Accelerating Local LLMs on Resource-Constrained Edge Devices via Distributed Prompt Caching

Hiroki Matsutani
Dept. of ICS, Keio University
Yokohama, Japan
matutani@arc.ics.keio.ac.jp

Naoki Matsuda
Dept. of ICS, Keio University
Yokohama, Japan
matsuda@arc.ics.keio.ac.jp

Naoto Sugiura
Dept. of ICS, Keio University
Yokohama, Japan
nsugiura@arc.ics.keio.ac.jp

Abstract

Since local LLM inference on resource-constrained edge devices imposes a severe performance bottleneck, this paper proposes distributed prompt caching to enhance inference performance by cooperatively sharing intermediate processing states across multiple low-end edge devices. To fully utilize prompt similarity, our distributed caching mechanism also supports partial matching. As this approach introduces communication overhead associated with state sharing over a wireless network, we introduce a Bloom-filter-based data structure, referred to as a catalog, to determine whether a remote server possesses the desired internal states, thereby suppressing unnecessary communication. Experiments using the Gemma-3 270M model and the MMLU dataset on the Raspberry Pi Zero 2W platform demonstrate that the proposed approach reduces TTFT (Time to First Token) and TTLT (Time to Last Token) by 93.12% and 50.07% on average, respectively.

CCS Concepts: • **Computing methodologies** → *Cooperation and coordination.*

Keywords: Local LLM, Edge LLM, Distributed cache, KV cache

ACM Reference Format:
Hiroki Matsutani, Naoki Matsuda, and Naoto Sugiura. 2026. Accelerating Local LLMs on Resource-Constrained Edge Devices via Distributed Prompt Caching. In *Sixth European Workshop on Machine Learning and Systems (EuroMLSys '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3805621.3807647

1 Introduction

A local LLM (Large Language Model) refers to the deployment of an LLM in a local environment, in contrast to cloud-based LLM services; typical examples include those running on on-premises servers and mobile devices. Although local LLMs are often limited in size due to available compute resources, they offer distinct benefits. First, as queries and prompts are processed in a private environment, security and privacy concerns can be effectively addressed; this is particularly appealing for applications dealing with privacy-sensitive data, such as personal assistants and surveillance cameras. Second, as services do not rely on the cloud or Internet connectivity, response times are more predictable and typically offer low latency; this is well-suited for control applications in edge environments.

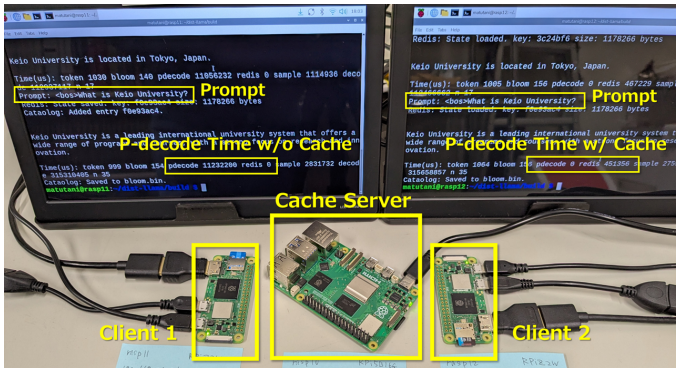

Figure 1. System overview. A local LLM on Client 1 (left) first processes a prompt and uploads its internal states to Cache Server (middle), accelerating a subsequent query on Client 2 (right).

The expansion of local LLM applications to the edge necessitates deployment on resource-constrained platforms. For instance, we have demonstrated local LLMs on Raspberry Pi Zero 2W, which is known as a $15 computer, targeting embedded and wearable applications [1][2]. Accordingly, this paper focuses on this class of low-cost edge computers. However, as demonstrated in Section 5, the performance is quite limited due to constrained compute resources, requiring tens of seconds for simple queries; this negates the benefits of local LLMs mentioned above.

