CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion

Jiayi Yao
University of Chicago/CUHK Shenzhen

Hanchen Li
University of Chicago

Yuhan Liu
University of Chicago

Siddhant Ray
University of Chicago

Yihua Cheng
University of Chicago

Qizheng Zhang
Stanford University

Kuntai Du
University of Chicago

Shan Lu
Microsoft Research / University of Chicago

Junchen Jiang
University of Chicago

Presentation: Shouwei Gao Date: June/16/2025

1. Introduction

- Large language models can provide higher quality answers with more relevant information. A typical example is RAG.
- In RAG, a user query will be prepended by multiple **text chunks** from the knowledge base to form the LLM input.
- The context text chunks will significantly slow down the LLM **prefill**, specifically **the time to first token(TTFT)**.
- Speed up the prefill of LLM: re-use the stored KV caches.

1. Introduction

Limitations of existing solutions:

1. Prefix caching: only stores and reuses the KV cache of the prefix of the LLM input.
2. Full KV reuse: When a reused text is not at the input prefix, it still reuses the KV cache by adjusting its positional embedding so that the LLM generation will produce meaningful output. However, this method approximates the cross-attention between the reuse

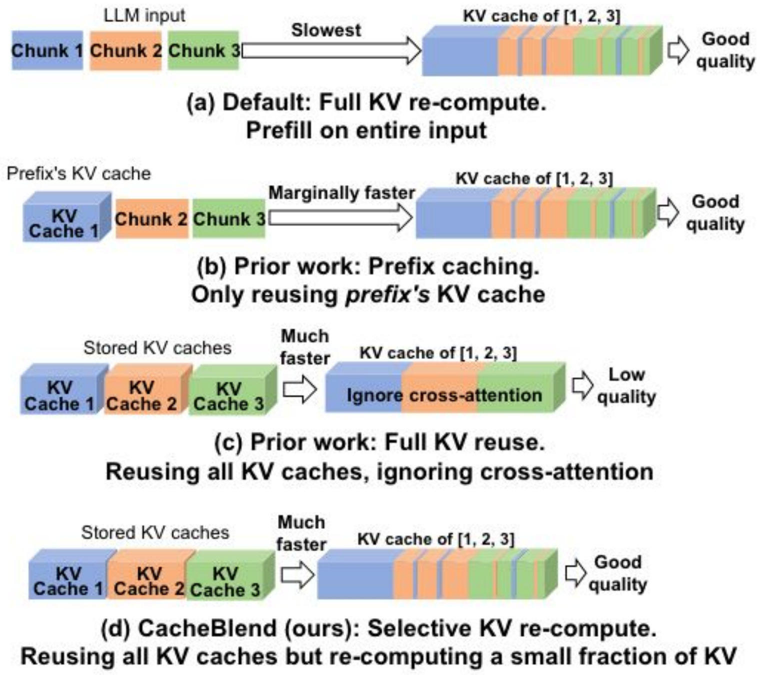


Figure 1. *Contrasting full KV recompute, prefix caching, full KV reuse, and CacheBlend's selective KV recompute.*

2. Motivation

1. Why is prefix caching insufficient?

prefix caching can only save the prefill of the first text chunk, so the saving will be marginal when the LLM input includes more text chunks, even if they are reused.

2. Why is full KV reuse insufficient?

The absence of cross-attention in full KV reuse causes significant discrepancies in the forward attention matrix, which contains the attention between context tokens and the last few tokens, and directly affects the generated tokens.

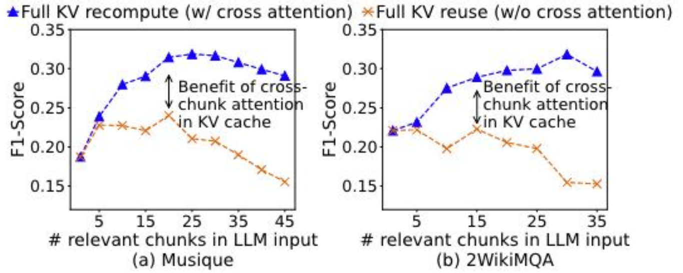
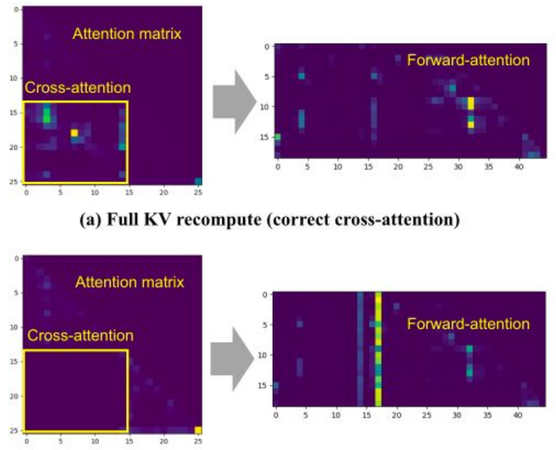


