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

Abstract

Large language models (LLMs) often incorporate multiple text chunks in their inputs to provide the necessary contexts. To speed up the prefill of the long LLM inputs, one can *pre*-compute the KV cache of a text and *re*-use the KV cache when the context is reused as the prefix of another LLM input. However, the reused text chunks are *not* always the input prefix, which makes precomputed KV caches not directly usable since they ignore the text's *cross-attention* with the preceding texts. Thus, the benefits of reusing KV caches remain largely unrealized.

This paper tackles just one challenge: when an LLM input contains *multiple* text chunks, *how to quickly combine their precomputed KV caches* in order to achieve the same generation quality as the expensive full prefill (*i.e.,* without reusing KV cache)? This challenge naturally arises in retrieval-augmented generation (RAG) where the input is supplemented with multiple retrieved texts as the context. We present CacheBlend, a scheme that reuses the precomputed KV caches, regardless prefix or not, and *selectively recomputes the KV values of a small subset of tokens* to partially update each reused KV cache. In the meantime, the small extra delay for recomputing some tokens can be pipelined with the retrieval of KV caches within the same job, allowing CacheBlend to store KV caches in slower devices with more storage capacity while retrieving them *without* increasing the inference delay. By comparing CacheBlend with the state-of-the-art KV cache reusing schemes on three open-source LLMs of various sizes and four popular benchmark datasets of different tasks, we show that CacheBlend reduces time-to-first-token (TTFT) by 2.2–3.3× and increases the inference throughput by 2.8-5× from full KV recompute without compromising generation quality. The code is available at **https://github.com/LMCache/LMCache**.

CCS Concepts • **Computing methodologies** → **Natural language processing**; • **Networks** → **Cloud computing**; • **Information systems** → *Data management systems.*

Keywords Large Language Models, KV Cache, Retrieval-Augmented-Generation

ACM Reference Format:
Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3689031.3696098

1 Introduction

For their remarkable capabilities, large language models (LLMs) are widely used in personal assistance, AI healthcare, and question answering [1, 3, 4, 9]. To ensure high-quality and consistent responses, applications often supplement the user query with additional texts to provide the necessary *context* of domain knowledge or user-specific information. A typical example is Retrieval-Augmented Generation (RAG) where a user query will be prepended by multiple *text chunks* retrieved from a database to form the LLM input.

These context text chunks, however, significantly slow down LLM inference. This is because, before generating any token, an LLM first uses *prefill* to go through the entire LLM input to produce the *KV cache*—concatenation of tensors associated with each input token that embeds the token's "attention" with its preceding tokens. Thus, the prefill delay determines the time to first token (TTFT). We refer to it as

EuroSys '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/25/03…$15.00
https://doi.org/10.1145/3689031.3696098

CacheBlend：用于带缓存知识融合的快速大型语言模型服务的RAG

贾怡 耀 芝加哥大学/香港中文大学（深圳）
李汉辰 芝加哥大学
刘宇涵 芝加哥大学
西达汉特·雷 芝加哥大学
伊华城 芝加哥大学
张启正 斯坦福大学
昆泰杜 芝加哥大学
山璐 微软研究院 / 芝加哥大学
江俊辰 芝加哥大学

摘要

大型语言模型（LLMs）通常在其输入中包含多个文本块，以提供必要的上下文。为了加快长LLM输入的预填充，可以预先计算文本的KV缓存，并在上下文作为另一个LLM输入的前缀时重用KV缓存。然而，重用的文本块并不总是输入前缀，这使得预计算的KV缓存无法直接使用，因为它们忽略了文本与前面文本的交叉注意力。因此，重用KV缓存的好处在很大程度上仍未实现。

本文解决了一个挑战：当LLM输入包含多个文本块时，如何快速结合它们的预计算KV缓存，以实现与昂贵的完整预填充相同的生成质量（即，不重用KV缓存）？这个挑战自然出现在检索增强生成（RAG）中，其中输入通过多个检索文本作为上下文进行补充。我们提出了CacheBlend，一种重用预计算KV缓存的方案，无论是否有前缀，并选择性地重新计算小部分标记的KV值，以部分更新每个重用的KV缓存。与此同时，重新计算某些标记的小额外延迟可以与同一作业中的KV缓存检索进行流水线处理，使CacheBlend能够在存储容量更大的较慢设备中存储KV缓存，同时在不增加推理延迟的情况下检索它们。通过比较CacheBlend

通过在三种不同规模的开源 LLM 和四个不同任务的流行基准数据集上使用最先进的 KV 缓存重用方案，我们展示了 CacheBlend 将首次令牌时间 (TTFT) 减少了 2.2–3.3×，并将推理吞吐量提高了 2.8-5×，而不影响生成质量。代码可在 https://github.com/LMCache/LMCache 获取。

CCS 概念 • 计算方法 → 自然语言处理; • 网络 → 云计算; • 信息系统 → 数据管理系统。

关键词 大型语言模型，KV缓存，检索增强生成

ACM 参考格式:
贾怡尧，李汉辰，刘宇涵，西丹特·雷，程怡华，张启正，杜坤泰，卢珊，姜俊辰。2025年。《CacheBlend：用于RAG的快速大型语言模型服务与缓存知识融合》。在第二十届欧洲计算机系统会议（EuroSys '25），2025年3月30日至4月3日，荷兰鹿特丹。ACM，纽约，NY，美国，16页。https://doi.org/10.1145/3689031.3696098

1 引言

由于其卓越的能力，大型语言模型（LLMs）在个人助手、人工智能医疗和问答系统中得到了广泛应用 [1, 3, 4, 9]。为了确保高质量和一致的响应，应用程序通常会通过附加文本来补充用户查询，以提供必要的领域知识或用户特定信息的上下文。一个典型的例子是检索增强生成（RAG），在这种情况下，用户查询将由从数据库中检索的多个文本块预先添加，以形成 LLM 输入。

然而，这些上下文文本块显著减慢了LLM推理。这是因为，在生成任何标记之前，LLM首先使用预填充遍历整个LLM输入，以生成KV缓存——与每个输入标记相关联的张量的连接，嵌入了标记与其前面标记的“注意力”。因此，预填充延迟决定了第一个标记的时间（TTFT）。我们称之为

EuroSys '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/25/03…$15.00
https://doi.org/10.1145/3689031.3696098

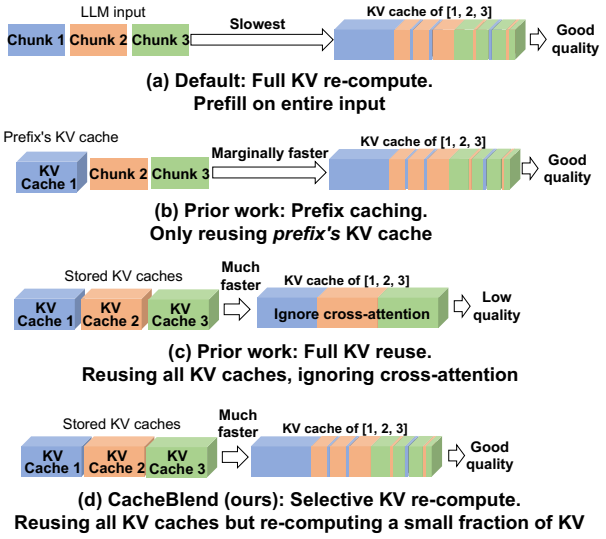


Figure 1. *Contrasting full KV recompute, prefix caching, full KV reuse, and CacheBlend's selective KV recompute.*

full KV recompute (Figure 1(a)). Despite many optimizations, the delay and computation of prefill grow super-linearly with the input length, and can easily slow down the service, especially on long LLM inputs (*e.g.,* in RAG) [11, 53, 60].

So, how do we speed up the prefill of LLM inputs? Recent optimizations embrace the fact that same context texts are often reused by different LLM inputs. They then *pre*-compute the KV caches of these texts *once* and *re*-use the stored KV caches to avoid repeated prefill on these reused texts.

Limitations of existing methods: There are currently two approaches to KV cache reusing, but they both have limitations.

First, *prefix caching* only stores and reuses the KV cache of the prefix of the LLM input [33, 36, 41, 42, 59] (Figure 1(b)). Because the prefix's KV cache is independent of the succeeding texts, prefix caching does not hurt generation quality. However, many applications, such as RAG, include *multiple* text chunks, rather than one, in the LLM input to provide all necessary contexts to ensure good response quality. Thus, only the first text chunk is the prefix, and other reused texts' KV caches are not reused. As a result, the speed of prefix caching will be almost as slow as full KV recompute when input consists of many reused text chunks.

Second, *full KV reuse* aims to address this shortcoming (Figure 1(c)). 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 [24]. However, this method ignores the important *cross-attention*—the attention between tokens in one chunk with tokens in its preceding chunks. The cross-attention information cannot be pre-computed as the preceding chunks are not known in advance. Yet, cross-attention can be vital to answer queries (*e.g.,* about geopolitics) that naturally require understanding information from multiple chunks *jointly* (*e.g.,* chunks about geography and chunks about politics). §3.3 offers concrete examples to illustrate when prefix caching and modular caching are insufficient.

Our approach: This paper tackles only one challenge: when an LLM input includes multiple text chunks, how to *quickly* combine their individually pre-computed KV caches, in order to achieve the same generation *quality* as the expensive full prefill? In other words, we seek to have both the speed of *full KV reuse* and the generation quality of *full KV recompute*.

We present CacheBlend, a system that fuses multiple pre-computed KV caches, regardless of prefix or not, by *selectively recomputing* the KV cache of a small fraction of tokens, based on the preceding texts in the specific LLM input. We refer to it as **selective KV recompute** (Figure 1(d)). At a high level, selective KV recompute performs prefill on the input text in a traditional layer-by-layer fashion; however, in each layer, it updates the KV of only a small fraction of tokens while reusing the KV of other tokens.

Comparing with full KV recompute, an update fraction of less than 15% can typically generate same-quality responses based on our experience. The deeper reason why it suffices to only updating a small fraction of KV is due to the sparsity of attention matrices (see §4.3).

Comparing with full KV reuse, CacheBlend achieves higher generation quality with a small amount of extra KV update. Moreover, this small extra computation does not increase the inference latency, because CacheBlend parallelizes partial KV update on one layer with the fetching of the KV cache on the next layer into GPU memory. Such pipelining enables CacheBlend to store KV caches in slower non-volatile devices (*e.g.,* disk) *without* incurring extra delay, allowing more KV caches to be stored and reused.

To put our contribution in context, CacheBlend enables the reusing of KV caches of multiple text chunks in one LLM input, without compromising generation quality. This is complementary to the recent work that reduces KV cache storage sizes [28, 35, 42, 43, 45, 58] and optimizes the access patterns of KV cache [33, 59].

We implemented CacheBlend on top of vLLM and compared CacheBlend with state-of-the-art KV cache reusing schemes on three open-source LLMs of various sizes and three popular benchmark datasets of two LLM tasks (RAG and QA). We show that compared to prefix caching, CacheBlend reduces time-to-first-token (TTFT) by 2.2–3.3× and increases the inference throughput by 2.8–5×, without compromising generation quality or incurring more storage cost. Compared to full KV reuse, CacheBlend achieves almost the same TTFT but 0.1-0.2 higher absolute F1-scores on QA tasks and 0.03-0.25 higher absolute Rouge-L scores on summarization.

2 Background

Most LLM services today use transformers [13, 16, 52]. After receiving the input tokens, the LLM first uses the prefill

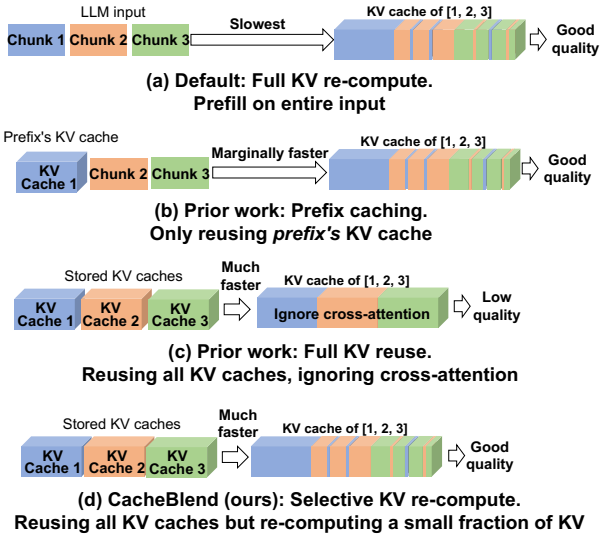


图 1. 对比完整的 KV 重新计算、前缀缓存、完整的 KV 重用和 CacheBlend 的选择性 KV 重新计算。

完整的 KV 重新计算（图 1(a)）。尽管进行了许多优化，预填充的延迟和计算随着输入长度的增加而呈超线性增长，并且很容易减慢服务，特别是在长 LLM 输入（例如，在 RAG 中）时 [11, 53, 60]。

那么，我们如何加快 LLM 输入的预填充呢？最近的优化利用了相同上下文文本通常被不同 LLM 输入重用的事实。它们首先对这些文本的 KV 缓存进行预计算，然后重用存储的 KV 缓存，以避免对这些重用文本的重复预填充。

现有方法的局限性：目前有两种KV缓存重用的方法，但它们都有局限性。

首先，前缀缓存仅存储和重用 LLM 输入的前缀的 KV 缓存 [33, 36, 41, 42, 59]（图 1(b)）。由于前缀的 KV 缓存独立于后续文本，前缀缓存不会影响生成质量。然而，许多应用程序，如 RAG，在 LLM 输入中包含多个文本块，而不是一个，以提供所有必要的上下文以确保良好的响应质量。因此，只有第一个文本块是前缀，其他重用文本的 KV 缓存不会被重用。因此，当输入由许多重用文本块组成时，前缀缓存的速度几乎与完整的 KV 重新计算一样慢。

其次，完整的KV重用旨在解决这一缺陷（图1(c)）。当重用的文本不在输入前缀时，它仍然通过调整其位置嵌入来重用KV缓存，以便LLM生成将产生有意义的输出[24]。然而，这种方法忽略了重要的交叉注意力——一个块中的标记与其前面块中的标记之间的注意力。交叉注意力信息无法预先计算，因为前面的块在事先并不知道。然而，交叉注意力对于回答自然需要的查询（例如，关于地缘政治的查询）可能至关重要。

从多个块共同理解信息（例如，关于地理的块和关于政治的块）。§3.3提供了具体示例，以说明何时前缀缓存和模块缓存不足。

我们的方法：本文仅解决一个挑战：当LLM输入包含多个文本块时，如何快速组合它们各自预计算的KV缓存，以实现与昂贵的完整预填充相同的生成质量？换句话说，我们希望同时拥有完整KV重用的速度和完整KV重新计算的生成质量。

我们介绍了CacheBlend，一个融合多个预计算KV缓存的系统，无论是否有前缀，通过基于特定LLM输入中前面的文本选择性地重新计算一小部分令牌的KV缓存。我们将其称为选择性KV重新计算（图1(d)）。从高层次来看，选择性KV重新计算以传统的逐层方式对输入文本进行预填充；然而，在每一层中，它仅更新一小部分令牌的KV，同时重用其他令牌的KV。

与完整的 KV 重新计算相比，根据我们的经验，更新比例低于 15% 通常可以生成相同质量的响应。仅更新小部分 KV 的更深层原因在于注意力矩阵的稀疏性（见§4.3）。

与完全的 KV 重用相比，CacheBlend 在少量额外的 KV 更新下实现了更高的生成质量。此外，这小部分额外的计算并不会增加推理延迟，因为 CacheBlend 在一层上并行化部分 KV 更新，同时在下一层将 KV 缓存提取到 GPU 内存中。这种流水线使得 CacheBlend 能够将 KV 缓存存储在较慢的非易失性设备（例如，磁盘）中，而不会产生额外的延迟，从而允许存储和重用更多的 KV 缓存。

为了将我们的贡献置于背景中，CacheBlend 使得在一个 LLM 输入中重用多个文本块的 KV 缓存成为可能，而不影响生成质量。这与最近的工作相辅相成，这些工作减少了 KV 缓存的存储大小 [28, 35, 42, 43, 45, 58] 并优化了 KV 缓存的访问模式 [33, 59]。

我们在 vLLM 的基础上实现了 CacheBlend，并将其与三种不同规模的开源 LLM 和两个 LLM 任务（RAG 和 QA）的三个流行基准数据集上的最先进 KV 缓存重用方案进行了比较。我们展示了与前缀缓存相比，CacheBlend 将首次令牌时间（TTFT）减少了 2.2–3.3×，并将推理吞吐量提高了 2.8–5×，而不影响生成质量或增加更多存储成本。与完全 KV 重用相比，CacheBlend 实现了几乎相同的 TTFT，但在 QA 任务上 F1 分数绝对值提高了 0.1-0.2，在摘要任务上 Rouge-L 分数绝对值提高了 0.03-0.25。

2 背景

大多数LLM服务今天使用变换器[13, 16, 52]。在接收到输入标记后，LLM首先使用预填充

phase (explained shortly) to transform the tokens into key (K) and value (V) vectors, *i.e., KV cache*. After prefill, the LLM then iteratively decodes (generates) the next token with the current KV cache and appends the new K and V vectors of the new tokens to the KV cache for the next iteration.

The prefill phase computes the KV cache layer by layer. The input tokens embeddings on each layer are first transformed into query (Q), key (K), and value (V) vectors, of which the K and V vectors form one layer of the KV cache. The LLM then multiplies Q and K vectors to obtain the *attention matrix*—the attention between each token and its preceding tokens—and does another dot product between the (normalized and masked) attention matrix with the V vector. The resulting vector will go through multiple neural layers to obtain the tokens' embeddings on the next layer.

When the KV cache of a prefix is available, the prefill phase will only need to compute the *forward attention* matrix (between the suffix tokens and the prefix tokens) on each layer which directly affects the generated token.

The prefill phase can be slow, especially on long inputs. For instance, on an input of four thousand tokens (a typical context length in RAG [33]), running prefill can take three (or six) seconds for Llama-34B (or Llama-70B) on one A40 GPU. This causes a substantial delay that users have to wait before seeing the first word generated. Recent work also demonstrates that prefill can be a throughput bottleneck, by showing that getting rid of the prefill phase can double the throughput of an LLM inference system [60].

3 Motivation

3.1 Opportunities of reusing KV caches

Recent systems try to alleviate the prefill overhead by leveraging the observation that in many LLM use cases, the same texts are used repeatedly in different LLM inputs. This allows reusing KV caches of these reused texts (explained shortly).

Text reusing is particularly prevalent when same texts are included in the LLM input to provide necessary contexts to ensure high and consistent response quality. To make it more concrete, let's consider two scenarios.

