CaR: An Efficient KV Cache Reuse System for Large Language Model Inference

Kexin Chu, Tzechinh Liu, Yunding Li, Pengchao Yuan, and Wei Zhang

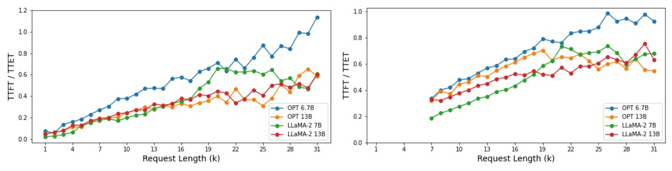
University of Connecticut

***Abstract*—The KV cache in the current LLM serving system is mainly used for accelerating the processing of a request. After the response is generated, the KV cache is aggressively deleted. However, the KV cache can be reused across the requests in some scenarios such as virtual assistants and multi-turn conversations, which can dramatically reduce the computation cost and improve the serving latency. Caching the historical tokens raises substantial memory requirements. Further, in the existing serving system, the request scheduler and KV cache are treated separately and independently. However, they are tightly coupled together. a CaR is a multi-tier cache system to enable the KV cache reuse and share across the requests. Instead of using DRAM, CaR leverages CXL (Compute Express Link) as the external memory with the GPU-CXL direct data transfers, which can avoid the bandwidth contention and interference caused by the tasks running on the CPU. To wisely use the fast-tier HBM, we co-design the KV cache manager and scheduler to cooperate with the request scheduling and token placement across the tiers. To hide the reload time, CaR designs a pipeline prefetcher to overlap the communication and computation. Further, CaR proposes a quality-aware sparsification algorithm to compress the KV cache in each layer with a heterogeneous manner. It not only reduces the data transfer size but also reduces the KV cache size. To remove the data offload from a request's critical path, we design the asynchronous offload engine to swap out the data from HBM to CXL in the background. Our experiment shows that CaR can reduce TTFT by about 30%, especially in long context scenarios, where TTFT of OPT-13B is reduced by more than 60%.**

I. Introduction

Large Language Models (LLMs) such as GPT-3 [5], OPT [36] and Llama [31] bring a groundbreaking shift with the generative capabilities [15], [18], [32]. Following this success, we have seen a surge in LLM-based serving systems like ChatGPT, Copilot, etc. These LLM serving systems usually face a large number of users and are expected to provide low latency and high throughput [2], [6], [14], [25], [26], [34]. A request will be processed through the prefill and decode stages. A token generation in the decode stage depends on all of the previous tokens in the request. To avoid recomputation, the existing systems typically leverage the KV cache to accelerate the processing by storing the previous tokens in a request [18]. The KV cache takes a substantial HBM space in the GPU.

To limit the memory usage of the KV cache, the current LLM inference systems directly discard it once the request is completed [3], [15], [18], [23], [28], [41]. However, the KV cache can be reused and shared across the requests [39]. By reusing the KV cache from the previous requests, the expensive computation in the prefill stage can be avoided and substantially reduce the latency, which will be more beneficial for long prompts. As shown in Figure 1, the prefill latency dramatically increases as the length grows.



(a) ShareGPT (b) LongAlign

Fig. 1: The compare of prefilling latency(TTFT) and end-to-end latency(TTET).

To accommodate the historical KV cache, especially for the long prompt [4], [8], [12], we have to leverage the external memory [19] and wisely manage the token placement across the tiering memory. However, it will bring several challenges: 1) where to place the KV cache? 2) How to limit the data transfer overhead? 3) When should kick off the data transfers?

To address the above challenges, we propose a multi-tier cache system called CaR, which can wisely manage the historical prompts to reuse the KV cache across the requests and efficiently migrate the data across the fast-tier HBM and slow-tier CXL [17], [30], [35]. From our preliminary results, reloading the KV cache from the slow-tier memory to HBM can improve the prefill latency by 30% rather than directly recomputing the tokens, which consumes a lot of GPU computation resources. We believe the latency can be greatly improved with our proposed techniques and solutions. Our contributions are:

- We identify the main sources of latency when processing long prompts and the major design defect in existing LLM serving systems.
- We build a multi-tier KV cache system, designated as CaR, which utilizes a modern CXL device for external memory. The system enables direct data transfer between the GPU HBM and CXL, thereby avoiding interference from tasks running on the CPU.
- We co-design the scheduler and cache management system, encompassing techniques such as cache replacement and prefetching, to achieve high throughput.
- We propose an improved algorithm for compressing the KV cache in each layer in a heterogeneous manner. This algorithm can reduce the KV cache size, thereby

mitigating storage stress and improving data transmission from the CXL to the GPU.