To overcome this limitation, this paper proposes distributed prompt caching to enhance the performance of local LLMs, specifically Time to First Token (TTFT), on resource-constrained edge devices. This approach enables the cooperative sharing

[1] https://www.youtube.com/watch?v=PjXGlZcVzDA
[2] Gemma-3 270M model is used for the demonstration. Although the model performance is limited, it is useful as a base model for specific applications through fine-tuning.

EuroMLSys '26, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2605-7/2026/04
https://doi.org/10.1145/3805621.3807647

of intermediate processing states across devices, leveraging prompt similarity. As shown in Figure 1, for instance, a local LLM on Client 1 (left) first processes a prompt and uploads its internal states to the Cache Server (middle), thereby accelerating a subsequent query on Client 2 (right). The contributions of this work are summarized as follows:

- We identify the performance limitations of local LLMs on low-end edge devices using the MMLU dataset.
- We extend the concept of prompt prefix caching to a multi-device environment to address these performance bottlenecks.
- We introduce a local *catalog* to effectively eliminate the communication overhead associated with state sharing over a wireless network.
- We analyze the break-even point of the proposed approach on real edge hardware to evaluate its practical feasibility.

This paper is organized as follows. Section 2 briefly reviews caching techniques for LLM inference. Sections 3 and 4 present the design and implementation of the proposed distributed prompt caching mechanism. Section 5 discusses experimental results, and Section 6 concludes the paper.

2 Background and Related Work

A typical LLM architecture consists of dozens of Transformer blocks, each primarily comprising Multi-Head Attention (MHA) and Feed-Forward Network (FFN) layers [8]. In particular, MHA is the core component that captures dependencies within a sequence of past tokens to enable next-token prediction; this process is known as autoregressive token generation. In the autoregressive token generation process, the Key-Value (KV) cache [6] is a widely used technique for storing the K and V tensors of past tokens, thereby avoiding redundant computations when generating subsequent tokens.

While conventional KV caches are typically maintained within a single context, they can also be reused across multiple contexts; such a technique is known as prompt caching [2]. This allows systems to bypass all or part of the prompt decoding process when input prompts share a substantial overlap, such as when they are generated from the same template. To further increase cache reuse opportunities, research primarily follows two directions: prefix caching and semantic caching. In prefix caching, common prefixes are pre-defined by service providers in [4], whereas they are dynamically managed using a prefix-tree structure in [9]. Similarly, the KV cache is managed as a radix-tree structure in the SGLang framework [10]. Semantic caching, on the other hand, retrieves reusable KV states based on the cosine similarity between the incoming and cached prompts [7].

In addition, various research efforts have explored enhancing KV cache efficiency. For instance, the caches are compressed to accelerate LLM serving [5]. To reuse KV caches across multi-turn conversations, the caches are saved in cost-effective memory and storage mediums [1].

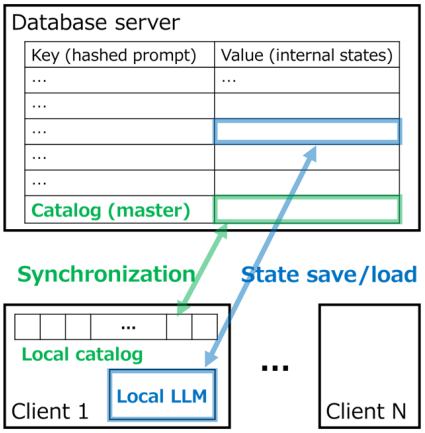

Figure 2. Data structures. The internal states and the master *catalog* are stored in the remote server. Local inference on the clients utilizes the cached states when available. Each local *catalog* is synchronized with the master on the server.

These approaches mentioned above are promising, particularly for LLM servers that process a high volume of queries from clients. In the context of edge-based LLMs, this is especially relevant in scenarios where edge devices serve as clients querying remote servers; in such cases, LLM services rely heavily on server-side infrastructure. In contrast, this paper focuses on local LLM inference on resource-constrained edge devices, where LLM tasks are executed on-device. To this end, we propose a distributed prompt caching mechanism to enhance local inference efficiency by extending the concept of prompt caching to a cooperative framework across multiple low-end edge devices.