- In a company that uses LLM to manage internal records, two queries can be *"who in the IT department proposed using RAG to enhance the customer service X during the last all-hands meeting?"* and *"who from the IT department were graduates from college Y?"* While seemingly different, both queries involve *the list of employees in the IT department* as a necessary context to generate correct answers.
- Similarly, in an LLM-based application that summarizes Arxiv papers, two queries can be *"what are the trending RAG techniques on Arxiv?"* and *"what datasets are used recently to benchmark RAG-related papers on Arxiv?"* They both need the *recent Arxiv papers about RAG* as the necessary context to generate correct results.

Since the reused contexts typically contain more information than the user queries, the prefill on the "context" part of the input accounts for the bulk of prefill overhead [22, 33]. Thus, it would be ideal to store and reuse the KV caches of reused texts, in order to avoid the prefill overhead when these texts are used again in different LLM inputs.

3.2 Why is prefix caching insufficient?

Indeed, several recent systems are developed to reduce prefill delay by reusing KV caches. For example, in prefix caching, the KV cache of a reusable text chunk is precomputed once, and if the text chunk is at the *prefix* of an LLM input, then the precomputed KV cache can be reused to avoid prefill on the prefix. The advantage of prefix caching is that the KV cache of a prefix is not affected by the succeeding text, so the generation result will be identical to full KV recompute (without the KV cache). Several systems have followed this approach, *e.g.,* vLLM [36], SGLang [59], and RAGCache [33].

The disadvantage of prefix caching is also clear. To answer one query, applications, such as RAG, often prepend ***multiple*** text chunks in the LLM input to provide different contexts necessary for answering the query.[1] As a result, ***except the first chunk, all other chunks' KV caches are not reused since they are not the prefix of the LLM input***.

Let us think about the queries from §3.1. To answer "*who in the IT department proposed using RAG to enhance the customer service X during the last all-hands meeting?*", we need contexts from *multiple* sources, including IT department's employees, information about service X, and meeting notes from the all-hands meeting. Similarly, in the Arxiv-summarization app, answering the example queries will require the LLM to read *several* recent RAG-related Arxiv papers as the contexts. Being on different topics, these contexts are unlikely to appear together in one text chunk. They are separate text chunks used together only when answering a particular query.

To empirically show the needs for including multiple text chunks in LLM inputs, we use two popular multi-hop QA datasets, Musique and 2WikiMQA. These datasets consist of queries and multiple associated context texts needed to answer the queries. Following the common practice of RAG, we first create a vector database by splitting the contexts into chunks of 128 tokens (a popular number [29]) using the text chunking mechanism from Langchain [5]. For each query, we embed the query using SentenceTransformers [49],

[1] Prepending all contexts in an LLM input is a popular way of augmenting user queries (*e.g.,* "stuff" mode in Langchain and LlamaIndex). By default, this paper uses this mode. Other RAG methods like "MapReduce" or "Rerank" do not fit in this category. They first process each context text chunk separately before combining the results from each context. Since each chunk is always the prefix when processed separately, prefix caching works well. However, "MapReduce" is slow since it needs to summarize every chunk before generating the answer from summaries. "Rerank" suffers from low generation quality if multiple chunks contain relevant information as it processes every chunk individually. We also empirically evaluate these methods and compare them with CacheBlend in §7.

阶段（简要说明）将标记转换为键（K）和值（V）向量，即 KV 缓存。在预填充之后，LLM 然后迭代解码（生成）下一个标记，使用当前的 KV 缓存，并将新标记的新 K 和 V 向量附加到 KV 缓存中，以便进行下一次迭代。

预填充阶段逐层计算KV缓存。每层的输入令牌嵌入首先被转换为查询（Q）、键（K）和值（V）向量，其中K和V向量形成KV缓存的一层。然后，LLM将Q和K向量相乘以获得注意力矩阵——每个令牌与其前置令牌之间的注意力——并对（归一化和掩蔽的）注意力矩阵与V向量进行另一个点积。结果向量将经过多个神经层以获得下一层的令牌嵌入。

当前缀的KV缓存可用时，预填充阶段只需在每一层计算前向注意力矩阵（在后缀标记和前缀标记之间），这直接影响生成的标记。

预填充阶段可能很慢，尤其是在长输入的情况下。例如，在四千个标记的输入上（RAG [33] 中的典型上下文长度），在一台 A40 GPU 上运行预填充可能需要三（或六）秒钟，针对 Llama-34B（或 Llama-70B）。这导致用户在看到生成的第一个单词之前必须等待相当长的时间。最近的研究还表明，预填充可能是一个吞吐量瓶颈，表明去掉预填充阶段可以使 LLM 推理系统的吞吐量翻倍 [60]。

3 动机

3.1 重用 KV 缓存的机会

最近的系统试图通过利用观察到的事实来减轻预填充开销，即在许多 LLM 使用案例中，相同的文本在不同的 LLM 输入中被重复使用。这允许重用这些重复文本的 KV 缓存（稍后会解释）。

文本重用在将相同文本包含在LLM输入中以提供必要的上下文以确保高质量和一致的响应时尤为普遍。为了使其更具体，我们考虑两种情况。

- 在一家使用LLM管理内部记录的公司中，两个查询可以是“在上一次全员会议中，IT部门中谁提议使用RAG来增强客户服务X？”和“IT部门中谁是大学Y的毕业生？”虽然看似不同，但这两个查询都涉及IT部门员工的名单，作为生成正确答案的必要背景。

- 类似地，在一个基于LLM的应用中，用于总结Arxiv论文的两个查询可以是“Arxiv上流行的RAG技术是什么？”和“最近使用了哪些数据集来基准测试Arxiv上的RAG相关论文？”它们都需要关于RAG的最新Arxiv论文作为生成正确结果的必要上下文。

由于重用的上下文通常包含比用户查询更多的信息，因此输入的“上下文”部分的预填充占据了大部分的预填充开销 [22, 33]。因此，理想的做法是存储和重用重用文本的 KV 缓存，以避免在这些文本在不同的 LLM 输入中再次使用时产生的预填充开销。

3.2 为什么前缀缓存不足？

确实，最近开发了几个系统，通过重用KV缓存来减少预填充延迟。例如，在前缀缓存中，可重用文本块的KV缓存被预先计算一次，如果文本块位于LLM输入的前缀处，则可以重用预计算的KV缓存以避免在前缀上的预填充。前缀缓存的优点在于，前缀的KV缓存不受后续文本的影响，因此生成结果将与完全的KV重新计算（没有KV缓存）相同。几个系统遵循了这种方法，例如，vLLM [36]、SGLang [59]和RAGCache [33]。

前缀缓存的缺点也很明显。为了回答一个查询，应用程序（如 RAG）通常会在 LLM 输入中预先添加多个文本块，以提供回答查询所需的不同上下文。[1] 结果是，除了第一个块，所有其他块的 KV 缓存都没有被重用，因为它们不是 LLM 输入的前缀。

让我们思考一下§3.1中的查询。要回答“在上一次全员会议中，IT部门的谁提议使用RAG来增强客户服务X？”，我们需要来自多个来源的上下文，包括IT部门的员工、关于服务X的信息，以及全员会议的会议记录。同样，在Arxiv摘要应用中，回答示例查询将需要LLM阅读几篇最近与RAG相关的Arxiv论文作为上下文。由于主题不同，这些上下文不太可能在一个文本块中同时出现。它们是单独的文本块，仅在回答特定查询时一起使用。

为了实证展示在LLM输入中包含多个文本块的需求，我们使用了两个流行的多跳问答数据集，Musique和2WikiMQA。这些数据集由查询和多个与查询相关的上下文文本组成，这些文本是回答查询所需的。遵循RAG的常见做法，我们首先通过使用Langchain的文本分块机制将上下文拆分为128个标记的块（一个流行的数字[29]）来创建一个向量数据库。对于每个查询，我们使用SentenceTransformers [49] 嵌入查询，

[1] Prepending all contexts in an LLM input is a popular way of augmenting user queries (*e.g.,* “stuff” mode in Langchain and LlamaIndex). By default, this paper uses this mode. Other RAG methods like “MapReduce” or “Rerank” do not fit in this category. They first process each context text chunk separately before combining the results from each context. Since each chunk is always the prefix when processed separately, prefix caching works well. However, “MapReduce” is slow since it needs to summarize every chunk before generating the answer from summaries. “Rerank” suffers from low generation quality if multiple chunks contain relevant information as it processes every chunk individually. We also empirically evaluate these methods and compare them with CacheBlend in §7.

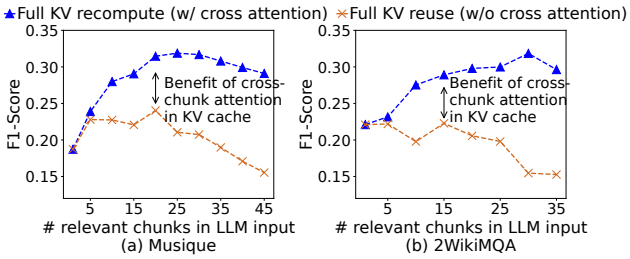


Figure 2. *Generation quality improves as more text chunks are retrieved.*

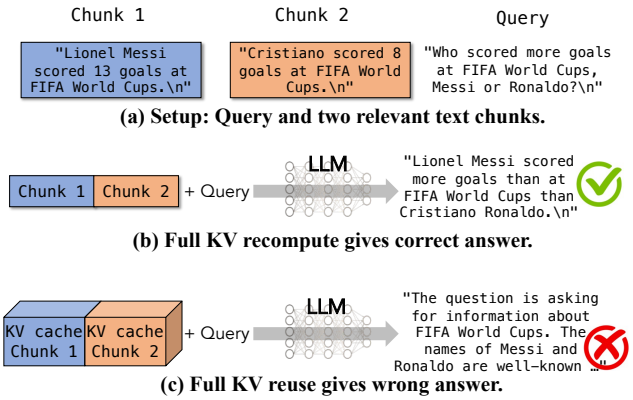


Figure 3. *An illustrative example of an LLM input with two text chunks prepended to a query. Full KV recompute (b), without reusing KV cache, is slow but gives the correct answer. Full KV reuse (c), however, gives the wrong answer as it neglects cross-attention between the chunks (Figure 4).*

and fetch top-k relevant chunks from the database, based on the least L2 distance between the embeddings of the query and the chunk respectively. Figure 2 shows the generation quality, measured using a standard *F1-score* metric, with an increasing number of selected text chunks. We can see that the quality improves significantly as more text chunks are retrieved to supplement the LLM input, though including too many chunks hurts quality due to the well-known lost-in-the-middle issue [40, 54].

In short, *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.

3.3 Why is full KV reuse insufficient?

Full KV reuse is proposed to address this very problem. This approach is recently pioneered by PromptCache [24]. It concatenates independently precomputed KV caches of recurring text chunks with the help of *buffers* to maintain the positional accuracy of each text chunk. For instance, to concatenate the KV caches of chunks C_1 and C_2, PromptCache first needs to precompute the KV cache of C_2 by running prefill on a hypothetical input that prepends C_2 with a dummy prefix of length greater or equal to C_1. This way, even if C_2 is not the prefix, we still correctly preserve the positional information of C_2's KV cache, though each chunk's KV cache will have to be precomputed multiple times.

However, even with the positional information preserved, a more fundamental problem is that the KV cache of non-prefix text chunk (*e.g.*, C_2) ***ignores the cross-attention*** between the chunk and the preceding text (*e.g.*, C_1). This is because the preceding text is not known when precomputing the KV cache.

Ignoring cross-attention can lead to a wrong response. Figure 3 shows an illustrative example, where a user query *"How many goals did Messi score more than Cristiano Ronaldo at FIFA World Cups?"* is prepended by the two text chunks of the players' career statistics. With full prefill or prefix caching, the result is clear and correct. With full KV reuse the KV caches of the two text chunks are precomputed, with each chunk having the right positional embedding, and then concatenated to form the KV cache. However, if the LLM uses this KV cache to generate the answer, it will start to ramble and not produce the right answer.

To understand why, we take a closer look at the **attention matrix** (explained in §2), particularly the cross-attention between the two text chunks talking about the players' statistics. Figure 4 visualizes the attention matrix resulting from the KV cache of the original (full) prefill and the KV cache of full KV reuse. Since full KV reuse precomputes each chunk separately, the cross attention between two chunks is completely missed (never computed) when the KV caches are precomputed. In this example, the first chunk contains Messi's goal count and the second chunk contains Ronaldo's. The LLM is queried to compare the goal counts between Messi and Ronaldo. Neglecting the interaction (cross-attention) between two chunks would lead to a flawed answer.

In fairness, it should be noted that full KV reuse does work when the cross-attention between chunks is low. This can commonly occur with prompt templates which are the main target application of PromptCache [24].

The absence of cross-attention in full KV reuse causes significant discrepancies in the **forward attention** matrix (explained in §2), which contains the attention between context tokens and the last few tokens, and directly affects the generated tokens.

To show the prevalence of cross-attention in multi-chunk LLM inputs, Figure 2 contrasts the response quality (in F1 score) between full KV recompute (with cross-attention) and full KV reuse (without cross-attention). We can see that as the number of relevant chunks increases, the disparity between full prefill and modular caching becomes more pronounced. This is because, with a larger number of chunks, the amount of cross-referencing and interdependency between different parts of the input (cross-attention) increases.

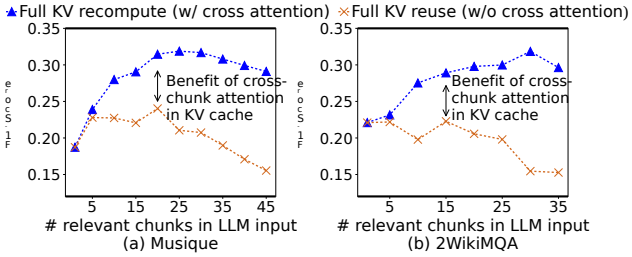


图2. 随着检索到的文本块数量增加，生成质量提高。

并从数据库中获取与查询和块的嵌入之间的最小 L2 距离相关的前 k 个块。图 2 显示了生成质量，使用标准 F1 分数度量，随着所选文本块数量的增加。我们可以看到，随着检索到更多文本块以补充 LLM 输入，质量显著提高，尽管包含过多块会由于众所周知的中间丢失问题而影响质量 [40, 54]。

简而言之，前缀缓存只能保存第一个文本块的预填充，因此当LLM输入包含更多文本块时，节省的效果将是微不足道的，即使它们被重用。

3.3 为什么完全的KV重用不足?

提出了完全的KV重用以解决这个问题。这种方法最近由PromptCache [24] 首创。它通过缓冲区将独立预计算的KV缓存的重复文本块连接起来，以保持每个文本块的位置信息的准确性。例如，为了连接块C_1和C_2的KV缓存，PromptCache首先需要通过在一个假设的输入上运行预填充来预计算C_2的KV缓存，该输入在C_2前面加上一个长度大于或等于C_1的虚拟前缀。这样，即使C_2不是前缀，我们仍然可以正确保留C_2的KV缓存的位置信息，尽管每个块的KV缓存必须被预计算多次。

然而，即使保留了位置信息，一个更根本的问题是非前缀文本块（例如，C_2）的KV缓存忽略了该块与前面的文本（例如，C_1）之间的交叉注意力。这是因为在预计算KV缓存时，前面的文本是未知的。

忽略交叉注意力可能导致错误的响应。图3展示了一个说明性的例子，其中用户查询“梅西在国际足联世界杯上比克里斯蒂亚诺·罗纳尔多多进了多少球？”前面加上了两段球员职业统计数据的文本块。使用完整的预填充或前缀缓存，结果清晰且正确。使用完整的KV重用时，两个文本块的KV缓存是预先计算的，每个块都有正确的位置嵌入，然后连接形成KV缓存。然而，如果LLM

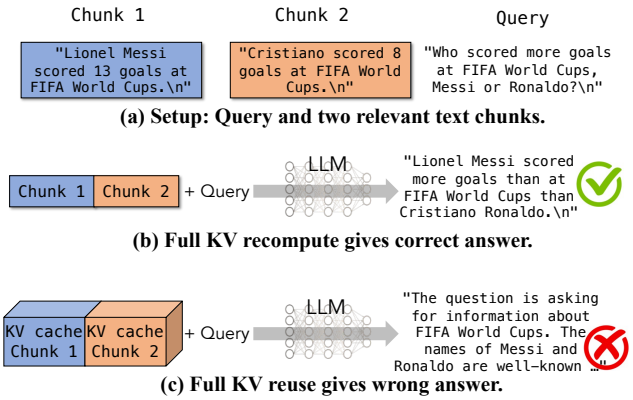


图3. 一个LLM输入的示例，其中两个文本块被添加到查询前面。完全的KV重新计算(b)，不重用KV缓存，速度较慢但给出正确答案。然而，完全的KV重用(c)给出了错误的答案，因为它忽略了块之间的交叉注意力（图4）。

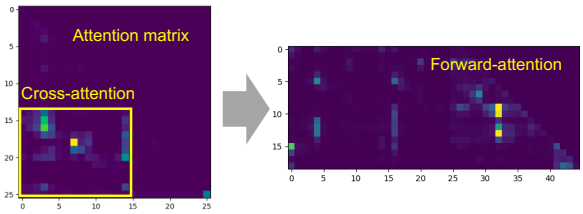
使用这个KV缓存生成答案时，它会开始胡言乱语，而无法产生正确的答案。

为了理解原因，我们更仔细地查看注意力矩阵（在§2中解释），特别是关于球员统计数据的两个文本块之间的交叉注意力。图4可视化了来自原始（完整）预填充的KV缓存和完整KV重用的KV缓存所产生的注意力矩阵。由于完整的KV重用分别预计算每个块，因此在KV缓存被预计算时，两个块之间的交叉注意力完全被忽略（从未计算）。在这个例子中，第一个块包含梅西的进球数，第二个块包含罗纳尔多的进球数。LLM被查询以比较梅西和罗纳尔多之间的进球数。忽视两个块之间的交互（交叉注意力）将导致错误的答案。

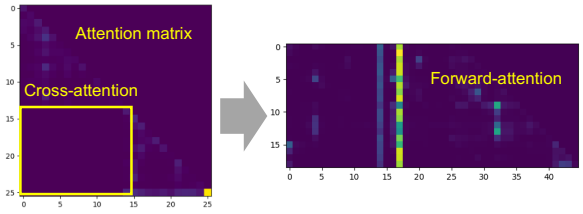
公平地说，应该指出，当块之间的交叉注意力较低时，完全的KV重用确实有效。这通常发生在提示模板中，而提示模板是PromptCache [24] 的主要目标应用。

在完全的KV重用中缺乏交叉注意力会导致前向注意力矩阵（在§2中解释）出现显著差异，该矩阵包含上下文标记与最后几个标记之间的注意力，并直接影响生成的标记。

为了展示跨注意力在多块 LLM 输入中的普遍性，图 2 对比了全 KV 重新计算（带跨注意力）和全 KV 重用（不带跨注意力）之间的响应质量（以 F1 分数计）。我们可以看到，随着相关块数量的增加，完全预填充和模块缓存之间的差异变得更加明显。这是因为，随着块数量的增加，输入不同部分之间的交叉引用和相互依赖（跨注意力）的数量也增加。



(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.*

4 Fast KV Cache Fusing

Given that *full KV recompute* (*i.e.,* full prefill or prefix caching) can be too slow while *full KV reuse* has low quality, a natural question then is how to have both the speed of full KV reuse *and* the quality of full KV recompute. Our goal, therefore, is the following:

Goal. *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.*

To achieve our goal, we present CacheBlend, which *recomputes the KV of a selective subset of tokens* on each layer while reusing other tokens' KV.[2] This section explains CacheBlend in three parts. We begin with the notations (§4.1), and then describe *how to recompute* the KV of only a small subset of tokens (§4.2), and finally explain *how to select* the tokens on each layer whose KV will be recomputed (§4.3).