II. Background and Motivation

A. Generative LLM Inference

Generative LLM inference comprises two phases [1], [16], [24], [32]: prefill and decoding. In the prefill phase, LLM models compute all input tokens in a single pass. Subsequently, in the decode phase, LLM models utilize previous tokens, including both the prompt and newly generated ones, as input to generate the subsequent output token. Consequently, the decode phase unfolds iteratively, processing one token at a time until either the total number of generated tokens reaches the predefined maximum or the special EOS token is produced.

When generating a new token, all KV tensors of preceding tokens in that sequence are required for computing self-attention. The length of the sequence quadratically amplifies the size of the KV tensors, thereby increasing execution time. To alleviate this quadratic overhead in LLM inference, the Key and Value tensors are typically cached in GPUs for reuse in subsequent generation steps, a practice known as KV caching [9]–[11], [21], [29], [33].

B. Opportunities and Challenges

The KV cache in the existing LLM serving system primarily accelerates processing within a request. Unfortunately, it's aggressively discarded once the request concludes. However, there are scenarios where the KV cache can be reused [9]. For instance, in a virtual assistant system, the same request may originate from different users. In ChatGPT, users engage in multi-round conversations with dependent requests and responses. If we intelligently manage historical tokens and enable sharing of the KV cache across requests, we can significantly optimize the time-consuming and compute-intensive prefill processing, particularly for long prompt requests [20].

To assess the prefill cost within the overall serving time for a request, we measure both the Time to First Token (TTFT) during the prefill phase and the Time to End Token (TTET) during the decoding phase, leveraging data from the ShareGPT and LongAlign datasets. It's notable that as the prompt length increases, the TTFT experiences a significant surge, as illustrated in Figure 1, thus emerging as the primary processing cost. Given the prevalence of lengthy prompts in complex tasks, there's a pressing need to optimize performance within the prefill stage of the LLM serving system. Our preliminary experiments, detailed in Section IV, focus on evaluating the benefits of cache reuse instead of simply recomputing everything in the prefill stage. These experiments underscore the promising potential of cache reuse across requests as a strategy for performance optimization.

However, storing the historical KV cache requires a significant amount of memory space [17]. For instance, a request containing 2K tokens can occupy approximately 2.6GB of KV cache space with the OPT 30B model. Considering that the model parameters in OPT 30B already consume 56GB of memory space, it becomes apparent that a system equipped with 4 A100-40GB GPUs can only accommodate 40 active requests with 2K tokens each, severely limiting system throughput. To address this limitation, leveraging external storage space becomes imperative. The emergence of a new external memory device - CXL - presents a promising opportunity to expand memory space beyond the GPU HBM. CXL facilitates direct migration of the KV cache between the CXL and GPU through GPU-direct. However, in a multi-tier KV cache system characterized by heterogeneous speeds, several significant challenges are discussed below.

Where to place the data across the tiers?

The HBM on GPUs is very expensive while extremely performant. CXL is slower than HBM, however, it provides much larger capacities. It is not trivial to wisely manage the data placement and replacement across the tiers, and then efficiently use the HBM capacity and improve the fast-tier hit ratio. The KV cache size for a request varies a lot and incurs different migration costs. This is different from the page replacement in the operating system, which is typically faced with a fixed normal or huge page size and has the same migration cost. How can we apply these kinds of unique characteristics to cache replacement policy design other than the factors of access frequency and access time?

How to limit the migration overhead across the tiers?

The data in the CXL device is required to reload into HBM for use. We want to avoid the GPU stall waiting for the data to be loaded. This not only adds to the latency but also incurs low GPU utilization. Although CXL 3.0 can support 64 GB/s bandwidth, the access speed still much lags behind the GPU's processing capability. How can we allow the GPU to get the KV cache data in on time from the CXL layer by controlling the data migration time and data size? A request keeps generating new tokens to be saved into the KV cache. To avoid the fast-tier becoming full and then blocking the request's continuous processing, how should we do the offload to make the available HBM space and avoid the interruption while the GPU is busy with the request processing?

When to migrate the data across the tiers?