3 Distributed Prompt Caching Mechanism

To accelerate local LLM inference on resource-constrained edge devices, in this paper, internal states generated during prompt decoding (also known as prefill) are stored in a remote database server and shared among the same or different devices, as demonstrated in Figure 1. The underlying data structures are detailed in Figure 2. By reusing these cached internal states for local LLM inference, the prompt decoding process for all or part of a given prompt can be bypassed. This approach is beneficial if there is a sufficient similarity between the current and cached prompts, particularly when repeatedly utilizing the identical prompt templates or system instructions. This reduction in prompt decoding time is crucial, as Time to First Token (TTFT) directly impacts the user experience.

3.1 Catalog

The internal states, including the KV cache associated with a specific prompt, are hereafter referred to as the prompt

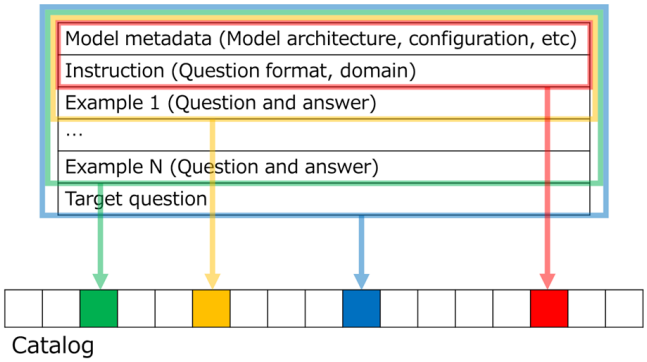

Figure 3. Example of *catalog*. Four distinct ranges of a prompt consisting of the instruction, few-shot examples, and target question are registered in the *catalog*.

cache. The size of each prompt cache entry depends on the context length and the model architecture used; it is often significant (e.g., several megabytes or more) for resource-constrained devices. This data volume imposes substantial communication overhead, specifically the latency associated with uploading and downloading cache entries, potentially outweighing the benefits of prompt cache sharing. Since edge devices are often battery-powered and rely on wireless connectivity, remote database access must be minimized to reduce both power consumption and latency.

To suppress unnecessary database access from edge devices, this paper introduces a Bloom-filter-based *catalog* that compactly summarizes the set of cached internal states stored on the remote server. As shown in Figure 2, each client maintains its local *catalog*, which is synchronized with the master *catalog* on the server. Figure 3 illustrates an example of the *catalog*, where each cell in the bottom part indicates whether the internal states corresponding to a given prompt are cached in the database server.

With the proposed *catalog*, local LLM inference on edge devices follows the steps below:

- Step 1: The edge device tokenizes the input prompt.
- Step 2: It queries the local *catalog* to determine whether the remote server has cached the corresponding states for this prompt.
- Step 3: If the local *catalog* hits, the edge device downloads the prompt cache from the server (see the blue arrow in Figure 2); otherwise, it decodes the prompt locally, uploads the resulting internal states to the server, and correspondingly updates its local *catalog* to reflect the new entry.
- Step 4: It decodes response tokens and outputs them.

The local *catalog* is synchronized with the server asynchronously to reflect updates from other edge devices so as not to impact inference latency (see the green arrow in Figure 2).

To query or update the local *catalog* in Steps 2 and 3, a sequence of token IDs generated from a given prompt is hashed to generate a unique lookup key. To ensure the integrity of the retrieved cache, additional metadata, such as the model name and its configuration parameters, is incorporated into the hash input, as shown in the top part of Figure 3. This approach distinguishes cached states from those generated under different model architectures or quantization settings.

3.2 Partial Matching

To further increase the cache hit ratio, the proposed *catalog* supports partial prompt matching. This enables the reuse of cached prefix states by leveraging the logical structure of prompts. Taking multiple-choice questions as an example, prompts typically consist of the following three parts:

- Instruction: Specifies the task format (i.e., multiple-choice question) and the domain (e.g., *astronomy*).
- Few-shot examples: Contain N pairs of question and answer.
- Target question: The actual question to be answered.