Figure 2. *Generation quality improves as more text chunks are retrieved.*



(a) Full KV recompute (correct cross-attention)

(b) Full KV reuse (ignoring cross-attention)

Figure 4. *Contrasting the attention matrices of (a) full KV recompute and (b) full KV reuse. The yellow boxes highlight the cross-attention. The right-hand side plots show the resulting forward attention matrices whose discrepancies are a result of the different cross-attention between the two methods.*

2. Motivation

- Fast KV Cache Fusing: When an LLM input includes multiple re-used text chunks, how to quickly update the pre-computed KV cache, such that the forward attention matrix (and subsequently the output text) has minimum difference with the one produced by full KV recompute.
- KV deviation: We define the KV deviation of a KV cache KV on layer i of token j as the absolute difference between two tokens. It measures how much different the given KV is on a particular token and layer compared to the full-prefilled KV cache.
- Attention deviation: Similarly, for the forward attention matrix A_i on layer i, we define the attention deviation to be the L-2 norm of its difference with full compute.

2. Motivation

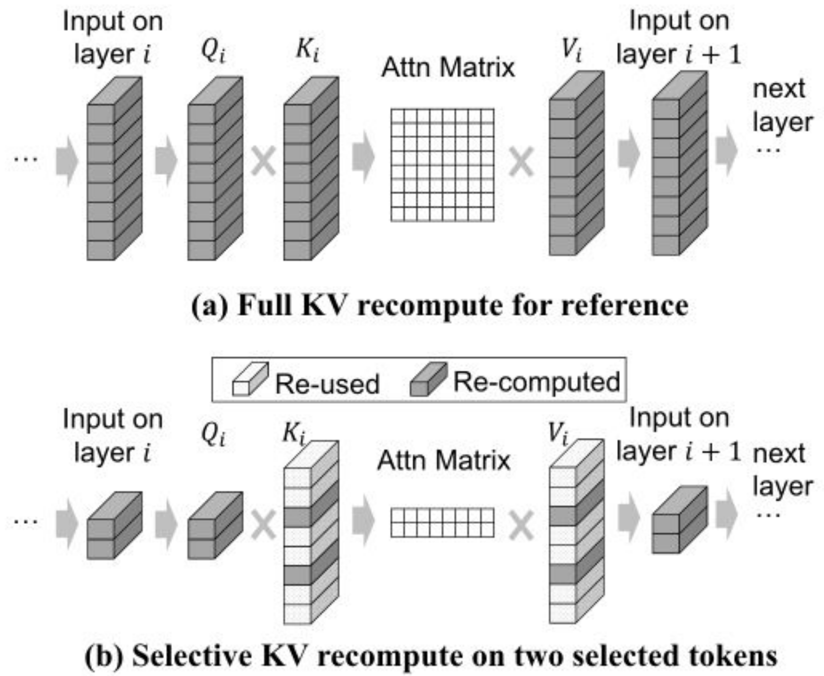


Figure 5. *Illustrated contrast between (a) full KV recompute and (b) selective KV recompute on one layer.*

- It first applies a mask on the input of each layer i to reduce it to a subset of selected tokens.
- It then transforms the reduced input into the Qi , Ki and Vi vectors will also be restricted to the selected tokens.
- It then expands the Ki vector and Vi vector by reusing the KV cache entries associated with the un-selected tokens on layer i, so that the attention matrix includes attention between selected tokens and all other tokens.
- Finally, it runs the same attention module to produce the input of the next layer.