4.1 Terminology

Table 1 summarizes the notations used in this section. For a given list of N text chunks, we use KV^{full} to denote the KV cache from full KV recompute, KV^{pre} to denote the pre-computed KV cache, and KV^{new} to denote the CacheBlend-updated KV cache. Here, each of these KV caches is a concatenation of KV caches associated with different text chunks. Each layer i of the KV cache, KV_i, produces the forward attention matrix A_i.

[2]For simplicity, we use the terms KV and KV cache interchangeably.

Notation	*Description*
i	Layer index
j	Token index
KV	KV cache
KV_i	KV on layer i
$KV_i[j]$	KV on layer i at token j
KV^{full}	Fully recomputed KV cache
KV^{pre}	Pre-computed KV cache
KV^{new}	CacheBlend-updated KV cache
A_i	Forward attention matrix on layer i
A_i^{full}	Forward attention matrix of full KV recompute
A_i^{pre}	Forward attention matrix of full KV reuse
A_i^{new}	Forward attention matrix with CacheBlend
$\Delta_{\text{kv}}(KV_i, KV_i^{\text{full}})[j]$	KV deviation between $KV_i[j]$ and $KV_i^{\text{full}}[j]$
$\Delta_{\text{attn}}(A_i, A_i^{\text{full}})$	Attention deviation between A_i and A_i^{full}

Table 1. Summary of terminology

The difference between the full KV recompute (full prefill) and the full KV re-use is two-fold.

- **KV deviation:** We define the *KV deviation* of a KV cache KV on layer i of token j as the absolute difference between $KV_i[j]$ and $KV_i^{\text{full}}[j]$, denoted as $\Delta_{\text{kv}}(KV_i, KV_i^{\text{full}})[j]$. It measures how much different the given KV is on a particular token and layer compared to the full-prefilled KV cache. We will later use the KV deviation to identify which tokens' KV has higher deviation and thus need to be updated.
- **Attention deviation:** Similarly, for the forward attention matrix A_i on layer i, we define the *attention deviation*, denoted as $\Delta_{\text{attn}}(A_i, A_i^{\text{full}})$, to be the L-2 norm of its difference with A_i^{full}. Recall from §3.3 that full KV reuse suffers from deviation in the forward attention matrix (illustrated in Figure 4) due to the absence of cross-attention.

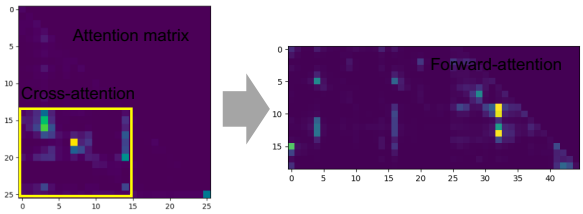
Using these notations, our goal can be formulated as how to quickly update the precomputed KV cache KV^{pre} to the new KV cache KV^{new}, such that the attention deviation $\Delta_{\text{attn}}(A_i^{\text{new}}, A_i^{\text{full}})$ on any layer i, is minimized.

4.2 Selectively recomputing KV cache

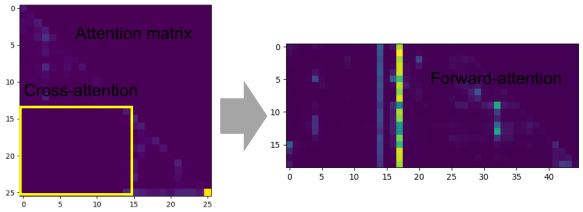
For now, let us assume we have already selected a subset of tokens to recompute on each layer (we will explain how to select them in §4.3). Here, we describe how CacheBlend recomputes the KV of these selected tokens on each layer.

Workflow: The default implementation of prefill (depicted in Figure 5(a)) does not "skip" tokens while only computes the KV of a subset of tokens. Instead, CacheBlend runs the following steps (depicted in Figure 5(b)):

- 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 Q_i, K_i and V_i vectors will also be restricted to the selected tokens.
- It then expands the K_i vector and V_i vector by reusing the KV cache entries associated with the un-selected tokens



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



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

图4. 对比(a)完整KV重新计算和(b)完整KV重用的注意力矩阵。黄色框突出显示了交叉注意力。右侧的图表显示了结果前向注意力矩阵，其差异是两种方法之间不同交叉注意力的结果。

4 快速 KV 缓存融合

考虑到完全的KV重新计算（即完全的预填充或前缀缓存）可能太慢，而完全的KV重用质量较低，那么一个自然的问题是如何同时拥有完全KV重用的速度和完全KV重新计算的质量。因此，我们的目标是以下内容：

目标。当LLM输入包含多个重复使用的文本块时，如何快速更新预计算的KV缓存，以使前向注意力矩阵（以及随后生成的输出文本）与通过完全KV重新计算生成的矩阵之间的差异最小。

为了实现我们的目标，我们提出了CacheBlend，它在每一层上重新计算选择子集的令牌的KV，同时重用其他令牌的KV。[2] 本节将CacheBlend分为三个部分进行解释。我们首先介绍符号（§4.1），然后描述如何仅重新计算小子集令牌的KV（§4.2），最后解释如何选择每一层上将重新计算KV的令牌（§4.3）。

4.1 术语

表1总结了本节中使用的符号。对于给定的N文本块列表，我们使用KV^{full}表示来自完整KV重新计算的KV缓存，使用KV^{pre}表示预计算的KV缓存，使用KV^{new}表示CacheBlend更新的KV缓存。在这里，这些KV缓存都是与不同文本块相关的KV缓存的连接。KV缓存的每一层i，KV_i，产生前向注意力矩阵A_i。

[2]For simplicity, we use the terms KV and KV cache interchangeably.

Notation	*Description*
i	Layer index
j	Token index
KV	KV cache
KV_i	KV on layer i
$KV_i[j]$	KV on layer i at token j
KV^{full}	Fully recomputed KV cache
KV^{pre}	Pre-computed KV cache
KV^{new}	CacheBlend-updated KV cache
A_i	Forward attention matrix on layer i
A_i^{full}	Forward attention matrix of full KV recompute
A_i^{pre}	Forward attention matrix of full KV reuse
A_i^{new}	Forward attention matrix with CacheBlend
$\Delta_{\text{kv}}(KV_i, KV_i^{\text{full}})[j]$	KV deviation between $KV_i[j]$ and $KV_i^{\text{full}}[j]$
$\Delta_{\text{attn}}(A_i, A_i^{\text{full}})$	Attention deviation between A_i and A_i^{full}

表1. 术语总结

全KV重新计算（完全预填充）与全KV重用之间的区别有两个方面。

- KV 偏差：我们定义在令牌 j 的层 i 上的 KV 缓存 KV 的 KV 偏差为 $KV_i[j]$ 和 $KV_i^{\text{full}}[j]$ 之间的绝对差值，记作 $\Delta_{\text{kv}}(KV_i, KV_i^{\text{full}})[j]$。它衡量给定的 KV 在特定令牌和层上与完全预填充的 KV 缓存相比有多大的不同。我们稍后将使用 KV 偏差来识别哪些令牌的 KV 偏差较高，因此需要更新。
- 注意偏差：类似地，对于层 i 上的前向注意矩阵 A_i，我们定义注意偏差，记作 $\Delta_{\text{attn}}(A_i, A_i^{\text{full}})$，为其与 A_i^{full} 之间差异的 L-2 范数。回想 §3.3，完整的 KV 重用由于缺乏交叉注意，导致前向注意矩阵（如图 4 所示）出现偏差。

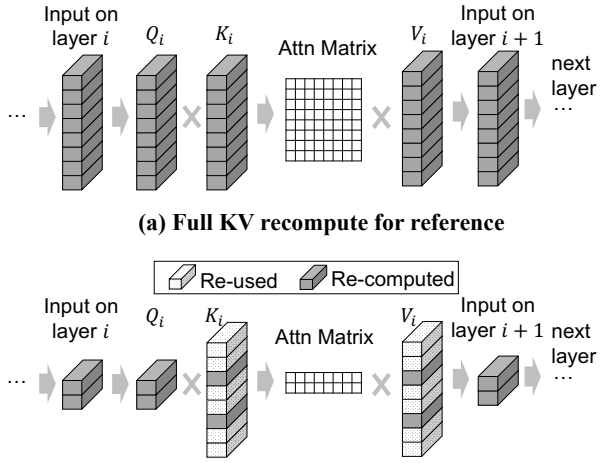
使用这些符号，我们的目标可以表述为如何快速将预计算的 KV 缓存 KV^{pre} 更新为新的 KV 缓存 KV^{new}，以使任何层 i 上的注意力偏差 $\Delta_{\text{attn}}(A_i^{\text{new}}, A_i^{\text{full}})$ 最小化。

4.2 有选择地重新计算 KV 缓存

目前，让我们假设我们已经选择了一组令牌在每一层上进行重新计算（我们将在§4.3中解释如何选择它们）。在这里，我们描述CacheBlend如何在每一层上重新计算这些选定令牌的KV。

工作流程：预填充的默认实现（如图5(a)所示）并不会“跳过”标记，而只是计算一部分标记的KV。相反，CacheBlend执行以下步骤（如图5(b)所示）：

- 它首先对每一层的输入 i 应用一个掩码，将其减少到选定标记的子集。
- 它随后将减少的输入转换为 Q_i、K_i 和 V_i 向量，这些向量也将限制在所选的标记中。
- 它然后通过重用与未选择的标记相关联的KV缓存条目来扩展K_i向量和V_i向量。



(a) Full KV recompute for reference

(b) Selective KV recompute on two selected tokens

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

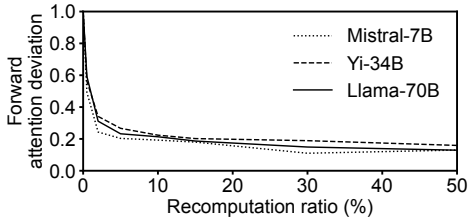


Figure 6. *Attention deviation reduces as we recompute the KV of more tokens on each layer. Importantly, the biggest drop in attention deviation results from recomputing the KV of the tokens with the highest KV deviation (i.e., HKVD tokens).*

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.

These changes make little assumption on the exact transformer process and can be integrated with many popular transformers (more details in §6). It is important to notice that the compute overhead is proportional to the number of selected tokens. This is because it only runs computation associated with the selected tokens. If we recompute r% of tokens per layer, the total compute overhead will be r% of full prefill.

4.3 Selecting which tokens to recompute

Next, we explain how to choose the tokens whose KV should be recomputed on each layer in order to reduce the attention deviation on each layer, which results from the KV deviation. Our intuition, therefore, is to prioritize recomputing the KV of tokens who have high KV deviations. Of course, this intuitive scheme is *not* feasible as it needs to know full-prefilled KV cache, and we will make it practical shortly.

To show the effectiveness of selecting tokens with high KV deviations, Figure 6 uses three models on the dataset of Musique (please see §7.1 for details). It shows the change of average attention deviation across all layers i, $\Delta_{\text{attn}}(A_i, A_i^{\text{full}})$, after we use the aforementioned scheme (§4.2) to select and recompute the r% of tokens j who have the highest KV deviation $\Delta_{\text{kv}}(KV_i, KV_i^{\text{full}})[j]$. As the recompute ratio (r) increases, we can see that the attention deviation gradually reduces, and the biggest drops happen when the top few tokens with the highest KV deviations are recomputed. Empirically, it suggests the following insight.

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.

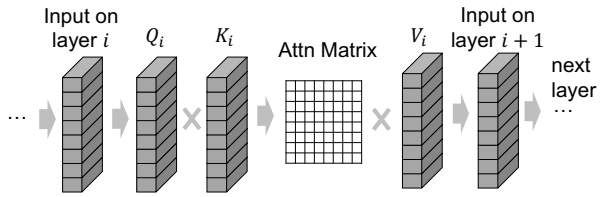
Do we need to recompute KV for most tokens? In §7, we empirically show that choosing 10-20% tokens as HKVD tokens and recomputing their KV suffices to greatly reduce the attention deviation and preserve generation quality.

This can be intuitively explained by *attention sparsity*, a well-studied property observed in many transformer models by prior research [14, 15, 43, 58]. It says that in an attention matrix, high attention typically only occurs between a small number of tokens and their preceding tokens. To validate this observation, Figure 7 uses the same models and dataset as Figure 6. It shows the distribution of KV deviation on one layer. We can see that a small fraction, about 10-15%, of tokens have much higher KV deviation than others, which corroborates the sparsity of cross-attention.

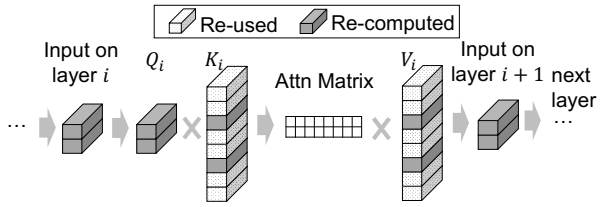
If a token has very low attention with other chunks' tokens (*i.e.,* low cross-attention with other chunks), the KV deviation between A^{pre} and A^{full} will be low and thus do not need to be recomputed. Only when a token has a high attention with other chunks (high KV deviation compared with ground truth), should its KV be recomputed.

How to identify the HKVD tokens without knowing the true KV values or attention matrix? Naively, to identify the HKVD tokens, one must know the fully recomputed KV_i^{full} of each layer i in the first place, but doing so is too

[3] In the precomputed KV cache, the K vector of each chunk must be adjusted with the correct positional embedding. In SOTA positional embedding scheme (Rotary Positional Embedding or ROPE [50]), this correction is done simply by multiplying the K vector by a rotation matrix of $\begin{pmatrix}\cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta\end{pmatrix}$. (The n-dimensional case in Appendix A) This step has negligible overhead since the multiplication is performed only once.



(a) Full KV recompute for reference



(b) Selective KV recompute on two selected tokens

图5. (a) 完全KV重新计算与(b) 单层选择性KV重新计算之间的对比。

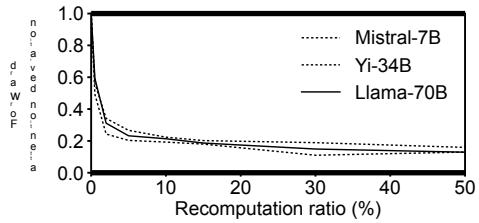


图6. 随着我们在每一层上重新计算更多标记的KV，注意力偏差减少。重要的是，注意力偏差的最大下降来自于重新计算具有最高KV偏差的标记的KV（即HKVD标记）。

在层 i 上，以便注意力矩阵包括所选标记与所有其他标记之间的注意力。

- 最后，它运行相同的注意力模块以生成下一层的输入。

这些变化对精确的变换器过程几乎没有假设，并且可以与许多流行的变换器集成（更多细节见§6）。重要的是要注意，计算开销与所选标记的数量成正比。这是因为它仅运行与所选标记相关的计算。如果我们在每一层重新计算 r% 的标记，总的计算开销将是 r% 的完整预填充。

4.3 选择重新计算哪些标记

接下来，我们将解释如何选择在每一层中需要重新计算 KV 的令牌，以减少每一层因 KV 偏差而导致的注意力偏差。因此，我们的直觉是优先重新计算具有高 KV 偏差的令牌的 KV。当然，这种直观的方案并不可行，因为它需要知道完整的预填充 KV 缓存，我们将很快使其变得可行。

为了展示选择具有高KV偏差的标记的有效性，图6在Musique数据集上使用了三种模型（详细信息请参见§7.1）。它显示了在我们使用上述方案（§4.2）选择并重新计算具有最高KV偏差的r%标记j后，所有层的平均注意力偏差的变化i，$\Delta_{\text{attn}}(A_i, A_i^{\text{full}})$。随着重新计算比例（$r$）的增加，我们可以看到注意力偏差逐渐减少，最大的下降发生在重新计算具有最高KV偏差的前几个标记时。经验上，这表明了以下见解。

洞察 1。在层 i 上，重新计算具有更高 KV 偏差的令牌 j 的 KV（即 $\Delta_{kv}(KV_i, KV_i^{full})[j]$）会以更大的幅度减少注意力偏差（即 $\Delta_{attn}(A_i, A_i^{full})$）。

因此，如果我们重新计算层 i 上 10% 的令牌的 KV，我们应该选择 KV 偏差最高的 10% 的令牌。[3] 我们将这些令牌称为层 i 上的高 KV 偏差（或 HKVD）令牌。

现在我们知道应该重新计算HKVD令牌的KV，两个自然的问题出现了。

我们是否需要为大多数标记重新计算KV？在§7中，我们通过实验证明，选择10-20%的标记作为HKVD标记并重新计算它们的KV足以大大减少注意力偏差并保持生成质量。

这可以通过注意力稀疏性直观地解释，这是先前研究[14, 15, 43, 58]在许多变换器模型中观察到的一个经过充分研究的特性。它表明，在一个注意力矩阵中，高注意力通常只发生在少数几个标记及其前面的标记之间。为了验证这一观察，图7使用与图6相同的模型和数据集。它显示了一个层上KV偏差的分布。我们可以看到，约10-15%的标记的KV偏差远高于其他标记，这证实了交叉注意力的稀疏性。

如果一个标记与其他块的标记的注意力非常低（即，与其他块的交叉注意力低），则 A^{pre} 和 A^{full} 之间的 KV 偏差将很低，因此不需要重新计算。只有当一个标记与其他块的注意力很高（与真实值相比，KV 偏差高）时，才应该重新计算其 KV。

如何在不知道真实 KV 值或注意力矩阵的情况下识别 HKVD 令牌？简单来说，要识别 HKVD 令牌，首先必须知道每一层 KV_i^{full} 的完全重新计算 i，但这样做太过复杂。

[3] In the precomputed KV cache, the K vector of each chunk must be adjusted with the correct positional embedding. In SOTA positional embedding scheme (Rotary Positional Embedding or ROPE [50]), this correction is done simply by multiplying the K vector by a rotation matrix of $\begin{pmatrix}\cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta\end{pmatrix}$. (The n-dimensional case in Appendix A) This step has negligible overhead since the multiplication is performed only once.