In this case, multiple prompt ranges derived from a single input, such as the instruction alone, the instruction with examples, or the entire prompt, can be registered in the *catalog* during Step 3. As shown in Figure 3, we consider four distinct prompt ranges: 1) the instruction alone, 2) the instruction with the first example, 3) the instruction with all examples, and 4) the entire prompt.

Since a longer sequence of reused tokens results in a more substantial reduction in decoding time, the lookup strategy in Step 2 is adapted to examine these multiple ranges. Specifically, if a match of sufficient length is identified among the examined ranges, the edge device initiates the retrieval of the longest matching prompt cache from the server.

3.3 False Positives

There is a possibility of false positives inherent in the Bloom-filter-based *catalog*. In such cases, edge devices may retrieve an unintended prompt cache from the server; however, since this cache will not match the current prompt, the decoding process cannot be bypassed and is instead fully computed at the edge. Thus, these false positives do not impact the logical correctness while increasing the processing time. This impact is discussed in Section 5.

4 Implementation

Table 1 details the specifications of the server and client devices. The database server runs on a Raspberry Pi 5 Model B with 16GB DRAM, utilizing Redis 8.0.2 (see Cache Server in Figure 1). Redis snapshotting is disabled to minimize disk I/O overhead. For the clients, local LLMs are deployed on two edge devices: Raspberry Pi Zero 2W with 512MB DRAM (low-end setting; see Clients in Figure 1) and Raspberry Pi 5 Model B with 4GB DRAM (high-end setting).

Table 1. Specifications of server and client devices.

	Server	Client (high-end)	Client (low-end)
Platform	Raspberry Pi 5 Model B		Zero 2W
CPU	ARM Cortex-A76 2.4GHz		Cortex-A53 1GHz
DRAM	16GB	4GB	512MB
OS	Raspberry Pi OS 13 (Trixie), 64-bit		

A lightweight custom client program is implemented in C++ and compiled with g++ 14.2.0. The LLM inference, Redis access, and Bloom filter functionalities are implemented using llama.cpp b7957 [3], Hiredis 1.2.0 [4], and libbloom 2.0 [5], respectively. Specifically, `llama_state_get_data()` is used to extract internal states of local LLMs after prompt decoding, while `llama_state_set_data()` is used to restore the saved states. The Bloom filter is configured with a capacity of 1M entries and a target false-positive ratio of 1%; in this setting, its size is only 1.20MB.

Finally, the server and clients are connected via 2.4GHz Wi-Fi 4, as shown in Figure 1.

5 Evaluations

5.1 Evaluation Methodology

The proposed distributed prompt caching is evaluated in terms of Time to First Token (TTFT) and Time to Last Token (TTLT) using two edge computing platforms: Raspberry Pi Zero 2W [6] with 512MB DRAM (low-end) and Raspberry Pi 5 Model B with 4GB DRAM (high-end). For the model configurations, we employ Gemma-3 270M [7] for the low-end setting and the 1B variant for the high-end setting. Greedy sampling is used as the decoding strategy across all experiments.

In the experiments, prompts are generated using the MMLU dataset, which consists of 57 domains [3]. Within each domain, the instruction and few-shot examples are shared across prompts. These examples and the target questions are sampled from the `val` and `test` sets of the same domain. Since the length of the question-answer pairs significantly affects processing time, our experiments specifically focus on pairs with 256 words or fewer; as a result, 6,434 prompts are tested in total.

Given a prompt, all or part of the prompt may hit the distributed prompt cache. We consider the following five cases:

- Case 1: No cache hit (i.e., miss).
- Case 2: Hits the instruction part only.
- Case 3: Hits the instruction part and the first example.
- Case 4: Hits the instruction part and all examples (N = 5).
- Case 5: Hits the entire prompt (i.e., full hit).