2. Motivation

Insight 1. *On layer i, recomputing the KV of token j who has a higher KV deviation (i.e., $\Delta_{kv}(KV_i, KV_i^{full})[j]$) reduces the attention deviation (i.e., $\Delta_{attn}(A_i, A_i^{full})$) by a greater amount.*

Thus, if we recompute the KV of, say 10%, of tokens on a layer i, we should choose the 10% of tokens which have the highest KV deviations.[3] We refer to these tokens as the **High-KV-Deviation** (or **HKVD**) tokens on layer i.

Now that we know we should recompute KV for the HKVD tokens, two natural questions arise.

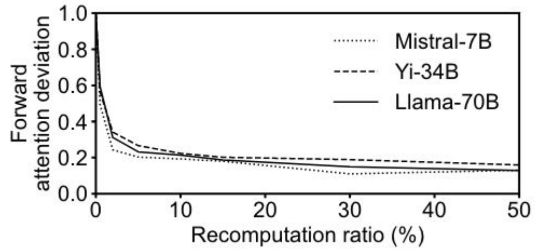


Figure 6. *Attention deviation reduces as we recompute the KV of more tokens on each layer. Importantly, the biggest drop in attention deviation results from recomputing the KV of the tokens with the highest KV deviation (i.e., HKVD tokens).*

Do we need to recompute KV for most tokens? (10-20% tokens as HKVD)

How to identify the HKVD tokens without knowing the true KV values or attention matrix? (HKVD tokens on different layers are not independent)

2. Motivation

Insight 2. Tokens with the highest KV deviations on one layer are likely to have the highest KV deviations on the next layer.

Perform prefill on the first layer first, pick the HKVD tokens of the first layer, and only update their KV on all other layers.

That said, using only the attention deviation of different tokens on the first layer may not be statistically reliable to pick HKVD tokens of all layers, especially deeper layers.

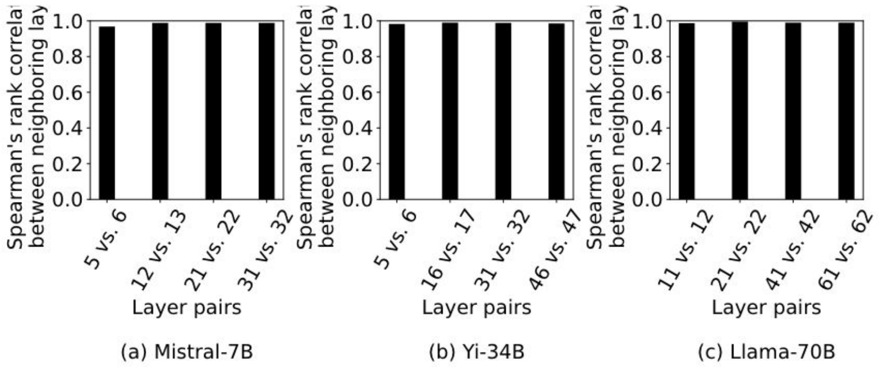


Figure 8. *Rank correlation of the KV deviation per token between two consecutive layers.*

2. Motivation

Thus, we opt for a *gradual filtering* scheme (depicted in Figure 9). If on average we want to pick $r\%$ HKVD tokens per layer, we will pick $r_1\%$ tokens based on the token-wise attention deviation on the first layer, with r_1 being slightly higher than r, and use them as the HKVD tokens on the second layer. Then we recompute the KV of these $r_1\%$ HKVD tokens on the second layer and pick $r_2\%$ tokens that have the highest token-wise attention deviation, with r_2 slightly less than r_1, as the HKVD tokens on the next layer, and so forth. Intuitively, this gradual-filtering scheme eventually