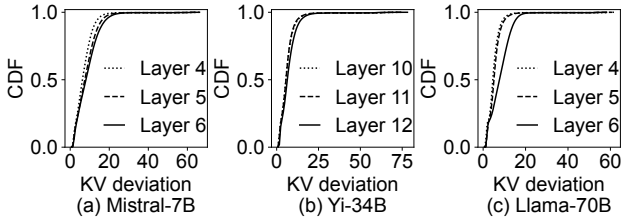


Figure 7. *Distribution of KV deviation of different tokens on one layer.*

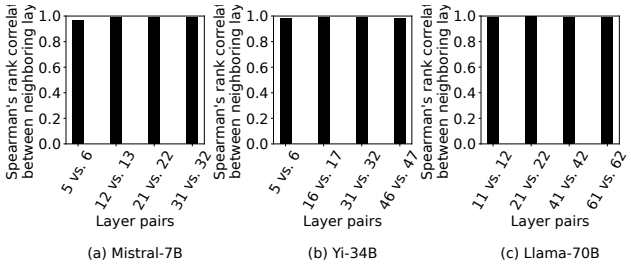


Figure 8. *Rank correlation of the KV deviation per token between two consecutive layers.*

expensive and defeats the purpose of selective KV recompute. Instead, we observe that the HKVD tokens on different layers are not independent:

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

For instance, if the HKVD tokens on the first layer are tokens 2, 3, and 5, these three tokens will likely also have higher KV deviations than most other tokens on the second layer.

Figure 8 uses the same setting as Figure 7 and shows Spearman's rank correlation score between the KV deviation of tokens between two neighboring layers. The figure shows a consistently high similarity of HKVD tokens between different layers.[4]

The intuition behind this correlation lies in the previous observation that the input embedding of each token changes slowly between layers in transformer models [44, 47]. Therefore, KV cache between layers should also bear similarity as KV cache is generated from the input embedding with a linear transformation.

Given the substantial correlation between the HKVD tokens, a straightforward solution is that we can perform prefill on the first layer first, pick the HKVD tokens of the first layer, and only update their KV on all other layers. Since an LLM usually has over 30 layers, this process can save most of the compute compared to full KV recompute. 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.

[4]We should clarify that although the HKVD tokens are similar across layers, the attention matrices between layers can still be quite different.

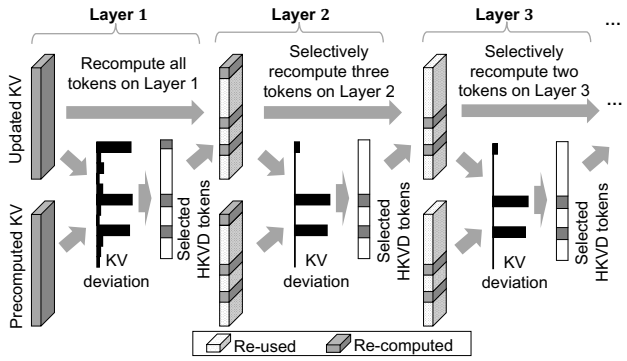


Figure 9. *CacheBlend selects the HKVD (high KV deviation) tokens of one layer by computing KV deviation of only the HKVD tokens selected from the previous layer and selecting the tokens among them with high KV deviation.*

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 picks the HKVD tokens who have high attention deviation, not only on the first layer but also on multiple layers, which empirically is statistically more reliable to identify the HKVD tokens on each layer.

Although the KV-cache space of the layer i performing the HKVD calculation holds both Updated-KV and Precomputed-KV, layer-i's extra Precomputed-KV is immediately discarded once the inference proceeds to layer i+1. This makes the memory overhead in HKVD negligible.

5 CacheBlend System Design

We present a concrete system design for CacheBlend, which reduces the impact of the selective KV recompute using the following basic insight.

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.*

Pipelining KV loading and recompute: In CacheBlend, the selective recompute of one layer can start immediately after pre-computed the KV cache of the previous layer is loaded into the GPU. This is because which tokens' KV to recompute on one layer only depends on the KV deviation of the previous layer's tokens. As a result, if loading the pre-computed KV for one layer is faster or equal to selective KV

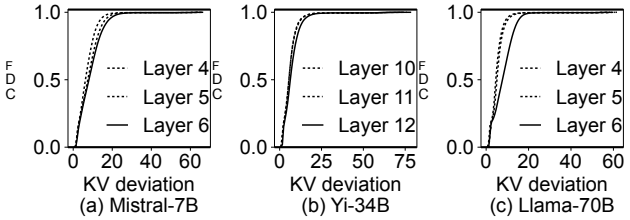


图7. 一层上不同标记的KV偏差分布。

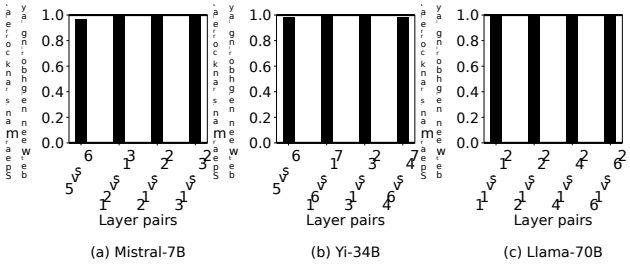


图8. 两个连续层之间每个标记的KV偏差的等级相关性。

昂贵且违背了选择性KV重新计算的目的。相反，我们观察到不同层上的HKVD令牌并不是独立的：

洞察 2. 在一层中具有最高 KV 偏差的标记在下一层中很可能也会具有最高的 KV 偏差。

例如，如果第一层的HKVD令牌是令牌2、3和5，那么这三个令牌在第二层上的KV偏差可能也会高于大多数其他令牌。

图8使用与图7相同的设置，显示了相邻层之间令牌的KV偏差的斯皮尔曼等级相关系数。该图显示了不同层之间HKVD令牌的一致高相似性。[4]

这种相关性的直觉在于之前的观察，即在变换模型中，每个标记的输入嵌入在层之间缓慢变化。因此，层之间的KV缓存也应该具有相似性，因为KV缓存是通过对输入嵌入进行线性变换生成的。

鉴于HKVD令牌之间的显著相关性，一个简单的解决方案是我们可以先在第一层进行预填充，选择第一层的HKVD令牌，并仅在所有其他层更新它们的KV。由于LLM通常有超过30层，与完全重新计算KV相比，这个过程可以节省大部分计算资源。也就是说，仅使用第一层不同令牌的注意力偏差可能在统计上不可靠，以选择所有层的HKVD令牌，特别是更深的层。

[4]We should clarify that although the HKVD tokens are similar across layers, the attention matrices between layers can still be quite different.

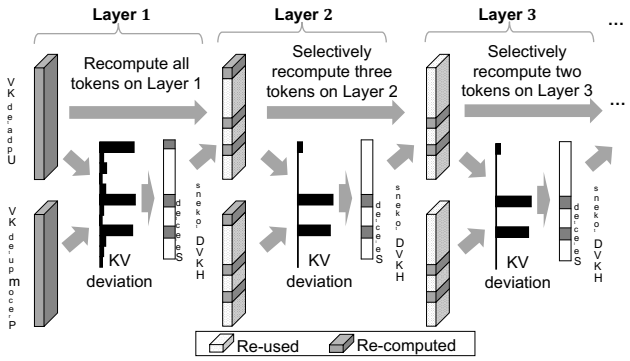


图9。CacheBlend通过计算仅从上一层选择的HKVD（高KV偏差）令牌的KV偏差，选择一层的HKVD令牌，并在其中选择具有高KV偏差的令牌。

因此，我们选择逐步过滤方案（如图9所示）。如果我们平均想要每层选择 r% 的 HKVD 令牌，我们将根据第一层的令牌级注意力偏差选择 r_1 % 的令牌，其中 r_1 稍高于 r，并将它们用作第二层的 HKVD 令牌。然后，我们重新计算第二层这些 r_1 % HKVD 令牌的 KV，并选择 r_2 % 具有最高令牌级注意力偏差的令牌，其中 r_2 稍低于 r_1，作为下一层的 HKVD 令牌，依此类推。直观上，这种逐步过滤方案最终选择了在第一层和多个层上都具有高注意力偏差的 HKVD 令牌，这在经验上在统计上更可靠地识别每层的 HKVD 令牌。

尽管执行 HKVD 计算的第 i 层的 KV-cache 空间同时包含 Updated-KV 和 Precomputed-KV，但一旦推理进入层 i+1，第 i 层的额外 Precomputed-KV 会立即被丢弃。这使得 HKVD 的内存开销可以忽略不计。

5 缓存混合系统设计

我们提出了一个具体的CacheBlend系统设计，它通过以下基本见解减少了选择性KV重新计算的影响。

基本见解：如果选择性KV重新计算（§4.3）的延迟快于将KV加载到GPU内存中的延迟，那么合理地将选择性KV重新计算和KV加载进行流水线处理，使得KV重新计算的额外延迟可以忽略不计。

流水线KV加载和重计算：在CacheBlend中，选择性重计算一层可以在前一层的KV缓存加载到GPU后立即开始。这是因为在一层上重计算哪些令牌的KV仅依赖于前一层令牌的KV偏差。因此，如果加载一层的预计算KV的速度快于或等于选择性KV

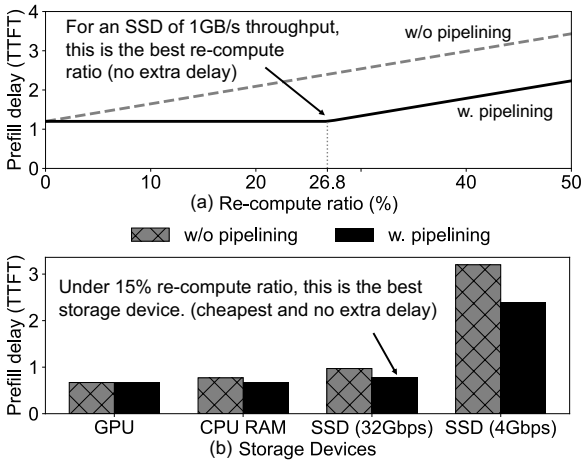


Figure 10. *(a) Smartly picking the recompute ratio will not incur an extra delay. (b) Smartly picking storage device(s) to store KVs saves cost while not increasing delay.*

recompute of one layer, the KV-loading delay should be able to hide the selective recompute delay, *i.e.,* without incurring any extra delay on time-to-first-token (TTFT).

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).

5.1 Key Components

To realize the benefit of pipelining KV loading and recompute, our system has three major components.

Loading Controller: We face two design questions in practice: First, *given a fixed storage device to use, how to choose a recompute ratio (what fraction of tokens to recompute KV per layer) without incurring extra delay to time-to-first-token (TTFT)?* Figure 10(a) illustrates an example that, if we select a recompute ratio wisely, the recompute should not cause *any* extra delay to loading if loading is slow.

For this, the controller uses two delay estimators to find an idealized recompute ratio, such that the recompute delay is close to the loading delay. Given the recompute ratio r, length of context to be loaded L, and LLM, the *recompute delay estimator* calculates the expected delay $T_{recompute}(r\%, LLM, L)$[5].

[5] $T_{recompute}(r\%, LLM, L) = r\% \times Prefill(LLM, L)$. $Prefill(LLM, L)$ is offline profiled.

The *loading delay estimator* estimates the loading delay of the KV cache of one layer, $T_{load}(LLM, L, storage\_device)$[6], based on the LLM, the storage device's speed (which is measured offline), and the length of context L.

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 recompute. 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.

In practice, CacheBlend faces another challenge: which *storage devices* should the developer use? To solve this challenge, we present a more formulated question to the loading controller: *If we only do KV recompute of a fixed selective recompute ratio (ex.* 15%*), how can we choose the right storage device to store KVs such that no extra delay is caused?* As shown in Figure 10(b), under a fixed recompute ratio, the controller should pick the cheapest storage device among all devices that do not increase the delay.

In CacheBlend, the system developers can provide a list of potential storage device(s), and the controller uses a *storage cost estimator* which estimates the cost of storing KVs for each device, namely $C_{store}(LLM, L, T, storage\_device)$, based on the LLM, length of context L and time duration T needed to store it (if it is cloud storage). Then it uses $T_{recompute}(15\%, LLM, L)$ and $T_{load}(LLM, L, storage\_device)$ to estimate the recompute and loading delays for all devices. Lastly, it finds out which storage device is the cheapest where $T_{recompute} \geq T_{load}$. In this way, if the developer can navigate through the different storage devices for the KV caches given a fixed recomputation target that satisfies the generation quality requirement.

KV cache store *(mapping LLM input to KV caches)*: The KV cache store splits an LLM input into multiple text chunks, each of which can be reused or new. For instance, a RAG input typically consists of multiple retrieved context chunks (likely of a fixed length) and the user input. The splitting of LLM inputs is specific to the application, and we implement the same strategy as described in recent work [24, 38]. Once the input is split into text chunks, each chunk is hashed to find their corresponding KV cache, in the same way as the block hashing is implemented in vLLM [36]. The KV caches of new chunks generated by the fusor (explained soon) are added to the devices. When the storage devices are full, we evict the least recently used KV cache. In this paper, we only focus on storing KV cache in one single level of storage device such as CPU RAM or SSD.

[6] $T_{load}(LLM, L, storage\_device) = \frac{PerTokenKVSize(LLM) \times L}{Throughput(storage\_device)}$.

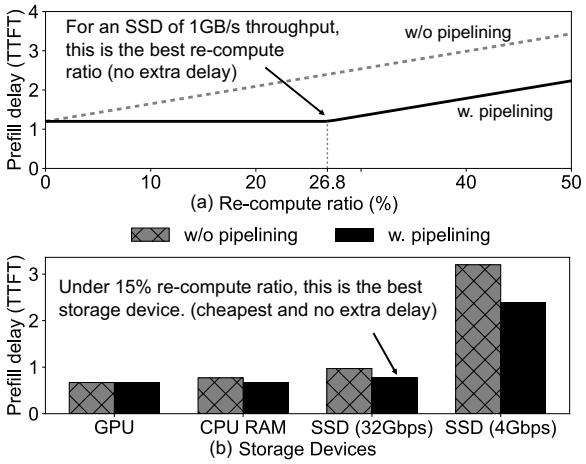


图10. (a) 智能选择重计算比例不会导致额外延迟。 (b) 智能选择存储设备来存储 KVs 可以节省成本，同时不会增加延迟。

重新计算一层时，KV加载延迟应该能够隐藏选择性重新计算延迟，即在首次令牌时间（TTFT）上不产生任何额外延迟。

使用Llama-7B模型和4K长的上下文，仅重新计算15%的标记（默认重新计算比例）每层只需3毫秒，而从NVME SSD加载一层的KV缓存需要16毫秒（§7）。在这种情况下，KV加载可以掩盖对15%标记的KV重新计算的延迟，即KV重新计算不会产生额外的延迟。重新计算更多的标记，虽然可以稍微提高生成质量，但只要延迟低于16毫秒，也可能不会产生额外的延迟。相反，使用另一个模型Llama-70B，重新计算15%的标记需要7毫秒，但从NVME SSD加载一层的KV只需4毫秒。在这里，KV加载并没有完全掩盖重新计算的延迟。总之，需要一个控制器来智能地选择重新计算比例以及存储KV缓存的位置（如果适用）。

5.1 关键组成部分

为了实现管道化KV加载和重新计算的好处，我们的系统有三个主要组件。

加载控制器：在实践中，我们面临两个设计问题：首先，给定一个固定的存储设备，如何选择重计算比例（每层重计算多少比例的令牌 KV），而不增加首次令牌时间（TTFT）的额外延迟？图 10(a) 说明了一个例子，如果我们明智地选择重计算比例，重计算在加载缓慢的情况下不应导致任何额外的加载延迟。

为此，控制器使用两个延迟估计器来找到理想的重新计算比率，使得重新计算延迟接近加载延迟。给定重新计算比率 r、要加载的上下文长度 L 和 LLM，重新计算延迟估计器计算预期延迟 $T_{recompute}(r\%, LLM, L)$[5]。

加载延迟估计器根据LLM、存储设备的速度（离线测量）和上下文长度 L 估计一层的KV缓存的加载延迟 $T_{load}(LLM, L, storage\_device)$[6]。

控制器计算一个理想化的重新计算比率，使得加载延迟可以隐藏重新计算延迟，而不降低推理质量。它首先选择重新计算比率 $r\%$，使得 $T_{recompute}(r\%, LLM, L)$ 等于 $T_{load}(LLM, L, storage\_device)$，然后取 $r\%$ 和 $r^*\%$ 的最大值，其中 $r^*\%$ 是经验上从完整 KV 重新计算中具有低可忽略质量下降的最小重新计算比率。实际上，我们发现 $r^*\%$ 是来自图 16 的 15%。这意味着即使存储设备是快速设备（例如 CPU RAM），延迟也将受到最小重新计算的下限，以保证质量。

在实践中，CacheBlend 面临另一个挑战：开发者应该使用哪些存储设备？为了解决这个挑战，我们向加载控制器提出了一个更具公式化的问题：如果我们仅以固定的选择性重计算比例（例如 15%）进行 KV 重计算，我们如何选择合适的存储设备来存储 KVs，以确保不造成额外的延迟？如图 10(b) 所示，在固定的重计算比例下，控制器应该在所有不增加延迟的设备中选择最便宜的存储设备。

在CacheBlend中，系统开发人员可以提供潜在存储设备的列表，控制器使用存储成本估算器来估算每个设备存储KV的成本，即$C_{store}(LLM, L, T, storage_device)$，基于LLM、上下文长度$L$和存储所需的时间持续$T$（如果是云存储）。然后，它使用$T_{recompute}(15\%, LLM, L)$和$T_{load}(LLM, L, storage_device)$来估算所有设备的重新计算和加载延迟。最后，它找出哪个存储设备是最便宜的，其中$T_{recompute} \geq T_{load}$。通过这种方式，如果开发人员能够在给定满足生成质量要求的固定重新计算目标的情况下，浏览不同的KV缓存存储设备。

KV 缓存存储（将 LLM 输入映射到 KV 缓存）：KV 缓存存储将 LLM 输入拆分为多个文本块，每个文本块可以被重用或是新的。例如，RAG 输入通常由多个检索到的上下文块（可能是固定长度）和用户输入组成。LLM 输入的拆分是特定于应用的，我们实施与最近的工作 [24, 38] 中描述的相同策略。一旦输入被拆分为文本块，每个块都会被哈希以找到其对应的 KV 缓存，方式与 vLLM [36] 中实现的块哈希相同。由融合器（稍后解释）生成的新块的 KV 缓存会被添加到设备中。当存储设备满时，我们会驱逐最近最少使用的 KV 缓存。在本文中，我们仅关注在单一层级的存储设备中存储 KV 缓存，例如 CPU RAM 或 SSD。

[5] $T_{recompute}(r\%, LLM, L) = r\% \times Prefill(LLM, L)$. $Prefill(LLM, L)$ is offline profiled.

