CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving

Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang*, Kuntai Du, Jiayi Yao, Shan Lu†, Ganesh Ananthanarayanan†, Michael Maire, Henry Hoffmann, Ari Holtzman, Junchen Jiang
University of Chicago †*Microsoft* **Stanford University*

Abstract

As large language models (LLMs) take on complex tasks, their inputs are supplemented with *longer contexts* that incorporate domain knowledge. Yet using long contexts is challenging as nothing can be generated until the whole context is processed by the LLM. While the context-processing delay can be reduced by reusing the KV cache of a context across different inputs, fetching the KV cache, which contains large tensors, over the network can cause high extra network delays.

CacheGen is a fast context-loading module for LLM systems. First, CacheGen uses a custom tensor encoder, leveraging KV cache's distributional properties to *encode* a KV cache into more compact bitstream representations with negligible decoding overhead, to save bandwidth usage. Second, CacheGen *adapts* the compression level of different parts of a KV cache to cope with changes in available bandwidth, in order to maintain low context-loading delay and high generation quality. We test CacheGen on popular LLMs and datasets. Compared to the recent systems that reuse the KV cache, CacheGen reduces the KV cache size by 3.5-4.3x and the total delay in fetching and processing contexts by 3.2-3.7x with negligible impact on the LLM response quality. Our code is at: **https://github.com/UChi-JCL/CacheGen**.

CCS Concepts

• **Computing methodologies** → **Natural language generation**; • **Networks** → **Application layer protocols**; • **Information systems** → **Information systems applications**.

Keywords

Large Language Models, KV Cache, Compression

ACM Reference Format:

Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, Junchen Jiang. 2024. CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving. In *SIGCOMM '24, August 4–August 8, 2024, Sydney, Australia. ACM, New York, NY, USA, 18 pages*

ACM SIGCOMM '24, August 4–8, 2024, Sydney, NSW, Australia

ACM ISBN 979-8-4007-0614-1/24/08
https://doi.org/10.1145/3651890.3672274

1 Introduction

With impressive generative quality, large language models (LLMs) are ubiquitously used [22, 38, 46, 128] in personal assistance, AI healthcare, and marketing. The wide use of LLM APIs (*e.g.,* OpenAI GPT-4 [108]) and the industry-quality open-source models (*e.g.,* Llama [129]), combined with popular application frameworks (*e.g.,* HuggingFace [10], Langchain [83]), further boosts LLMs' popularity.

To perform complex tasks, users or applications often prepend an LLM input with a *long context* containing thousands of tokens or more. For example, some context supplements user prompts with domain-knowledge text so that the LLM can generate responses using specific knowledge not embedded in the LLM itself. As another example, a user prompt can be supplemented with the conversation histories accumulated during the interactions between the user and the LLM. Though short inputs are useful [94, 124], longer inputs often improve response quality and coherence [31, 32, 35, 45, 67, 116, 130, 141], which has fueled the ongoing race to train LLMs that accept ever longer inputs, from 2K tokens in ChatGPT to 100K in Claude [24].

Using long contexts poses a challenge to the response generation *latency*, as no response can be generated until the whole context is loaded and processed by the LLM. The amount of computation in processing a long context grows super-linearly with the context length [31, 47, 116, 131, 150]. While some recent works increase the throughput of processing long context [17], the *delay* of processing the context can still be several seconds for long contexts (2 seconds for a 3K context) [17, 58]. In response, many systems reduce the context-processing delay by storing and reusing the *KV cache* of the context to skip redundant computation when the context is used again (*e.g.,* [23, 58, 82, 156]).

Yet, the KV cache of a reused context may *not* always be in local GPU memory when the next input comes; instead, the KV cache may need to be retrieved from another machine(s) first, causing extra network delays (Figure 1a). For instance, a database of background documents might reside in a separate storage service, and the documents (*i.e.,* context) assisting LLM inference are only to be selected and fetched to the LLM when a relevant query is received [27, 31, 36, 84, 110].

The extra network delay for fetching the KV cache has not yet received much attention. Previous systems assume the KV cache of a context is always kept in the same GPU memory between different requests sharing the same context [58], or the KV cache is small enough to be sent quickly by a fast interconnection [111, 157]. Yet, as elaborated in §3, the delay for fetching a KV cache can be non-trivial, since a KV cache consists of large high-dimensional floating-point tensors, whose sizes grow with both the context length and model size and can easily reach 10s GB. The resulting network delay can

CacheGen：用于快速大型语言模型服务的KV缓存压缩和流式传输

刘宇涵，李汉辰，程怡华，西丹特·雷，黄宇扬，张启正*，杜坤泰，姚佳怡，卢珊†，甘尼什·阿南坦纳拉扬†，迈克尔·梅尔，亨利·霍夫曼，阿里·霍尔茨曼，蒋俊辰 芝加哥大学 †微软 *斯坦福大学

摘要

随着大型语言模型（LLMs）承担复杂任务，它们的输入被补充了包含领域知识的更长上下文。然而，使用长上下文是具有挑战性的，因为在整个上下文被 LLM 处理之前，无法生成任何内容。虽然通过在不同输入之间重用上下文的 KV 缓存可以减少上下文处理延迟，但通过网络获取包含大张量的 KV 缓存可能会导致高额外网络延迟。

CacheGen 是一个快速的上下文加载模块，适用于 LLM 系统。首先，CacheGen 使用自定义张量编码器，利用 KV 缓存的分布特性将 KV 缓存编码为更紧凑的比特流表示，解码开销可以忽略不计，从而节省带宽使用。其次，CacheGen 调整 KV 缓存不同部分的压缩级别，以应对可用带宽的变化，以保持低上下文加载延迟和高生成质量。我们在流行的 LLM 和数据集上测试了 CacheGen。与最近重用 KV 缓存的系统相比，CacheGen 将 KV 缓存大小减少了 3.5-4.3 倍，并将获取和处理上下文的总延迟减少了 3.2-3.7 倍，对 LLM 响应质量的影响可以忽略不计。我们的代码在：https://github.com/UChi-JCL/CacheGen。

CCS 概念

• 计算方法 → 自然语言生成; • 网络 → 应用层协议; • 信息系统 → 信息系统应用。

关键词

大型语言模型，KV缓存，压缩

ACM 参考格式：
刘宇涵，李汉辰，程怡华，西丹特·雷，黄宇扬，张启正，杜坤泰，姚佳怡，卢珊，甘尼什·阿南坦纳亚南，迈克尔·梅尔，亨利·霍夫曼，阿里·霍尔茨曼，蒋俊辰。2024年。CacheGen：用于快速大型语言模型服务的KV缓存压缩和流式传输。在SIGCOMM '24，2024年8月4日至8月8日，澳大利亚悉尼。ACM，纽约，NY，美国，18页

ACM SIGCOMM '24, August 4–8, 2024, Sydney, NSW, Australia

ACM ISBN 979-8-4007-0614-1/24/08
https://doi.org/10.1145/3651890.3672274

1 引言

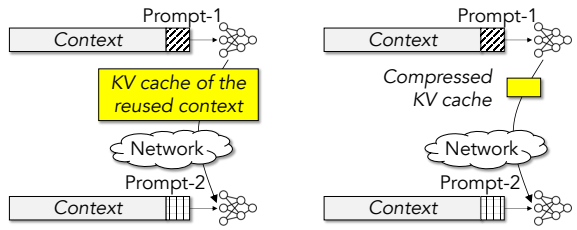
凭借令人印象深刻的生成质量，大型语言模型（LLMs）在个人助手、人工智能医疗和营销中被广泛使用 [22, 38, 46, 128]。LLM API（例如，OpenAI GPT-4 [108]）的广泛使用以及行业质量的开源模型（例如，Llama [129]），结合流行的应用框架（例如，HuggingFace [10]，Langchain [83]），进一步提升了LLMs的受欢迎程度。

为了执行复杂任务，用户或应用程序通常会在 LLM 输入前添加一个包含数千个标记或更多的长上下文。例如，一些上下文通过领域知识文本补充用户提示，以便 LLM 可以使用未嵌入在 LLM 本身中的特定知识生成响应。另一个例子是，用户提示可以通过在用户与 LLM 之间的交互中积累的对话历史进行补充。尽管短输入是有用的 [94, 124]，但较长的输入通常会提高响应质量和连贯性 [31, 32, 35, 45, 67, 116, 130, 141]，这推动了训练能够接受越来越长输入的 LLM 的持续竞争，从 ChatGPT 的 2K 标记到 Claude 的 100K [24]。

使用长上下文会对响应生成延迟构成挑战，因为在整个上下文加载并被 LLM 处理之前，无法生成任何响应。处理长上下文的计算量随着上下文长度的增加而超线性增长 [31, 47, 116, 131, 150]。尽管一些近期的研究提高了处理长上下文的吞吐量 [17]，但处理长上下文的延迟仍然可能达到几秒钟（对于 3K 上下文为 2 秒） [17, 58]。作为回应，许多系统通过存储和重用上下文的 KV 缓存来减少上下文处理延迟，以在上下文再次使用时跳过冗余计算（例如，[23, 58, 82, 156]）。

然而，当下一个输入到来时，重用上下文的 KV 缓存可能并不总是在本地 GPU 内存中；相反，KV 缓存可能需要先从其他机器检索，这会导致额外的网络延迟（图 1a）。例如，背景文档的数据库可能位于一个单独的存储服务中，只有在收到相关查询时，才会选择并提取协助 LLM 推理的文档（即上下文）[27, 31, 36, 84, 110]。

获取 KV 缓存的额外网络延迟尚未受到太多关注。之前的系统假设同一上下文的 KV 缓存在不同请求之间始终保存在同一 GPU 内存中 [58]，或者 KV 缓存足够小，可以通过快速互连迅速发送 [111, 157]。然而，正如 §3 中详细说明的那样，获取 KV 缓存的延迟可能并不微不足道，因为 KV 缓存由大型高维浮点张量组成，其大小随着上下文长度和模型大小的增加而增长，容易达到数十 GB。由此产生的网络延迟可以



(a) Baseline: Sharing the full-size KV cache tensor is slow

(b) CacheGen: Speeding up by compressing (encoding) KV cache

Figure 1: *When the context is reused, CacheGen speeds up the sharing of its KV cache by compressing (encoding) the KV cache.*

be 100s milliseconds to over 10 seconds, hurting the interactive user experience [1, 2, 87]. In short, when loading contexts' KV cache from other machines, solely optimizing computational delay may cause *higher* response latency, as loading the KV cache increases the network delay.

There have been a few recent efforts to reduce the *run-time* size of KV cache in GPU memory in order to fit the memory limit or LLM's input limit. Some drop unimportant tokens from KV cache or context text [71, 72, 95, 153], and others apply smart quantization on KV cache tensor [62, 78, 97]. In contrast, we want to reduce the *transmission-time* size of KV cache to reduce the *network delay*. Thus, we do *not* need to keep the tensor format of KV cache and, instead, can encode it into more compact bitstreams.

We present CacheGen, a fast context-loading module in LLM systems for reducing the network delay in fetching and processing long contexts (Figure 1b). It entails two techniques.