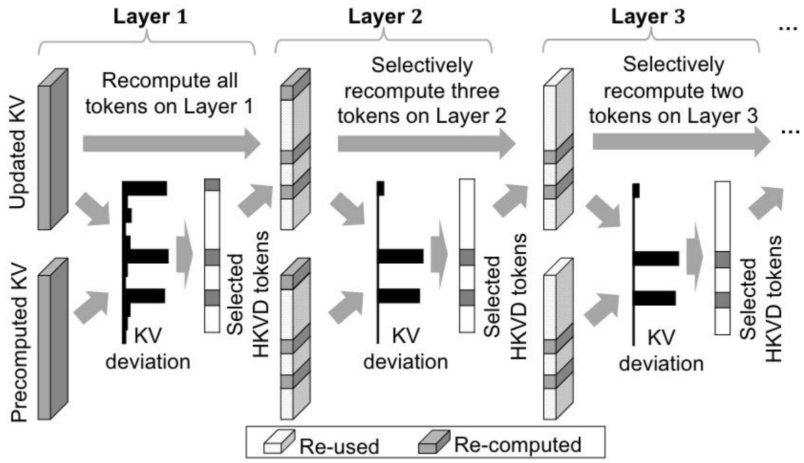


Figure 9. *CacheBlend selects the HKVD (high KV deviation) tokens of one layer by computing KV deviation of only the HKVD tokens selected from the previous layer and selecting the tokens among them with high KV deviation.*

3. CacheBlend System Design

Basic insight: If the delay for selective KV recompute (§4.3) is faster than the loading of KV into GPU memory, then properly pipelining the selective KV recompute and KV loading makes the extra delay of KV recompute negligible.

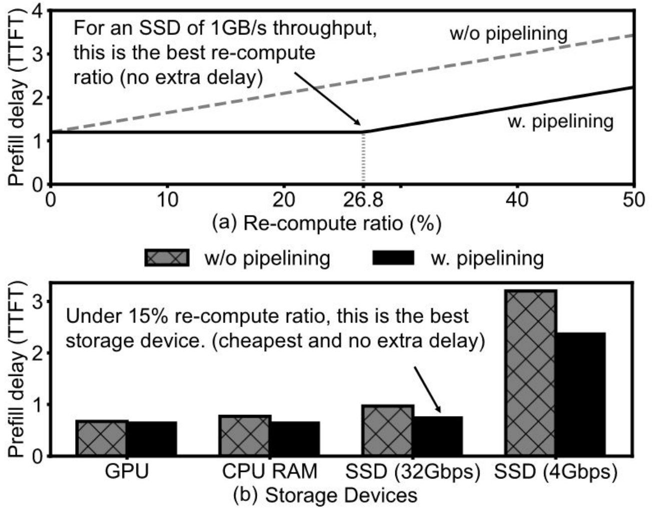


Figure 10. *(a) Smartly picking the recompute ratio will not incur an extra delay. (b) Smartly picking storage device(s) to store KVs saves cost while not increasing delay.*

Take the Llama-7B model and a 4K-long context, recomputing 15% of the tokens (the default recompute ratio) only takes 3 ms per layer, while loading one layer's KV cache takes 16 ms from an NVME SSD (§7). In this case, KV loading can hide the delay for KV recompute on 15% of the tokens, *i.e.*, KV recompute incurs no extra delay. Recomputing more tokens, which can slightly improve generation quality, may not incur extra delay either, as long as the delay is below 16 ms. On the contrary, with another model, Llama-70B, recomputing 15% of tokens takes 7 ms, but it only takes 4 ms to load one layer's KV from an NVME SSD. Here KV loading does not completely hide the recompute delay. In short, a controller is needed to intelligently pick the recompute ratio as well as where to store the KV cache (if applicable).

3. CacheBlend System Design

Basic insight: If the delay for selective KV recompute is faster than the loading of KV into GPU memory, then properly pipelining the selective KV recompute and KV loading makes the extra delay of KV recompute negligible.