When reloading the data into HBM is not trivial. Reloading too early before the next request starts to be used can reduce HBM usage efficiency because the data has to wait until a related request gets scheduled. Reloading too late impacts a request's serving time because the request has to be temporarily suspended until the data is ready inside the HBM. The request scheduler determines what requests will form a batch to be served on the GPU. There is a tight relationship between the scheduler and the reload engine. How to co-design the cache manager and the scheduler to optimize request serving time and data usage time, and then reduce the data idle time (sitting in the HBM and waiting for to be used) and request suspend time (waiting for data to be loaded into HBM)?

III. System Design

A. Overview

In this paper, we introduce CaR, a multi-tier cache system to enable historical KV cache reusing across requests, instead of

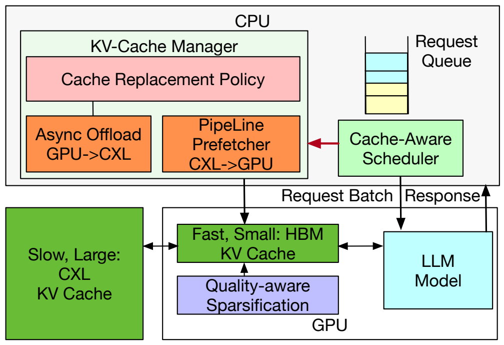


Fig. 2: The System Architecture of CaR

discarding them once the decoding is finished. CaR maintains a cache table to record whether the KV cache of the prompts is stored inside HBM or CXL-based external memory. If a new request has a cache hit by looking up the table, the KV cache can be directly retrieved from the memory instead of going through the time-costly and compute-costly recomputation. In this way, CaR can significantly reduce the prefill latency and GPU computation consumption, improving GPU throughput.

We show CaR's architecture with the modules highlighted in figure 2. Instead of treating memory manager and scheduler separately in the traditional system, CaR co-designs scheduler and cache manager due to the tightly coupled relationships. Scheduler feeds the requests' information going to run on GPU to the cache manager. The score-based cache policy determines what data should be offloaded from the HBM to CXL and reloaded from CXL to HBM. The pipelined load engine reloads the required KV cache layer by layer to overlap the migration and GPU execution. The asynchronous offload engine runs in the background to mask the data transfer contention with the users' requests. The quality-aware sparsification module further reduces the data transfer size and KV cache memory consumption.

B. Co-design of Scheduler and Cache Management

1) Scheduler: As shown in Figure 3, the scheduler employs a priority queue to determine the order of execution once the GPU becomes available. To increase the system's overall throughput, we categorize requests into three groups based on the size of the KV cache to be transmitted from the CXL to the GPU, and assign a priority to each group accordingly:

- Requests that can reuse the KV cache already present in the HBM are given the highest priority.
- Requests that do not involve any historical data follow, with the second-highest priority.
- Requests falling outside of these two scenarios are assigned the lowest priority.

It should be noted that requests capable of partially utilizing the KV cache from the CXL are assigned a lower priority than those with no cache at all. This is because the absence of cache in the latter case avoids the need for data transfer, thereby allowing more time for subsequent data movement. Additionally, this approach prevents requests that can only partially use the KV cache from being prioritized over those

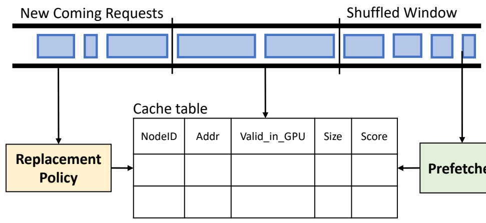


Fig. 3: The Co-design of Scheduler and Cache Manager

with no available cache, avoiding the risk of both types of requests entering a waiting state due to untimely transfers, which would result in wasted GPU time.

While this approach ensures minimal transmission latency, the conventional priority queue may result in certain low-priority requests waiting for extended periods before execution. Another issue arises from the constantly changing queue, which can confuse the prefetcher. To address these challenges, we maintain a fixed-size window at the front of the queue, known as the Shuffled Window. This ensures that all requests are executed after an acceptable waiting time and provides ample opportunities for the prefetcher to fetch the required KV cache in a timely manner.