KV cache *encoding and decoding*: CacheGen encodes a pre-computed KV cache into more compact *bitstream* representations, rather than keeping the tensor shapes of the KV cache. This greatly saves bandwidth and delays when sending a KV cache. Our KV cache encoder employs a custom quantization and arithmetic coding strategy to leverage the distributional properties of KV cache, such as locality of KV tensors across nearby tokens and different sensitivities towards quantization losses at different layers of a KV cache. Furthermore, the decoding (decompression) of KV caches is accelerated by a GPU-based implementation, and the decoding is *pipelined* with transmission to further reduce its impact on the overall inference delay.

KV cache *streaming*: CacheGen streams the encoded bitstreams of a KV cache in a way that adapts to changes in network conditions. Before a user query arrives, CacheGen splits a long context into chunks and encodes the KV of each chunk separately at various compression levels (similar to video streaming). When sending a context's KV cache, CacheGen fetches the chunks one by one and adapts the per-chunk compression level to maintain high generation quality while keeping the network delay within a Service-Level Objective (SLO). When the bandwidth is too low, CacheGen can also fall back to sending a chunk in text format and leave it to the LLM to recompute the KV cache of the chunk.

In short, unlike prior systems that optimize the KV cache in GPU memory, CacheGen focuses on the *network* delay for sending the KV cache. We compare CacheGen with a range of baselines, including KV quantization [120], loading contexts in text form, and state-of-the-art context compression [72, 153], using three popular

Technique	KV cache size (in MB, lower the better)	Accuracy (higher the better)
8-bit quantization	622	1.00
CacheGen (this paper)	176	0.98
H2O [153]	282	0.97
CacheGen on H2O	71	0.97
LLMLingua [72]	492	0.94
CacheGen on LLMLingua	183	0.94

Table 1: *Performance of CacheGen and the baselines on Mistral-7B with LongChat dataset [90]. Full results are shown in §7.*

LLMs of various sizes (from 7B to 70B) and four datasets of long contexts (662 contexts with 1.4 K to 16 K tokens). Table 1 gives a preview of the results. Our key findings are:

- In terms of the delay of transmitting and processing contexts (*i.e.,* time-to-first-token), CacheGen is 3.2-3.7× faster than the quantization baseline at the similar generation quality (F1 score and perplexity), and 3.1-4.7× faster than loading the text contexts with less than 2% accuracy drop. Notably, compared with 8-bit quantization, a nearly lossless KV cache compression, CacheGen is still able to reduce the delay of loading context by 1.67-1.81×.
- In terms of the bandwidth usage for sending KV cache, CacheGen achieves the same generation quality while using 3.5-4.3× less bandwidth than the quantization baseline.
- When combined with the recent context compression methods [72, 153], CacheGen further reduces the bandwidth usage for sending their KV caches by 3.3-4.2×.

This work does not raise any ethical issues.

2 Background and Motivation

2.1 Large language model basics

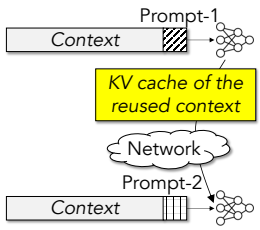
Transformers [37, 44, 131] are the de facto models for most large language model (LLM) services. At a high level, a transformer takes a sequence of input tokens[1] and generates a sequence of output tokens through two phases.

During the prefill phase, an attention neural network takes in the input token. Then each of the l layers in the attention module produces two two-dimensional tensors, a key (K) tensor and a value (V) tensor. These K and V tensors contain information essential for LLM to utilize the context later. All the KV tensors across different layers are together called the *KV cache*.

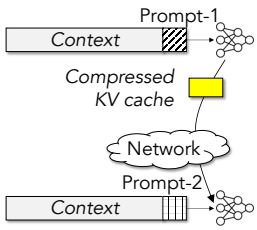
During the generation phase, also called the decoding phase, the KV cache is used to compute the attention score between every pair of tokens, which constitute the attention matrix, and generate output tokens in an autoregressive manner. For performance reasons, the KV cache, which has a large memory footprint [82], is usually kept in GPU memory during this phase and released afterward. Some emergent optimizations save and reuse the KV cache across different LLM requests, as we will explain shortly.

In all mainstream models, the compute overhead of the prefill phase grows superlinearly with the input length. Since the prefill phase must be completed before generating the first output token, its duration is called *Time-to-First-Token* (*TTFT*). This paper

[1] A "token" can be a punctuation, a word, or a part of a word. Tokenizing an input is much faster than the generation process.



(a) Baseline: Sharing the full-size KV cache tensor is slow



(b) CacheGen: Speeding up by compressing (encoding) KV cache

图 1：当上下文被重用时，CacheGen 通过压缩（编码）KV 缓存来加速其 KV 缓存的共享。

在100毫秒到超过10秒之间，影响交互式用户体验[1, 2, 87]。简而言之，当从其他机器加载上下文的KV缓存时，仅优化计算延迟可能会导致更高的响应延迟，因为加载KV缓存会增加网络延迟。

最近有一些努力旨在减少GPU内存中KV缓存的运行时大小，以适应内存限制或LLM的输入限制。一些方法从KV缓存或上下文文本中删除不重要的标记[71, 72, 95, 153]，而其他方法则对KV缓存张量应用智能量化[62, 78, 97]。相比之下，我们希望减少KV缓存的传输时间大小，以降低网络延迟。因此，我们不需要保持KV缓存的张量格式，而是可以将其编码为更紧凑的比特流。

我们介绍了CacheGen，这是LLM系统中的一个快速上下文加载模块，用于减少获取和处理长上下文时的网络延迟（图1b）。它包含两种技术。

KV缓存编码和解码：CacheGen将预计算的KV缓存编码为更紧凑的比特流表示，而不是保持KV缓存的张量形状。这大大节省了发送KV缓存时的带宽和延迟。我们的KV缓存编码器采用自定义量化和算术编码策略，以利用KV缓存的分布特性，例如相邻标记之间KV张量的局部性以及KV缓存不同层对量化损失的不同敏感性。此外，KV缓存的解码（解压缩）通过基于GPU的实现加速，并且解码与传输进行流水线处理，以进一步减少其对整体推理延迟的影响。

KV缓存流：CacheGen以适应网络条件变化的方式流式传输KV缓存的编码比特流。在用户查询到达之前，CacheGen将长上下文拆分成多个块，并以不同的压缩级别分别编码每个块的KV（类似于视频流）。在发送上下文的KV缓存时，CacheGen逐个获取这些块，并调整每个块的压缩级别，以保持高生成质量，同时将网络延迟控制在服务水平目标（SLO）之内。当带宽过低时，CacheGen还可以回退到以文本格式发送一个块，并将其交给LLM重新计算该块的KV缓存。

简而言之，与之前优化GPU内存中KV缓存的系统不同，CacheGen专注于发送KV缓存的网络延迟。我们将CacheGen与一系列基准进行比较，包括KV量化[120]、以文本形式加载上下文以及最先进的上下文压缩[72, 153]，使用三种流行的

Technique	KV cache size (in MB, lower the better)	Accuracy (higher the better)
8-bit quantization	622	1.00
CacheGen (this paper)	176	0.98
H2O [153]	282	0.97
CacheGen on H2O	71	0.97
LLMLingua [72]	492	0.94
CacheGen on LLMLingua	183	0.94

表1：CacheGen和基线在Mistral-7B上使用LongChat数据集的性能[90]。完整结果见§7。

各种规模的LLM（从7B到70B）和四个长上下文的数据集（662个上下文，包含1.4K到16K个标记）。表1提供了结果的预览。我们的主要发现是：

- 在传输和处理上下文的延迟方面（即首次令牌的时间），CacheGen 比量化基线快 3.2-3.7×，在相似的生成质量（F1 分数和困惑度）下，并且比加载文本上下文快 3.1-4.7×，准确率下降不到 2%。值得注意的是，与 8 位量化相比，几乎无损的 KV 缓存压缩，CacheGen 仍然能够将加载上下文的延迟减少 1.67-1.81×。
- 在发送KV缓存的带宽使用方面，CacheGen在使用比量化基线少3.5-4.3×的带宽的同时，实现了相同的生成质量。
- 当与最近的上下文压缩方法 [72, 153] 结合时，CacheGen 进一步将发送其 KV 缓存的带宽使用减少了 3.3-4.2×。

这项工作没有提出任何伦理问题。

2 背景与动机

2.1 大型语言模型基础

变压器 [37, 44, 131] 是大多数大型语言模型 (LLM) 服务的事实标准模型。从高层次来看，变压器接收一系列输入标记 {v*} 并通过两个阶段生成一系列输出标记。

在预填充阶段，注意力神经网络接收输入标记。然后，注意力模块中的每个 l 层生成两个二维张量，一个键（K）张量和一个值（V）张量。这些 K 和 V 张量包含了 LLM 后续利用上下文所需的关键信息。不同层中的所有 KV 张量统称为 KV 缓存。

在生成阶段，也称为解码阶段，KV 缓存用于计算每对标记之间的注意力分数，这构成了注意力矩阵，并以自回归的方式生成输出标记。出于性能考虑，KV 缓存通常在此阶段保留在 GPU 内存中，并在之后释放。一些新兴的优化方法在不同的 LLM 请求之间保存和重用 KV 缓存，正如我们稍后将解释的那样。

在所有主流模型中，预填充阶段的计算开销随着输入长度的增加而超线性增长。由于预填充阶段必须在生成第一个输出标记之前完成，因此其持续时间被称为首次标记时间（TTFT）。本文

[1]A "token" can be a punctuation, a word, or a part of a word. Tokenizing an input is much faster than the generation process.

focuses on reducing TTFT during prefilling while not changing the decoding process.

2.2 Context in LLM input

LLMs may generate low-quality or hallucinated answers when the response requires knowledge not already embedded in the models. Thus, many LLM applications and users supplement the LLM input with additional texts, referred to as the **context** [53, 89]. The LLM can read the context first and use its in-context learning capability to generate high-quality responses.[2]

The contexts in LLM input can be used for various purposes.

(i) a user question can be supplemented with a document about specific domain knowledge, to produce better answers [3, 7, 117], including using latest news to answer fact-checking inquiries [8, 9], using case law or regulation documents to offer legal assistance [118, 125], etc.; *(ii)* code analysis applications retrieve context from a code repository to answer questions or generate a summary about the repository [30, 69, 73], and similarly financial companies use LLMs to generate summaries or answer questions based on detailed financial documents [105]; *(iii)* gaming applications use the description of a particular character as context so that the LLM can generate character dialogues or actions matching the character personality [110, 121, 140]; *(iv)* in few-shot learning, a set of question-answer pairs are used as context to teach the LLM to answer certain types of questions [18, 99, 123]; *(v)* in chatting apps, the conversational history with a user is often prepended as the context to subsequent user input to produce consistent and informed responses [26, 76].

We observe that in practice, contexts are often **long** and often **reused** to supplement different user inputs.