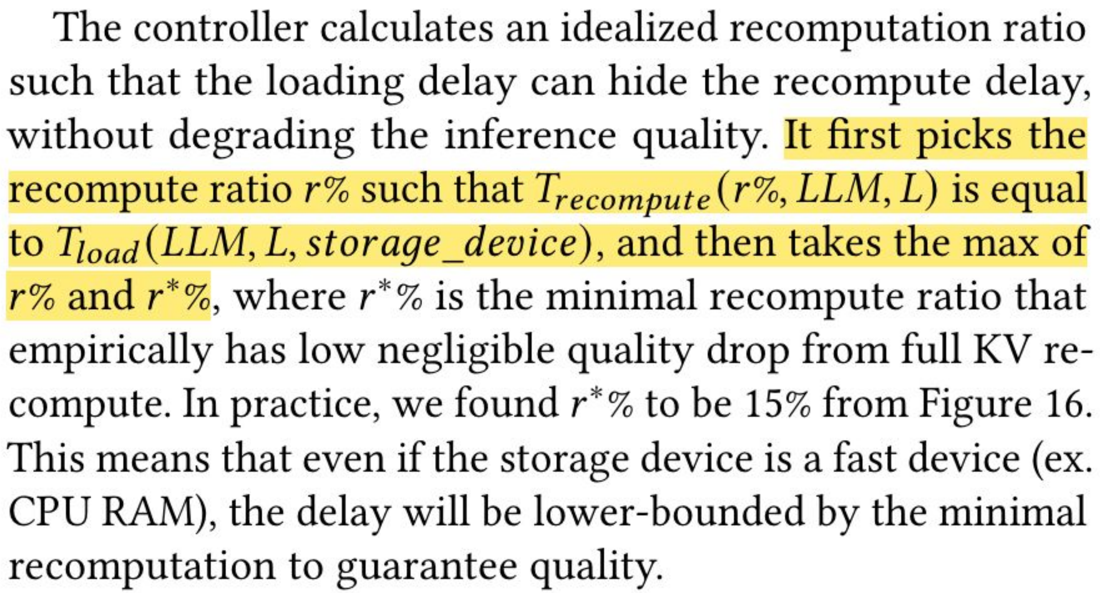
Three major components:

- Loading Controller
- KV cache store(mapping LLM input to KV caches)
- Fusor

3. CacheBlend System Design

Loading Controller:

- Recompute delay estimator
- Loading delay estimator
- Storage cost estimator



The controller calculates an idealized recomputation ratio such that the loading delay can hide the recompute delay, without degrading the inference quality. It first picks the recompute ratio $r\%$ such that $T_{recompute}(r\%, LLM, L)$ is equal to $T_{load}(LLM, L, storage\_device)$, and then takes the max of $r\%$ and $r^*\%$, where $r^*\%$ is the minimal recompute ratio that empirically has low negligible quality drop from full KV re-compute. In practice, we found $r^*\%$ to be 15% from Figure 16. This means that even if the storage device is a fast device (ex. CPU RAM), the delay will be lower-bounded by the minimal recomputation to guarantee quality.

3. CacheBlend System Design

KV cache store: The KV cache store splits an LLM input into multiple text chunks, each of which can be reused or new.

Fusor: The cache fusor (§4) merges pre-computed KV caches via selective recompute.

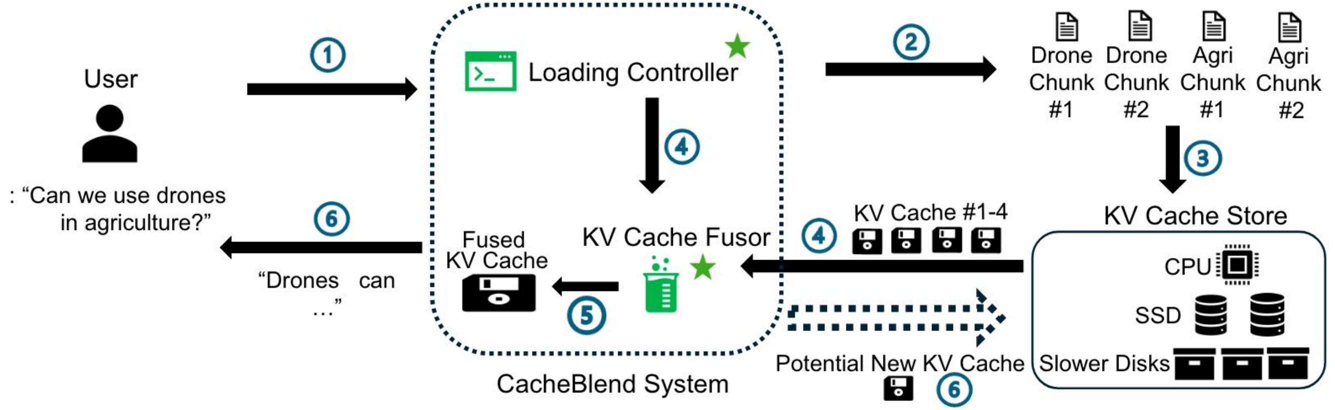


Figure 11. *CacheBlend system (green stared) in light of LLM context augmented generation for a single request. CacheBlend uses text provided by the retriever, interacts with the storage device(s), and provides KV cache on top of LLM inference engines.*

4. Implementaion

- CacheBlend is implemented on top of vLLM using 3K lines of Python code.
- Fusor integration has three interfaces: *fetch_kv*, *prefill_layer*, and *synchronize*.
- *fetch_kv* loads the KV cache to GPU, returning -1 if not in the system.
- *prefill_layer* performs layer-wise partial prefill using input data and KV cache.
- CacheBlend manages KV caches, storing or loading them efficiently to GPU/CPU.