Cases 2, 3, 4, and 5 correspond to the red, yellow, green, and blue parts in Figure 3, respectively. To avoid excessive processing time, N is set to 1 for the low-end setting, whereas it is set to 5 for the high-end setting, unless otherwise noted.

[3] https://github.com/ggml-org/llama.cpp
[4] https://github.com/redis/hiredis
[5] https://github.com/jvirkki/libbloom
[6] https://www.raspberrypi.com
[7] https://deepmind.google/models/gemma/gemma-3

Table 2. TTFT and TTLT [sec] on low-end and high-end settings under Case 1 (cache miss) and Case 5 (full hit).

	TTFT			TTLT		
Case	1	5	[%]	1	5	[%]
Low-end	12.59	0.87	6.88	23.74	11.86	49.93
High-end	2.70	2.89	107.08	2.77	2.97	107.10

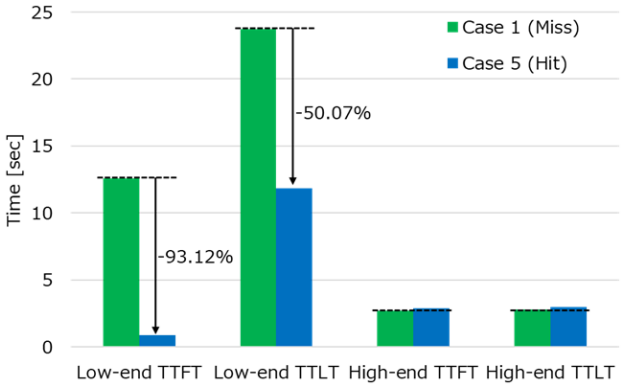

Figure 4. Performance comparison between cache miss and hit cases (based on Table 2).

5.2 Evaluation Results

5.2.1 Benefit vs. Overhead. Table 2 presents the TTFT and TTLT for both low-end and high-end edge settings under Case 1 (cache miss) and Case 5 (full hit). These latency values are averaged over 6,434 prompts. Note that TTFT and TTLT include database access overhead over Wi-Fi (e.g., cache check, download, upload); a detailed breakdown of these components is provided later in this subsection. In the low-end setting, the latency reduction is significant under Case 5; as shown in Figure 4, TTFT and TTLT are reduced by 93.12% and 50.07% on average, respectively. In contrast, the high-end setting shows a different trend; specifically, TTFT and TTLT increase by 7.08% and 7.10% on average, respectively, due to the database access overhead analyzed below.

Table 3 details the latency breakdown for the low-end and high-end edge settings by considering the following six components:

- Token: The time required to tokenize a given prompt.
- Bloom: The time required to determine whether the remote server has cached a given prompt by querying the local *catalog*.
- P-decode: The time required to decode a given prompt.

Table 3. Latency breakdowns [msec] on low-end and high-end settings under Case 1 (cache miss) and Case 5 (full hit).

	Token	Bloom	P-decode	Redis	R-decode	Sample	N	# tokens	State size [MB]
Low-end (Case 1)	3.46	0.30	12580.85	2.42†	11061.04	95.69	1	65.27	2.25
Low-end (Case 5)	3.46	0.19	0.00	861.92	10904.67	84.82	1	65.27	2.25
High-end (Case 1)	1.61	0.00	2688.17	7.84†	72.59	1.45	5	334.11	9.94
High-end (Case 5)	1.56	0.00	0.00	2887.04	78.12	1.67	5	334.11	9.94

† Induced by rate false positives.

- Redis: The time required to download/upload a corresponding prompt cache from/to the remote database server over Wi-Fi.
- R-decode: The time required to decode a response token. In the case of multiple-choice questions, the output token length is typically one (e.g., "A", "B", "C", "D").
- Sample: The time required to sample the response token using a greedy sampling strategy.

