). After engine A receives all necessary KV, it proceeds with the remaining prefill and begins decoding (with `start_generate`). Different compositions of the APIs can support more patterns as discussed below.

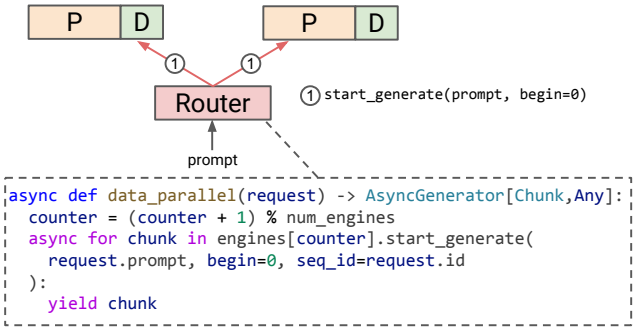


Figure 2. Data parallel via microserving

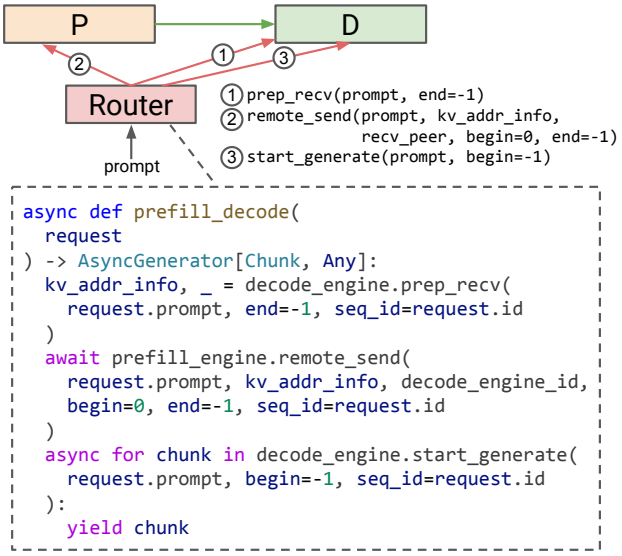
3.2 Programmable Router

With fine-grained REST APIs enabling granular control over engine actions, the router can transform a coarse-grained request-level API into a set of fine-grained REST APIs based on the current scheduling pattern.

Reconfiguration between scheduling patterns only occurs on the router side and does not require engine reconfiguration.

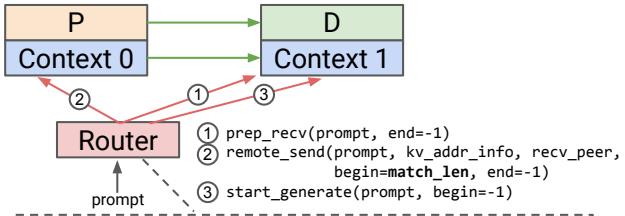
We showcase several examples of how the router can be programmed to support various scheduling patterns using concise asyncio Python code. These examples can be generalized to a router that dynamically reconfigures its strategy by inspecting an internal policy updated by the traffic.

Example 1: Data parallel Data parallel (Figure 2) is the strategy that dispatches incoming requests to engines in a round-robin fashion. It is straightforward to implement




```python
async def prefill_decode(
  request
) -> AsyncGenerator[Chunk, Any]:
  kv_addr_info, _ = decode_engine.prep_recv(
    request.prompt, end=-1, seq_id=request.id
  )
  await prefill_engine.remote_send(
    request.prompt, kv_addr_info, decode_engine_id,
    begin=0, end=-1, seq_id=request.id
  )
  async for chunk in decode_engine.start_generate(
    request.prompt, begin=-1, seq_id=request.id
  ):
    yield chunk
```

Figure 3. Prefill-decode disaggregation via microserving




```python
async def prefill_decode_with_context_cache(
  request
) -> AsyncGenerator[Chunk, Any]:
  kv_addr_info, match_len = decode_engine.prep_recv(
    request.prompt, end=-1, seq_id=request.id
  )
  await prefill_engine.remote_send(
    request.prompt, kv_addr_info, decode_engine_id,
    begin=match_len, end=-1, seq_id=request.id
  )
  async for chunk in decode_engine.start_generate(
    request.prompt, begin=-1, seq_id=request.id
  ):
    yield chunk
```

Figure 4. Context cache-aware prefill-decode disaggregation via microserving

because engines do not communicate with each other, making it the default strategy in many LLM serving systems. While data parallel helps achieve high throughput, it does not reduce latency except for the reduced decode batch size.

To implement data parallel in the router, we only need to maintain a counter indicating the next engine to dispatch the request to, and then send a `start_generate(prompt,begin=0)` API call to that engine. Figure 2 contains a reference router implementation of data parallel in just 5 lines of code.

Example 2: Prefill-decode disaggregation Prefill-decode disaggregation separates prefill and decode operations across LLM engines. The key idea is that colocating prefill and decode may result in strong interference and unnecessary coupling of resource allocation and parallelism between the two phases. Disaggregating prefill and decode leads to lower latency, better resource management, and more scalability. This scheduling strategy is challenging to implement in two aspects: efficiently transferring KV cache from the prefill engine to the decode engine, and coordinating between the two engines to minimize CPU runtime overhead. Our fine-grained REST API provides a clean and easy-to-use solution to the coordination issue, while our unified KV cache interface enables asynchronous and fast KV transfer (§3.6).