Long contexts are increasingly common in practice. For example, those contexts discussed above, such as case law documents, financial documents, news articles, code files, and chat history accumulated in a session, easily contain thousands of tokens or more. Intuitively, longer contexts are more likely to include the right information and hence may improve the quality of the response. Indeed, FiD [67] shows that the accuracy increases from 40% to 48% when the context increases from 1K tokens to 10K. Retro [35] similarly shows that the generation quality (perplexity) improves significantly when the context increases from 6K tokens to 24K. This paper focuses on contexts such as conversation histories accumulated in a chat session, or a single document input by the user to provide necessary information needed to accomplish the task.

These long contexts are often *reused* by different inputs. In the financial analysis example, consider two queries, "write a short summary based on the company's earning report last quarter" and "what were the company's top sources of revenue in the last quarter"; the same earning reports are likely to be supplemented to both queries as the contexts. Similarly, the same law enforcement document or latest news article can be used to answer many different queries in legal assistant or fact-checking apps. As another example, during a chat session, early chat content will keep getting reused as part of the context for every later chat input.

[2]An example of this process is retrieval-augmented generation (RAG), which uses a separate logic to select the context documents for a given query, It is well-studied in natural-language literature and widely used in industry.

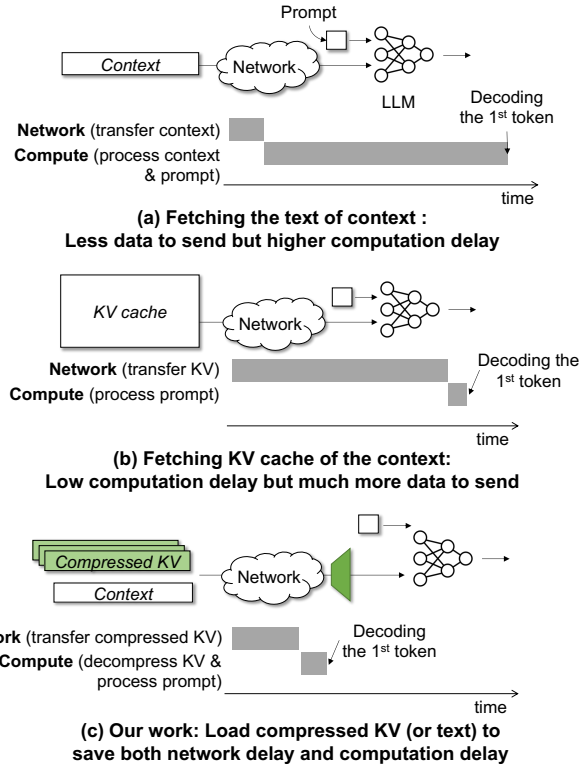


Figure 2: *How different ways of loading context affect the network delay (to transfer context or KV cache) and the computation delay (to run the attention module on the context).*

In short, longer contexts lead to higher prefill delays and hence longer TTFT, but since the same contexts are often reused, it is promising to reduce TTFT by caching the intermediate results (i.e., the KV cache) and hence avoid prefill recomputation. This solution has indeed been explored recently [23, 58, 82] and shown its potential with just one caveat, which we discuss in the next section.

3 The Hidden Network Bottleneck

While reusing the KV cache of a long context could drastically reduce TTFT, this benefit comes with a catch—the reused KV cache must be in the local GPU memory in the first place [23, 58, 82, 156].

Why KV cache needs to be loaded: In practice, however, the reused KV cache may need to be fetched from another machine(s). This is because GPU memory is likely not enough to store the KV caches of many repeated contexts. For example, in a financial assistance application, an LLM performs data analysis on long financial reports [107], which can have thousands or tens of thousands of tokens, leading to a large KV cache size. To make it concrete, processing Amazon's annual report for 2023, which has ~80,000 tokens [20], with the model of Llama-34B produces a KV cache of 19 GB, which is on par with the size of the LLM itself. As different queries that reuse a KV cache may be several hours apart, the reused KV cache may have to be offloaded to make space for fresh chat sessions. Moreover, as newer LLMs can accept ever longer contexts [51, 56, 63, 91, 138], storing them on dedicated storage servers, rather than CPUs or GPU, would be more practical and

专注于在预填充期间减少 TTFT，同时不改变解码过程。

2.2 LLM输入中的上下文

LLM在响应需要模型中未嵌入的知识时，可能会生成低质量或虚构的答案。因此，许多LLM应用和用户会用额外的文本来补充LLM输入，这被称为上下文[53, 89]。LLM可以先阅读上下文，并利用其上下文学习能力生成高质量的响应。2

LLM输入中的上下文可以用于多种目的。

(i) 用户问题可以通过关于特定领域知识的文档进行补充，以产生更好的答案 [3, 7, 117]，包括使用最新新闻来回答事实核查查询 [8, 9]，使用案例法或法规文件提供法律援助 [118, 125]，等等；(ii) 代码分析应用从代码库中检索上下文，以回答问题或生成关于该代码库的摘要 [30, 69, 73]，类似地，金融公司使用LLMs根据详细的财务文件生成摘要或回答问题 [105]；(iii) 游戏应用使用特定角色的描述作为上下文，以便LLM可以生成与角色个性相匹配的角色对话或动作 [110, 121, 140]；(iv) 在少量学习中，一组问答对被用作上下文，以教导LLM回答某些类型的问题 [18, 99, 123]；(v) 在聊天应用中，与用户的对话历史通常作为上下文附加到后续用户输入，以产生一致且有根据的响应 [26, 76]。

我们观察到，在实践中，上下文通常很长，并且经常被重复使用以补充不同的用户输入。

长上下文在实践中越来越常见。例如，上述讨论的上下文，如案例法文件、财务文件、新闻文章、代码文件和在会话中积累的聊天记录，轻易地包含数千个标记或更多。直观上，较长的上下文更有可能包含正确的信息，因此可能提高响应的质量。实际上，FiD [67] 显示，当上下文从 1K 标记增加到 10K 时，准确率从 40% 提高到 48%。Retro [35] 同样显示，当上下文从 6K 标记增加到 24K 时，生成质量（困惑度）显著改善。本文重点关注在聊天会话中积累的对话历史或用户输入的单个文档，以提供完成任务所需的必要信息。

这些长上下文通常会被不同的输入重复使用。在财务分析的例子中，考虑两个查询："根据公司上个季度的财报写一个简短的总结"和"公司上个季度的主要收入来源是什么"；相同的财报很可能会作为上下文补充到这两个查询中。同样，相同的执法文件或最新的新闻文章可以用于回答法律助手或事实核查应用中的许多不同查询。作为另一个例子，在聊天会话中，早期的聊天内容将不断被重复使用，作为每个后续聊天输入的上下文的一部分。

[2]An example of this process is retrieval-augmented generation (RAG), which uses a separate logic to select the context documents for a given query, It is well-studied in natural-language literature and widely used in industry.

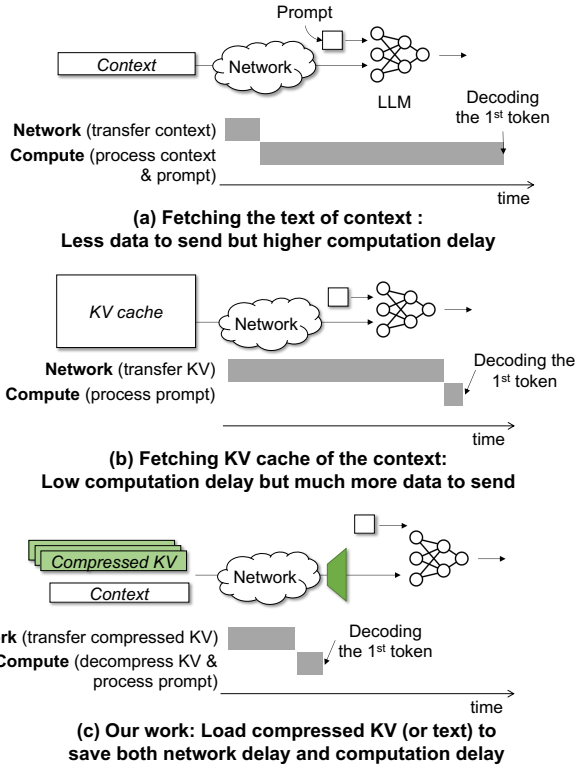


图 2：不同加载上下文的方式如何影响网络延迟（传输上下文或 KV 缓存）和计算延迟（在上下文上运行注意力模块）。

简而言之，较长的上下文会导致更高的预填充延迟，从而导致更长的 TTFT，但由于相同的上下文通常会被重复使用，因此通过缓存中间结果（即 KV 缓存）来减少 TTFT 是有希望的，从而避免预填充的重新计算。这个解决方案确实在最近被探索过 [23, 58, 82]，并显示出其潜力，只有一个警告，我们将在下一节中讨论。

3 隐藏的网络瓶颈

虽然重用长上下文的KV缓存可以大幅减少TTFT，但这个好处有一个前提——重用的KV缓存必须首先在本地GPU内存中[23, 58, 82, 156]。

为什么需要加载KV缓存：然而，在实践中，重用的KV缓存可能需要从其他机器获取。这是因为GPU内存可能不足以存储许多重复上下文的KV缓存。例如，在一个财务援助应用中，一个LLM对长达数千或数万标记的财务报告进行数据分析[107]，这会导致KV缓存的大小很大。具体来说，处理亚马逊2023年的年报，该报告有~80,000个标记[20]，使用Llama-34B模型生成的KV缓存为19 GB，这与LLM本身的大小相当。由于重用KV缓存的不同查询可能相隔数小时，因此重用的KV缓存可能必须被卸载以为新的聊天会话腾出空间。此外，由于更新的LLM可以接受越来越长的上下文[51, 56, 63, 91, 138]，将它们存储在专用存储服务器上，而不是CPU或GPU，将更为实际。

economical. Besides, different requests that reuse KV cache may not always hit the same GPU, which also requires the KV cache to be moved between machines.

Fetching KV cache from another machine causes a substantial delay, yet this network delay has not received sufficient attention.

Is it a new problem? Although some recent efforts also propose to send KV cache across GPUs to run multi-GPU inference, these systems assume that the KV cache is shared via high-speed links [111, 157], *e.g.*, direct NVLinks, which has bandwidth of up to several hundred Gbps. In these settings, the network delay to fetch KV cache can be negligible. However, KV caches also need to be fetched over lower-bandwidth links, such as between regular cloud servers, where the bandwidth is usually in the single-digit Gbps range [70]. As illustrated in Figure 2b, in this setting, the delay of fetching KV cache into GPU memory can be as long as (or even longer than) prefill without the KV cache.

Our approach: This paper focuses on reducing the network delay in fetching the KV cache. To this end, we compress the KV cache by ***encoding*** it into more compact bitstream representations (shown in Figure 2c). This goal may seem similar to the recent works that drop words (tokens) from the text context or quantize the KV cache tensors [62, 78, 95, 97, 153]. However, there is a key difference. These techniques reduce the ***run-time*** GPU memory footprint of KV cache, thus retaining the tensor shapes of KV cache. In contrast, we reduce the ***transmission-time*** size of KV cache by encoding it into compact bitstreams to reduce the network delay of sending it. Moreover, there is a natural synergy—the KV cache shrunk by these recent works can still be encoded to further reduce the KV cache size and the network delay of sending KV caches.