[6] $T_{load}(LLM, L, storage\_device) = \frac{PerTokenKVSize(LLM) \times L}{Throughput(storage\_device)}$.

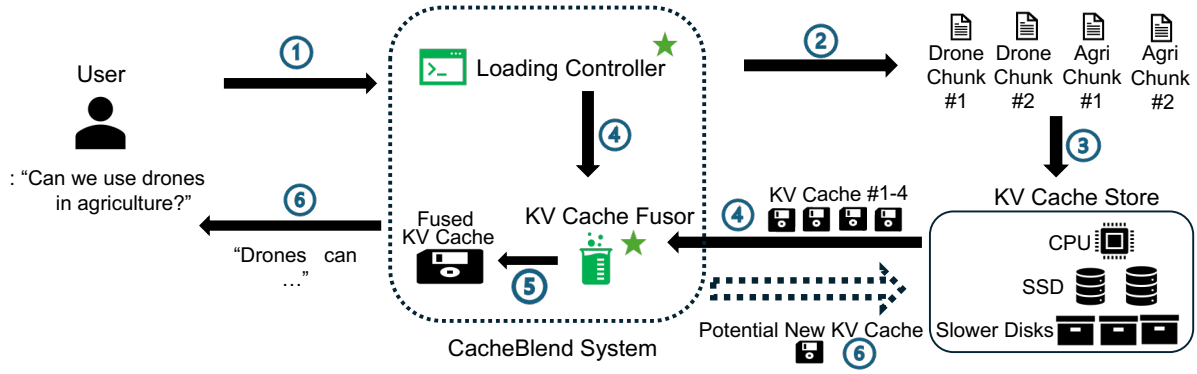


Figure 11. *CacheBlend system (green stared) in light of LLM context augmented generation for a single request. CacheBlend uses text provided by the retriever, interacts with the storage device(s), and provides KV cache on top of LLM inference engines.*

Fusor: The cache fusor (§4) merges pre-computed KV caches via selective recompute. Recall from §4.3, the decision of which tokens need to be recomputed for one layer depends on the recompute of the previous layer. Thus, the fusor waits until the recompute for the previous layer is done, and the KV caches for layer L are loaded into the queue on GPU memory and then perform selective recompute using the recompute ratio $r\%$ calculated by the loading controller. The fusor repeats this process until all the layers are recomputed.

5.2 Putting them together

We put the key components together in an LLM inference workflow in Figure 11. When a user of an LLM application submits a question, a list of relevant text chunks will be queried. The loading controller then queries the KV cache manager on whether the KV caches for those text chunks exist, and where they are stored. Next, the KV cache manager returns this information back to the loading controller and the controller computes the idealized selective recomputation ratio, sends it to the fusor, and loads the KV caches into a queue in GPU memory. The KV cache fusor continuously recomputes the KV caches in the queue, until all layers are recomputed. Lastly, the fused KV cache is input into the LLM inference engine, which generates the answer to the user question based on the KV cache.

6 Implementation

We implement CacheBlend on top of vLLM with about 3K lines of code in Python based on PyTorch v2.0.

Integrating Fusor into LLM serving engine: CacheBlend performs the partial prefill process in a layer-wise manner through three interfaces:

- `fetch_kv(text, layer_id) -> KVCache`: given a piece of text and a layer id, CacheBlend fetches the corresponding KV cache from KV store into the GPU. Returns -1 if the KV cache is not in the system.
- `prefill_layer(input_dict, KVCache) -> output_dict`: CacheBlend takes in the input and KV cache of this layer and performs the partial prefill process for this particular layer. The output is used as the input for the next layer.
- `synchronize()`: CacheBlend requires synchronization before prefilling every layer to make sure the KV cache of this layer has already been loaded into the GPU.

We implement these three interfaces inside vLLMs. For `fetch_kv`, we first calculate the hash of the text and search if it is inside the KV store system. If it is present, we call `torch.load()` to load it into GPU memory if KV cache is on disk or use `torch.cuda()` if the KV cache is inside CPU memory. For `prefill_layer`, we implement this interface on top of the original `layer` function in vLLM that performs one layer of prefill. Three key-value pairs are recorded in the `input_dict`: (1) the original input data `input_org` required for prefilling an LLM layer (e.g., input_tensor, input_metadata), (2) a `check_flag` indicating whether HKVD tokens will be selected in this layer, and (3) `HKVD_indices` that track the indices of HKVD tokens. If `check_flag` is `True`, the input tokens with the largest deviation between the newly computed KV cache and the loaded KV cache will be selected as the HKVD tokens. If `check_flag` is `False`, partial prefill will only be performed on the current HKVD tokens indicated by `HKVD_indices`. Only the KV cache of the HKVD tokens will be computed and updated at each layer. In the partial prefill for layer i, two threads are used to pipeline the computation (`prefill_layer`) of layer i and the KV cache loading (`fetch_kv`) of the next layer $i+1$. `synchronize` is called before `prefill_layer` to assure the KV cache needed for prefill has been loaded into GPU.

Managing KV cache: CacheBlend manages the KV caches such that: If KV cache is not inside the system and is recomputed by the LLM engine in the runtime, we will move the KV cache into CPU by `torch.cpu()` and open a thread to write it back to disk in the background with `torch.save()`. During `fetch_kv`, we go through through the hash tables to fetch KV cache for the fusor. The hash tables are kept in CPU for their relatively small size (16MB for one million chunks).

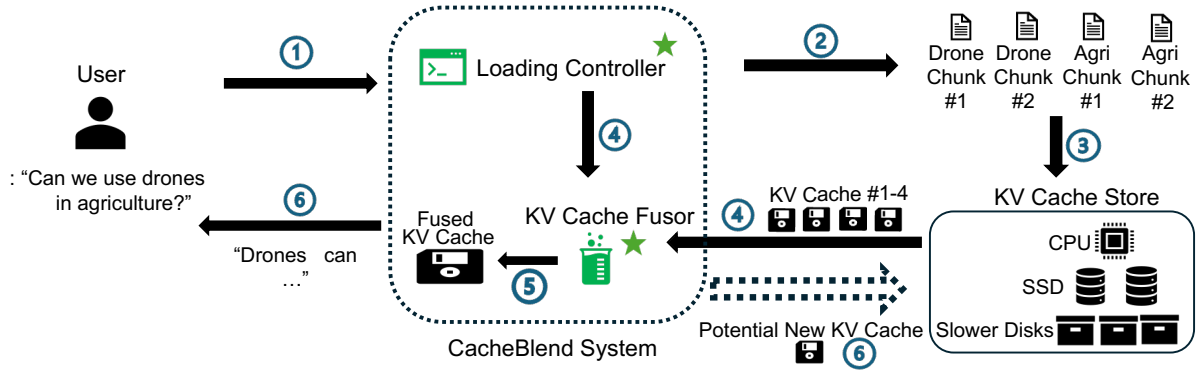


图11. CacheBlend系统（绿色星标）在LLM上下文增强生成单个请求的背景下。CacheBle nd
使用检索器提供的文本，与存储设备交互，并在LLM推理引擎之上提供KV缓存 抱歉，我无法处

Fusor：缓存融合器（§4）通过选择性重新计算合并预计算的KV缓存。回想一下§4.3，决定一个层需要重新计算哪些标记取决于前一层的重新计算。因此，融合器会等待前一层的重新计算完成，然后将层L的KV缓存加载到GPU内存的队列中，然后使用加载控制器计算的重新计算比例r%进行选择性重新计算。融合器重复此过程，直到所有层都被重新计算。

5.2 将它们结合在一起

我们在图11中将关键组件组合在一起，形成一个LLM推理工作流程。当LLM应用的用户提交问题时，将查询相关文本块的列表。加载控制器随后查询KV缓存管理器，了解这些文本块的KV缓存是否存在，以及它们存储的位置。接下来，KV缓存管理器将这些信息返回给加载控制器，控制器计算理想的选择性重计算比率，将其发送给融合器，并将KV缓存加载到GPU内存中的队列中。KV缓存融合器不断重计算队列中的KV缓存，直到所有层都被重计算。最后，融合后的KV缓存被输入到LLM推理引擎中，该引擎根据KV缓存生成用户问题的答案。

6 实施

我们在 vLLM 上实现了 CacheBlend，使用大约 3000 行基于 PyTorch v2.0 的 Python 代码。

将Fusor集成到LLM服务引擎中：CacheBlend通过三个接口以分层方式执行部分预填充过程：

- fetch_kv(text, layer_id) -> KVCache：给定一段文本和一个层 ID，CacheBlend 从 KV 存储中将相应的 KV 缓存提取到 GPU。如果 KV 缓存不在系统中，则返回 -1。
- prefill_layer(input_dict, KVCache) -> output_dict: CacheBlend 接收此层的输入和 KV 缓存并对该特定层执行部分预填充过程。输出用作下一层的输入。
- synchronize(): CacheBlend 在预填充每一层之前需要同步，以确保该层的 KV 缓存已经加载到 GPU 中。

我们在 vLLMs 中实现这三个接口。对于 fetch_kv，我们首先计算文本的哈希值，并检查它是否在 KV 存储系统中。如果存在，我们调用 torch.load() 将其加载到 GPU 内存中（如果 KV 缓存在磁盘上），或者使用 torch.cuda()（如果 KV 缓存在 CPU 内存中）。对于 prefill_layer，我们在 vLLM 的原始层函数之上实现此接口，该函数执行一层的预填充。在 input_dict 中记录了三个键值对：(1) 预填充 LLM 层所需的原始输入数据 input_org（例如，input_tensor，input_metadata），(2) 一个 check_flag，指示此层是否会选择 HKVD 令牌，以及 (3) HKVD_indices，用于跟踪 HKVD 令牌的索引。如果 check_flag 为 True，则将选择新计算的 KV 缓存与加载的 KV 缓存之间偏差最大的输入令牌作为 HKVD 令牌。如果 check_flag 为 False，则仅对 HKVD_indices 指示的当前 HKVD 令牌执行部分预填充。每层只会计算和更新 HKVD 令牌的 KV 缓存。在层 i 的部分预填充中，使用两个线程来管道层 i 的计算（prefill_layer）和下一层 i + 的 KV 缓存加载（fetch_kv）。在 prefill_layer 之前调用 synchronize，以确保所需的 KV 缓存已加载到 GPU 中。

管理KV缓存：CacheBlend管理KV缓存，使得：如果KV缓存不在系统内并且在运行时由LLM引擎重新计算，我们将通过torch.cpu()将KV缓存移动到CPU，并打开一个线程在后台使用torch.save()将其写回磁盘。在fetch_kv期间，我们遍历哈希表以获取fusor的KV缓存。哈希表由于其相对较小的大小（每百万个块16MB）而保留在CPU中。

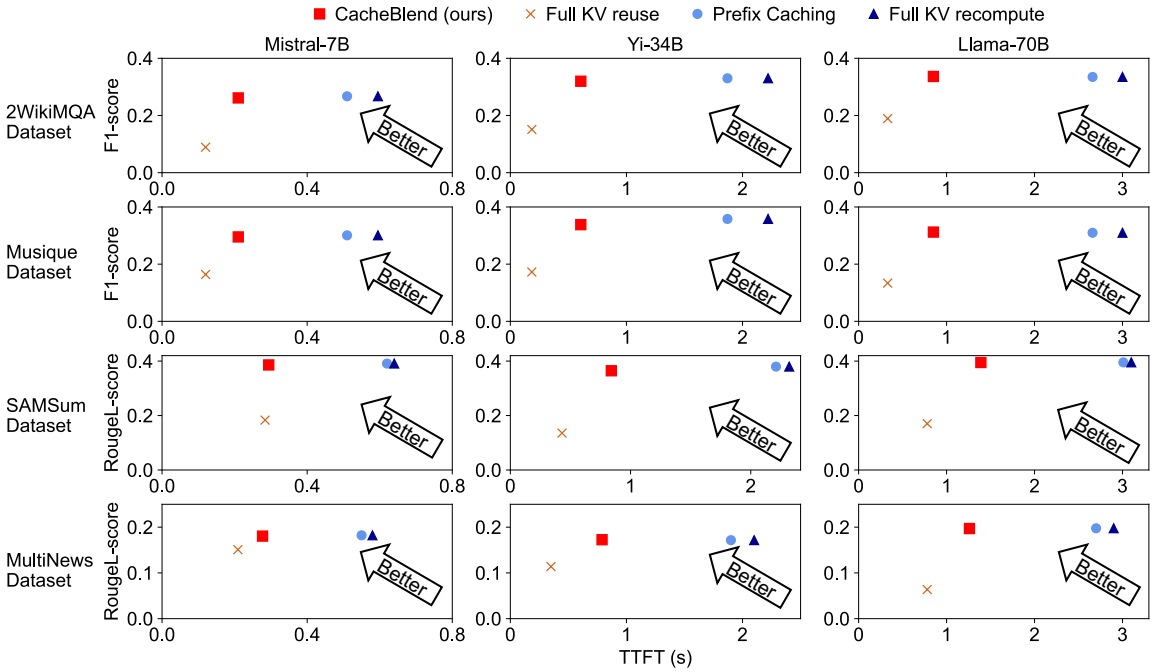


Figure 12. *CacheBlend reduces TTFT by 2.2-3.3× compared to full KV recompute with negligible quality drop across four datasets and three models.*

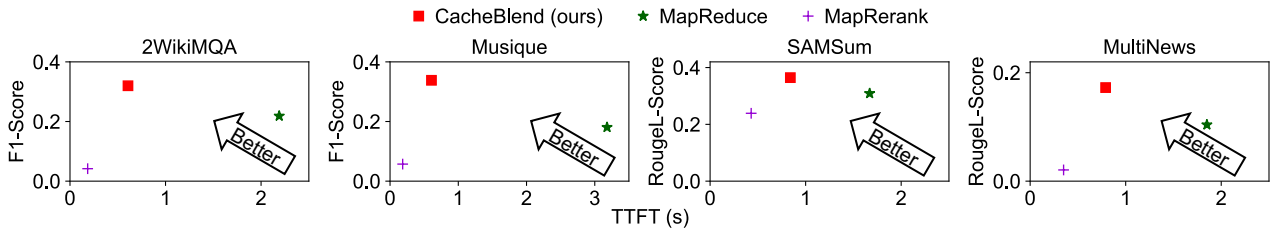


Figure 13. *Generation quality of CacheBlend with Yi-34B vs MapReduce and MapRerank.*

7 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.

7.1 Setup

Models and hardware settings: We evaluate CacheBlend on Mistral-7B[30], Yi-34B[56] and Llama-70B[2] to represent a wide scale of open source models. Note that we apply 8-bit model quantization to Llama-70B and Yi-34B. We run our end-to-end experiments on Runpod GPUs [10] with 128 GB RAM, 2 Nvidia A40 GPUs, and 1TB NVME SSD whose measured throughput is 4.8 GB/s. We use 1 GPU to serve Mistral-7B and Yi-34B, and 2 GPUs to serve Llama-70B.

Datasets: Our evaluation covers the following datasets.

- *2WikiMQA*[7] [27]: This dataset aims to test LLM's reasoning skills by requiring the model to read multiple paragraphs to answer a given question. We included 200 test cases, following the dataset size of previous work [12].
- *Musique*[7] [51]: This is a multi-document question-answering dataset. It is designated to test LLM's multi-hop reasoning ability where one reasoning step critically relies on information from another and contains 150 test cases.

[7]Since the standard answers for *2WikiMQA* and *Musique* are always less than 5 words, we append "Answer within 5 words." to their prompts to reduce the impact of answer length mismatch in F1 score calculation.