To illustrate the router-side implementation, let's consider a simple case where we do not have a context cache. As shown in Figure 3, the router first calls `prep_recv` on the decode engine. The argument `end=-1` instructs the decode engine to prepare space for the KV cache entries of `request.prompt[:-1]` (all but the last prompt token), and the returned `kv_addr_info` encodes the page index and slot index of the KV cache entries.

Next, the router calls `remote_send` on the prefill engine, with `begin=0` and `end=-1` to ask the prefill engine to run prefill computation to generate KV cache for `request.prompt[:-1]` and send KV cache to the corresponding entries on the decode engine side. The KV transfer here uses a remote memory access semantic, allowing the sender to directly write KV into the remote address. When the remote write is completed `remote_send` returns.

After `remote_send` returns, the KV transfer is complete. The router calls `start_generate` on the decode engine to prefill the last token, sample, and start decoding.

Figure 3 shows an API call graph illustrating these three steps and their implementation. The scheduling patterns discussed later have similar router-side implementations to prefill-decode disaggregation, differing only in the arguments of the APIs.

Example 3: Context cache-aware prefill-decode disaggregation This example extends the prefill-decode disaggregation to consider the context cache on each LLM engine. The implementation is similar to the no-context-cache case, with the main difference being that `match_len` is not necessarily 0 (Figure 4).

The key API call differences are:

- `prep_recv`: The decode engine matches the prompt with its local context cache, returning the matched prefix length `match_len` and allocating space for the un-

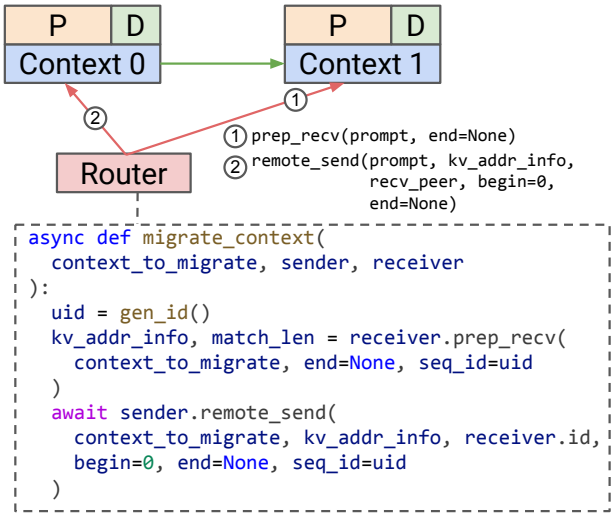


Figure 5. Context cache migration via microserving

matched portion.

- `remote_send`: The prefill engine matches the prompt with its local context cache and transfers the necessary KV data to the decode engine. This may involve direct transfer of cached data and/or prefill computation for unmatched portions.
- `start_generate`: Remains the same as in the no-context-cache case.

Therefore, the REST APIs allow efficient handling of different prefix-matching scenarios between the prefill and decode engines. For more details on the internal workings of KV transfer in various cases, refer to §3.5.

Example 4: Context Cache Migration When serving QA workloads, developers tend to place context cache of different categories into different engines and dispatch incoming traffic based on which context category it matches. Consider there are several engines, with some specialized for history context and others for science context. If there are more science requests than history requests, we may want to switch some history engines to science engines through context migration, and vice versa.

Context cache migration can be composed with our fine-grained REST APIs as well (Figure 5). By calling `prep_recv` on receiver and `remote_send` on sender, a context transfer is easily accomplished. Note that the router also needs to maintain a radix tree in addition to those on LLM engines, so that it can decide which engine to dispatch requests to and when to trigger a context switch.

3.3 Exploring Balanced Prefill-Decode Disaggregation

§3.2 shows LLM microserving's functionality to leverage the router programming model for supporting existing strategies for disaggregating inference and context migration. This section introduces how this programming flexibility enables LLM microserving to support new prefill-decode disaggregation strategies that haven't been considered by prior work. One issue of prefill-decode disaggregation is that the prefill and decode workloads can get unbalanced depending on the workload (Qin et al., 2024). When processing long prompts, the prefill engine can get over-utilized while the decode engine stays idle. Most prior approaches (Hu et al., 2024b; Jin et al., 2024; Wu et al., 2024) solve this problem by dynamically adapting the number of prefill and decode instances, which introduces significant migration overheads.