4 CacheGen: KV Cache Encoding and Streaming

The need to reduce KV cache transmission delay motivates a new module in LLM systems, which we call a *KV cache streamer*. The KV cache streamer serves three roles:

- *(1) Encoding* a given KV cache into more compact bitstream representations — *KV bitstreams*. This can be done offline.
- *(2) Streaming* the encoded KV bitstream through a network connection of varying throughput.
- *(3) Decoding* the received KV bitstream into the KV cache.

At first glance, our KV cache streamer may look similar to recent techniques (*e.g.*, [72, 95, 153]) that compress long contexts by dropping less important tokens. Yet, they differ in crucial ways:

Those recent techniques aim at reducing the *run-time* size of the KV cache to accommodate the GPU memory-size constraint or LLM input-window constraint, and yet we aim at reducing the *transmission-time* size of the KV cache to reduce network delay. As a result, previous techniques have to maintain the KV caches' shapes of large floating-point tensors so that the shrunk KV caches can be directly consumed by the LLM at the run-time; meanwhile, they can use information during the generation phase to know which tokens in the context are more important to the particular query under processing. In contrast, we need *not* to maintain the original tensor shapes, and can encode them into more compact bitstreams and adapt their representation to network bandwidth. Meanwhile, we have to decide which compression scheme to use before a particular query is processed, and hence, we cannot use information from the generation phase.

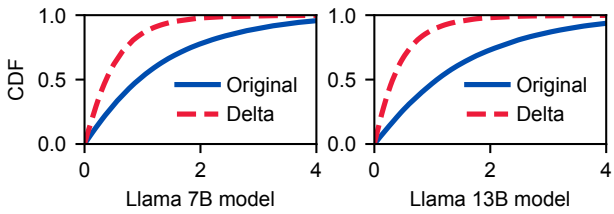


Figure 3: *Contrasting the distribution of the original values and the delta values. We model two Llama models with various long contexts (§5.1). We show absolute values for clarity.*

This paper presents **CacheGen**, a concrete design of the KV cache streamer. First, CacheGen uses a custom KV cache codec (encoder and decoder) to minimize the size of KV bitreams, by embracing several distributional properties of KV cache tensors (§5.1). This greatly reduces the bandwidth demand to transmit the KV cache, thus directly reducing TTFT. Second, when streaming the KV bitstreams under dynamic bandwidth, CacheGen dynamically switches between different encoding levels or computing the KV cache on demand, in order to keep the TTFT within a given deadline while maintaining a high response quality. The KV encoding/decoding incurs a negligible compute overhead and is pipelined with network transmission to minimize the impact on end-to-end delay.

5 CacheGen Design

We now describe the design of CacheGen, starting with the insights on KV cache (§5.1) that inspires KV cache encoder (§5.2), followed by how CacheGen adapts to bandwidth (§5.3).

5.1 Empirical insights of KV cache

We highlight three observations on the characteristics of KV cache values. Though it is intrinsically hard to prove they apply to any LLM with any context, here, we use a representative workload to empirically demonstrate the prevalence of these observations. The workload includes two LLMs of different capacities (Llama-7B and Llama-13B) and LongChat dataset [90] (which contains 100 long contexts between 9.2K and 9.6K tokens, randomly sampled from the whole set of 200 contexts), one of the largest datasets of long contexts. Details of this workload can be found in §7.1.

5.1.1 ***Token-wise locality.*** The first observation is about how the K and V tensor values change *across tokens* in a context. Specifically, we observe that

Insight 1. *Within the same layer and channel, tokens in closer proximity have more similar K/V tensor values compared to tokens that are further apart.*

For each model, we contrast the distribution of K (or V) tensors' original values and the distribution of the *deltas*—the differences between K (or V) tensors' values at the same layer and channel between every pair of consecutive tokens in the contexts. Figure 3 shows the distribution of absolute values in the original tensor and the deltas of one layer across all the contexts[3]. In both models across the contexts, we can see that the deltas are much more concentrated

[3]We randomly sampled a single layer from the K tensor because the values in the different layers have different ranges.

经济实惠。此外，重用KV缓存的不同请求可能并不总是命中同一GPU，这也需要在机器之间移动KV缓存。

从另一台机器获取KV缓存会导致显著的延迟，但这种网络延迟并没有受到足够的关注。这是一个新问题吗？尽管一些最近的努力也提议通过GPU之间发送KV缓存以进行多GPU推理，但这些系统假设KV缓存是通过高速链接共享的[111, 157]，例如，直接的NVLinks，其带宽可达到数百Gbps。在这些设置中，获取KV缓存的网络延迟可以忽略不计。然而，KV缓存也需要通过较低带宽的链接获取，例如在常规云服务器之间，带宽通常在个位数Gbps范围内[70]。如图2b所示，在这种情况下，将KV缓存提取到GPU内存中的延迟可能与没有KV缓存的预填充一样长（甚至更长）。

我们的方法：本文重点在于减少获取KV缓存的网络延迟。为此，我们通过将KV缓存编码为更紧凑的比特流表示来压缩KV缓存（如图2c所示）。这个目标可能看起来与最近一些从文本上下文中删除单词（标记）或量化KV缓存张量的工作相似[62, 78, 95, 97, 153]。然而，有一个关键的区别。这些技术减少了KV缓存的运行时GPU内存占用，从而保留了KV缓存的张量形状。相比之下，我们通过将KV缓存编码为紧凑的比特流来减少KV缓存的传输时间大小，以降低发送它的网络延迟。此外，这些最近工作的KV缓存缩小后仍然可以被编码，以进一步减少KV缓存的大小和发送KV缓存的网络延迟。

4 CacheGen: KV 缓存编码与流式传输

减少KV缓存传输延迟的需求促使了LLM系统中一个新模块的出现，我们称之为KV缓存流。KV缓存流承担三个角色：

- (1) 将给定的 KV 缓存编码为更紧凑的比特流表示——KV 比特流。这可以离线完成。
- (2) 通过不同吞吐量的网络连接流式传输编码的KV比特流。
- (3) 将接收到的KV比特流解码到KV缓存中。

乍一看，我们的 KV 缓存流媒体可能看起来与最近的技术（例如，[72, 95, 153]）相似，这些技术通过丢弃不太重要的标记来压缩长上下文。然而，它们在关键方面有所不同：

这些最近的技术旨在减少KV缓存的运行时大小，以适应GPU内存大小限制或LLM输入窗口限制，同时我们旨在减少KV缓存的传输时间大小，以降低网络延迟。因此，以前的技术必须保持大型浮点张量的KV缓存形状，以便缩小的KV缓存可以在运行时被LLM直接使用；同时，他们可以在生成阶段使用信息来了解上下文中哪些标记对正在处理的特定查询更为重要。相比之下，我们不需要保持原始张量形状，可以将其编码为更紧凑的比特流，并将其表示适应网络带宽。同时，我们必须决定使用哪种压缩方案。

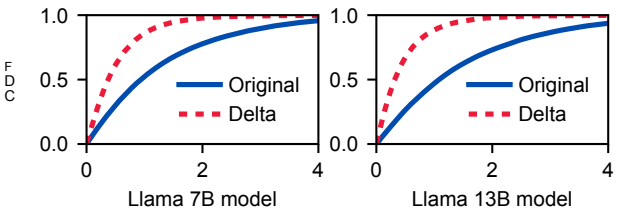


图3：对比原始值和增量值的分布。我们对两个具有不同长上下文的Llama模型进行建模（§5.1）。为了清晰起见，我们显示绝对值。

在处理特定查询之前，因此我们无法使用生成阶段的信息。

本文介绍了CacheGen，这是KV缓存流的具体设计。首先，CacheGen使用自定义的KV缓存编解码器（编码器和解码器）来最小化KV比特流的大小，通过利用KV缓存张量的几种分布特性（§5.1）。这大大减少了传输KV缓存所需的带宽，从而直接降低了TTFT。其次，在动态带宽下流式传输KV比特流时，CacheGen动态切换不同的编码级别或按需计算KV缓存，以便在保持高响应质量的同时将TTFT控制在给定的截止时间内。KV编码/解码产生的计算开销微不足道，并与网络传输进行流水线处理，以最小化对端到端延迟的影响。

5 CacheGen 设计

我们现在描述CacheGen的设计，首先是对KV缓存的见解（§5.1），这启发了KV缓存编码器（§5.2），接着是CacheGen如何适应带宽（§5.3）。

5.1 KV缓存的经验洞察

我们强调了关于KV缓存值特征的三个观察。尽管从本质上讲，很难证明它们适用于任何具有任何上下文的LLM，但在这里，我们使用一个代表性的工作负载来实证展示这些观察的普遍性。该工作负载包括两个不同容量的LLM（Llama-7B和Llama-13B）和LongChat数据集[90]（该数据集包含100个长上下文，长度在9.2K到9.6K个标记之间，随机从200个上下文的整个集合中抽样），这是最大的长上下文数据集之一。该工作负载的详细信息可以在§7.1中找到。

5.1.1 逐词局部性。第一个观察是关于 K 和 V 张量值在上下文中如何随词元变化。具体来说，我们观察到

洞察 1. 在同一层和通道内，距离较近的标记相比于距离较远的标记具有更相似的 K/V 张量值。

对于每个模型，我们对比了 K（或 V）张量的原始值的分布和增量的分布——即在上下文中每对连续标记之间同一层和通道的 K（或 V）张量值之间的差异。图 3 显示了原始张量和一个层的增量在所有上下文中的绝对值分布。在这两个模型的上下文中，我们可以看到增量更加集中。

[3]We randomly sampled a single layer from the K tensor because the values in the different layers have different ranges.

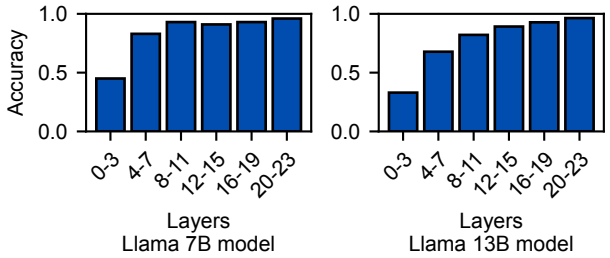


Figure 4: *Applying data loss to different layers of a KV cache has different impact on accuracy. (Same workload as Figure 3).*

around zero than the original values. Consequently, the variance of the deltas is 2.4-2.9× lower than that of the original values. The token-wise locality of K and V tensors inspires CacheGen to encode deltas rather than original values.

This token-wise locality can be intuitively explained by the transformer's self-attention mechanism, which computes the KV tensors. The mechanism is mathematically equivalent to calculating the KV tensors of one token based on the KV tensors of the previous token. This means KV tensors at one token are intrinsically correlated with those of the previous token.

