RAG Workloads Performance with KV Cache Partial Recomputation

Mayank Bhatia, Tzu-Yi Chan, Allison Ye, David Zhu
{mayankb3,tzuyic2,ay14,yuerzhu2}@illinois.edu
University of Illinois Urbana-Champaign, Urbana, Illinois, USA

ABSTRACT

As the growth of LLM workloads significantly outpaces the growth of GPU memory capacity, modern Retrieval Augmented Generation (RAG) frameworks consider CPU memory as a larger but slower memory layer. The serving frameworks mainly take advantage of the large CPU memory pool in two ways: (1) offloading portions of the KV cache to CPU memory to support longer sequence lengths [10, 13, 15], (2) offloading external knowledge-base KV caches [3, 12]. Offloading KV caches for reuse introduces an interesting tradeoff: the advantage is reduced redundant computation, and the disadvantage is increased latency due to high I/O bandwidth requirements when copying from host memory.

In this work, we implement the recently proposed strategies which recompute a prefix of the KV cache and asynchronously load the remainder to reduce generation latency[6, 8]. We explore their accuracy and overheads. We show that the integer linear program (ILP) approach from [6] can be faithfully simplified to a closed-form approximation formula to avoid NP-hard optimization in the pipeline. We also discover that the two-pointer approach from [8] designed for KV caches stored on disk still has performance benefits for KV caches in the CPU memory tier provided a sufficiently long input sequence length.

1 INTRODUCTION

Retrieval Augmented Generation (RAG) enhances large language models (LLMs) by incorporating external knowledge retrieved before generating a response. When a user inputs a query, the system searches a database (e.g., a vendor store) for relevant context and prepends the retrieved text to the query, forming the full input for the LLM. This process enables LLMs to generate outputs that extend beyond their training data and incorporate current or domain-specific information [3, 12]. However, incorporating this external context significantly increases inference latency due to the rapid growth of key-value (KV) caches, which store previously computed token representations.

After an extended input is provided to the model, inference proceeds in two stages: prefill and decode. In the prefill stage, the model computes query (Q), key (K), and value (V) representations for every token in the input via linear projections and attention operations. During the decode stage, the model sequentially generates new tokens by appending each newly generated token to the sequence and computing new Q, K, and V representations only for that token. Predicting each subsequent token requires attention computations involving previously generated tokens, whose K/V representations have already been computed. To avoid redundant computation, these previously computed K/V representations are stored in a KV cache [4, 7, 9, 17].

Large RAG workloads can quickly exceed GPU memory limitations, especially on consumer-grade and edge devices designed for cost and power efficiency. Not only does the KV cache grow due to the increased sequence lengths, but also some RAG workflows employ multiple KV caches to improve on cache hit rates. Consequently, it becomes essential to offload KV caches from limited GPU memory to the much larger CPU memory layer, necessitating frequent data transfers between the two devices [1, 11, 14, 15]. For example, InfiniGen [11] offloads most KV tokens to preserve model performance in 1-million-token inference. CacheBlend [17] is a recently proposed scheme that reduces inference delay through effective cache reuse and optimized system design.

In this work, we explore and improve upon strategies to speed up the acquisition of KV values for an input sequence by re-computing the prefix during otherwise idle GPU cycles and copying the remainder from host memory. We explore these partial recomputation strategies:

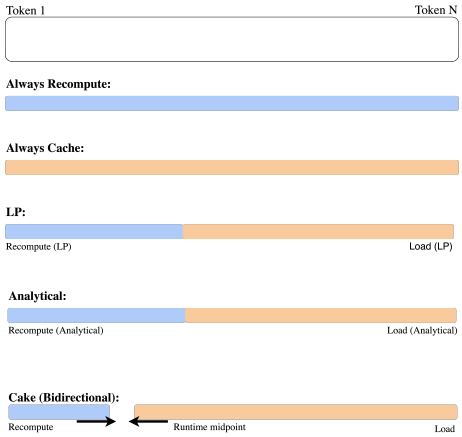

Figure 1: KV cache partial recomputation strategies.

(1) **Always recompute:** Always recompute KV values.
(2) **Always cache:** Always store results in the GPU KV cache and offload to CPU memory if GPU memory limit is exceeded.
(3) **LP:** Following the LP used by KVPR[6], find the recompute-load split using linear programming.
(4) **Cake[8]:** Using a two-pointer approach, recompute KV cache starting from the beginning of the sequence and I/O load from the end, meeting in the middle.
(5) **New:** We derive a simple approximation for the ILP introduced by KVPR[6], and improve on it by considering the overhead of computing Q when acquiring K and V.