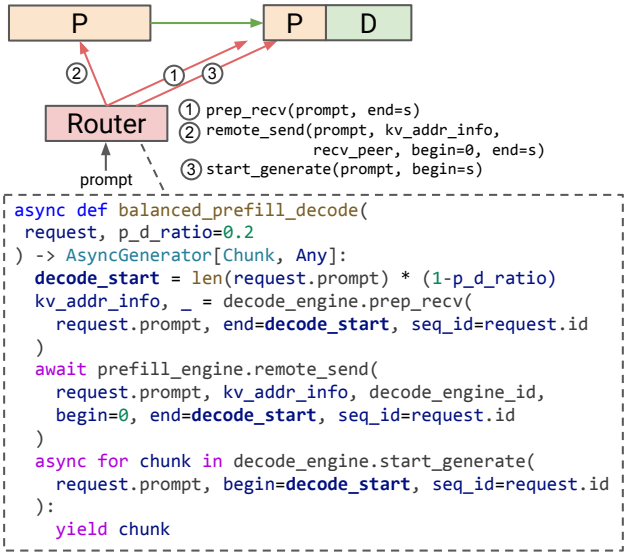


Figure 6. Balanced prefill-decode disaggregation via microserving

We explore a different strategy that *dynamically* moves a part of prefill computation into the decode engine, where the migrated prefill computation can be fused with the decode computation to improve overall throughput. We refer to this strategy as *balanced prefill-decode disaggregation*, and implement it in a few lines of code using the router's programming interface (see Figure 6). The router decides the subsequence (`prompt[:s]`) to compute on the prefill instance based on the monitored P:D distribution. When the prefill engine returns from `remote_send`, the decode engine has KV of `prompt[:s]`, so it continues to prefill the remaining part of `prompt[s:]`, does sampling, and starts decoding. This strategy enables more fine-grained load balancing between prefill and decode, making disaggregation applicable to a broader range of workloads.

3.4 Microserving with a Unified KV Cache Interface

The previous sections discuss the control plane by defining the semantics of the three fine-grained APIs and showing how they can schedule the system dynamically. This section

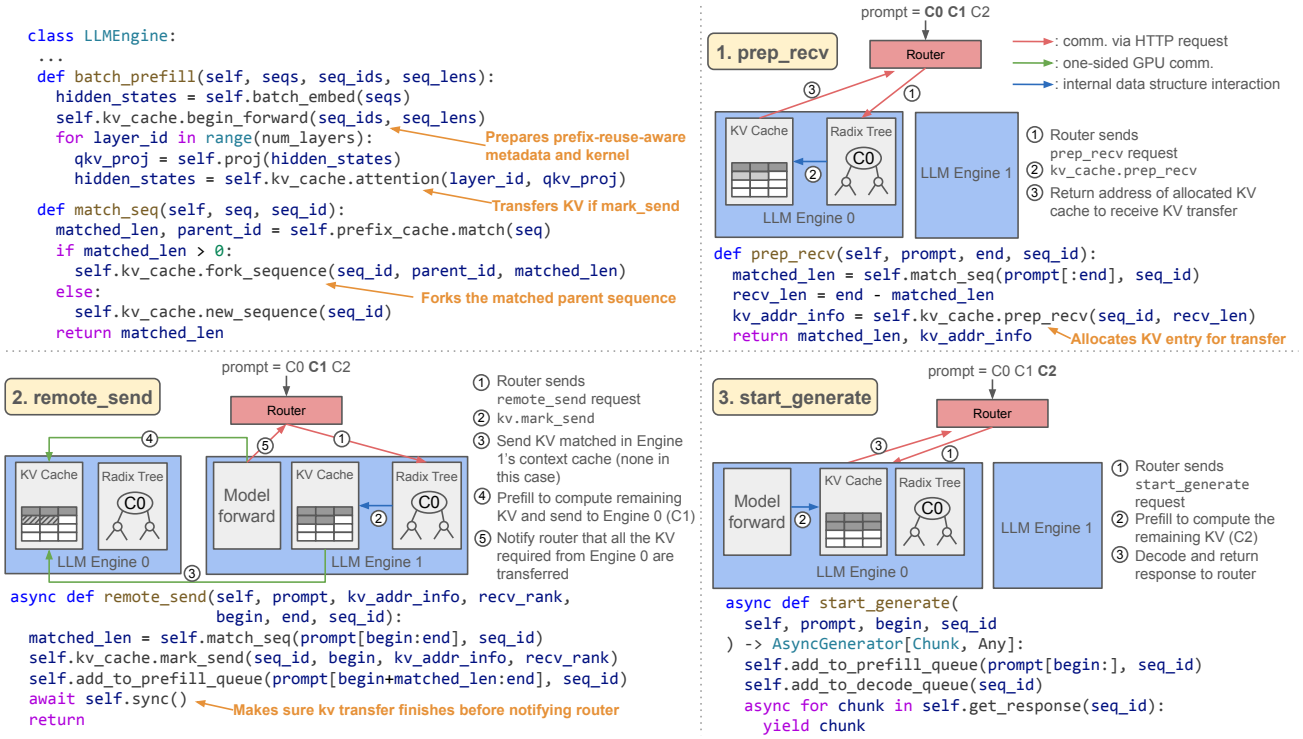


Figure 7. The implementation of each REST API in an LLM engine with a unified KV Cache interface. Here we depict prefilling a prompt with contexts C0 C1 C2 in a balanced P-D disaggregated pattern discussed in §3.3. Engine 0 is the decode engine, and Engine 1 is the prefill engine. The router instructs Engine 0 to prefill C2 to transfer part of prefill pressure. Both engines have C0 in their context cache. (1) The router sends `prep_recv` to Engine 0, which matches C0 and tells the KV cache to prepare for receiving C1. (2) The router sends `remote_send` to Engine 1. Engine 1 also matches C0, prefills C1, and sends the KV of C1 to Engine 0 with remote GPU memory write. (3) After the KV transfer of C1 finishes, the router sends `start_generate` to Engine 0, which prefills C2 first and then starts decoding.

goes over how an LLM microserving engine can implement and execute each of the three fine-grained APIs with a unified KV cache interface, as shown in Figure 7. The KV cache APIs are listed in Table 2.