5.1.2 ***Layer-wise sensitivity to loss.*** The second observation concerns how sensitive different values in the K and V tensors are to data loss. Our observation is the following:

Insight 2. *The output quality of the LLM is more sensitive to losses in the KV cache values of the shallower layers than to losses in those of the deeper layers.*

The heterogeneous loss sensitivity on different layers suggests that our KV cache encoder should compress different layers differently. Figure 4 shows how much accuracy is affected by applying data losses to the values of a specific layer group in the K and V tensors. Here, we apply rounding as the data loss, and we compute the average resulting response accuracy (defined in §7.1) across 100 contexts in the dataset. We can see that the average response accuracy drops significantly when the loss is applied to the early layers of a model while applying the same loss on the deeper layers has much less impact on the average response accuracy. This result holds consistently across different models we tested.

Intuitively, the deeper layers of a KV cache extract higher-level structures and knowledge than the shallower layers of a KV, which embed more primitive information [119, 132]. As a result, the loss of information by removing precision on the early-layer cache might propagate and affect the later-layer cache, and thus hinder the model's ability to grasp the higher-level structures necessary to produce quality responses.

5.1.3 ***Distribution along layers, channels, and tokens.*** Finally, regarding the distributions of values along the three dimensions of KV cache—layers, channels, and token positions—we make the following observation.

Insight 3. *Each value in a KV cache is indexed by its channel, layer, and token position. The information gain of grouping values by their channel and layer is significantly higher than the information gain of grouping values by their token position.*

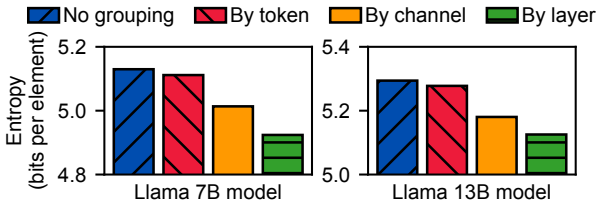


Figure 5: *Entropy (bits per element) when using different grouping strategies (Same workload as Figure 3.)*

Intuitively, this can be loosely interpreted as different KV values in the same channel (or layer) being more similar to each other than different KV values belonging to the same token position. A possible explanation is that different channels or layers capture various features in the input [49, 92]. Some channels capture subject-object relationships, while others focus on adjectives. As for different layers, later layers capture more abstract semantic information than earlier ones according to prior works [49, 92]. On the other hand, within a given layer and channel, the KV values for different tokens are more similar, likely because of the self-attention mechanism, wherein each token's KV is derived from all preceding tokens. We leave a more detailed examination to future work.

To empirically verify the insight, we first group the values in the KV caches produced by the two models and 100 contexts based on their layers, channels, or token positions, and then compute the entropy of each group. Figure 5 shows the average entropy (bits per element) when different grouping strategy is applied, including no grouping, grouping by tokens positions, grouping by channels, and grouping by layers. It shows grouping values by token positions reduces entropy much less than grouping by channel or layer.

5.2 KV cache encoding

The aforementioned insights inspire the design of CacheGen's KV cache encoder. The encoding consists of three high-level steps (elaborated shortly):

First, it calculates the *delta tensors* (defined later) between the K and V tensors of nearby tokens. This is inspired by the token-wise locality observation (§5.1.1) which suggests deltas between tokens might be easier to compress than the original values in the KV tensors.

Second, it applies different levels of quantization to different layers of the delta tensors. The use of different quantizations at different layers is inspired by the observation of heterogeneous loss sensitivity (§5.1.2).

Third, it runs a lossless arithmetic coder to encode the quantized delta tensors into bitstreams. Specifically, inspired by the observation in §5.1.3, the arithmetic coder compresses the values in each layer and channel separately (§5.1.3).

These steps may seem similar to video coding, which encodes pixels into bitstreams. Video coding also computes the delta between nearby frames, quantizes them, and encodes the delta by arithmetic coding [126]. Yet, blindly applying existing video codecs could not work well since they were only optimized for pixel values in natural video content. Instead, the exact design of CacheGen is inspired by domain-specific insights on LLM-generated KV cache (§5.1).

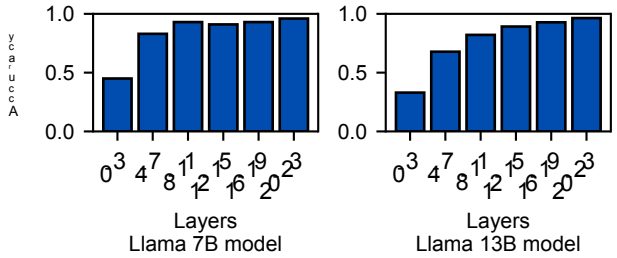


图4：对KV缓存的不同层应用数据丢失对准确性有不同的影响。（与图3相同的工作负载）。

接近零的原始值。因此，增量的方差比原始值低2.4-2.9×。K和V张量的逐个令牌局部性启发CacheGen编码增量而不是原始值。

这种逐个标记的局部性可以通过变换器的自注意力机制直观地解释，该机制计算KV张量。该机制在数学上等同于基于前一个标记的KV张量计算一个标记的KV张量。这意味着一个标记的KV张量与前一个标记的KV张量本质上是相关的。

5.1.2 层级敏感性对损失的影响。第二个观察涉及 K 和 V 张量中不同值对数据丢失的敏感性。我们的观察如下：

洞察 2. LLM 的输出质量对较浅层的 KV 缓存值的损失比对较深层的损失更为敏感。

不同层上的异质损失敏感性表明，我们的KV缓存编码器应该以不同的方式压缩不同的层。图4显示了在K和V张量的特定层组的值上应用数据损失对准确性的影响程度。在这里，我们将舍入作为数据损失，并计算数据集中100个上下文的平均响应准确性（在§7.1中定义）。我们可以看到，当损失应用于模型的早期层时，平均响应准确性显著下降，而在更深层应用相同的损失对平均响应准确性的影响要小得多。这个结果在我们测试的不同模型中始终保持一致。

直观上，KV缓存的深层提取的结构和知识比KV的浅层更高层次，这些浅层嵌入了更原始的信息[119, 132]。因此，通过降低早期层缓存的精度而导致的信息损失可能会传播并影响后期层缓存，从而妨碍模型掌握生成高质量响应所需的高层结构的能力。

5.1.3 沿层、通道和标记的分布。最后，关于 KV 缓存的三个维度——层、通道和标记位置的值分布，我们做出以下观察。

洞察 3. KV 缓存中的每个值都通过其通道、层和令牌位置进行索引。按通道和层对值进行分组的信息增益显著高于按令牌位置对值进行分组的信息增益。

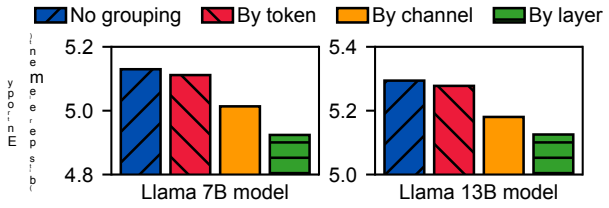


图 5：使用不同分组策略时的熵（每个元素的比特数）（与图 3 的工作负载相同。）

直观上，这可以被松散地解释为同一通道（或层）中的不同KV值彼此之间更相似，而不是属于同一标记位置的不同KV值。一个可能的解释是，不同的通道或层捕捉输入中的各种特征 [49, 92]。一些通道捕捉主语-宾语关系，而其他通道则专注于形容词。至于不同的层，根据先前的研究 [49, 92]，后面的层捕捉到比前面的层更抽象的语义信息。另一方面，在给定的层和通道内，不同标记的KV值更相似，这可能是由于自注意力机制，其中每个标记的KV是从所有前面的标记中派生出来的。我们将更详细的检查留给未来的工作。

为了实证验证这一见解，我们首先根据层、通道或令牌位置对两个模型和100个上下文中产生的KV缓存中的值进行分组，然后计算每个组的熵。图5显示了在应用不同分组策略时的平均熵（每个元素的比特数），包括不分组、按令牌位置分组、按通道分组和按层分组。结果表明，按令牌位置分组的值减少的熵远低于按通道或层分组。

5.2 KV缓存编码

上述见解激发了CacheGen的KV缓存编码器的设计。编码由三个高层步骤组成（稍后详细说明）：

首先，它计算附近标记的 K 和 V 张量之间的增量张量（稍后定义）。这受到逐标记局部性观察（§5.1.1）的启发，该观察表明，标记之间的增量可能比 KV 张量中的原始值更容易压缩。

其次，它对增量张量的不同层应用不同级别的量化。在不同层使用不同的量化是受到异构损失敏感性观察的启发 (§5.1.2)。

第三，它运行一个无损算术编码器，将量化的增量张量编码为比特流。具体来说，受到§5.1.3中的观察启发，算术编码器分别压缩每一层和每个通道中的值（§5.1.3）。

这些步骤可能看起来与视频编码相似，视频编码将像素编码为比特流。视频编码还计算相邻帧之间的差异，对其进行量化，并通过算术编码对差异进行编码 [126]。然而，盲目应用现有的视频编解码器可能效果不佳，因为它们仅针对自然视频内容中的像素值进行了优化。相反，CacheGen 的确切设计受到针对 LLM 生成的 KV 缓存的领域特定见解的启发 (§5.1)。

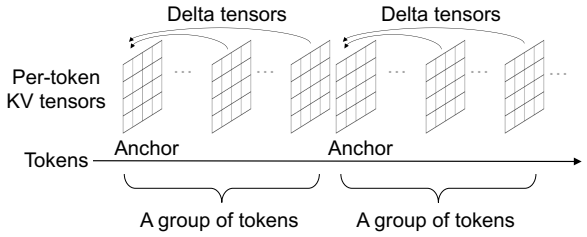


Figure 6: *Within a token group, CacheGen computes delta tensors between KV tensors of the anchor token and those of remaining tokens.*

Next, we explain the details of each step.

Change-based encoding: To leverage the token-wise locality, we first split the context into *groups of tokens* each containing ten contiguous tokens. As shown in Figure 6, in each group, we independently (*i.e.,* without referencing other tokens) compress the KV tensor of the first token, called the *anchor token*, and then compress and record the *delta tensors* with respect to the anchor token for every other token.

This process is analogous to video coding, where the frames are separated into groups of *pictures*, within which it runs similar delta-based encoding. The difference, however, is that instead of compressing the delta between each pair of consecutive tokens, we reference the same anchor token for every token in the chunk. This allows us to do compression and decompression in parallel and saves time.

Layer-wise quantization: After partitioning the tokens into groups, CacheGen uses quantization to reduce the precision of elements (floating points) in a KV cache so that they can be represented by fewer bits. Quantization has been used recently to reduce attention matrices to pack longer contexts in GPU memory [120]. However, in previous work, elements are uniformly quantized with the same number of bits without leveraging any unique properties of KV cache. Driven by the insight of heterogeneous loss sensitivity (§5.1.2), we apply more conservative quantization (i.e., using more bits) on the delta tensors of earlier layers. Specifically, we split the transformer layers into three layer groups, the first (earliest) 1/3 of layers, the middle 1/3 of layers, and the last 1/3 of layers, and apply different amounts of quantization bin size on the delta tensors at each layer group respectively. The size of the quantization bin grows larger (*i.e.,* larger quantization errors) from earlier to later layer groups. Following previous work [48], we use the `vectorwise` quantization method, which has been usually used for quantizing model weights.