Our main contributions are improving the efficiency and accuracy of the LP provided by KVPR[6] and demonstrating that Cake[8], while designed for disk, improves CPU caching as well.

2 RELATED WORK

The **transformer architecture**[16] is at the heart of most application-level optimizations such as this one. The transformer architecture is illustrated in Figure 2. We implement a transformer in CUDA to robustly test and measure the impact of various KV-optimization policies on the time taken to pass to feedforward neural network. In particular, we are interested in the acquisition of KV-values required to compute attention. Instead of computing these KV values repetitively at every autoregressive step or experiencing the full latency of CPU memory copies of these KV-values from the offloaded KV-cache, we explore hybrid approaches which overlay asynchronous reads from host CPU memory with computing a few of the KV-cache entries to reduce the overall latency between receiving input embeddings and beginning attention computation in the transformer block.

Recent works like **KVPR**[6] and **Cake**[8] look to dynamically balance KV cache recomputation and data transfer based on system profiling to address the PCIe bandwidth bottleneck present in I/O-heavy approaches.

KVPR introduces an I/O-aware approach that partitions the KV cache into re-computable and transferable segments. A profiler first analyzes input and hardware characteristics to extract key parameters like GPU compute speed and PCIe bus speed. Then, a scheduler finds the optimized split point in the sequence using an integer linear programming (ILP) solution. However, the scheduler's compile-time analysis did not account for runtime overhead, which could significantly impact overall performance. Our contribution is providing guidelines on how to efficiently utilize the ILP, an efficient approximation, and improving upon the original ILP by also Q-computation in addition to acquring K and V.

Cake targets the prefill stage by employing a bidirectional KV cache loader that concurrently loads prefix cache from disk and recomputes KV entries on the GPU block by block. It adapts to current compute and I/O bandwidth conditions, eliminating the need for manual tuning. However, this run-time split introduces additional overhead. At the end of each block, operations like pointer checks, stream stalls, and kernel relaunches cause unavoidable delays. Furthermore, Cake was designed for cache on disk. We evaluate the overheads of Cake on a CPU cache.

Additionally, **KVPR** and **Cake** do not cite GitHub repositories to evaluate their techniques. So, we re-implement their techniques as well as formulating our own modifications and best-practices.

3 PROPOSED METHOD

Our primary novel contributions are an analysis of the KVPR ILP, an improved modification of it, and highly accurate closed-form approximations from the ILP to avoid requiring an optimizer. First, we note that executing the ILP on various sequence lengths is highly redundant. This is because the sequence length term is common to all terms in the maximization in 1, so the ILP actually is best used as a one-time ratio-finder. Given a particular sequence length, the optimal partition is some fraction of the sequence length. All other sequence length inputs to the ILP will return different outputs which are the same ratio to the input sequence length. Therefore, the ILP only need be ran once with system configurations of v_{com} and v_{gpu} representing the communication and computation rates, after which the extracted ratio can simply be multiplied to all incoming sequence lengths, with the reasonable assumption that v_{com} and v_{gpu} don't change significantly during autoregression. However, we take this one step further and note that an ILP is not needed at all, proposing an analytical method that obtains the same output for a given sequence length.

Notably, we also implement the Cake algorithm designed for disk offloading to explore the impact of this algorithm which expects the KV-cache to be on disk to see relative speedups in a CPU-GPU environment.

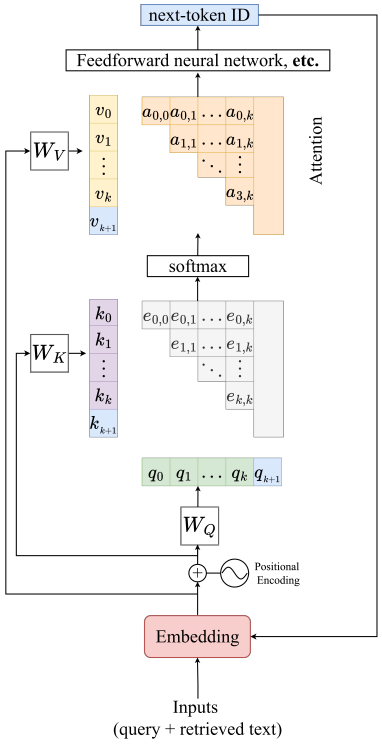

Figure 2: Self-attention within a transformer.

3.1 Analytical Method

Given the current sequence length s at the i-th decoder layer, let l be the recompute-load split point such that $0 \le l \le s$. Let $X^i[0:l]$ represent the activations that must first be transferred from CPU to GPu in order to recompute KV values, and let $K^i[l:s]$ and $V^i[l:s]$ represent the remaining KV cache to be loaded.