To implement the diverse semantics of the REST APIs, the same engine needs to support various attention compute, KV transfer, and KV reuse patterns. Under prefill-decode disaggregation, the engine needs to send part of the KV cache to the remote; while prefix matching can result in forking operations where multiple sequences share the same prefix, allowing the attention operation to load the KV of the shared prefix once. These may also result in different combinations (§3.5). Therefore, it is challenging to support the numerous patterns in the same model definition.

Our unified KV cache interface is yet another simple and effective solution for abstracting out such patterns. It provides a two-stage interface:

- In the *declaration* stage, the KV cache provides two main runtime APIs to declare the necessary information about the pending actions. `begin_forward` declares pending attention operations and asks the KV cache to *plan* the set of metadata and kerne kernel operations once for all attention layers. `mark_send` declares that the pending attention would invoke a KV transfer to remote.
- In the *computation* stage during model forward, the model definition calls `attention` with `qkv_data`. The KV cache performs attention with the already-prepared metadata, which informs the KV cache of outstanding KV transfers and opportunities to leverage common prefixes.

With such a two-stage KV interface, the engine can implement each of the three fine-grained REST APIs. We go over each implementation while referring to Figure 7.

To implement `prep_recv`, the engine calls the KV cache API `prep_recv` that allocates KV entries to prepare for receiving the unmatched parts of the prompt and returns the address that the peer should send to. There is no computation for this REST API.

For `remote_send`, the engine calls the KV cache API `mark_send` to declare that KV transfer is needed for this sequence. The engine then uses `begin_forward` to inform the KV cache of any prefix reuse opportunities for the upcoming compute. Upon `attention` in model forward, the KV cache sends this sequence's KVs to the peer via GPU communication, overlapped with attention computation.

Table 2. The KV Cache interface.

KV Interface	Parameters	Explanation
`new_sequence`	`seq_id`	Add a new sequence of `seq_id` with empty KV state to the cache.
`fork_sequence`	`seq_id, parent_id, offset`	Fork the KV state of parent sequence at `offset` to form a child sequence of `seq_id`, which has access to parent's KV state.
`begin_forward`	`seq_ids, append_lens`	Mark the start of the forward function with the ids of the sequences and their lengths to forward.
`prep_recv`	`seq_id, recv_len`	Prepare and allocate entries to receive `recv_len` KV for sequence `seq_id`. Return address to append KV.
`mark_send`	`seq_id, begin, kv_addr_info, recv_rank`	Internally mark that sequence `seq_id`'s KVs starting at `begin` needs to be sent to engine `recv_rank` at its `kv_addr_info`.
`attention`	`layer_id, qkv_data`	Compute attention with `qkv_data` at the given layer for the sequences declared in `begin_forward`. If `mark_send` called beforehand, launch the KV transfer kernel concurrently. Return attention output.

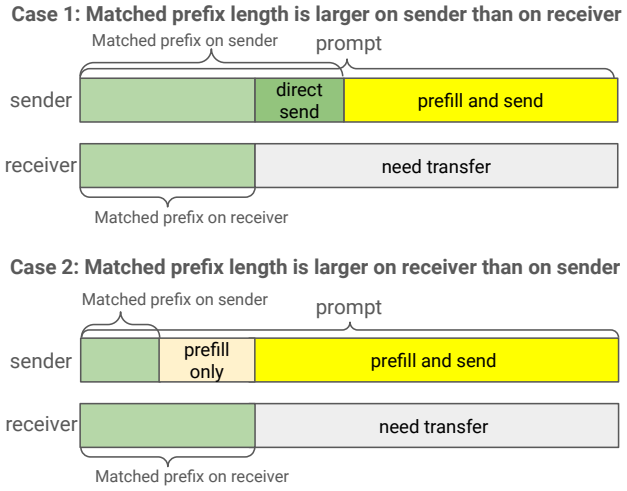


Figure 8. `attention` handles KV transfer in different prefix-matching scenarios.

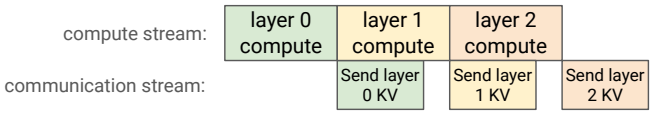


Figure 9. Overlapping of KV transfer and attention computation.

Upon the execution of `start_generate`, the engine already possesses the KVs both from its local KV cache and from the remote peer (if any). It uses the same `begin_forward` and `attention` steps to prefill any remaining part of the prompt and starts decoding.

3.5 KV Cache and Distributed Context Cache

To support prefix matching as promised by the REST APIs, the LLM engines need to maintain a context cache. While some existing frameworks implement this inside the KV cache, our KV cache API `fork_sequence` decouples context cache management from the KV cache. This allows for the coexistence of both local eviction policy from the engine and global eviction policy from the router, which may instruct the engine to "pin" certain important prefixes based on its global knowledge.

When `mark_send` is called, `attention` is responsible for sending the KVs to the receiver. However, this process is complicated by the possibility of prefix reuse. For instance, a part of the KVs to be sent may already exist in the sender's context cache, allowing direct transfer to the receiver. Here, we discuss KV cache transfer in different prefix-matching scenarios by considering two cases (Figure 8).