To facilitate seamless collaboration between the scheduler and the cache management module, a cache table has been meticulously implemented to save pertinent metadata. The NodeID field within each entry acts as a unique identifier for the KV cache node. The Addr field specifies the memory address of the KV cache node, pinpointing its location whether in the HBM or CXL-based external memory. The Size field provides a clear indication of the memory size allocated to the current KV cache node. To track the presence of KV cache nodes within the HBM, the Valid_in_GPU field is utilized. If a KV cache node is currently resident in HBM, this field is set to True. Lastly, the Score field captures the importance score assigned to each KV cache node, which is instrumental in guiding the cache's replacement policy.

2) Prefetcher: Within the Shuffled Window, for each request, the scheduler initiates by fetching the NodeID from memory, using this identifier to look up the corresponding entry in the cache table. If an entry's Addr is valid and the Valid_in_GPU status is False, it signifies that the KV cache node is currently residing on the CXL and is poised for forthcoming reuse. In such instances, the scheduler seamlessly relays the Addr and Size details to the prefetcher.

Through the method detailed in [40], we utilize the LLM model to predict the output sequence length and calculate TPOP (Time Per Output Token) dynamically. We then pass this information to the prefetcher. The prefetcher sets a timer for each request to monitor its progress. It decides when to prefetch by comparing the transmission latency with the GPU's service time for the next request. Periodically, the prefetcher decides whether to fetch based on the following formula's outcome:

$$fetch(t) = \mathbb{I}_{\min_{0\le i\le N}\{TPOP\times len_i-(t-timer_i)\}\le \frac{SIZE}{bandwidth}}(t)$$

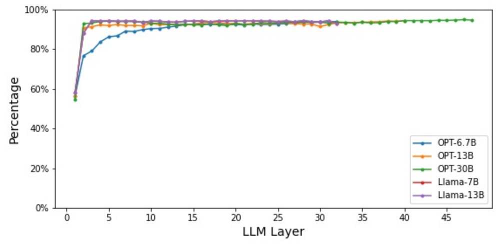


Fig. 4: Then sparsity rate of each layer in the LLM model.

Here, N represents the total number of requests currently being handled by the GPU, t denotes the current time, len_i is the predicted output length for request i, and $SIZE$ is the size of the KV cache required to be fetched.

3) Cache Replacement Policy: We propose a cache replacement policy designed to evict KV cache nodes from HBM when it reaches full capacity. Our policy targets the KV cache node with the smallest Score for eviction. The scheduler updates the Score according to the following rules

- When a request enters the Shuffled Window, the Score is increased by 1.
- when a request is sent to the GPU for inference, the Score is further increase by 1.
- when a request completes its autoregressive generation, the Score is decreased by 2.

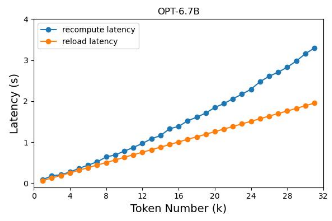
C. Efficient Data Transmission Module

1) Layer-wise Pipeline Loading: The inference process of LLM models on GPUs is performed layer by layer. We adopt a pipeline loading engine, allows loading the KV cache needed by subsequence layers while the GPU execute prefilling or decoding processes of first layer. By doing this, the KV cache loading time is overlaped with the computation time.

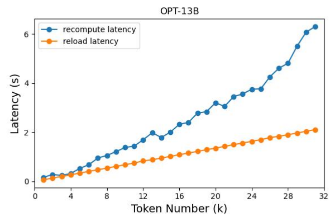
2) Asynchronous Data Offloading: To eliminate the offloading overhead from the request's critical path, we introduce an asynchronous data offloading mechanism to support active data offloading. Once the utilization of HBM exceeds 90%, the offloader begins actively offloading the KV cache node based on the Policy's Score. The entire data offloading process is performed asynchronously. When the offloading engine initiates an offloading task, a success signal is sent to CaR, enabling the GPU to continue executing the LLM inference or prefetching required KV cache data.