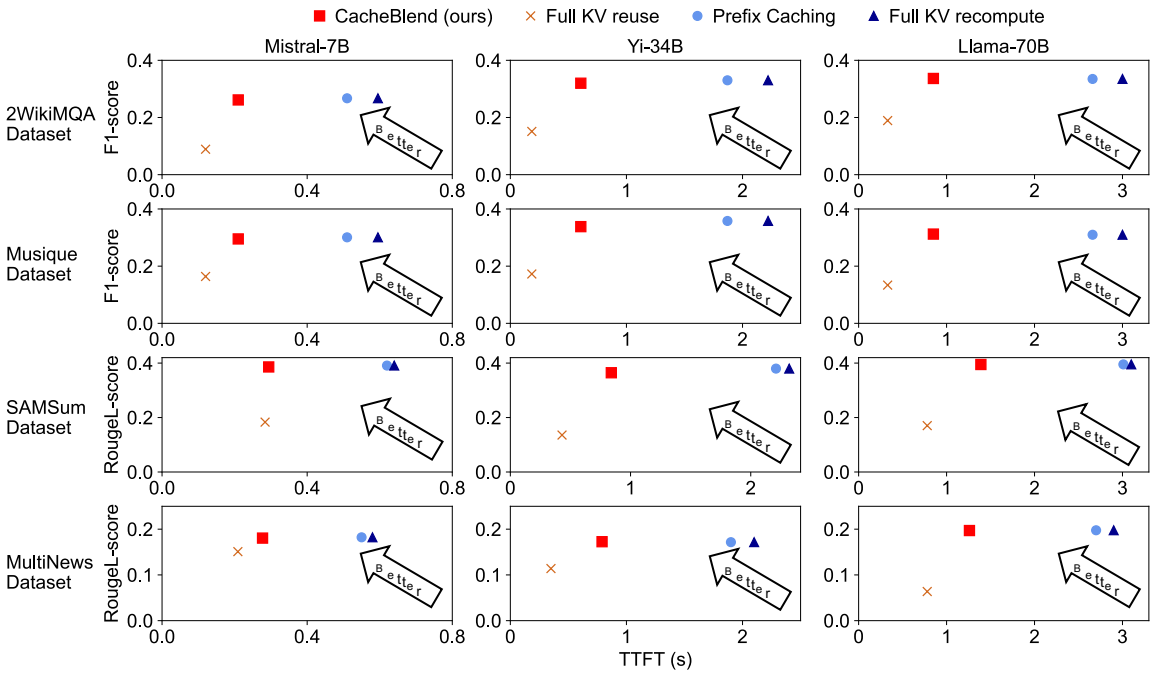


图12. 与完整的KV重新计算相比，CacheBlend将TTFT减少了2.2-3.3×，在四个数据集上质量下降微乎其微。 ets
和三个模型。

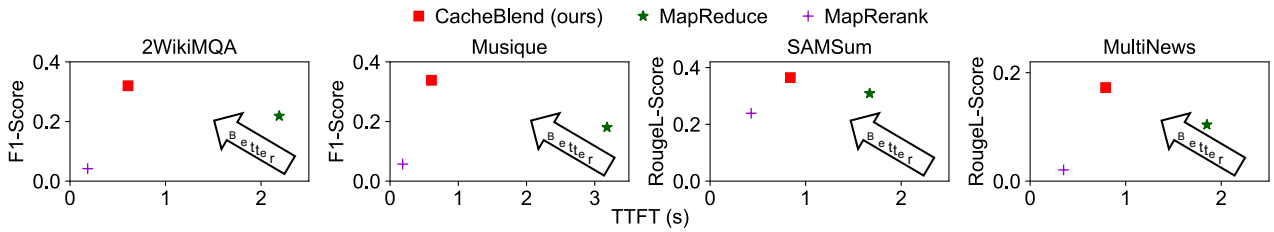


图13. CacheBlend与Yi-34B相比的生成质量， vs MapReduce和MapRerank。

7 评估

我们从评估中得到的主要结论是：

- TTFT 减少：与完整的 KV 重新计算相比，CacheBlend 在多个模型和任务中将 TTFT 减少了 2.2-3.3×。
- 高质量：与完全的KV重用相比，CacheBlend在F1分数和Rouge-L分数上将质量从0.15提高到0.35，同时与完全的KV重新计算和前缀缓存相比，质量下降不超过0.01-0.03。
- 更高的吞吐量：在相同的 TTFT 下，CacheBlend 可以将吞吐量提高至比完整 KV 重新计算高出 5×，比前缀缓存高出 3.3×。

7.1 设置

模型和硬件设置：我们在 Mistral-7B[30]、Yi-34B[56] 和 Llama-70B[2] 上评估 CacheBlend，以代表广泛的开源模型。请注意，我们应用 8 位模型量化到 Llama-70B 和 Yi-34B。我们在配备 128 GB RAM、2 个 Nvidia A40 GPU 和 1TB NVME SSD 的 Runpod GPU [10] 上进行端到端实验，其测得的吞吐量为 4.8 GB/s。我们使用 1 个 GPU 来服务 Mistral-7B 和 Yi-34B，使用 2 个 GPU 来服务 Llama-70B。

数据集：我们的评估涵盖以下数据集。

- 2WikiMQA[7] [27]: 该数据集旨在通过要求模型阅读多个段落来测试LLM的推理能力，以回答给定的问题。我们包含了200个测试案例，遵循之前工作的数据集大小[12]。
- Musique[7] [51]: 这是一个多文档问答数据集。它旨在测试大型语言模型的多跳推理能力，其中一个推理步骤在关键上依赖于来自另一个步骤的信息，并包含150个测试案例。

[7]Since the standard answers for *2WikiMQA* and *Musique* are always less than 5 words, we append "Answer within 5 words." to their prompts to reduce the impact of answer length mismatch in F1 score calculation.

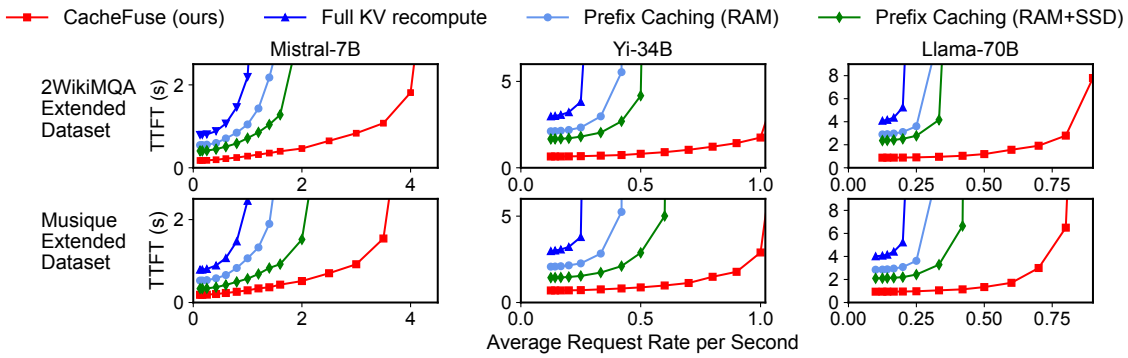


Figure 14. *CacheBlend achieves lower TTFT with higher throughput in RAG scenarios compared with baselines of similar quality.*

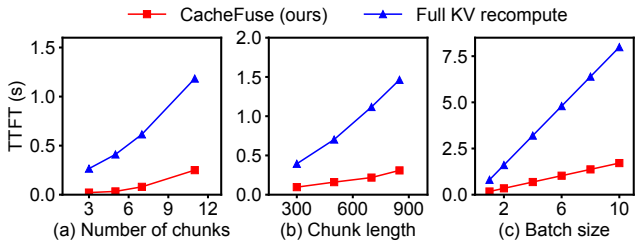


Figure 15. *CacheBlend outperforms baseline with varying chunk numbers, chunk lengths, and batch sizes.*

- *SAMSum* [25]: This dataset comprises multiple pairs of dialogues and summaries, and requires the LLM to output a summary to a new dialogue. It is intended to test the few-shot learning ability of language models and contains 200 test cases.
- *MultiNews* [20]: This dataset consists of news articles and human-written summaries of these articles from the site newser.com. Each summary is professionally written by editors and includes links to the original articles cited and contains 60 sampled cases.

We split contexts into 512-token chunks with Langchain and use the original 200-400 token chunks in SAMSum. We also create a synthetic dataset to simulate the chunk reuse in RAG scenarios. Specifically, we randomly pick 1500 queries in the Musique and 2WikiMQA datasets each and build a context chunk database by splitting each query's context into 512-token chunks [29] with Langchain [5]. For each query, we use GPT4 API to generate 3 more similar queries. In the 6000 queries (1500 original + 4500 simulated), we retrieve the top-6 chunks[8] based on L2 distance, in a random order[34]. We refer to these datasets as *Musique extended* and *2WikiMQA extended*. We only report for baselines with similar quality and skip the result for the first 1K queries as the initial storage is completely empty.

Quality metrics: We adopt the following standard metrics to measure the generation quality.

- *F1-score [6]* is used to evaluate *2WikiMQA* and *Musique* datasets [12]. It measures the similarity between the model's output and the ground-truth answer of the question based on the number of overlapping words.
- *Rouge-L score [39]* is used to evaluate *MultiNews* and *SAMSum* datasets [12]. It measures the similarity between the model's output and the ground-truth summaries based on the longest common sequence.

Baselines: We compare CacheBlend with the following baselines:

- *Full KV recompute*: The raw texts are fed into LLM as input. The LLM calculates KV cache of all tokens during prefill.
- *Prefix caching* [33, 36, 59]: We adopt the techniques from SGLang [59] to identify the frequently used prefix chunks and store their KV caches in both RAM and SSD. The KV cache of non-prefix tokens needs to be computed during prefill. We also make an idealized assumption in favor of prefix caching that there is no loading delay from RAM or SSD to GPU. This assumption makes it perform better than it would under real-world conditions.
- *Full KV reuse* [24]: We implement full KV reuse by using the approach proposed in PromptCache [24]. We append a buffer before the text to prepare its KV cache to be used in different positions with correct positional encoding. We did not compare with the scaffolding scheme since its application to RAG scenarios requires human users to manually select important chunks at runtime.
- *MapReduce* [7]: Different from traditional MapReduce [18], this is an alternative RAG method in LangChain. The LLM first summarises all chunks in parallel and concatenates them together. The concatenated summaries are then fed to the LLM again to generate the final answer.
- *MapRerank* [8]: This is another RAG method in LangChain. In MapRerank, the LLM independently generates an answer from each chunk along with a score based on its confidence that the answer is correct. The answer with the highest score is picked as the final output.

[8]Max number of chunks that fit into input token limit for Llama-70B.

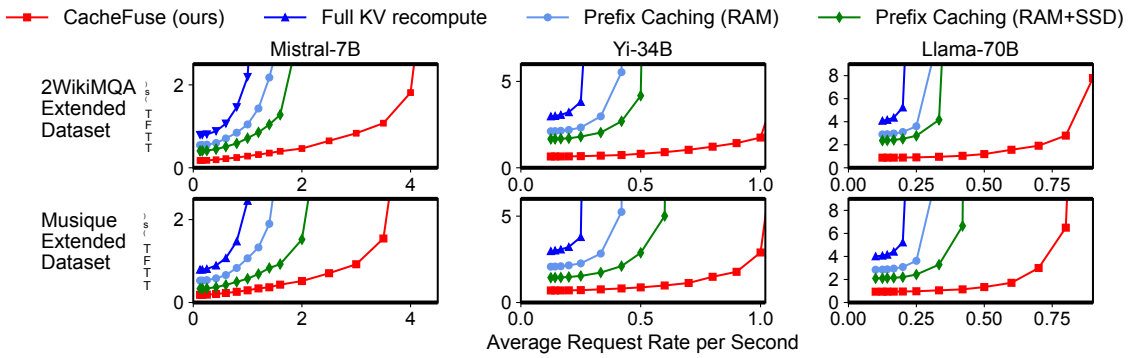


图14. 在RAG场景中，CacheBlend相比于相似质量的基线实现了更高的吞吐量和更低的TTFT。

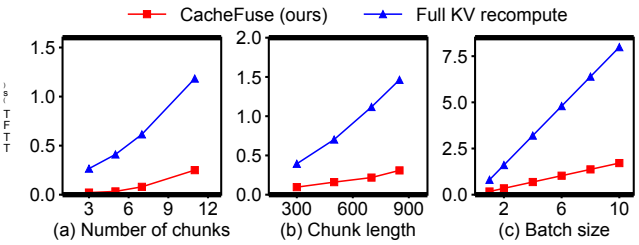


图15. CacheBlend在不同的块数量、块长度和批量大小下优于基线。

- SAMSum [25]：该数据集包含多个对话和摘要的配对，要求LLM为新的对话输出摘要。它旨在测试语言模型的少量学习能力，并包含200个测试案例。

- MultiNews [20]：该数据集由来自 newser.com 网站的新闻文章及其人工撰写的摘要组成。每个摘要均由编辑专业撰写，并包含引用的原始文章链接，共包含 60 个抽样案例。

我们将上下文分成512个标记的块，使用Langchain，并在SAMSum中使用原始的200-400个标记的块。我们还创建了一个合成数据集，以模拟RAG场景中的块重用。具体来说，我们在Musique和2WikiMQA数据集中随机选择1500个查询，并通过将每个查询的上下文分成512个标记的块[29]来构建上下文块数据库，使用Langchain [5]。对于每个查询，我们使用GPT4 API生成3个更相似的查询。在这6000个查询中（1500个原始+和4500个模拟），我们根据L2距离以随机顺序检索前6个块[8]。我们将这些数据集称为Musique扩展和2WikiMQA扩展。我们仅报告质量相似的基线，并跳过前1000个查询的结果，因为初始存储完全为空。

质量指标：我们采用以下标准指标来衡量生成质量。

- F1-score [6] 用于评估 2WikiMQA 和 Musique 数据集 [12]。它根据重叠单词的数量来衡量模型输出与问题的真实答案之间的相似性。

- Rouge-L 分数 [39] 用于评估 MultiNews 和 SAM-Sum 数据集 [12]。它基于最长公共序列测量模型输出与真实摘要之间的相似性。

基准：我们将CacheBlend与以下基准进行比较：

- 完整的 KV 重新计算：原始文本作为输入被输入到 LLM 中。LLM 在预填充期间计算所有标记的 KV 缓存。
- 前缀缓存 [33, 36, 59]：我们采用 SGLang [59] 的技术来识别频繁使用的前缀块，并将它们的 KV 缓存存储在 RAM 和 SSD 中。非前缀令牌的 KV 缓存需要在预填充期间计算。我们还做了一个理想化的假设，以支持前缀缓存，即从 RAM 或 SSD 到 GPU 没有加载延迟。这个假设使其表现优于现实条件下的表现。

- 完整的KV重用[24]：我们通过使用PromptCache[24]中提出的方法实现完整的KV重用。我们在文本前添加一个缓冲区，以准备其KV缓存，以便在不同位置使用正确的位置编码。我们没有与支架方案进行比较，因为其在RAG场景中的应用需要人工用户在运行时手动选择重要的片段。

- MapReduce [7]：与传统的 MapReduce [18] 不同，这是一种在 LangChain 中的替代 RAG 方法。LLM 首先并行总结所有块并将它们连接在一起。然后，将连接的摘要再次输入 LLM 以生成最终答案。

- MapRerank [8]：这是LangChain中的另一种RAG方法。在MapRerank中，LLM独立地从每个块生成一个答案，并根据其对答案正确性的信心生成一个分数。得分最高的答案被选为最终输出。

[8]Max number of chunks that fit into input token limit for Llama-70B.

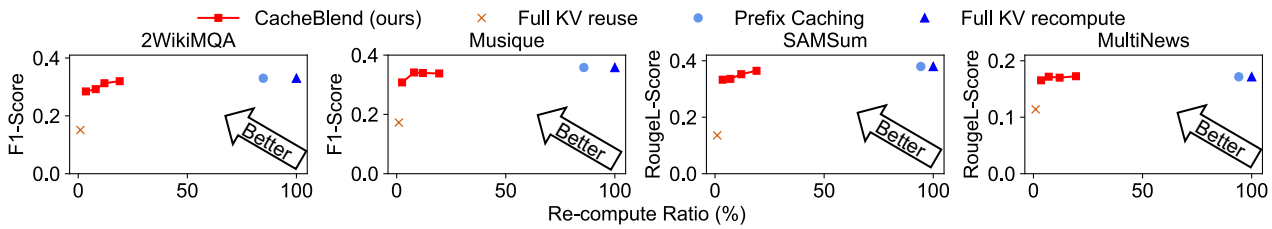


Figure 16. *CacheBlend has minimal loss in quality compared with full KV recompute, with 5%–18% selective recompute ratio, with Yi-34B.*

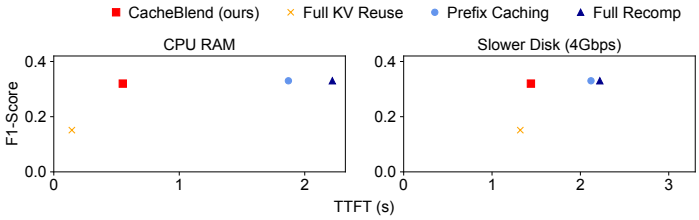


Figure 17. *CacheBlend's outperforms baselines when using RAM and slower disks*

7.2 Overall Improvement

Reduced TTFT with minimal quality drop: Figure 12 compares the average quality and TTFT across the requests, where each request has a context of 6 top chunks picked by lowest L2-distance between the respective embeddings generated by SentenceTransformers [49] (512 tokens per chunk). As shown in the graph, compared to the full KV recompute and prefix caching, CacheBlend's reduction in F1 and Rouge-L score is within 0.02, while it significantly reduces the TTFT by 2.2-3.3× across all models and datasets. While CacheBlend is slower than full KV reuse due to its selective recomputation, its quality stably outperforms full KV reuse by a large margin (in many cases more than 2×).

Figure 13 compares CacheBlend with RAG methods including MapReduce and MapRerank. Compared to MapReduce, CacheBlend has a 2-5× lower TTFT and higher F1 score.

Higher throughput with lower delay: In Figure 14, we compare CacheBlend with full KV recompute and prefix caching on Musique extended and 2WikiMQA datasets under different request rates. CacheBlend achieves lower delay with higher throughput by 2.8-5× than all the baselines across different models and datasets.

Understanding CacheBlend's improvement: CacheBlend is better than all baselines for different reasons. Compared to the full KV recompute, CacheBlend has a much lower delay and higher throughput due to only a small amount of tokens are recomputed. Compared to full KV reuse, although its delay is lower than CacheBlend, the quality drops a lot as full KV reuse did not perform any of the recompute, thus missing the cross-attention between different chunks. Compared to prefix caching, CacheBlend is also better in terms of higher throughput and lower delay as prefix caching needs to store *multiple versions* of KV caches for the same chunk if they have different prefixes. Thus, given the total storage space is fixed, prefix caching will incur a higher miss rate.

Finally, compared to other RAG methods, like MapReduce and MapRerank, CacheBlend is also better in terms of quality or delay. For MapReduce, it has a higher delay than CacheBlend due to additional LLM inference. Although MapRerank has slightly lower TTFT than CacheBlend, its quality is much worse, since processing the input chunks separately ignores the dependencies between chunks.

7.3 Sensitivity Analysis

For a better understanding of CacheBlend, we further analyze how varying the configurations impacts overall performance.

Varying chunk numbers and lengths: Figure 15a and 15b show the minimum compute time needed by CacheBlend to maintain generation quality (≤0.015 loss in F1-score) at different numbers of chunks and chunk lengths. The experiment is conducted on 2WikiMQA with Mistral-7B model. As shown in the figure, the compute time reduction ratio remains similar across different numbers of chunks and chunk length settings.

Varying recompute ratios: Figure 16 shows the impact of the recompute ratio on the quality-TTFT trade-off across all datasets on Yi-34B model. Across all the datasets, CacheBlend's loss in generation quality is at most 0.002 in F1 score or Rouge-L score compared to full KV recompute, with 5%~18% recomputation ratio. To put the number into context, the 5%~18% recomputation ratio can be translated to 4.1-6.6× TTFT reduction compared with full KV-recompute and 3.4-6.1× TTFT reduction compared with prefix caching.

Varying batch size: Figure 15c shows the compute time of prefill phase of different batch sizes. It is worth noting that the time of the decoding phase increases slower than the prefill phase when the batch size becomes larger [36, 60], making prefill overhead dominant with increasing batch size. Therefore, CacheBlend's improvement over the prefill phase becomes more prominent to the overall delay reduction with larger batch sizes.