Case 1: Matched prefix length is larger on sender than on receiver. In this scenario, the segment matched only by the sender (in dark green) can be directly sent to the receiver, while the segment matched by neither engine needs to be prefilled and then transferred (in yellow).

Case 2: Matched prefix length is larger on receiver than on sender. In this scenario, not all the KV segments needed for transfer exist in the sender's context cache. To compute the segment, the sender should prefill all the non-cached segments and transfer only the part required by the receiver.

Given the numerous possible combinations, it is crucial to have a unified interface that supports different KV transfer, reuse, and compute patterns.

3.6 End-to-end Implementation

We implement the LLM microserving engine and KV cache on top of the existing project MLC-LLM (team, 2023), which has a total of 13k lines of C++ and 6k lines of Python. Thanks to the flexibility of LLM microserving, we implement all the strategies in the router with *350 lines* of Python code.

For GPU communication, we use the NVSHMEM library (NVIDIA) which, unlike NCCL, supports one-sided `put` and `get` API. That is, the communication only requires SM on one side to participate. We overlap the KV transfer with attention computation by having a computation stream and a communication stream and by eagerly sending a layer's KV right after its computation finishes, as shown in Figure 9. The KV transfer is asynchronous and does not affect the ongoing requests of the receiver engine.

4 EVALUATION

This section provides an evaluation to answer the following questions:

- Can we easily program customized LLM disaggregated serving patterns with LLM microserving (§4.1)?
- Can LLM microserving support global context reuse and efficient KV migration between engines (§4.2)?
- What are the performance impacts of PD balance ratio (§4.3)?

4.1 Disaggregated LLM Inference Pattern Evaluation

This section evaluates LLM microserving's programmability for various LLM disaggregated serving patterns and the performance of each pattern. We implement three disaggregation patterns, including

- 1P1D: prefill-decode disaggregation with 1 prefill engine and 1 decode engine (Figure 3).
- 1P1D-balance: balanced prefill-decode disaggregation with 1 prefill engine and 1 decode engine (Figure 6).
- 1P2D: prefill-decode disaggregation with 1 prefill engine and 2 decode engines.

Notably, the programmability of LLM microserving allows for sharing the same engine implementation across these patterns, and the patterns only differ in how the router is implemented. We also include the data parallelism (DP) implementation in LLM microserving (Figure 2) and vLLM (v0.6.3.post1) (Kwon et al., 2023) as baselines, where vLLM runs with optimized latency configuration of 10-step scheduler. Each engine runs the Llama3.1 8B model on a single NVIDIA A100 SXM 40GB GPU. We use the balance ratio 0.2 for 1P1D-balance, which means the prefill of the last 20% tokens is assigned to the decode engine, and the prefill engine only computes and transfers KV for the first 80% of tokens. The request arrival follows the arrival time sampled from the Poisson distribution of different per-GPU request rates, in order to fairly compare disaggregation patterns that use different numbers of GPUs. For each request, we measure the metrics of time to first token (TTFT), time per output token (TPOT), and job completion time (JCT).

We study the system performance under the ShareGPT dataset and synthetic datasets. ShareGPT is a collection of user-shared conversations with ChatGPT and provides diverse examples of real interactions. It features short input and output lengths, with means being around 200 and 260 respectively. To study the performance with longer input lengths, we generate the synthetic dataset with the input and output lengths of each entry sampled from a normal distribution. The mean input length is 3000, and the mean output length is 100, with a standard deviation of 5 for both. This

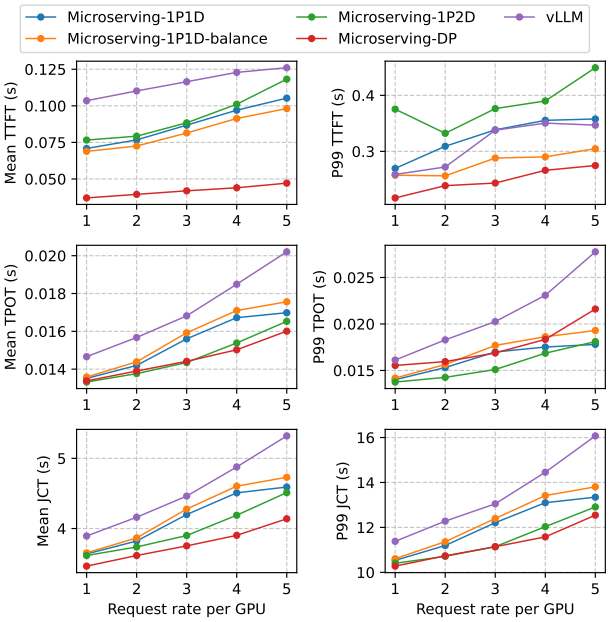


Figure 10. LLM inference evaluation on ShareGPT. Prefill-decode disaggregation has no observed benefit over data parallelism because the prefill engine is idle with the short input in ShareGPT.

configuration ensures that requests will not finish shortly after prefill, as typical QA datasets have few output tokens.