D. Quality-aware Compression

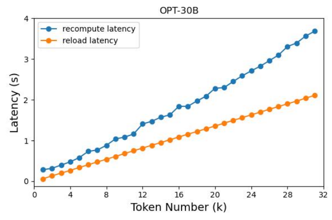
As the size of the KV cache increases linearly when the token numbers increase, long contexts result in high transmission overhead when being reloaded. Data compression offers a viable solution to mitigate this issue. However, the sparsity of the attention score matrix varies across different layers [37]. As shown in Figure 4, if we consider elements as zeros while they fall below 1% of the row-wise maximum value, the sparsity in the first layer is approximately 60%, then it gradually increases to about 95%. Based on this observation. We employ a quality-aware compression strategy to adaptively control the compression ratio of each layer. Maximize the overall KV



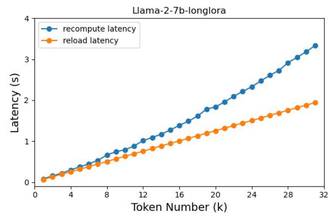
(a) OPT-6.7B



(b) OPT-13B



(c) OPT-30B



(d) Llama-2-7b-longlora

Fig. 5: The latency comparison of recompute and reload.

cache compression rate while ensuring accuracy. Upon loading a new LLM model into CaR, the system utilizes predefined batch prompts as input to execute inference and obtain the sparsity data of each layer. In the subsequent LLM inference process, CaR dynamically determines the compression ratio of each layer based on the acquired sparsity data. It utilizes the attention score matrix's scores to discard the KV cache corresponding to tokens with lower scores.

IV. Evaluation

In this section, we evaluate CaR using different LLM models such as finetuned OPT [36] 6.7B, 13B, 30B, and Llama-2 longlora [7] 7B, 13B. We utilize the following datasets: ShareGPT [27], [38], Longalign [13], and WikiText [22].

We implemented CaR based on vLLM [18] and deployed it on a system with 8 NVIDIA A100-40GB GPUs, 128 GB DRAM, and 10TB CXL disks.

A. Recomputation vs KV cache Sharing

We assess the latency of various LLM models across different request lengths,as shown in Figure 5. Our preliminary results show that leveraging stored KV cache for reuse yields considerable improvements over recomputation. the KV cache reuse demonstrates the potential to reduce TTFT by 30% in scenarios with long context. And with OPT-13B, this reduction even exceeds 60%. Although the latency of reloading remains significant, the delay in the critical path can be further reduced with the use of our designed prefetcher.

V. Conclusion

This Paper proposes CaR, a multi-tier cache system to support the reuse and sharing of the KV cache across the requests. CaR achieves a significant reduction in the recomputation overhead of the KV cache in LLMs. Our preliminary experiment shows that CaR can reduce TTFT by about 30%, especially in long context scenarios, where TTFT of OPT-13B is reduced by more than 60%.

References

[1] A. Agrawal, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, and R. Ramjee, "Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills," *arXiv preprint arXiv:2308.16369*, 2023.

[2] S. Ahmad, H. Guan, B. D. Friedman, T. Williams, R. K. Sitaraman, and T. Woo, "Proteus: A high-throughput inference-serving system with accuracy scaling," 2024.

[3] R. Y. Aminabadi, S. Rajbhandari, A. A. Awan, C. Li, D. Li, E. Zheng, O. Ruwase, S. Smith, M. Zhang, J. Rasley, and Y. He, "Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '22. IEEE Press, 2022.

[4] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *arXiv preprint arXiv:2004.05150*, 2020.

[5] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[6] C. Chen, S. Borgeaud, G. Irving, J.-B. Lespiau, L. Sifre, and J. Jumper, "Accelerating large language model decoding with speculative sampling," *arXiv preprint arXiv:2302.01318*, 2023.

[7] Y. Chen, S. Qian, H. Tang, X. Lai, Z. Liu, S. Han, and J. Jia, "Longlora: Efficient fine-tuning of long-context large language models," 2024.

[8] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating long sequences with sparse transformers," *arXiv preprint arXiv:1904.10509*, 2019.

[9] L. D. Corro, A. D. Giorno, S. Agarwal, B. Yu, A. Awadallah, and S. Mukherjee, "Skipdecode: Autoregressive skip decoding with batching and caching for efficient llm inference," 2023.

[10] T. Dao, "Flashattention-2: Faster attention with better parallelism and work partitioning," *arXiv preprint arXiv:2307.08691*, 2023.

[11] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *Advances in Neural Information Processing Systems*, vol. 35, pp. 16 344–16 359, 2022.

[12] J. Ding, S. Ma, L. Dong, X. Zhang, S. Huang, W. Wang, N. Zheng, and F. Wei, "Longnet: Scaling transformers to 1,000,000,000 tokens," *arXiv preprint arXiv:2307.02486*, 2023.

[13] Y. Ding, L. L. Zhang, C. Zhang, Y. Xu, N. Shang, J. Xu, F. Yang, and M. Yang, "Longrope: Extending llm context window beyond 2 million tokens," 2024.