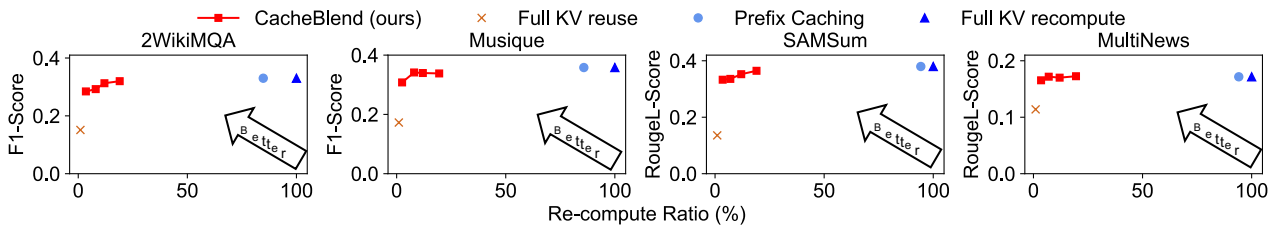


图16. 与完整的KV重新计算相比，CacheBlend在质量上损失最小，选择性重新计算比例为5%–18%，使用Yi-34B。

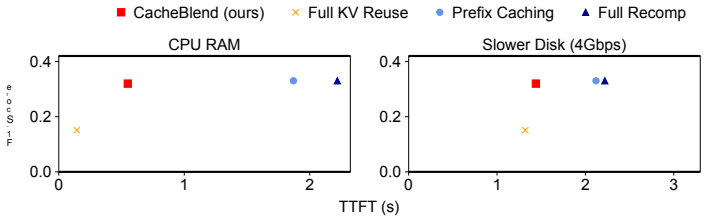


图17. 当使用RAM和较慢的磁盘时，CacheBlend的性能优于基线。

7.2 整体改进

减少TTFT且质量下降最小：图12比较了请求的平均质量和TTFT，其中每个请求的上下文为6个通过最低L2距离从SentenceTransformers [49]生成的各自嵌入中挑选的顶部块（每个块512个标记）。如图所示，与完整的KV重新计算和前缀缓存相比，CacheBlend在F1和Rouge-L分数上的减少在0.02以内，同时在所有模型和数据集上显著减少了TTFT，减少幅度为2.2-3.3×。虽然由于选择性重新计算，CacheBlend的速度比完整的KV重用慢，但其质量稳定地大幅超越完整的KV重用（在许多情况下超过2×）。

图13比较了CacheBlend与包括MapReduce和MapRerank在内的RAG方法。与MapReduce相比，CacheBlend的TTFT低2-5×，F1分数更高。

更高的吞吐量与更低的延迟：在图14中，我们比较了CacheBlend与全KV重新计算和前缀缓存在Musique扩展和2WikiMQA数据集下不同请求速率的表现。CacheBlend在不同模型和数据集上以2.8-5×的优势实现了更低的延迟和更高的吞吐量，优于所有基线。

理解 CacheBlend 的改进：CacheBlend 在不同的原因下优于所有基线。与完整的 KV 重新计算相比，CacheBlend 的延迟更低，吞吐量更高，因为只有少量的 token 被重新计算。与完整的 KV 重用相比，尽管其延迟低于 CacheBlend，但质量下降很多，因为完整的 KV 重用没有进行任何重新计算，从而错过了不同块之间的交叉注意力。与前缀缓存，CacheBlend 在更高的吞吐量和更低的延迟方面也表现更好，因为前缀缓存需要为相同的块存储多个版本的 KV 缓存，如果它们具有不同的前缀。因此，考虑到总存储空间是固定的，前缀缓存将导致更高的未命中率。

最后，与其他RAG方法相比，如MapReduce和MapRerank，CacheBlend在质量或延迟方面也更好。对于MapReduce，由于额外的LLM推理，它的延迟比CacheBlend更高。尽管MapRerank的TTFT略低于CacheBlend，但其质量要差得多，因为单独处理输入块忽略了块之间的依赖关系。

7.3 敏感性分析

为了更好地理解CacheBlend，我们进一步分析了不同配置如何影响整体性能。

变化的块数量和长度：图15a和15b显示了CacheBlend在不同块数量和块长度下维持生成质量所需的最小计算时间（≤0.015在F1分数中的损失）。实验是在2WikiMQA上使用Mistral-7B模型进行的。如图所示，计算时间的减少比例在不同块数量和块长度设置下保持相似。

变化的重新计算比率：图16显示了重新计算比率对Yi-34B模型在所有数据集上的质量-TTFT权衡的影响。在所有数据集中，与完整KV重新计算相比，CacheBlend在生成质量上的损失最多为0.002的F1分数或Rouge-L分数，重新计算比率为5%~18%。为了将这个数字放入上下文中，5%~18%的重新计算比率可以转化为与完整KV重新计算相比的4.1-6.6× TTFT减少，以及与前缀缓存相比的3.4-6.1× TTFT减少。

变化的批量大小：图15c显示了不同批量大小的预填充阶段的计算时间。值得注意的是，当批量大小增大时，解码阶段的时间增长速度比预填充阶段慢[36, 60]，这使得随着批量大小的增加，预填充的开销变得主导。因此，CacheBlend在预填充阶段的改进在较大批量大小下对整体延迟的减少变得更加显著。

Varying storage device: To study the effect of different storage types on CacheBlend, we modify the underlying storage devices for every method and conduct a similar experiment as Figure 12 for the Yi-34B model and 2WikiMQA dataset. As shown in Figure 17, CacheBlend consistently reduces TTFT with minimal quality degradation when the KV cache is stored in RAM or a slower SSD device. Notice that the delay gap between CacheBlend and Full KV reuse is smaller for slower storage since the delay of CacheBlend would be more dominated by the loading delay instead of its partial KV recomputation.

8 Related Work

Retrieval augmented generation (RAG): RAG [22, 23, 37, 46, 48] can enhance the accuracy and reliability of LLMs with text chunks fetched from external sources. However, processing these text chunks in the LLM can take a long time. CacheBlend reduces this overhead by storing and reusing the KV caches of these text chunks.

KV cache reuse across requests: Storing and reusing KV cache across different requests have been commonly studied in recent work [24, 33, 41, 42, 59]. Most of these works [33, 41, 42, 59] focus on prefix-only caching. PromptCache [24] allows KV cache to be reused at different positions but fails to maintain satisfying generation quality due to inaccurate postional encoding and ignorance of cross attention. CacheBlend adopts a novel partial recomputation framework to better retain positional accuracies and cross attention. Most of the existing work stores KV cache in volatile memory devices for guaranteed performance (e.g., GPU HBM, CPU DRAM). While there are emerging research trying to reuse high-speed NVME SSD for KV caches [21], CacheBlend is unique in pipelining loading with partial recomputation and its extension to even slower object store.

General-purpose LLM serving systems: Numerous general-purpose LLM serving systems have been developed [11, 36, 57, 60]. Orca [57] enables multiple requests to be processed in parallel with iteration-level scheduling. vLLM [36] further increases the parallelsim through more efficent GPU memory management. CacheBlend is complementary to these general-purpose LLM serving systems, empowering them with context resuing capabilities.

Context compression methods: Context compression techniques [19, 31, 32, 43, 55, 58] can be complementary to CacheBlend. Some of these techniques [31, 32] shorten the prompt length by prunining the unimportant tokens. CacheBlend is compatible with such methods in that it can take different chunk lengths as shown in §7.3. Another line of work [19, 43, 58] focus on dropping the unimportant KV vectors based on the attention matrix, which essentially reduce the KV cache size. CacheBlend can benifit from such techniques by storing and loading less KV cache.

9 Limitations and Future Work

Our method in this paper (e.g., the insights in § 4.3) currently only applies to language models with transformer structures. We leave investigation of architectures other than transformer such as Mamba [26] and Griffin [17] for future work. In our evaluation, we haven't tested CacheBlend's performance on more models and datasets with different quantization settings. For better understanding and improvement of the method, we have open-sourced our work to facilitate more effort in this direction.

In this paper, we integrated CacheBlend in vLLM but have not yet tested CacheBlend's performance on the latest serving engines like Distserve[60] or StableGen [11]. Nor have we studied how to apply CacheBlend to workloads that share KV cache across different compute nodes. Since CacheBlend is able to reduce the costly prefill phase, we believe combining CacheBlend with these new serving engines could potentially bring more savings. We leave the integration of CacheBlend into these novel inference frameworks for future work.

10 Conclusion

We present CacheBlend, a system that combines multiple pre-computed KV caches, when their corresponding texts are concatenated in the LLM input. To preserve generation quality, CacheBlend recovers the cross-attention among these texts by selectively recomputing the KV cache values of a small fraction of tokens. Through experiments across four datasets and three models, CacheBlend reduces TTFT by 2.2-3.3× and increases throughput by 2.8-5×, compared to full KV recompute, under negligible quality drop. The code is available at **https://github.com/LMCache/LMCache**.

11 Acknowledgement

We thank all the anonymous reviewers and our shepherd, Thaleia Dimitra Doudali, for their insightful feedback and suggestions. The project is funded by NSF CNS-2146496, CNS-2131826, CNS-2313190, CNS-1901466, CNS-2313190, CCF-2119184, CNS-1956180, and research awards from Google, CERES Center, Conviva, and Chameleon Cloud.

References

[1] 12 Practical Large Language Model (LLM) Applications - Techopedia. https://www.techopedia.com/12-practical-large-language-model-llm-applications. (Accessed on 09/21/2023).

[2] [2302.13971] llama: Open and efficient foundation language models. https://arxiv.org/abs/2302.13971. (Accessed on 09/21/2023).

[3] 7 top large language model use cases and applications. https://www.projectpro.io/article/large-language-model-use-cases-and-applications/887. (Accessed on 09/21/2023).

[4] Applications of large language models - indata labs. https://indatalabs.com/blog/large-language-model-apps. (Accessed on 09/21/2023).

变化的存储设备：为了研究不同存储类型对CacheBlend的影响，我们为每种方法修改底层存储设备，并进行与图12类似的实验，使用Yi-34B模型和2WikiMQA数据集。如图17所示，当KV缓存存储在RAM或较慢的SSD设备中时，CacheBlend始终以最小的质量下降减少TTFT。请注意，对于较慢的存储，CacheBlend与完全KV重用之间的延迟差距较小，因为CacheBlend的延迟更受加载延迟的影响，而不是其部分KV的重新计算。

8 相关工作

检索增强生成（RAG）：RAG [22, 23, 37, 46, 48] 可以通过从外部来源获取的文本块提高大型语言模型（LLMs）的准确性和可靠性。然而，在LLM中处理这些文本块可能需要很长时间。CacheBlend通过存储和重用这些文本块的KV缓存来减少这种开销。

KV缓存在请求之间的重用：在最近的研究中，跨不同请求存储和重用KV缓存已被广泛研究[24, 33, 41, 42, 59]。这些研究中的大多数[33, 41, 42, 59]专注于仅前缀缓存。Prompt-Cache [24]允许在不同位置重用KV缓存，但由于位置编码不准确和忽视交叉注意力，未能保持令人满意的生成质量。CacheBlend采用了一种新颖的部分重新计算框架，以更好地保留位置准确性和交叉注意力。现有的大多数工作将KV缓存存储在易失性内存设备中以保证性能（例如，GPU HBM，CPU DRAM）。虽然有新兴研究尝试将高速NVME SSD用于KV缓存[21]，但CacheBlend在将加载与部分重新计算流水线化及其扩展到更慢的对象存储方面是独特的。

通用 LLM 服务系统：已经开发了许多通用 LLM 服务系统 [11, 36, 57, 60]。Orca [57] 通过迭代级调度使多个请求能够并行处理。vLLM [36] 通过更高效的 GPU 内存管理进一步提高了并行性。CacheBlend 是这些通用 LLM 服务系统的补充，赋予它们上下文重用的能力。

上下文压缩方法：上下文压缩技术 [19, 31, 32, 43, 55, 58] 可以与 CacheBlend 互补。其中一些技术 [31, 32] 通过修剪不重要的标记来缩短提示长度。CacheBlend 与这些方法兼容，因为它可以采用不同的块长度，如 §7.3 所示。另一项工作 [19, 43, 58] 侧重于基于注意力矩阵丢弃不重要的 KV 向量，这本质上减少KV缓存大小。CacheBlend可以通过存储和加载更少的KV缓存来受益于此类技术。

9 限制与未来工作

我们在本文中的方法（例如，第4.3节中的见解）目前仅适用于具有变换器结构的语言模型。我们将对变换器以外的架构（如Mamba [26]和Griffin [17]）的研究留待未来工作。在我们的评估中，我们尚未测试CacheBlend在不同量化设置下对更多模型和数据集的性能。为了更好地理解和改进该方法，我们已将我们的工作开源，以促进在这一方向上的更多努力。

在本文中，我们将CacheBlend集成到vLLM中，但尚未测试CacheBlend在最新服务引擎如Distserve[60]或StableGen [11]上的性能。我们也没有研究如何将CacheBlend应用于跨不同计算节点共享KV缓存的工作负载。由于CacheBlend能够减少昂贵的预填充阶段，我们相信将CacheBlend与这些新的服务引擎结合可能会带来更多的节省。我们将CacheBlend集成到这些新颖推理框架的工作留待未来进行。

10 结论

我们介绍了CacheBlend，一个系统，它结合了多个预计算的KV缓存，当它们对应的文本在LLM输入中连接时。为了保持生成质量，CacheBlend通过选择性地重新计算少量标记的KV缓存值，恢复这些文本之间的交叉注意力。通过在四个数据集和三个模型上的实验，CacheBlend将TTFT减少了2.2-3.3×，并在质量下降可忽略不计的情况下将吞吐量提高了2.8-5×，与完全的KV重新计算相比。代码可在https://github.com/LMCache/LMCache获取。

11 感谢

我们感谢所有匿名评审和我们的指导者Thaleia Dimitra Doudali，感谢他们的深刻反馈和建议。该项目由NSF CNS-2146496、CNS-2131826、CNS-2313190、CNS-1901466、CNS-2313190、CCF-2119184、CNS-1956180以及来自Google、CERES中心、Conviva和Chameleon Cloud的研究奖项资助。

参考文献

[1] 12个实用的大型语言模型（LLM）应用 - Techopedia. https://www.techopedia.com/12-practical-large-language-model-llm-applications.（访问日期：2023年9月21日）。 [2] [2302.13971] llama：开放且高效的基础语言模型. https://arxiv.org/abs/2302.13971.（访问日期：2023年9月21日）。 [3] 7个顶级大型语言模型的用例和应用. https://www.projectpro.io/article/large-language-model-use-cases-and-applications/887.（访问日期：2023年9月21日）。 [4] 大型语言模型的应用 - indata labs. https://indatalabs.com/blog/large-language-model-apps.（访问日期：2023年9月21日）。

[5] Chains. https://python.langchain.com/docs/modules/chains/.

[6] Evaluating qa: Metrics, predictions, and the null response. https://github.com/fastforwardlabs/ff14_blog/blob/master/_notebooks/2020-06-09-Evaluating_BERT_on_SQuAD.ipynb.

[7] Langchain: Map reduce. https://api.python.langchain.com/en/latest/chains/langchain.chains.combine_documents.map_reduce.MapReduceDocumentsChain.html#langchain.chains.combine_documents.map_reduce.MapReduceDocumentsChain.

[8] Langchain: Map rerank. https://api.python.langchain.com/en/latest/chains/langchain.chains.combine_documents.map_rerank.MapRerankDocumentsChain.html.

[9] Real-world use cases for large language models (llms) | by cellstrat | medium. https://cellstrat.medium.com/real-world-use-cases-for-large-language-models-llms-d71c3a577bf2. (Accessed on 09/21/2023).

[10] Runpod: Cloud compute made easy. https://www.runpod.io/, 2024. Accessed: 2024-05-21.

[11] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills, 2023.

[12] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*, 2023.

[13] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[14] Beidi Chen, Tri Dao, Eric Winsor, Zhao Song, Atri Rudra, and Christopher Ré. Scatterbrain: Unifying sparse and low-rank attention. *Advances in Neural Information Processing Systems*, 34:17413–17426, 2021.

[15] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794*, 2020.

[16] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[17] Soham De, Samuel L. Smith, Anushan Fernando, Aleksandar Botev, George Cristian-Muraru, Albert Gu, Ruba Haroun, Leonard Berrada, Yutian Chen, Srivatsan Srinivasan, Guillaume Desjardins, Arnaud Doucet, David Budden, Yee Whye Teh, Razvan Pascanu, Nando De Freitas, and Caglar Gulcehre. Griffin: Mixing gated linear recurrences with local attention for efficient language models, 2024.

[18] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[19] Harry Dong, Xinyu Yang, Zhenyu Zhang, Zhangyang Wang, Yuejie Chi, and Beidi Chen. Get more with less: Synthesizing recurrence with kv cache compression for efficient llm inference. *arXiv preprint arXiv:2402.09398*, 2024.

[20] Alexander R Fabbri, Irene Li, Tianwei She, Suyi Li, and Dragomir R Radev. Multi-news: A large-scale multi-document summarization dataset and abstractive hierarchical model. *arXiv preprint arXiv:1906.01749*, 2019.

[21] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Attentionstore: Cost-effective attention reuse across multi-turn conversations in large language model serving, 2024.

[22] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. Retrieval-augmented generation for large language models: A survey, 2023.

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

[24] In Gim, Guojun Chen, Seung seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference, 2023.

[25] Bogdan Gliwa, Iwona Mochol, Maciej Biesek, and Aleksander Wawer. Samsum corpus: A human-annotated dialogue dataset for abstractive summarization. *arXiv preprint arXiv:1911.12237*, 2019.

[26] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces, 2023.

[27] Xanh Ho, Anh-Khoa Duong Nguyen, Saku Sugawara, and Akiko Aizawa. Constructing a multi-hop qa dataset for comprehensive evaluation of reasoning steps. *arXiv preprint arXiv:2011.01060*, 2020.

[28] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *arXiv preprint arXiv:2401.18079*, 2024.

[29] Muhammad Jan. Optimize rag efficiency with llamaindex: The perfect chunk size. https://datasciencedojo.com/blog/rag-with-llamaindex/, october 2023.

[30] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[31] Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Llmlingua: Compressing prompts for accelerated inference of large language models. *arXiv preprint arXiv:2310.05736*, 2023.

[32] Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression, 2023.

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

[34] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs, 2017.

[35] Hao Kang, Qingru Zhang, Souvik Kundu, Geonhwa Jeong, Zaoxing Liu, Tushar Krishna, and Tuo Zhao. Gear: An efficient kv cache compression recipe for near-lossless generative inference of llm, 2024.

[36] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[37] Huayang Li, Yixuan Su, Deng Cai, Yan Wang, and Lemao Liu. A survey on retrieval-augmented text generation. *arXiv preprint arXiv:2202.01110*, 2022.

[38] Chaofan Lin, Chengruidong Zhang Zhenhua Han, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of llm-based applications with semantic variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, Santa Clara, CA, July 2024. USENIX Association.

[39] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics.

[40] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172*, 2023.

[41] Shu Liu, Asim Biswal, Audrey Cheng, Xiangxi Mo, Shiyi Cao, Joseph E. Gonzalez, Ion Stoica, and Matei Zaharia. Optimizing llm queries in relational workloads, 2024.