Note that the TTFT under Case 1 comprises Token, Bloom, and P-decode, whereas under Case 5 it includes Token, Bloom, and Redis. The TTLT consists of R-decode and Sample in addition to the components comprising the TTFT. The Redis access time depends on the cache entry size and available network bandwidth, where the cache entry size is determined by the model used. In this experiment, the cache entry size is 2.25MB for the 270M model (low-end setting), whereas it is 9.94MB for the 1B model (high-end setting).

As shown in Table 3, for the low-end setting, P-decode requires 12.58 seconds under Case 1, which is replaced by Redis access requiring only 0.86 seconds under Case 5. Consequently, the proposed distributed prompt caching significantly reduces both TTFT and TTLT for the low-end setting in Case 5. In contrast, for the high-end setting, P-decode requires 2.69 seconds under Case 1, whereas Redis access requires 2.89 seconds under Case 5. This indicates that the benefit of a prompt cache hit is offset by the Redis access overhead over Wi-Fi in the high-end setting. In summary, the distributed prompt cache is highly effective for local LLM inference on resource-constrained computing platforms, such as Raspberry Pi Zero 2W.

5.2.2 Benefit of Partial Matching.

Here, we discuss how partial matching of the prompt cache affects the total decoding time (i.e., the sum of P-decode and R-decode). For this analysis, as shown in Figure 3, we selected a single prompt comprising an instruction, five examples (i.e., $N = 5$), and a target question from the *astronomy* domain. The total number of tokens in this prompt is 405. In this analysis, the numbers of matched tokens are 1, 10, 57, 340, and 405 for Cases 1, 2, 3, 4, and 5, respectively.

Table 4 presents the total decoding time (excluding Redis access overhead) under these five cases for the low-end and high-end settings. As the number of matched tokens increases, the total decoding time decreases as expected. For the low-end setting, Cases 2, 3, 4, and 5 reduce the total decoding time by 0.92, 2.61, 13.86, and 15.98 seconds, respectively, compared to Case 1. Figure 5 shows the performance comparison on the low-end setting assuming that the Redis access overhead is 0.86 seconds (represented by the blue part in the graph) as mentioned earlier. Under this condition, the partial prompt caching is proven to be effective especially for Cases 4 and 5 even when this overhead is taken into account.

Table 4. Total decoding time [msec] on low-end and high-end settings under partial matching cases.

	# matched	% matched	T-decode
Low-end (Case 1)	1	0.25	27203.96
Low-end (Case 2)	10	2.47	26288.23
Low-end (Case 3)	57	14.07	24590.09
Low-end (Case 4)	340	83.95	13344.96
Low-end (Case 5)	405	100.00	11220.95
High-end (Case 1)	1	0.25	3361.88
High-end (Case 2)	10	2.47	3280.38
High-end (Case 3)	57	14.07	2918.08
High-end (Case 4)	340	83.95	643.35
High-end (Case 5)	405	100.00	62.9

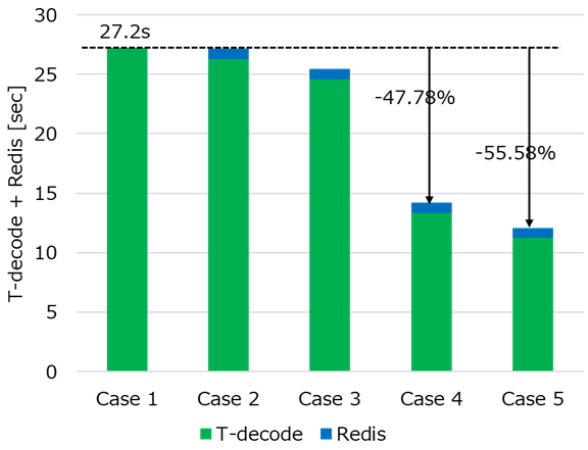

Figure 5. Performance comparison between partial matching cases on low-end setting (based on Table 4).