[14] C. Guo, J. Tang, W. Hu, J. Leng, C. Zhang, F. Yang, Y. Liu, M. Guo, and Y. Zhu, "Olive: Accelerating large language models via hardware-friendly outlier-victim pair quantization," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–15.

[15] Y. Jin, C.-F. Wu, D. Brooks, and G.-Y. Wei, "Increasing gpu utilization during generative inference for higher throughput," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[16] J. Juravsky, B. Brown, R. Ehrlich, D. Y. Fu, C. Ré, and A. Mirhoseini, "Hydragen: High-throughput llm inference with shared prefixes," *arXiv preprint arXiv:2402.05099*, 2024.

[17] B. Kim, S. Cha, S. Park, J. Lee, S. Lee, S.-h. Kang, J. So, K. Kim, J. Jung, J.-G. Lee *et al.*, "The breakthrough memory solutions for improved performance on llm inference," *IEEE Micro*, 2024.

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

[19] Y. Lee, J. Chung, and M. Rhu, "Smartsage: training large-scale graph neural networks using in-storage processing architectures," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 932–945.

[20] D. Li, R. Shao, A. Xie, Y. Sheng, L. Zheng, J. Gonzalez, I. Stoica, X. Ma, and H. Zhang, "How long can context length of open-source llms truly promise?" in *NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following*, 2023.

[21] B. Lin, T. Peng, C. Zhang, M. Sun, L. Li, H. Zhao, W. Xiao, Q. Xu, X. Qiu, S. Li *et al.*, "Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache," *arXiv preprint arXiv:2401.02669*, 2024.

[22] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," 2016.

[23] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[24] P. Patel, E. Choukse, C. Zhang, Íñigo Goiri, A. Shah, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative llm inference using phase splitting," 2023.

[25] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[26] K. Santhanam, D. Raghavan, M. S. Rahman, T. Venkatesh, N. Kunjal, P. Thaker, P. Levis, and M. Zaharia, "Alto: An efficient network orchestrator for compound ai systems," *arXiv preprint arXiv:2403.04311*, 2024.

[27] ShareGPT, "Sharegpt," https://sharegpt.com.

[28] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang, "Flexgen: high-throughput generative inference of large language models with a single gpu," in *Proceedings of the 40th International Conference on Machine Learning*, ser. ICML'23. JMLR.org, 2023.

[29] J. Stojkovic, E. Choukse, C. Zhang, I. Goiri, and J. Torrellas, "Towards greener llms: Bringing energy-efficiency to the forefront of llm inference," *arXiv preprint arXiv:2403.20306*, 2024.

[30] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong *et al.*, "Demystifying cxl memory with genuine cxl-ready systems and devices," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 105–121.

[31] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023.

[32] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[33] H. Wang, Z. Zhang, and S. Han, "Spatten: Efficient sparse attention architecture with cascade token and head pruning," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 97–110.

[34] H. Wang, H. Xu, Y. Wang, and Y. Han, "Cta: Hardware-software co-design for compressed token attention mechanism," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 429–441.

[35] L. Xiang, Z. Lin, W. Deng, H. Lu, J. Rao, Y. Yuan, and R. Wang, "Matryoshka: Non-exclusive memory tiering via transactional page migration," *OSDI*, 2024.

[36] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "Opt: Open pre-trained transformer language models," 2022.

[37] Z. Zhang, Y. Sheng, T. Zhou, T. Chen, L. Zheng, R. Cai, Z. Song, Y. Tian, C. Re, C. Barrett, Z. Wang, and B. Chen, "H2o: Heavy-hitter oracle for efficient generative inference of large language models," in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. [Online]. Available: https://openreview.net/forum?id=RkRrPp7GKO

[38] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. P. Xing, H. Zhang, J. E. Gonzalez, and I. Stoica, "Judging llm-as-a-judge with mt-bench and chatbot arena," 2023.

[39] L. Zheng, L. Yin, Z. Xie, J. Huang, C. Sun, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng, "Efficiently programming large language models using sglang," 2023.

[40] Z. Zheng, X. Ren, F. Xue, Y. Luo, X. Jiang, and Y. You, "Response length perception and sequence scheduling: An llm-empowered llm inference pipeline," 2023.

[41] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving," *OSDI*, 2024.