Figure 10 shows the evaluation results with the ShareGPT dataset. When the requests follow ShareGPT input and output length distribution, data parallel remains a strong baseline overall, and we do not observe benefits of prefill-decode disaggregation. This is mainly because the ShareGPT dataset features the same magnitude of input and output lengths, so the prefill load is relatively low, causing the prefill engine not to be fully utilized, while the data parallel engine is always kept busy. Among the disaggregation patterns, 1P1D-balance underperforms the vanilla 1P1D due to the same reason of the input-output length ratio, as 1P1D-balance further reduces the prefill engine workload and increases the decode engine workload. 1P2D brings lower TPOT and JCT than 1P1D by doubling the decode engine and amortizing the decode pressure onto two engines.

Figure 11 shows the evaluation results of the synthetic dataset, which features longer input lengths that increase the prefill engine workload. Prefill-decode disaggregation demonstrates the capability of reducing the JCT compared to data parallelism by up to 21% for mean JCT and up to 47% for P99 JCT. This significant speedup attributes to the substantial reduction of TPOT in disaggregation, achieved by eliminating the decode interference caused by long prefill in data parallelism. As the per-GPU request rate increases to 2.5 req/s, the heavier traffic puts more pressure on the prefill

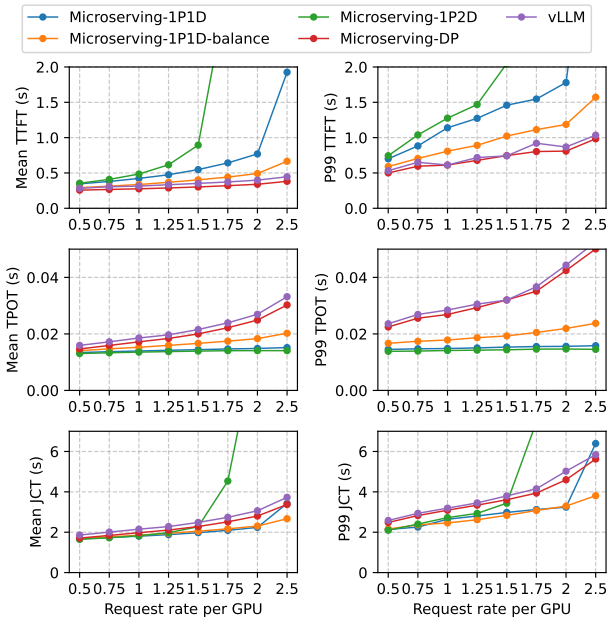


Figure 11. LLM inference evaluation on synthetic data with average input length 3000 and average output length 100. The new disaggregation pattern 1P1D-balance reduces up to 47% of job completion time, which benefits from transferring part of prefill engine pressure to the decode engine.

engine and causes performance degradation in 1P1D due to the increase of TTFT. In this case, the 1P1D-balance pattern helps by transferring some of the prefill engine pressure to the decode engine, thus maintaining a low JCT.

The disaggregated LLM inference pattern study shows that different patterns have different preferences on request input-output length ratios and traffic, and striking a workload balance between the prefill and decode engines is a key to better performance. LLM microserving provides microserving APIs and programmability, supporting the representation of various disaggregation patterns and the dynamic reconfiguration among these patterns.

4.2 KV Migration Efficiency Evaluation

In this section, we evaluate the efficiency of KV migration from one LLM engine `E1` to another engine `E2` when context cache is available (Figure 4). The evaluation method is: First prefill a context on `E1`. Then create a new prompt by concatenating the context with unique text of 500 tokens, and run its prefill on `E1` and decode on `E2`. When processing the prefill, `E1` recognizes the context reuse by context cache match, and then automatically migrates the context KV data to `E2`. `E1` also computes and transfers the KV of the unique text. As a comparison, we evaluate the prefill time of this request when no global context cache is available. In this baseline, the KV of the full input (including the

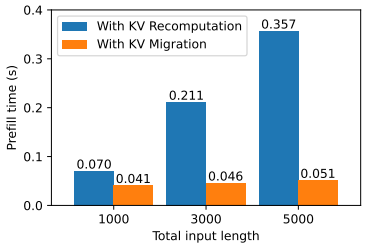


Figure 12. Llama3.1 8B prefill time comparison between "with KV recomputation" and "with KV migration" implemented in LLM microserving. Context lengths are 500/2500/4500 respectively, and the length of unique text is 500 tokens. KV migration keeps the prefill time at a low level, compared to the prefill time that recomputes the full KV of the input.

context and the unique text) is recomputed. We evaluate the context lengths of 500, 2500, and 4500 respectively (so the total input lengths are 1k, 3k, and 5k).

Figure 12 demonstrates LLM microserving's ability to leverage global context cache for KV migration between engines, significantly reducing prefill computation time. With context cache, KV migration ensures that only the unique text of 500 tokens requires computation, as opposed to prefilling the entire input. For a total input length of 1000 tokens, KV migration effectively halves the prefill length, resulting in 1.7× speedup in prefill time. Despite the increasing load of attention computation with longer input contexts, the prefill time with KV migration shows only a slight increase. In contrast, prefill time without KV migration (i.e. with recomputation) grows linearly as input length increases from 1000 to 5000 tokens. This growing disparity underscores the importance of global context reuse and KV migration in LLM microserving.