5.2.3 Benefit of Local Bloom Filter. In the distributed prompt caching mechanism, clients first query the local *catalog* to determine whether the remote server has cached a given prompt, thereby suppressing unnecessary database access. As mentioned above, the Redis access overhead is not negligible (e.g., 0.86 seconds for the low-end setting). Without the proposed *catalog*, every LLM inference would incur this overhead, which would offset the benefit of the distributed prompt caching, especially under a low cache-hit scenario. By introducing the *catalog*, the Redis access overhead is incurred only when a corresponding prompt cache is identified as available, regardless of the actual hit ratio, with the exception of Bloom filter false positives, which are discussed in the next subsection.

5.2.4 Impact of Bloom Filter False Positives. The *catalog* is implemented based on the assumption that the Bloom filter false positive ratio is 1%. In the event of a false positive, the TTFT and TTLT in Case 1 would incur the Redis access overhead. Specifically, for the low-end setting, the expected TTFT and TTLT for Case 1 increase by only 0.86×0.01 seconds, which has a negligible impact on the break-even point of the proposed distributed prompt cache (see also Figure 5).

5.3 Practical Considerations

In Figure 1, local LLMs are deployed on two Raspberry Pi Zero 2W boards (the left and right nodes). As demonstrated above, the proposed distributed prompt caching is particularly beneficial for these devices due to their severe resource constraints. Specifically, the limited CPU performance, reflected in long TTFT and TTLT, is mitigated by reusing internal states from the same or different nodes. Furthermore, the limited DRAM capacity (e.g., only 512MB) is addressed by utilizing the storage of a remote server. We employ an off-the-shelf Redis running on Raspberry Pi 5 (the middle node) as a *cache box*. As shown in Figure 1, by simply adding this *cache box* to the setup, overall performance is significantly improved. Please note that local LLM inference on the left and right nodes remains functional even if the middle node is unavailable, although performance will be significantly degraded due to the loss of the proposed distributed caching.

6 Summary

Since local LLM inference on low-end edge devices imposes a severe performance bottleneck, this paper proposes a distributed prompt caching mechanism by extending the concept of prompt prefix caching to a multi-device setting to fully utilize prompt similarity. While our approach introduces communication overhead associated with state sharing over Wi-Fi, the experiments demonstrate that it is highly effective in the low-end setting, where TTFT and TTLT are reduced by 93.12% and 50.07% on average, respectively. In contrast, the approach is less beneficial for high-end edge devices. Furthermore, the experiments show that partial matching of our distributed cache effectively increases cache reuse opportunities while reducing latency. Finally, thanks to the proposed local *catalog*, communication takes place only when a corresponding cache is likely to exist on the server. This suppresses unnecessary communication overhead, favorably shifting the break-even point of this approach.

A demonstration video of the proposed system running on Raspberry Pi Zero 2W is available at: https://www.youtube.com/watch?v=3qcExKXdkvQ

References

[1] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. Cost-Efficient Large Language Model Serving for Multi-turn Conversations with CachedAttention. In *Proceedings of the USENIX Annual Technical Conference (ATC'24)*. 111––126.

[2] In Gim, Guojun Chen, Seung-Seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt Cache: Modular Attention Reuse for Low-Latency Inference. In *Proceedings of the Annual Conference on Machine Learning and Systems (MLSys'24)*. 325–338.

[3] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring Massive Multitask Language Understanding. In *Proceedings of the International Conference on Learning Representations (ICLR'21)*.

[4] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'23)*. 611–626.

[5] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. 2024. CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM'24)*. 38––56.

[6] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently Scaling Transformer Inference. In *Proceedings of the Annual Conference on Machine Learning and Systems (MLSys'23)*. 606–624.

[7] Sajal Regmi and Chetan Phakami Pun. 2024. *GPT Semantic Cache: Reducing LLM Costs and Latency via Semantic Embedding Caching*. arXiv:2411.05276

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS'17)*.

[9] Lu Ye, Ze Tao, Yong Huang, and Yang Li. 2024. ChunkAttention: Efficient Self-Attention with Prefix-Aware KV Cache and Two-Phase Partition. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL'24)*. 11608–11620.

[10] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS'24)*.