5. Evaluation

Our key takeaways from the evaluation are:

• TTFT reduction: Compared to full KV recompute, CacheBlend reduces TTFT by 2.2-3.3× over several models and tasks.

• High quality: Compared with full KV reuse, CacheBlend improves quality from 0.15 to 0.35 in F1-score and Rouge-L score, while having no more than 0.01-0.03 quality drop compared to full KV recompute and prefix caching.

• Higher throughput: At the same TTFT, CacheBlend can increase throughput by up to 5× compared with full KV recompute and 3.3× compared with prefix caching.

5. Evaluation

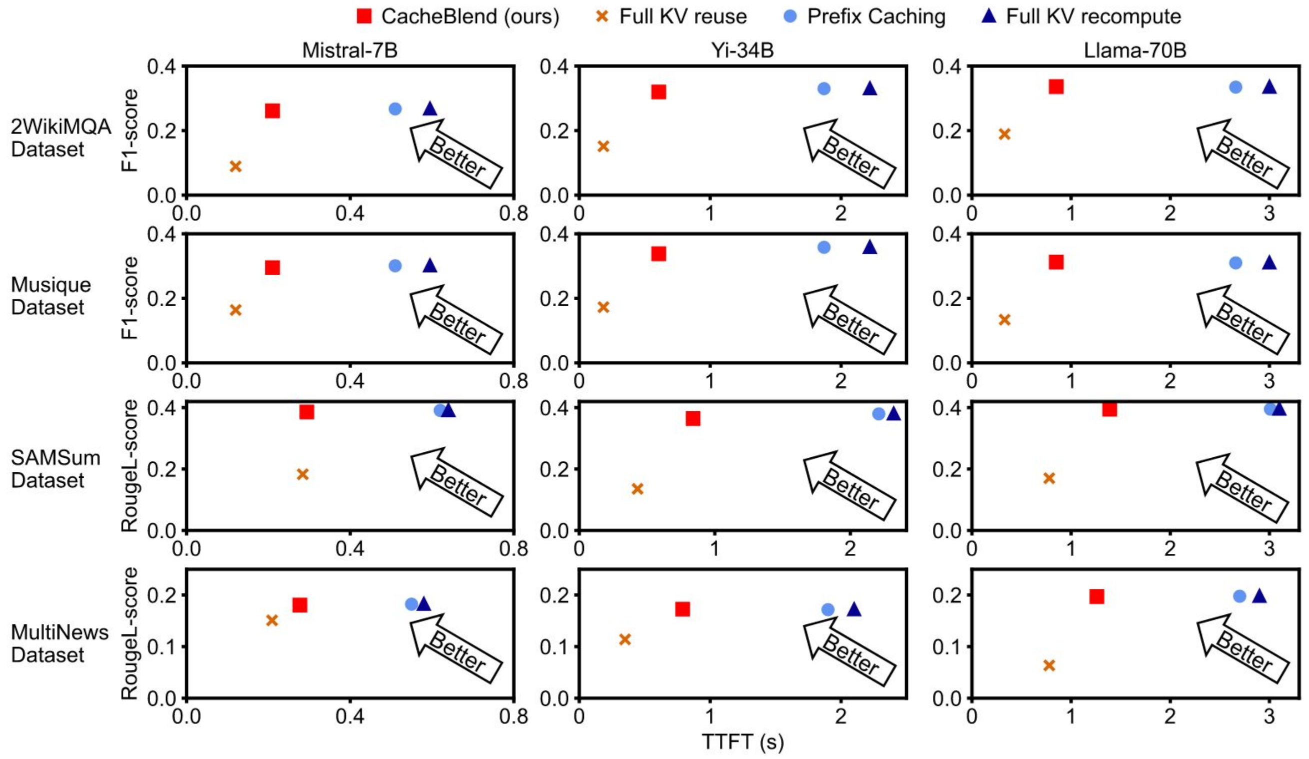


Figure 12. *CacheBlend reduces TTFT by 2.2-3.3× compared to full KV recompute with negligible quality drop across four datasets and three models.*

5. Evaluation

- **MapReduce** summarizes text chunks in parallel, then feeds them to the LLM for the final answer.
- **MapRerank** generates answers from each chunk and selects the one with the highest confidence score.