To further assess KV transfer efficiency during prefill, we evaluate transfer times under the same context cache settings. Table 3 presents the prefill and KV transfer times per model layer in Llama3.1 8B. LLM microserving overlaps the KV transfer with the prefill computation by employing the NVSHMEM library for GPU communication. As the total input length increases from 1000 to 5000, the KV transfer time overlap ratio rises from 15.8% to 55.4%. This increase occurs because the effective prefill length remains constant at 500 tokens (the length of unique text), while the size of transferred KV data grows linearly. Notably, LLM microserving's use of one-sided NVSHMEM primitives allows KV transfer to overlap with ongoing decode computations on the receiver engine, minimizing communication overhead.

4.3 Impact of PD Balance Ratio

We have observed the performance benefit of the 1P1D-balance disaggregation pattern in §4.1. In this section, we

Table 3. Per-layer prefill time and KV transfer time of Llama3.1 8B. In LLM microserving, KV transfer fully overlaps with prefill computation as long as the transfer time does not exceed the computation time.

Input Length	**1000**	**3000**	**5000**
$T_{\text{per-layer}}$ (ms)	1.247	1.391	1.564
$T_{\text{KV-transfer}}$ (ms)	0.197	0.533	0.867
Transfer Time Ratio	15.8%	38.3%	55.4%

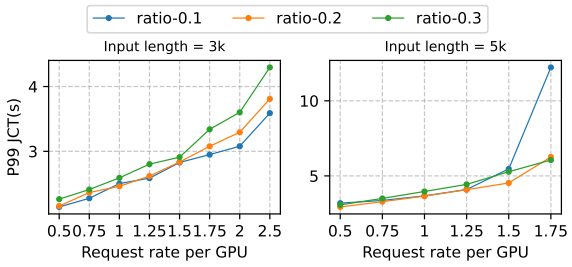


Figure 13. Impact of the PD balance ratio under different input lengths. Longer input requires higher PD balance ratio in order to further reduce the prefill engine pressure.

study the impact of different balance ratios that decide how much prefill workload will be transferred from the prefill engine to the decode engine. We tested the balance ratios of 0.1, 0.2 and 0.3 under input lengths 3000 and 5000. Results are shown in Figure 13. For the input length of 3000, the balance ratio of 0.1 offers up to 1.19 $\times$ speedup in P99 JCT than higher ratios. When the input length is increased to 5000, it is more beneficial to transfer more prefill workload to decode engine, especially in the scenario of a high request rate. This is because a high request rate aggravates the pressure on the prefill engine. As a result, requests can easily queue up on the prefill engine, and this congestion significantly slows down TTFT. A higher balance ratio helps better alleviate the congestion and bring the system back to a prefill-decode balanced state.

5 Related Work

Disaggregated Serving Recent advancements in LLM serving have focused on disaggregating the prefill and decode phases to optimize performance. SplitWise (Patel et al., 2024) introduced a disaggregated inference approach that separates prefill and decode operations. DistServe (Zhong et al., 2024) presented a distributed serving system that leverages disaggregation to improve goodput. TetriServe (Hu et al., 2024b) further refined this approach by introducing adaptive resource scheduling in disaggregated LLM serving. P/D-Serve (Jin et al., 2024) dynamically adjusts P/D ratios and forward queued prefill for better performance. LoongServe (Wu et al., 2024) elastically adjusts the degree of parallelism to quickly serve variable-length requests in prefill and decode phases. LLM microserving complements these works by providing a simple yet effective programming model to support and dynamically reconfigure these disaggregated strategies.

KV Cache Optimization (Wu & Tu, 2024) proposed Layer-Condensed KV Cache, which computes and caches KVs for only a small number of layers, reducing memory consumption while maintaining competitive performance. vLLM (Kwon et al., 2023) developed PagedAttention, a transparent cache management layer that minimizes GPU memory fragmentation, improving inference speed. Infinite-LLM (Lin et al., 2024) extended PagedAttention to enable distributed deployment across servers. Our system can leverage some of these optimizations and bring them to microserving scenarios.

Context Caching Context caching has emerged as a powerful technique to improve LLM serving performance. CacheGen (Liu et al., 2024) introduced a novel approach to compress the KV cache and optimize its streaming, addressing bandwidth limitations in fetching large caches. EPIC (Hu et al., 2024c) presented a position-independent context caching system that enables modular KV cache reuse regardless of token chunk position. MemServe (Hu et al., 2024a) introduced an elastic memory pool (MemPool) that manages distributed memory and KV caches across serving instances, combining context caching with disaggregated inference. LLM microserving brings fine-grained API and flexible programming model that can enable effective combination and reconfiguration of the context caching and disaggregated inference strategies.

LLM Decoding Techniques Besides context caching techniques that optimize for prefill stage performance, many LLM decoding techniques improve the decode stage efficiency by breaking the memory-bound auto-regressive decoding into local batch processing, in order to benefit from the batching effects of LLMs. Speculative decoding (Chen et al., 2023; Leviathan et al., 2023; Cai et al., 2024; Li et al., 2024) employs a small draft model to provide multiple draft tokens at a time for batched verification. Sarathi-Serve (Agrawal et al., 2024) introduces chunked-prefills and stall-free scheduling to optimize the throughput-latency tradeoff in LLM inference. With engine-level support, LLM microserving can benefit from these techniques in performance and bring them to microserving without any router-level changes.