[42] Yuhan Liu, Hanchen Li, Kuntai Du, Jiayi Yao, Yihua Cheng, Yuyang Huang, Shan Lu, Michael Maire, Henry Hoffmann, Ari Holtzman, et al. Cachegen: Fast context loading for language model applications. *arXiv*

[5] 链接。 https://python.langchain.com/docs/modules/chains/. [6] 评估 qa：指标、预测和空响应。 https://github.com/fastforwardlabs/ff14_blog/blob/master/_notebooks/2020-06-09-Evaluating_BERT_on_SQuAD.ipynb. [7] Langchain：映射归约。 https://api.python.langchain.com/en/latest/chains/langchain.chains.combine_documents.map_reduce.MapReduceDocumentsChain.html#langchain.chains.combine_documents.map_reduce.MapReduceDocumentsChain. [8] Langchain：映射重排序。 https://api.python.langchain.com/en/latest/chains/langchain.chains.combine_documents.map_rerank.MapRerankDocumentsChain.html. [9] 大型语言模型（llms）的真实世界用例 | 由 cellstrat | medium。 https://cellstrat.medium.com/real-world-use-cases-for-large-language-models-llms-d71c3a577bf2.（访问时间：2023年9月21日）。 [10] Runpod：简化的云计算。 https://www.runpod.io/，2024。访问时间：2024年5月21日。 [11] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani 和 Ramachandran Ramjee。 Sarathi：通过搭载解码与分块预填充实现高效的 llm 推理，2023。 [12] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang 和 Juanzi Li。 Longbench：一个用于长上下文理解的双语多任务基准。 arXiv 预印本 arXiv:2308.14508，2023。 [13] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell 等。语言模型是少量学习者。神经信息处理系统进展，33:1877–1901，2020。 [14] Beidi Chen, Tri Dao, Eric Winsor, Zhao Song, Atri Rudra 和 Christopher Ré。Scatterbrain：统一稀疏和低秩注意力。神经信息处理系统进展，34:17413–17426，2021。 [15] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser 等。重新思考与表演者的注意力。arXiv 预印本 arXiv:2009.14794，2020。 [16] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann 等。Palm：通过路径扩展语言建模。arXiv 预印本 arXiv:2204.02311，2022。 [17] Soham De, Samuel L. Smith, Anushan Fernando, Aleksandar Botev, George Cristian-Muraru, Albert Gu, Ruba Haroun, Leonard Berrada, Yutian Chen, Srivatsan Srinivasan, Guillaume Desjardins, Arnaud Doucet, David Budden, Yee Whye Teh, Razvan Pascanu, Nando De Freitas 和 Caglar Gulcehre。Griffin：将门控线性递归与局部注意力混合以实现高效的语言模型，2024。 [18] Jeffrey Dean 和 Sanjay Ghemawat。Mapreduce：在大型集群上简化数据处理。ACM 通讯，51(1):107–113，2008。 [19] Harry Dong, Xinyu Yang, Zhenyu Zhang, Zhangyang Wang, Yuejie Chi 和 Beidi Chen。以更少的获取更多：通过 kv 缓存压缩合成递归以实现高效的 llm 推理。arXiv 预印本 arXiv:2402.09398，2024。 [20] Alexander R Fabbri, Irene Li, Tianwei She, Suyi Li 和 Dragomir R Radev。Multi-news：一个大规模多文档摘要数据集和抽象层次模型。arXiv 预印本 arXiv:1906.01749，2019。 [21] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu 和 Pengfei Zuo。Attentionstore：在大型语言模型服务中跨多轮对话的成本效益注意力重用，2024。 [22] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun 和 Haofen Wang。针对大型语言模型的检索增强生成：一项调查，2023。 [23] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun 和 Haofen Wang。检索增强生成

对于大型语言模型的调查。arXiv 预印本 arXiv:2312.10997，2023年。[24] 在Gim，Guojun Chen，Seung seob Lee，Nikhil Sarda，Anurag Khandelwal和Lin Zhong的研究中。提示缓存：低延迟推理的模块化注意力重用，2023年。[25] Bogdan Gliwa，Iwona Mochol，Maciej Biesek和Aleksander Wawer。Samsum语料库：用于抽象摘要的人类注释对话数据集。arXiv 预印本 arXiv:1911.12237，2019年。[26] Albert Gu和Tri Dao。Mamba：具有选择性状态空间的线性时间序列建模，2023年。[27] Xanh Ho，Anh-Khoa Duong Nguyen，Saku Sugawara和Akiko Aizawa。构建多跳问答数据集以全面评估推理步骤。arXiv 预印本 arXiv:2011.01060，2020年。[28] Coleman Hooper，Sehoon Kim，Hiva Mohammadzadeh，Michael W Mahoney，Yakun Sophia Shao，Kurt Keutzer和Amir Gholami。Kvquant：通过kv缓存量化实现1000万上下文长度的llm推理。arXiv 预印本 arXiv:2401.18079，2024年。[29] Muhammad Jan。使用llamaindex优化rag效率：完美的块大小。https://datasciencedojo.com/blog/rag-with-llamaindex/，2023年10月。[30] Albert Q Jiang，Alexandre Sablayrolles，Arthur Mensch，Chris Bamford，Devendra Singh Chaplot，Diego de las Casas，Florian Bressand，Gianna Lengyel，Guillaume Lample，Lucile Saulnier等。Mistral 7b。arXiv 预印本 arXiv:2310.06825，2023年。[31] Huiqiang Jiang，Qianhui Wu，Chin-Yew Lin，Yuqing Yang和Lili Qiu。Llmlingua：压缩提示以加速大型语言模型的推理。arXiv 预印本 arXiv:2310.05736，2023年。[32] Huiqiang Jiang，Qianhui Wu，Xufang Luo，Dongsheng Li，Chin-Yew Lin，Yuqing Yang和Lili Qiu。Longllmlingua：通过提示压缩加速和增强长上下文场景中的llm，2023年。[33] Chao Jin，Zili Zhang，Xuanlin Jiang，Fangyue Liu，Xin Liu，Xuanzhe Liu和Xin Jin。Ragcache：高效的知识缓存用于检索增强生成。arXiv 预印本 arXiv:2404.12457，2024年。[34] Jeff Johnson，Matthijs Douze和Hervé Jégou。使用GPU进行十亿规模的相似性搜索，2017年。[35] Hao Kang，Qingru Zhang，Souvik Kundu，Geonhwa Jeong，Zaoxing Liu，Tushar Krishna和Tuo Zhao。Gear：一种高效的kv缓存压缩方案，用于近无损的llm生成推理，2024年。[36] Woosuk Kwon，Zhuohan Li，Siyuan Zhuang，Ying Sheng，Lianmin Zheng，Cody Hao Yu，Joseph Gonzalez，Hao Zhang和Ion Stoica。使用pagedattention进行大型语言模型服务的高效内存管理。在第29届操作系统原理研讨会论文集中，页码611–626，2023年。[37] Huayang Li，Yixuan Su，Deng Cai，Yan Wang和Lemao Liu。关于检索增强文本生成的调查。arXiv 预印本 arXiv:2202.01110，2022年。[38] Chaofan Lin，Chengruidong Zhang Zhenhua Han，Yuqing Yang，Fan Yang，Chen Chen和Lili Qiu。Parrot：使用语义变量高效服务基于llm的应用。在第18届USENIX操作系统设计与实现研讨会（OSDI 24），加利福尼亚州圣克拉拉，2024年7月。USENIX协会。[39] Chin-Yew Lin。ROUGE：用于自动评估摘要的包。在文本摘要分支中，页码74–81，西班牙巴塞罗那，2004年7月。计算语言学协会。[40] Nelson F Liu，Kevin Lin，John Hewitt，Ashwin Paranjape，Michele Bevilacqua，Fabio Petroni和Percy Liang。迷失在中间：语言模型如何使用长上下文。arXiv 预印本 arXiv:2307.03172，2023年。[41] Shu Liu，Asim Biswal，Audrey Cheng，Xiangxi Mo，Shiyi Cao，Joseph E. Gonzalez，Ion Stoica和Matei Zaharia。优化关系工作负载中的llm查询，2024年。[42] Yuhan Liu，Hanchen Li，Kuntai Du，Jiayi Yao，Yihua Cheng，Yuyang Huang，Shan Lu，Michael Maire，Henry Hoffmann，Ari Holtzman等。Cachegen：语言模型应用的快速上下文加载。arXiv

preprint arXiv:2310.07240, 2023.

[43] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *Advances in Neural Information Processing Systems*, 36, 2024.

[44] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, et al. Deja vu: Contextual sparsity for efficient llms at inference time. In *International Conference on Machine Learning*, pages 22137–22176. PMLR, 2023.

[45] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*, 2024.

[46] Yuning Mao, Pengcheng He, Xiaodong Liu, Yelong Shen, Jianfeng Gao, Jiawei Han, and Weizhu Chen. Generation-augmented retrieval for open-domain question answering. *arXiv preprint arXiv:2009.08553*, 2020.

[47] Jason Phang, Haokun Liu, and Samuel R Bowman. Fine-tuned transformers show clusters of similar representations across layers. *arXiv preprint arXiv:2109.08406*, 2021.

[48] Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. In-context retrieval-augmented language models. *Transactions of the Association for Computational Linguistics*, 11:1316–1331, 2023.

[49] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019.

[50] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

[51] Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. Musique: Multihop questions via single-hop question composition, 2022.

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, 2023.

[53] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism, 2024.

[54] Peng Xu, Wei Ping, Xianchao Wu, Lawrence McAfee, Chen Zhu, Zihan Liu, Sandeep Subramanian, Evelina Bakhturina, Mohammad Shoeybi, and Bryan Catanzaro. Retrieval meets long context large language models. *arXiv preprint arXiv:2310.03025*, 2023.

[55] Wangsong Yin, Mengwei Xu, Yuanchun Li, and Xuanzhe Liu. Llm as a system service on mobile devices, 2024.

[56] Alex Young, Bei Chen, Chao Li, Chengen Huang, Ge Zhang, Guanwei Zhang, Heng Li, Jiangcheng Zhu, Jianqun Chen, Jing Chang, et al. Yi: Open foundation models by 01. ai. *arXiv preprint arXiv:2403.04652*, 2024.

[57] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[58] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[59] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Efficiently programming large language models using sglang. *arXiv preprint arXiv:2312.07104*, 2023.

[60] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *arXiv preprint arXiv:2401.09670*, 2024.

预印本 arXiv:2310.07240, 2023. [43] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis 和 Anshumali Shrivastava. Scissorhands: 利用重要性假设的持久性进行 LLM KV 缓存压缩的测试时间. 神经信息处理系统进展, 36, 2024. [44] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re 等. Déjà vu: 在推理时高效 LLM 的上下文稀疏性. 在国际机器学习会议上, 页码 22137–22176. PMLR, 2023. [45] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen 和 Xia Hu. Kivi: 一种无调优的非对称 2 位量化用于 KV 缓存. arXiv 预印本 arXiv:2402.02750, 2024. [46] Yuning Mao, Pengcheng He, Xiaodong Liu, Yelong Shen, Jianfeng Gao, Jiawei Han 和 Weizhu Chen. 生成增强检索用于开放域问答. arXiv 预印本 arXiv:2009.08553, 2020. [47] Jason Phang, Haokun Liu 和 Samuel R Bowman. 微调的变换器在各层之间显示出相似表示的聚类. arXiv 预印本 arXiv:2109.08406, 2021. [48] Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown 和 Yoav Shoham. 上下文检索增强语言模型. 计算语言学协会会刊, 11:1316–1331, 2023. [49] Nils Reimers 和 Iryna Gurevych. Sentence-bert: 使用孪生 BERT 网络的句子嵌入. 在 2019 年自然语言处理实证方法会议的论文集中. 计算语言学协会, 2019 年 11 月. [50] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo 和 Yunfeng Liu. Roformer: 带有旋转位置嵌入的增强变换器. 神经计算, 568:127063, 2024. [51] Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot 和 Ashish Sabharwal. Musique: 通过单跳问题组合的多跳问题, 2022. [52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser 和 Illia Polosukhin. 注意力是你所需要的一切, 2023. [53] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu 和 Xin Jin. Loongserve: 通过弹性序列并行高效服务长上下文大型语言模型, 2024. [54] Peng Xu, Wei Ping, Xianchao Wu, Lawrence McAfee, Chen Zhu, Zihan Liu, Sandeep Subramanian, Evelina Bakhturina, Mohammad Shoeybi 和 Bryan Catanzaro. 检索与长上下文大型语言模型相遇. arXiv 预印本 arXiv:2310.03025, 2023. [55] Wangsong Yin, Mengwei Xu, Yuanchun Li 和 Xuanzhe Liu. LLM 作为移动设备上的系统服务, 2024. [56] Alex Young, Bei Chen, Chao Li, Chengen Huang, Ge Zhang, Guanwei Zhang, Heng Li, Jiangcheng Zhu, Jianqun Chen, Jing Chang 等. Yi: 由 01.ai 提供的开放基础模型. arXiv 预印本 arXiv:2403.04652, 2024. [57] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim 和 Byung-Gon Chun. Orca: 一种用于 {基于变换器的}生成模型的分布式服务系统. 在第 16 届 USENIX 操作系统设计与实现研讨会 (OSDI 22) 上, 页码 521–538, 2022. [58] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett 等. H2o: 用于大型语言模型高效生成推理的重型命中预言机. 神经信息处理系统进展, 36, 2024. [59] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez 等人。使用 sglang 高效编程大型语言模型。arXiv 预印本 arXiv:2312.07104，2023。[60] 邹尹敏，刘胜宇，陈俊达，胡建波，朱怡博，刘旭展，金鑫，张浩。Distserve：为优化良率的大型语言模型服务分离预填充和解码。arXiv 预印本 arXiv:2401.09670，2024。

A N-dimensional positional recovery

Here we prove our positional recovery method can work in the N-dimensional scenario. We start with the definition of RoPE in N-dimensional space.

Definition 1 (Rotary Positional Encoding, ROPE[50]). *Let vectors $q, k \in \mathbb{R}^d$ denote the query vector and key vector need to be embedded at some position m as $q_m, k_m \in \mathbb{R}^d$. Rope encodes the positional information as the following:*

$$q_m, k_m = \mathbb{R}^d_{\Theta,m}\{q, k\}$$

where

$$\mathbb{R}^d_{\Theta,m} = \begin{pmatrix} \cos m\theta_0 & -\sin m\theta_0 & \dots & 0 & 0 \\ \sin m\theta_0 & \cos m\theta_0 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \vdots & \cos m\theta_{\frac{d}{2}-1} & -\sin m\theta_{\frac{d}{2}-1} \\ 0 & 0 & \vdots & \sin m\theta_{\frac{d}{2}-1} & \cos m\theta_{\frac{d}{2}-1} \end{pmatrix}$$

is the rotary matrix with hyperparameter $\Theta \in \{\theta_i = 10000^{-2id}, i \in [0, 1, ..., \frac{d}{2} - 1]\}$

The reason why our positional recovery method can work is because attention score between a pair of tokens is invaraint to their absolute positions. Below is the proof of this invariance.

Proposition A.1 (Rope only depends on relative position). *Let vector $k \in \mathbb{R}^d$ denote a key vector and $k_m \in \mathbb{R}^d$ denote the key vector embedded at the fixed position m. And let vector $q \in \mathbb{R}^d$ denote a query vector and $q_{m+l} \in \mathbb{R}^d$ denote the query vector embedded at position (m+l). Then attention score $q_{m+l}k_m$ is derived as follow*

$$\begin{aligned} q_{m+l}k_m &= (\mathbb{R}^d_{\Theta,m+l}q)^T(\mathbb{R}^d_{\Theta,m}k) \\ &= \sum_{i=0}^{d/2-1} (q_{[2i]}k_{[2i]}\cos(m+l-m)\theta_i \\ &\quad + q_{[2i+1]}k_{[2i+1]}\cos(m+l-m)\theta_i) \\ &= \sum_{i=0}^{d/2-1} (q_{[2i]}k_{[2i]} + q_{[2i+1]}k_{[2i+1]})\cos l\theta_i \end{aligned} \tag{1}$$

where $\{q, k\}_{[i]}$ denotes i-th entry of vectors $\{q, k\}$ and h_i denotes dot product q_ik_i. The attention score $q_{m+l}k_m$ only depends on the relative distance l rather than the absolute position m.

N维位置恢复

在这里，我们证明了我们的定位恢复方法可以在N维场景中工作。我们从N维空间中RoPE的定义开始。

定义 1（旋转位置编码，ROPE[50]）。设向量 $q, k \in \mathbb{R}^d$ 表示查询向量和键向量，需要在某个位置 m 嵌入为 $q_m, k_m \in \mathbb{R}^d$。Rope 将位置信息编码为以下形式：

$$q_m, k_m = \mathbb{R}^d_{\Theta,m}\{q, k\}$$

哪里

$$\mathbb{R}^d_{\Theta,m} = \begin{pmatrix} \cos m\theta_0 & -\sin m\theta_0 & \dots & 0 & 0 \\ \sin m\theta_0 & \cos m\theta_0 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \vdots & \cos m\theta_{\frac{d}{2}-1} & -\sin m\theta_{\frac{d}{2}-1} \\ 0 & 0 & \vdots & \sin m\theta_{\frac{d}{2}-1} & \cos m\theta_{\frac{d}{2}-1} \end{pmatrix}$$

是具有超参数 $\Theta = \{\theta_i = 10000^{-2id}, i \in [0, 1, ..., \frac{d}{2} - 1]\}$ 的旋转矩阵

我们的位置恢复方法之所以有效，是因为一对标记之间的注意力得分对它们的绝对位置是不变的。下面是这种不变性的证明。

命题 A.1（绳索仅依赖于相对位置）。设向量 $k \in \mathbb{R}^d$ 表示关键向量，$k_m \in \mathbb{R}^d$ 表示嵌入在固定位置 m 的关键向量。设向量 $q \in \mathbb{R}^d$ 表示查询向量，$q_{m+l} \in \mathbb{R}^d$ 表示嵌入在位置 (m+l) 的查询向量。那么注意力得分 $q_{m+l}k_m$ 的推导如下：

$$\begin{aligned} q_{m+l}k_m &= (\mathbb{R}^d_{\Theta,m+l}q)^T(\mathbb{R}^d_{\Theta,m}k) \\ &= \sum_{i=0}^{d/2-1} (q_{[2i]}k_{[2i]}\cos(m+l-m)\theta_i \\ &\quad + q_{[2i+1]}k_{[2i+1]}\cos(m+l-m)\theta_i) \\ &= \sum_{i=0}^{d/2-1} (q_{[2i]}k_{[2i]} + q_{[2i+1]}k_{[2i+1]})\cos l\theta_i \end{aligned} \tag{1}$$

其中 $\{q, k\}_{[i]}$ 表示向量 $\{q, k\}$ 的第 i 个元素，而 h_i 表示点积 $q_i k_i$。注意力分数 $q_{m+l}k_m$ 仅依赖于相对距离 l 而不是绝对位置 m。