Note that we still use 8-bit quantization, a relatively high precision, on the KV cache of the anchor token (the first token of a token chunk). This is because these anchor tokens account for a small fraction of all tokens, but their precision affects the distribution of all delta tensors of the remaining tokens in a chunk. Thus, it is important to preserve higher precision just for these anchor tokens.

Arithmetic coding: After quantizing the KV cache into discrete symbols, CacheGen uses *arithmetic coding* [135] (AC) to losslessly compress the delta tensors and anchor tensors of a context into bitstreams. Like other entropy coding schemes, AC assigns fewer bits to encode more frequent symbols and more bits to encode less frequent symbols. For it to be efficient, AC needs *accurate, low-entropy* probability distributions of the elements in the KV cache.

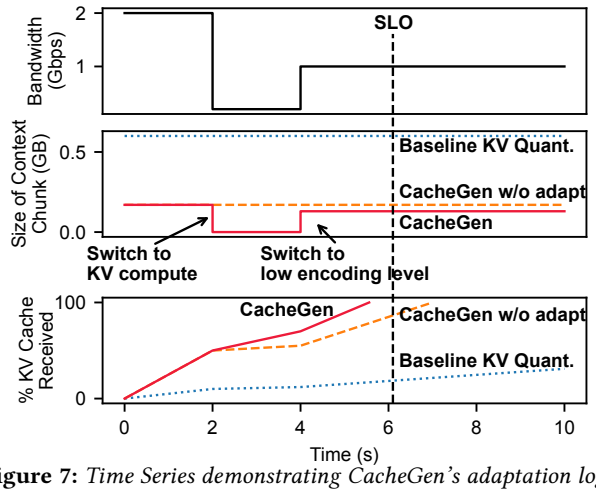


Figure 7: *Time Series demonstrating CacheGen's adaptation logic under bandwidth variation.*

Driven by the observation of the KV value distributions along layers, channels, and token positions (§5.1.3), we group KV values by channel and layer to obtain probability distributions. Specifically, our KV encoder offline profiles a separate probability distribution for each channel-layer combination of delta tensors and another for anchor tensors produced by an LLM, and uses the same distributions for all KV caches produced by the same LLM. CacheGen uses modified AC library [101] with CUDA to speed up encoding and decoding (§6). In §7.5, we empirically show that our method reduces the bitstream size by up to 53% compared to the strawman of using one global symbol distribution.

5.3 KV cache streaming adaptation

Since the transmission of a KV cache may take up to hundreds of milliseconds to a few seconds, the available bandwidth may fluctuate during a transmission. Thus, streaming the encoded KV bitstreams at a fixed encoding level may violate a given service-level objective (SLO) [33] of fetching the KV cache.[4] In Figure 7, for example, at the start of the transmission, the available throughput is 2 Gbps, and if the bandwidth remains at 2 Gbps, sending a KV stream of 1 GB can meet the SLO of 4 seconds. However, at $t = 2s$, the throughput drops to 0.2 Gbps and only increases to 1 Gbps at $t = 4s$, so the actual transmission delay increases from 4 seconds to 7 seconds, which violates the SLO.

Workflow: To handle variations in bandwidth, CacheGen splits a context into multiple *context chunks* (or **chunks** for short) of consecutive tokens and uses the KV cache encoder to encode each chunk into multiple bitstreams of different encoding (quantization) levels that can be decoded independently (explained shortly). This can be done offline. When fetching a context, CacheGen sends these chunks one by one, and each chunk can choose one of several *streaming configuration* (or **configurations** for short): it can be sent at one of the encoding levels or can be sent in the text format to let the LLM recompute K and V tensors.

[4] In practice, SLO is defined on TTFT. Once the KV cache of the long context is loaded in GPU, the remaining delay of one forward pass is marginal [82].

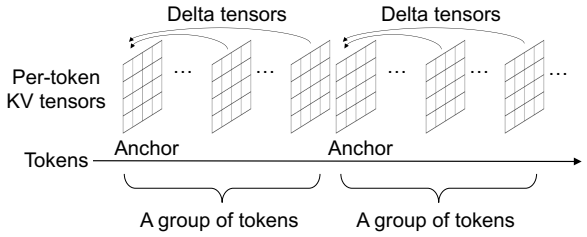


图6：在一个令牌组内，CacheGen计算锚令牌的KV张量与其余令牌的KV张量之间的增量张量。

接下来，我们将解释每个步骤的细节。

基于变化的编码：为了利用基于令牌的局部性，我们首先将上下文分成每组包含十个连续令牌的组。如图6所示，在每组中，我们独立地（即不参考其他令牌）压缩第一个令牌的KV张量，称为锚令牌，然后压缩并记录相对于锚令牌的每个其他令牌的增量张量。

这个过程类似于视频编码，其中帧被分成图像组，在这些组内运行类似的基于增量的编码。然而，区别在于，我们不是压缩每对连续标记之间的增量，而是对块中的每个标记引用相同的锚标记。这使我们能够并行进行压缩和解压缩，从而节省时间。

逐层量化：在将令牌分组后，CacheGen 使用量化来降低 KV 缓存中元素（浮点数）的精度，以便用更少的位表示它们。最近，量化已被用于减少注意力矩阵，以便在 GPU 内存中打包更长的上下文 [120]。然而，在之前的工作中，元素是以相同的位数均匀量化的，而没有利用 KV 缓存的任何独特属性。受到异构损失敏感性 (§5.1.2) 的启发，我们对早期层的增量张量应用更保守的量化（即，使用更多的位）。具体而言，我们将变换器层分为三个层组，第一（最早）1/3 的层，中间 1/3 的层，以及最后 1/3 的层，并分别对每个层组的增量张量应用不同数量的量化箱大小。量化箱的大小从早期层组到后期层组逐渐增大（即，量化误差增大）。遵循之前的工作 [48]，我们使用逐向量化方法，该方法通常用于量化模型权重。

请注意，我们仍然在锚标记（一个标记块的第一个标记）的KV缓存上使用8位量化，这是一种相对较高的精度。这是因为这些锚标记只占所有标记的一小部分，但它们的精度会影响块中其余标记的所有增量张量的分布。因此，仅为这些锚标记保留更高的精度是很重要的。算术编码：在将KV缓存量化为离散符号后，CacheGen使用算术编码[135]（AC）无损压缩上下文的增量张量和锚张量为比特流。与其他熵编码方案一样，AC为更频繁的符号分配更少的比特，而为不太频繁的符号分配更多的比特。

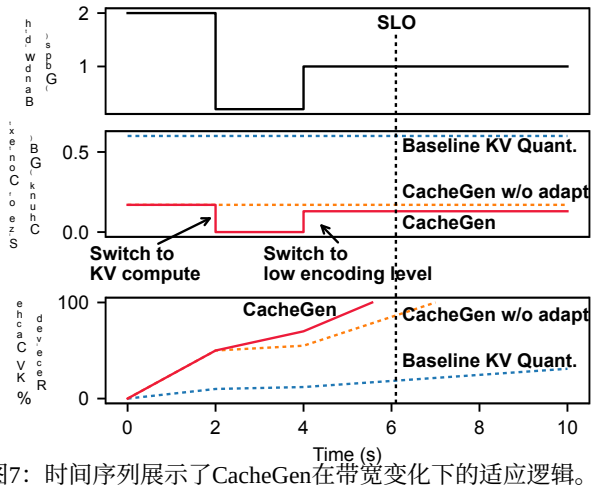


图7：时间序列展示了CacheGen在带宽变化下的适应逻辑。

频繁符号。为了提高效率，AC 需要 KV 缓存中元素的准确、低熵概率分布。

通过观察KV值在层、通道和标记位置的分布（§5.1.3），我们按通道和层对KV值进行分组，以获得概率分布。具体而言，我们的KV编码器离线为每个通道-层组合的增量张量配置一个单独的概率分布，并为由LLM生成的锚张量配置另一个概率分布，并对由同一LLM生成的所有KV缓存使用相同的分布。CacheGen使用修改过的AC库[101]和CUDA来加速编码和解码（§6）。在§7.5中，我们通过实验证明，我们的方法相比于使用一个全局符号分布的草案，能够将比特流大小减少多达53%。

5.3 KV缓存流适应

由于KV缓存的传输可能需要几百毫秒到几秒钟的时间，因此在传输过程中可用带宽可能会波动。因此，以固定编码级别流式传输编码的KV比特流可能会违反获取KV缓存的服务级别目标（SLO）[33]。例如，在图7中，在传输开始时，可用吞吐量为2 Gbps，如果带宽保持在2 Gbps，发送1 GB的KV流可以满足4秒的SLO。然而，在t = 2s时，吞吐量降至0.2 Gbps，并且在t = 4s时仅增加到1 Gbps，因此实际传输延迟从4秒增加到7秒，这违反了SLO。

工作流程：为了处理带宽的变化，CacheGen将上下文分割成多个连续标记的上下文块（简称块），并使用KV缓存编码器将每个块编码成多个不同编码（量化）级别的比特流，这些比特流可以独立解码（稍后解释）。这可以离线完成。当获取上下文时，CacheGen逐个发送这些块，每个块可以选择几种流配置之一（简称配置）：它可以以某种编码级别发送，或者可以以文本格式发送，以便让LLM重新计算K和V张量。

[4]In practice, SLO is defined on TTFT. Once the KV cache of the long context is loaded in GPU, the remaining delay of one forward pass is marginal [82].

CacheGen adapts the configuration of each chunk while streaming the KV cache to keep the transmission delay within an SLO. Figure 7 illustrates an example adaptation where CacheGen switches to sending text context and recomputing KV cache from the text at $t = 2s$ due to the bandwidth drop, and at $t = 4s$, since the bandwidth increases back to 1 Gbps, and CacheGen switch to sending KV bitstreams of subsequent chunks at a smaller size. With our adaptation logic (specific algorithm in §C.1), CacheGen can meet the SLO.

However, to adapt efficiently, several questions remain.

First, *how to stream multiple chunks at different streaming configurations without affecting compression efficiency?* To encode the chunks offline, CacheGen first computes the KV cache of the entire context (*i.e.,* prefill) and splits the K and V tensors of the KV cache along the token dimension into sub-tensors, each of which contains the layers and channels of the tokens in the same chunk. It then uses the KV encoder to encode the K or V sub-tensor of a chunk with different encoding (quantization) levels. Each chunk is encoded *independent* to other chunks *without* affecting the compression efficiency as long as a chunk is longer than a group of tokens. This is because encoding the KV tensor of a token only depends on itself and its delta with the anchor token of the group of tokens (§5.2). Thus, chunks sent with different encoding levels can be independently decoded and then concatenated to reconstruct the KV cache. In the case that a chunk is sent in text format, the LLM will compute its K and V tensors based on the previous chunk's KV tensors that have been received and decoded.[5]

Would streaming chunks at different configurations affect generation quality? If one chunk is sent at a smaller-sized encoding level than other chunks (due to low bandwidth), it will have high compression loss on *that* single chunk, but this will not affect the compression loss of other chunks. That said, we acknowledge that if the bandwidth is too low to send most chunks at a high encoding level, the quality will still suffer.