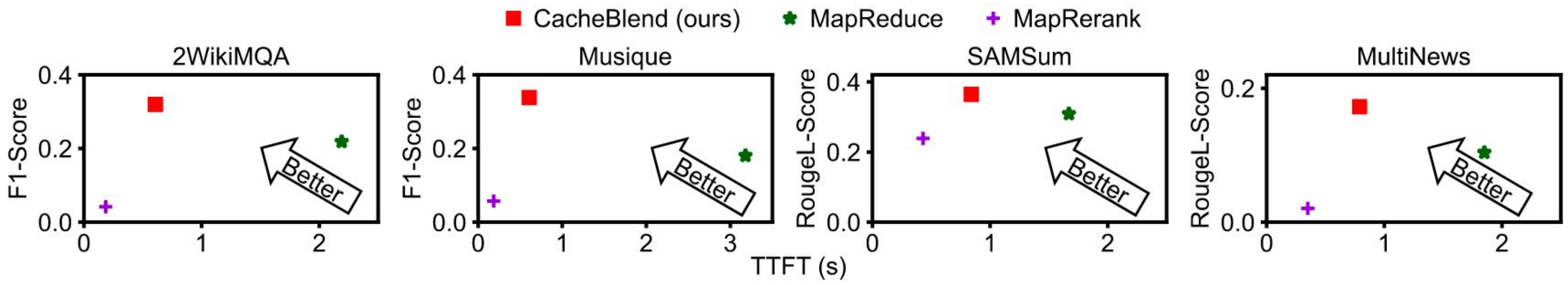


Figure 13. *Generation quality of* CacheBlend *with Yi-34B vs MapReduce and MapRerank.*

5. Evaluation

- **CacheBlend achieves lower delay with higher throughput by 2.8-5× than all the baselines across different models and datasets.**

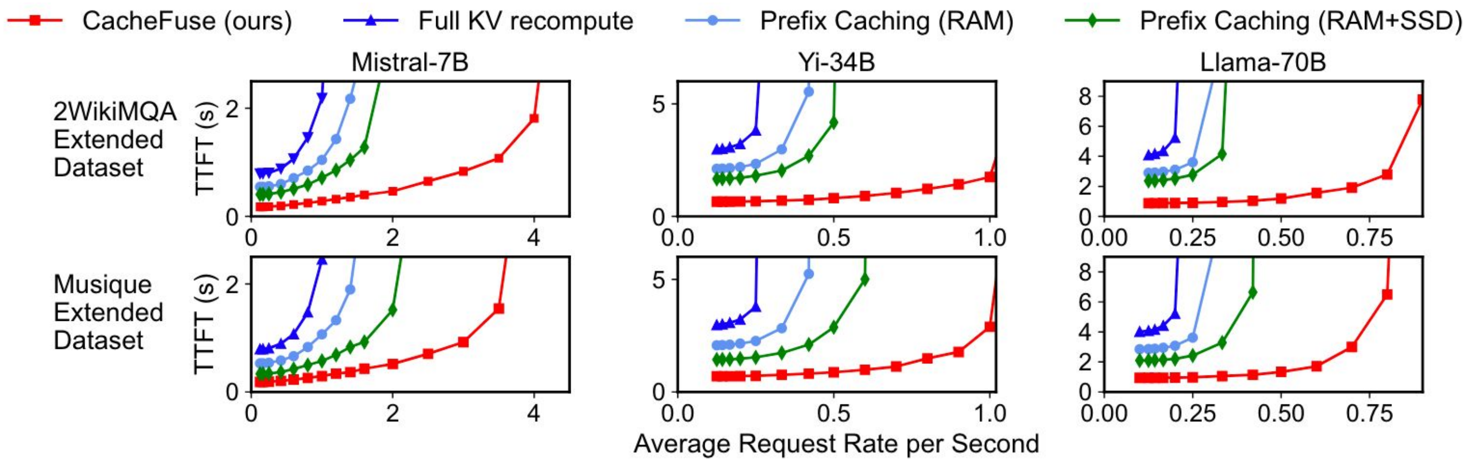


Figure 14. *CacheBlend achieves lower TTFT with higher throughput in RAG scenarios compared with baselines of similar quality.*