6 Conclusion

We introduce LLM microserving, a multi-level architecture for structuring and programming LLM inference services.

By providing fine-grained REST APIs, and a unified KV cache interface, LLM microserving allows easy implementation and customization of various disaggregation strategies while maintaining competitive performance. We hope this work will encourage additional studies of dynamic reconfiguration and orchestration strategies in microserving LLM workloads.

REFERENCES

Agrawal, A., Kedia, N., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., Tumanov, A., and Ramjee, R. Taming throughput-latency tradeoff in llm inference with sarathi-serve. *arXiv preprint arXiv:2403.02310*, 2024.

Cai, T., Li, Y., Geng, Z., Peng, H., Lee, J. D., Chen, D., and Dao, T. Medusa: Simple llm inference acceleration framework with multiple decoding heads, 2024. URL https://arxiv.org/abs/2401.10774.

Chen, C., Borgeaud, S., Irving, G., Lespiau, J.-B., Sifre, L., and Jumper, J. Accelerating large language model decoding with speculative sampling, 2023. URL https://arxiv.org/abs/2302.01318.

Hu, C., Huang, H., Hu, J., Xu, J., Chen, X., Xie, T., Wang, C., Wang, S., Bao, Y., Sun, N., et al. Memserve: Context caching for disaggregated llm serving with elastic memory pool. *arXiv preprint arXiv:2406.17565*, 2024a.

Hu, C., Huang, H., Xu, L., Chen, X., Xu, J., Chen, S., Feng, H., Wang, C., Wang, S., Bao, Y., Sun, N., and Shan, Y. Inference without interference: Disaggregate llm inference for mixed downstream workloads, 2024b. URL https://arxiv.org/abs/2401.11181.

Hu, J., Huang, W., Wang, H., Wang, W., Hu, T., Zhang, Q., Feng, H., Chen, X., Shan, Y., and Xie, T. Epic: Efficient position-independent context caching for serving large language models, 2024c. URL https://arxiv.org/abs/2410.15332.

Jin, Y., Wang, T., Lin, H., Song, M., Li, P., Ma, Y., Shan, Y., Yuan, Z., Li, C., Sun, Y., Wu, T., Chu, X., Huan, R., Ma, L., You, X., Zhou, W., Ye, Y., Liu, W., Xu, X., Zhang, Y., Dong, T., Zhu, J., Wang, Z., Ju, X., Song, J., Cheng, H., Li, X., Ding, J., Guo, H., and Zhang, Z. P/d-serve: Serving disaggregated large language model at scale, 2024. URL https://arxiv.org/abs/2408.08147.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pp. 611–626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Leviathan, Y., Kalman, M., and Matias, Y. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

Li, Y., Wei, F., Zhang, C., and Zhang, H. Eagle: Speculative sampling requires rethinking feature uncertainty, 2024. URL https://arxiv.org/abs/2401.15077.

Lin, B., Zhang, C., Peng, T., Zhao, H., Xiao, W., Sun, M., Liu, A., Zhang, Z., Li, L., Qiu, X., Li, S., Ji, Z., Xie, T., Li, Y., and Lin, W. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache, 2024. URL https://arxiv.org/abs/2401.02669.

Liu, Y., Li, H., Cheng, Y., Ray, S., Huang, Y., Zhang, Q., Du, K., Yao, J., Lu, S., Ananthanarayanan, G., Maire, M., Hoffmann, H., Holtzman, A., and Jiang, J. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, pp. 38–56, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706141. doi: 10.1145/3651890.3672274. URL https://doi.org/10.1145/3651890.3672274.

NVIDIA. Nvshmem. URL https://docs.nvidia.com/nvshmem/api/index.html.

Patel, P., Choukse, E., Zhang, C., Shah, A., Goiri, Í., Maleki, S., and Bianchini, R. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132. IEEE, 2024.

Qin, R., Li, Z., He, W., Zhang, M., Wu, Y., Zheng, W., and Xu, X. Mooncake: Kimi's kvcache-centric architecture for llm serving. *arXiv preprint arXiv:2407.00079*, 2024.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. Code llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950.

Sun, B., Huang, Z., Zhao, H., Xiao, W., Zhang, X., Li, Y., and Lin, W. Llumnix: Dynamic scheduling for large language model serving. In Gavrilovska,

A. and Terry, D. B. (eds.), *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pp. 173–191. USENIX Association, 2024. URL https://www.usenix.org/conference/osdi24/presentation/sun-biao.

team, M. MLC-LLM, 2023. URL https://github.com/mlc-ai/mlc-llm.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M.-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models, 2023. URL https://arxiv.org/abs/2307.09288.

Wu, B., Liu, S., Zhong, Y., Sun, P., Liu, X., and Jin, X. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism. *arXiv preprint arXiv:2404.09526*, 2024.

Wu, H. and Tu, K. Layer-condensed kv cache for efficient inference of large language models, 2024. URL https://arxiv.org/abs/2405.10637.

Zheng, L., Yin, L., Xie, Z., Huang, J., Sun, C., Hao Yu, C., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., et al. Efficiently programming large language models using sglang. *arXiv e-prints*, pp. arXiv–2312, 2023.

Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., and Zhang, H. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *arXiv preprint arXiv:2401.09670*, 2024.