Second, *how long should a context chunk be?* We believe that the chunk length depends on two considerations.

1. The encoded KV bitstream of a chunk size should not be too big because, otherwise, it cannot react to bandwidth changes in a timely manner.
2. The chunk should not be too small either since then we can not fully utilize the batching ability of GPU to compute KV tensors if text format is chosen.

With these considerations in mind, we empirically pick 1.5K tokens as the default chunk length in our experiments[6], though more optimization may find better chunk lengths.

Thirdly, *how does CacheGen decide the streaming configuration of the next chunk?* CacheGen estimates the bandwidth by measuring the throughput of the previous chunk. It assumes this throughput will remain constant for the remaining chunks and calculates the expected delay for each streaming configuration accordingly. The expected delay is calculated by dividing its size by the throughput (more details in §C). If there are bandwidth fluctuations, CacheGen's reaction will be delayed by at most one chunk. Since one chunk is a small subset of the entire KV cache, this reaction is sufficiently fast to meet SLO (details in §7.4). It then picks the configuration that has the least compression loss (*i.e.,* text format or lowest encoding level) with an expected delay still within the SLO, and uses the configuration to send the next chunk. For the first chunk, if some prior knowledge of the network throughput is available, CacheGen will use it to choose the configuration of the first chunk the same way. Otherwise, CacheGen starts with a default medium encoding level (140 MB per chunk for Llama 7B, detailed setting in §C.2).

Finally, *how does CacheGen handle the streaming of multiple requests?* When multiple requests arrive concurrently within T seconds, CacheGen batches and streams them together. It can batch up to B requests, which is the maximum number that the GPU server can handle simultaneously. Each request is divided into chunks of the same size, even though the total number of chunks may differ among requests. For each chunk index c, CacheGen determines the number of requests N_c that include chunk c. Using the throughput measured for the previous chunk $c-1$, CacheGen calculates the expected delays for each configuration by multiplying N_c by the delay for a single request. On the GPU servers, the requests are batched by padding their KV caches and processing them together.

6 Implementation

We implement CacheGen with about 2K lines of code in Python, and about 1K lines of CUDA kernel code, based on PyTorch v2.0 and CUDA 12.0.

Integration into LLM inference framework: CacheGen operates the LLM through two interfaces:

- `calculate_kv(context) -> KVCache`: given a piece of context, CacheGen invokes LLM through this function to get the corresponding KV cache.
- `generate_with_kv(KVCache) -> text`: CacheGen passes a KV cache to the LLM and lets it generate the tokens while skipping the prefilling of the context.

We implement these two interfaces in HuggingFace models using the `transformers` library [64] with about 500 lines of Python code. Both interfaces are implemented based on the `generate` function provided by the library. For `calculate_kv`, we let LLM only calculate the KV cache without generating new text, by passing the options of `max_length = 0` and `return_dict_in_generate = True` when getting the KV cache. The `generate_with_kv` is implemented by simply passing the KV cache via the `past_key_values` argument when calling the `generate` function. Similar integrations are also applicable to other LLM libraries, such as FastChat [155], llama.cpp [98], and GGML [57].

We have also integrated CacheGen in LangChain [83], a popular LLM application framework. CacheGen is activated in the `_generate` function of LangChain's `BaseLLM` module. CacheGen first checks whether the KV cache of the current context already exists (explained shortly). If so, CacheGen invokes `generate_with_kv` to start generating new texts. Otherwise, CacheGen will invoke `calculate_kv` to create the KV cache first before generating new texts.

KV cache management in CacheGen: To manage the KV cache, CacheGen implements two modules:

[5] A similar concept has been used to split LLM input into prefill chunks for more efficient batching [17].

[6] The chunk length is also long enough for the KV bitstream of each chunk to fill the sender's congestion window in our experiment setting.

CacheGen 在流式传输 KV 缓存时调整每个块的配置，以保持传输延迟在 SLO 之内。图 7 说明了一个适应示例，其中 CacheGen 在带宽下降时切换到发送文本上下文并从文本重新计算 KV 缓存，发生在 $t = 2s$，而在 $t = 4s$ 时，由于带宽恢复到 1 Gbps，CacheGen 切换到以更小的大小发送后续块的 KV 比特流。通过我们的适应逻辑（具体算法见 §C.1），CacheGen 可以满足 SLO。

然而，为了有效适应，仍然存在几个问题。

首先，如何在不同的流配置下流式传输多个块而不影响压缩效率？为了离线编码这些块，CacheGen 首先计算整个上下文的 KV 缓存（即预填充），并沿着令牌维度将 KV 缓存的 K 和 V 张量拆分为子张量，每个子张量包含同一块中令牌的层和通道。然后，它使用 KV 编码器以不同的编码（量化）级别对块的 K 或 V 子张量进行编码。每个块的编码是独立于其他块的，只要一个块的长度超过一组令牌，就不会影响压缩效率。这是因为对令牌的 KV 张量的编码仅依赖于它自身及其与该组令牌的锚点令牌的差异 (§5.2)。因此，使用不同编码级别发送的块可以独立解码，然后连接以重建 KV 缓存。如果一个块以文本格式发送，LLM 将根据之前接收和解码的块的 KV 张量计算其 K 和 V 张量。

流式传输不同配置的块会影响生成质量吗？如果一个块以比其他块更小的编码级别发送（由于带宽低），那么该单个块将会有较高的压缩损失，但这不会影响其他块的压缩损失。也就是说，我们承认如果带宽过低，无法以高编码级别发送大多数块，质量仍然会受到影响。

第二，语境块应该有多长？我们认为块的长度取决于两个因素。

1. 一段大小的编码KV比特流不应过大，因为否则它无法及时响应带宽变化。

2. 块也不应太小，因为如果选择文本格式，我们就无法充分利用 GPU 的批处理能力来计算 KV 张量。

考虑到这些因素，我们在实验中经验性地选择1.5K个标记作为默认的块长度6，尽管更多的优化可能会找到更好的块长度。

第三，CacheGen 如何决定下一个数据块的流配置？CacheGen 通过测量前一个数据块的吞吐量来估计带宽。它假设这个吞吐量在剩余的数据块中将保持不变，并相应地计算每个流配置的预期延迟。预期延迟通过将其大小除以吞吐量来计算（更多细节见 §C）。如果带宽波动，CacheGen 的反应最多会延迟一个数据块。由于一个数据块是一个整个KV缓存的小子集，这种反应足够快以满足SLO（详细信息见§7.4）。然后，它选择压缩损失最小的配置（即文本格式或最低编码级别），并且预期延迟仍在SLO范围内，并使用该配置发送下一个块。对于第一个块，如果有网络吞吐量的先前知识可用，CacheGen将以相同的方式选择第一个块的配置。否则，CacheGen将从默认的中等编码级别开始（对于Llama 7B，每个块140 MB，详细设置见§C.2）。

最后，CacheGen 如何处理多个请求的流式传输？当多个请求在 T 秒内同时到达时，CacheGen 会将它们批量处理并一起流式传输。它最多可以批量处理 B 个请求，这是 GPU 服务器可以同时处理的最大数量。每个请求被划分为相同大小的块，尽管请求之间的块总数可能不同。对于每个块索引 c，CacheGen 确定包含块 c 的请求数量 N_c。使用之前块 $c-1$ 测量的吞吐量，CacheGen 通过将 N_c 乘以单个请求的延迟来计算每个配置的预期延迟。在 GPU 服务器上，请求通过填充它们的 KV 缓存并一起处理来进行批量处理。

6 实施

我们在Python中实现了大约2000行代码的CacheGen，并在基于PyTorch v2.0和CUDA 12.0的基础上实现了大约1000行的CUDA内核代码。

集成到LLM推理框架中：CacheGen通过两个接口操作LLM：

- calculate_kv(context) -> KVCache：给定一段上下文，CacheGen通过此函数调用LLM以获取相应的KV缓存。
- generate_with_kv(KVCache) -> 文本：CacheGen 将 KV 缓存传递给 LLM，并让其生成令牌，同时跳过上下文的预填充。

我们在HuggingFace模型中使用transformers库[64]实现了这两个接口，代码大约有500行Python。两个接口都是基于库提供的generate函数实现的。对于calculate_kv，我们让LLM仅计算KV缓存而不生成新文本，通过在获取KV缓存时传递max_length = 0和return_dict_in_generate = True的选项。generate_with_kv的实现是通过在调用generate函数时简单地通过past_key_values参数传递KV缓存。类似的集成也适用于其他LLM库，如FastChat [155]、llama.cpp [98]和GGML [57]。

我们还在 LangChain [83] 中集成了 CacheGen，这是一个流行的 LLM 应用框架。CacheGen 在 LangChain 的 BaseLLM 模块的 _generate 函数中被激活。CacheGen 首先检查当前上下文的 KV 缓存是否已经存在（稍后会解释）。如果存在，CacheGen 将调用 generate_with_kv 开始生成新文本。否则，CacheGen 将先调用 calculate_kv 创建 KV 缓存，然后再生成新文本。

在CacheGen中的KV缓存管理：为了管理KV缓存，CacheGen实现了两个模块：

[5]A similar concept has been used to split LLM input into prefill chunks for more efficient batching [17].
[6]The chunk length is also long enough for the KV bitstream of each chunk to fill the sender's congestion window in our experiment setting.

Dataset	Size	Med.	Std.	P95
LongChat [90]	200	9.4K	164	9.6K
TriviaQA [75]	200	9.3K	4497	15K
NarrativeQA [81]	200	14K	1916	15K
WikiText [102]	62	5.9K	4548	14.8K

Table 2: *Size and context lengths of datasets in the evaluation.*

- `store_kv(LLM) -> {chunk_id: encoded_KV}`: calls `caculate_kv`, splitting the returned KV cache into context chunks, and encodes each chunk. Then, it stores a dictionary on the storage server, where it maps the `chunk_id` to the encoded bitstreams for the K and V tensors for the corresponding chunk.
- `get_kv(chunk_id) -> encoded_KV` fetches the encoded KV tensors corresponding to `chunk_id` on the storage server and transmits it to the inference server.

Whenever a new piece of context comes in, CacheGen first calls `store_kv`, which first generates the KV cache, and then stores the encoded bitstreams on the storage server. At run time, CacheGen calls `get_kv` to fetch the corresponding chunk of KV cache and feed into `generate_with_kv`.

Speed optimization for CacheGen: To speed up the encoding and decoding of KV cache, we implemented a GPU-based AC library [101] with CUDA to speed up encoding and decoding. Specifically, each CUDA thread is responsible for encoding/decoding the KV cache from the bitstream of one token. The probability distributions are obtained by counting the frequencies of quantized symbols in the KV feature for the corresponding context. We also pipeline the transmission of context chunk i with the decoding of context chunk $i-1$.