The LP problem from the KVPR[6] paper seeks to minimize the total processing time as follows:

$$\min_{l} \quad \left(\frac{M_{X^i[0:l]}}{v_{\text{com}}} + \max\left(t^i_{\text{recomp}}, \frac{M_{KV^i[l:s]}}{v_{\text{com}}} \right) \right) \tag{1}$$

$$\text{s.t.} \quad 0 \le l \le s, \quad \forall i \in \{1, \ldots, n\}$$

where

- $\frac{M_{X^i[0:l]}}{v_{com}} = \frac{b \times l \times h \times p}{v_{com}}$ = activation transfer time
- $t^i_{recomp} = \frac{4 \times b \times l \times h^2}{v_{gpu}}$ = recomputation time

- $\frac{M_{KV^i[l:s]}}{v_{com}} = \frac{2\times b\times(s-l)\times h\times p}{v_{com}}$ = KV cache loading time
- v_{gpu} = GPU performance in FLOPs/sec
- v_{com} = GPU-CPU PCIe bus speed in bytes/sec
- b, h, p = batch size, embedding dimension, and precision

Instead of solving the LP problem, we transform this into an analytical solution by directly setting recomputation time equal to loading time and rounding our final result to the nearest integer.

$$\frac{4blh^2}{v_{gpu}} = \frac{2b(s-l)hp}{v_{com}}$$

Simplifying, we get that the optimal split point l^* is

$$l^* = \lfloor c_1 \frac{spv_{gpu}}{2hv_{com} + pv_{gpu}} \rceil$$

where $c_1 = \frac{1}{1.013}$ is an empirical normalizing constant.

However, these formulas do not include query recomputation cost. After adding this to the total recomputation time, we get the following formula:

$$t^i_{recomp} = \frac{N_{KV^i[0:l]} + N_{Q^i}}{v_{gpu}} = \frac{4blh^2 + 2bsh^2}{v_{gpu}}$$

Our new equations and optimal split point are as follows (with empirically normalizing constant $c_2 = \frac{1}{1.026}$):

$$\frac{4blh^2 + 2bsh^2}{v_{gpu}} = \frac{2b(s-l)hp}{v_{com}}$$

$$l^* = \lfloor c_2 \frac{s(pv_{gpu} - hv_{com})}{2hv_{com} + pv_{gpu}} \rceil \tag{2}$$

3.2 Implementation Details

3.2.1 Cake Implementation. Due to no open source code, we implement Cake from scratch. We began with a CPU prototype using Python's `multiprocessing` library to simulate parallelism and test correctness. Transitioning to the GPU implementation introduced several challenges, particularly in coordinating concurrent compute and memory transfer operations.

In our CUDA version, we utilize two streams: one for compute and one for data transfer. Each stream maintains its own asynchronous queue for kernel launches and `cudaMemcpyAsync` calls. To avoid race conditions or redundant work, we ensure that no sequence block is submitted to both streams simultaneously. This coordination is achieved using CUDA events: a compute event and a copy event track the progress of each stream. New work is only issued to a stream when its corresponding event signals that the stream is ready (i.e., has completed previous tasks and its queue is empty).

The pseudocode below outlines the core logic of our overlapped QKV computation and cache loading approach:

3.2.2 Other Details. We also implement a RAG pipeline that retrieves context embeddings from the first 100k entries of the WikiSnippets dataset[5] to augment the original input. We use this pipeline to simulate the large input sequences obtained in RAG workflows, justifying our setting.

Algorithm 1 `qkv_cake`: Overlapped QKV Computation and Cache Loading

```
procedure QKV_CAKE
    Initialize CUDA streams and events
    Divide sequence into N blocks
    compute_idx ← 0, load_idx ← N − 1
    while compute_idx ≤ load_idx do
        if compute stream is ready then
            if compute_idx = 0 then
                Compute full Q from X using linear_kernel
            Compute K, V for block compute_idx using linear_kernel
            Record compute event
            compute_idx ← compute_idx + 1
        if copy stream is ready then
            Copy K, V block load_idx from host to device using cudaMemcpyAsync
            Record copy event
            load_idx ← load_idx − 1
    Synchronize both streams
    Destroy events
```

3.3 Evaluation Strategy

We evaluate our proposed closed-form solution, integer linear program from KVPR and the two-pointer approach from Cake, by measuring the time required to compute and load the Q, K, V matrices in parallel. Additionally, we use NVIDIA Nsight Systems to profile kernel activities and verify the compute/load parallelism.