5. Evaluation

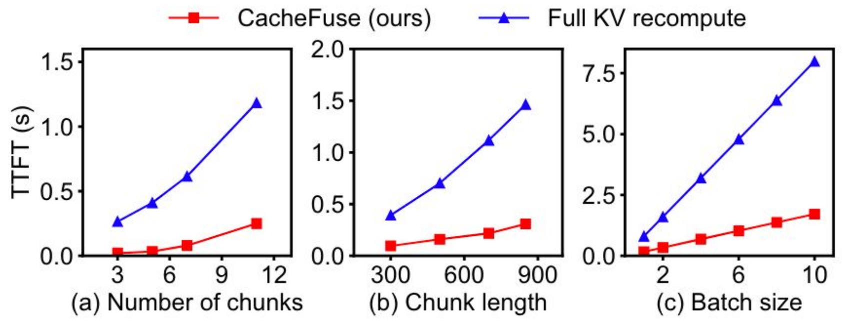


Figure 15. *CacheBlend outperforms baseline with varying chunk numbers, chunk lengths, and batch sizes.*

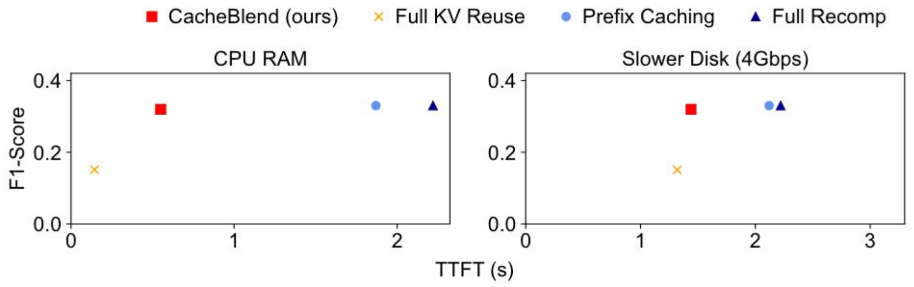


Figure 17. *CacheBlend's outperforms baselines when using RAM and slower disks*

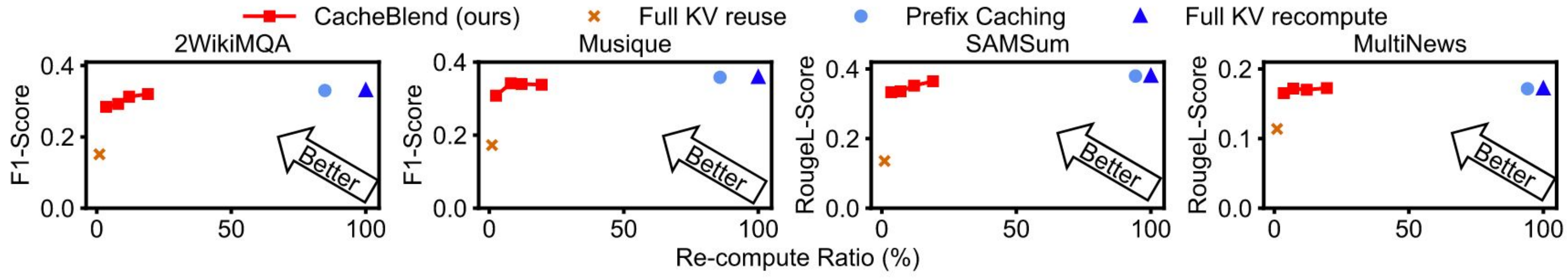


Figure 16. *CacheBlend has minimal loss in quality compared with full KV recompute, with 5%–18% selective recompute ratio, with Yi-34B.*

6. Limitations and Future Work

- CacheBlend currently applies only to transformer-based language models.
- Performance on more models, datasets, and quantization settings needs further testing.
- Integration with newer serving engines like Distserve and StableGen is a future direction.
- Applying CacheBlend to workloads with shared KV cache is also future work.
- The work has been open-sourced to facilitate more development in this area.