4 RESULTS

We evaluate our approach on c240g5 nodes obtained for free from CloudLab[2]. They have the following notable features with respect to our experiments:

- NVIDIA Tesla P100 GPU, 12 GB memory
- PCIe 3.0 connectivity
- 192 GB ECC DDR4-2666 RAM

Our overall result is in the following plot.

Here, the dotted red line $y = 1$ represents the latency of a pure-cache approach at different sequence lengths. All improvements are therefore below the dotted red line. There are many interesting observations to make. First, the split point determined by the ILP from KVPR is sub-optimal. This is not merely because of implementation details, but rather because of the overhead introduced by computing Q. We require the computation of Q in addition to the acquistion of KV, as Q and K are needed in the subsequent step of self-attention. We profile the experiment on NVIDIA Nsight systems and include some descriptive insights in figures (cite figures). We can clearly see that the overwhelming runtime bottleneck is one linear kernel, which represents the computation of Q. As long as computing Q takes longer from the KV cache, it doesn't seem to make much sense to partially recompute. If the requirement to compute Q were removed, significantly "better" results would be observed. However, we believe that computing Q is an important

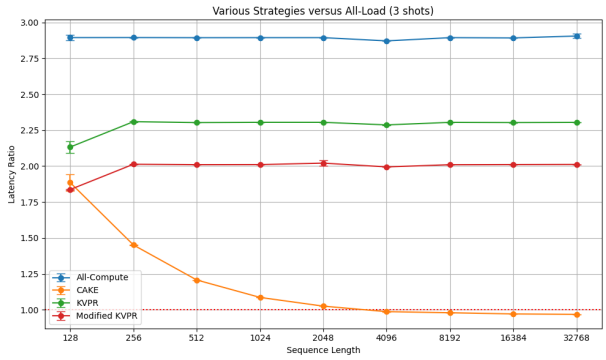

Figure 3: Time ratio of all compute methods over full QKV load across varying sequence lengths. Error bars are small and indicate the standard deviation over three experimental runs.

prerequisite to following steps in self-attention, so we include it in our experiments and benchmarking.

Furthermore, our improvement upon Cake by considering Q does decrease latency. However, the improvement is still not significant enough to beat a cache-only approach, and so our ultimate result is that a pure-cache approach is preferred over these methods based on this experiment.

On the other hand, Cake is actually quite performant for CPU caching. As the sequence length increases, Cake starts to perform slightly better than a pure-cache approach. Cake is attractive because there are no hardware configuration parameters required by the algorithm. The simple two-pointer approach is extremely portable, not requiring re-specification when porting between different hardware.

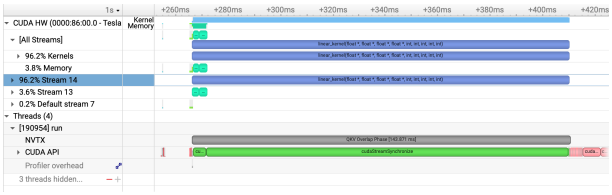

Figure 4: NVIDIA Nsight Systems result. Sequence Length = 500, Partial Recomputation = 0 tokens.

Finally, we also take a closer inspection of the analytical approximations of the ILP and modified ILP by KVPR. Even though we did not see a performance improvement over a cache-only strategy using these approaches, it is still interesting to see that we could provide a closed-form faithful approximation and avoid the NP-hard problem of integer linear programming. The following two plots show the percent error of our approximation compared to the ILP's results for different sequence lengths. We see virtually no difference in the partition choice, and this is after the empirically estimated constant percent shift appended to the analytical formula described in Equation 2.

In conclusion, it appears that the latency of retrieving from CPU memory is not significant enough to warrant partial recomputation

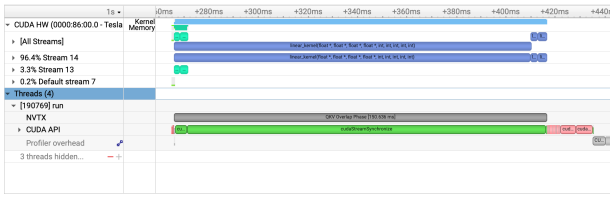

Figure 5: NVIDIA Nsight Systems result. Sequence Length = 500, Partial Recomputation = 10 tokens. This is less performant than the iteration with 0 tokens for partial recomputation, even though asynchronous memory copy and GPU matrix multiplication are correctly simultaneous.

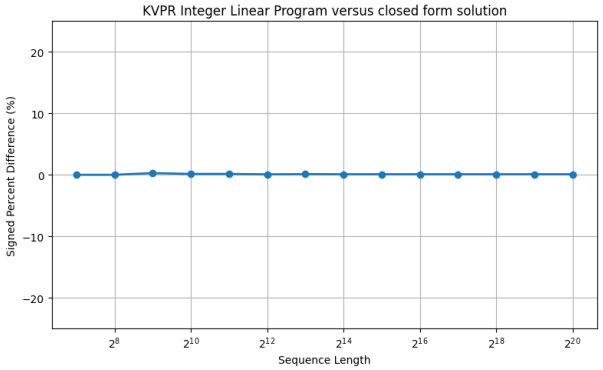

Figure 6: Error of our closed-form solution compared to the integer linear program provided by KVPR.

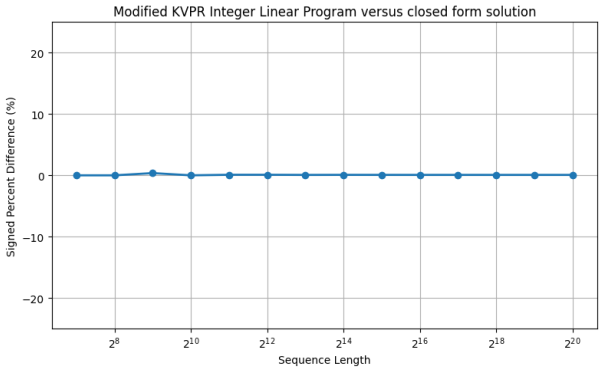

Figure 7: Error of our modified closed-form solution compared to our modification of the integer linear program from KVPR.

with these strategies. Notably, this experiment emphasizes the requirement of computing Q, whereas previous works seem to ignore computing Q. In doing so, we see drops in performance. We still provide closed form solutions for integer linear programs to reduce theoretical complexity of such an approach, as well as demonstrate that the Cake algorithm is advantageous for large enough sequence lengths, while also being simple and extremely portable.

REFERENCES

[1] Reza Abbasi and Sernam Lim. 2024. Superpipeline: A Universal Approach for Reducing GPU Memory Usage in Large Models. arXiv:2410.08791 [cs.LG] https://arxiv.org/abs/2410.08791

[2] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[3] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Haofen Wang, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* 2 (2023).

[4] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems* 6 (2024), 325–338.

[5] Mandy Guo, Zihang Dai, Denny Vrandecic, and Rami Al-Rfou. 2020. Wiki-40B: Multilingual Language Model Dataset. In *LREC 2020*.

[6] Chaoyi Jiang, Lei Gao, Hossein Entezari Zarch, and Murali Annavaram. 2024. Efficient LLM Inference with I/O-Aware Partial KV Cache Recomputation. arXiv:2411.17089 [cs.LG] https://arxiv.org/abs/2411.17089

[7] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. 2024. Ragcache: Efficient knowledge caching for retrieval-augmented generation. *arXiv preprint arXiv:2404.12457* (2024).

[8] Shuowei Jin, Xueshen Liu, Qingzhao Zhang, and Z Morley Mao. 2024. Compute or load kv cache? why not both? *arXiv preprint arXiv:2410.03065* (2024).

[9] 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 29th Symposium on Operating Systems Principles*. 611–626.

[10] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. arXiv:2406.19707 [cs.LG] https://arxiv.org/abs/2406.19707

[11] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. arXiv:2406.19707 [cs.LG] https://arxiv.org/abs/2406.19707

[12] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[13] Cheng Luo, Zefan Cai, Hanshi Sun, Jinqi Xiao, Bo Yuan, Wen Xiao, Junjie Hu, Jiawei Zhao, Beidi Chen, and Anima Anandkumar. 2025. HeadInfer: Memory-Efficient LLM Inference by Head-wise Offloading. arXiv:2502.12574 [cs.LG] https://arxiv.org/abs/2502.12574

[14] Cheng Luo, Zefan Cai, Hanshi Sun, Jinqi Xiao, Bo Yuan, Wen Xiao, Junjie Hu, Jiawei Zhao, Beidi Chen, and Anima Anandkumar. 2025. HeadInfer: Memory-Efficient LLM Inference by Head-wise Offloading. arXiv:2502.12574 [cs.LG] https://arxiv.org/abs/2502.12574

[15] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: high-throughput generative inference of large language models with a single GPU. In *Proceedings of the 40th International Conference on Machine Learning (ICML'23)*.

[16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL] https://arxiv.org/abs/1706.03762

[17] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2024. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. *arXiv preprint arXiv:2405.16444* (2024).