7 Evaluation

The key takeaways of our evaluation are:

- Across four datasets and three models, CacheGen can reduce TTFT (including both network and compute delay) by 3.1-4.7× compared to prefill from text context, and by 3.2-3.7× compared to the quantization baseline (§7.2).
- CacheGen's KV encoder reduces the bandwidth for transferring KV cache by 3.5-4.3× compared to the quantization baseline (§7.2).
- CacheGen's reduction in bandwidth usage is still effective when applied to recent context compression baselines [72, 153]. CacheGen further reduces the bandwidth usage by 3.3-4.2×, compared to applying quantization on context compression baselines (§7.2).
- CacheGen's improvement is significant across various workloads, including different context lengths, network bandwidths, and numbers of concurrent requests (§7.3).
- CacheGen's decoding overhead is minimal, in delay and compute, compared with LLM inference itself (§7.5).

7.1 Setup

Models: We evaluate CacheGen on three models of different sizes, specifically the fine-tuned versions of Mistral-7B, Llama-34B, and Llama-70B. All models are fine-tuned such that they can take long contexts (up to 32K). We did not test CacheGen on other LLMs (*e.g.,* OPT, BLOOM) because there are no public fine-tuned versions for long contexts to our best knowledge.

Datasets: We evaluate CacheGen on 662 contexts from four different datasets with different tasks (Table 2):

- *LongChat:* The task is recently released [90] to test LLMs on queries like "What was the first topic we discussed?" by using all the previous conversations as the context. Most contexts are around 9.2-9.6K tokens.
- *TriviaQA:* The task tests the reading comprehension ability of the LLMs [29], by giving the LLMs a single document (context), and letting it answer questions based on it. The dataset is part of the LongBench benchmark [29] suite.
- *NarrativeQA:* The task is used to let LLMs answer questions based on stories or scripts, provided as a single document (context). The dataset is also part of LongBench.
- *Wikitext:* The task is to predict the probability of the next token in a sequence based on the context consisting of relevant documents that belong to a specific Wiki page [102].

The dataset we used to design CacheGen's encoder is a subset of the datasets we used to evaluate CacheGen. This is for showing the insights in §5.1 are generalizable to different datasets.

Quality metrics: We measure generation quality using the standard metric of each dataset.

- *Accuracy* is used to evaluate the model's output on the LongChat dataset. The task predicts the first topic in the conversational history between the user and the LLM. The accuracy is defined as the percentage of generated answers that exactly includes the ground-truth topic.
- *F1 score* is used to evaluate the model's response in the TriviaQA and NarrativeQA datasets. It measures the probability that the generated answer matches the ground-truth answer of the question-answering task.
- *Perplexity* is used to evaluate the model's performance on the Wikitext dataset. The perplexity is defined as the exponentiated average negative log-likelihood of the next token [28, 41]. A low perplexity means that the model likely generates the next token correctly. While perplexity does not equate to text-generation quality, it is widely used as a proxy [13] to test the impact of pruning or quantizing LLMs on generation performance [48, 96, 116, 142].

System metrics: We compare CacheGen with baselines with two system-wise metrics.

- *Size of KV cache* is the size of the KV cache after compression, this measures the bandwidth needed to load KV caches.
- *Time-to-first-token (TTFT)* is the time from the arrival of the user query to the generation of the first token. This includes the loading delay of the KV cache and the prefill delay of the new questions. This is a metric widely used in industry [14, 25, 77] and recent works [58, 93].

Baselines: We compare CacheGen with baselines that do not change the contexts or model (more baselines in §7.5).

- *"Default quantization"* uses the uniform quantization of KV cache, specifically the same quantization level (*i.e.,* 3, 4, 8 bits) for every layer in the LLM (which was used in [120]).

Dataset	Size	Med.	Std.	P95
LongChat [90]	200	9.4K	164	9.6K
TriviaQA [75]	200	9.3K	4497	15K
NarrativeQA [81]	200	14K	1916	15K
WikiText [102]	62	5.9K	4548	14.8K

表 2：评估中数据集的大小和上下文长度。

- store_kv(LLM) –> {chunk_id: encoded_KV}：调用caculate_kv，将返回的KV缓存拆分为上下文块，并对每个块进行编码。然后，它在存储服务器上存储一个字典，将chunk_id映射到对应块的K和V张量的编码比特流。
- get_kv(chunk_id) –> 编码的 KV 从存储服务器获取与 chunk_id 对应的编码 KV 张量，并将其传输到推理服务器。

每当有新的上下文进来时，CacheGen 首先调用 store_kv，后者首先生成 KV 缓存，然后将编码的比特流存储在存储服务器上。在运行时，CacheGen 调用 get_kv 以获取相应的 KV 缓存块并输入到 generate_with_kv 中。

CacheGen的速度优化：为了加快KV缓存的编码和解码，我们实现了一个基于GPU的AC库[101]，使用CUDA来加速编码和解码。具体来说，每个CUDA线程负责从一个令牌的比特流中编码/解码KV缓存。通过计算相应上下文中量化符号的频率来获得概率分布。我们还将上下文块i的传输与上下文块$i-1$的解码进行流水线处理1。

7 评估

我们评估的关键要点是：

- 在四个数据集和三个模型中，与从文本上下文进行预填充相比，CacheGen可以将TTFT（包括网络和计算延迟）减少3.1-4.7×，与量化基线相比减少3.2-3.7×（§7.2）。
- CacheGen的KV编码器将KV缓存的传输带宽减少了3.5-4.3×，与量化基线相比（§7.2）。
- CacheGen 在应用于最近的上下文压缩基准时，带宽使用的减少仍然有效 [72, 153]。与在上下文压缩基准上应用量化相比，CacheGen 进一步减少了 3.3-4.2× 的带宽使用 (§7.2)。
- CacheGen的改进在各种工作负载中都很显著，包括不同的上下文长度、网络带宽和并发请求的数量（§7.3）。
- CacheGen的解码开销在延迟和计算方面相较于LLM推理本身是最小的（§7.5）。

7.1 设置

模型：我们在三种不同规模的模型上评估了 CacheGen，具体来说是 Mistral-7B、Llama-34B 和 Llama-70B 的微调版本。所有模型都经过微调，以便能够处理长上下文（最多 32K）。据我们所知，我们没有在其他 LLM（例如 OPT、BLOOM）上测试 CacheGen，因为没有公开的适用于长上下文的微调版本。

数据集：我们在来自四个不同数据集的662个上下文中评估CacheGen，这些数据集具有不同的任务（表2）：

- LongChat：该任务最近发布[90]，旨在通过使用所有先前的对话作为上下文，测试LLM在诸如“我们讨论的第一个主题是什么？”这样的查询。大多数上下文大约在9.2-9.6K个标记之间。
- TriviaQA：该任务通过给LLMs提供一份单一文档（上下文），并让其根据该文档回答问题，来测试LLMs的阅读理解能力[29]。该数据集是LongBench基准[29]套件的一部分。
- NarrativeQA：该任务用于让LLMs根据故事或剧本回答问题，这些内容以单个文档（上下文）的形式提供。该数据集也是LongBench的一部分。
- 任务是根据属于特定维基页面的相关文档的上下文，预测序列中下一个标记的概率 {v*}。

我们用于设计CacheGen编码器的数据集是我们用于评估CacheGen的数据集的一个子集。这是为了表明§5.1中的见解可以推广到不同的数据集。

质量指标：我们使用每个数据集的标准指标来衡量生成质量。

- 准确性用于评估模型在 LongChat 数据集上的输出。该任务预测用户与 LLM 之间对话历史中的第一个主题。准确性定义为生成的答案中完全包含真实主题的百分比。
- F1 分数用于评估模型在 Trivi- aQA 和 NarrativeQA 数据集中的响应。它衡量生成的答案与问答任务的真实答案匹配的概率。
- 困惑度用于评估模型在Wikitext数据集上的表现。困惑度被定义为下一个标记的指数化平均负对数似然 [28, 41]。低困惑度意味着模型可能正确生成下一个标记。虽然困惑度并不等同于文本生成质量，但它被广泛用作测试修剪或量化LLM对生成性能影响的代理 [13] [48, 96, 116, 142]。

系统指标：我们将CacheGen与基线进行比较，使用两个系统级指标。

- KV缓存的大小是压缩后KV缓存的大小，这测量了加载KV缓存所需的带宽。
- 首次令牌时间（TTFT）是指从用户查询到达到生成第一个令牌的时间。这包括KV缓存的加载延迟和新问题的预填充延迟。这是一个在行业中广泛使用的指标 [14, 25, 77] 和最近的研究 [58, 93]。

基线：我们将CacheGen与不改变上下文或模型的基线进行比较（更多基线见§7.5）。

- “默认量化”使用KV缓存的均匀量化，具体来说，对于LLM中的每一层使用相同的量化级别（即3、4、8位）（在[120]中使用）。

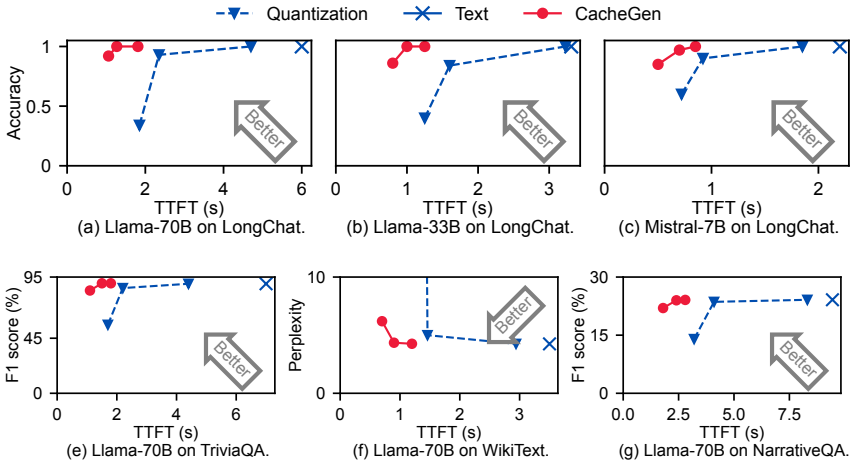


Figure 8: Time-to-first-token (TTFT): *Across different models and different datasets, CacheGen reduces TTFT with little negative impacts on quality (in accuracy, perplexity or F1 score).*

- *"Text context"* fetches the text of the context and feeds it to LLM to generate the KV cache for it. It represents the design of minimizing data transmission but at the expense of high computation overhead. We use the state-of-the-art inference engine, vLLM [82], to run the experiments. vLLM's implementation already uses xFormers [85], which includes speed and memory-optimized Transformers CUDA kernels and has shown much faster prefill delay than HuggingFace Transformers. This is a very competitive baseline.
- *"Context compression"* either drops tokens in the text context (LLMlingua [72]) or in the KV cache (H2O [153]).

Hardware settings: We use an NVIDIA A40 GPU server with four GPUs to benchmark our results. The server is equipped with 384GB of memory and two Intel(R) Xeon(R) Gold 6130 CPUs with Hyper-threading and Turbo Boost enabled by default.

7.2 Overall improvement

We first show the improvement of CacheGen over the baselines, as described in §7.1.

TTFT reduction: Figure 8 demonstrate CacheGen's ability to reduce TTFT, across three models and four datasets. Under bandwidth of 3 Gbps, compared to text context, CacheGen is able to reduce TTFT by 3.1-4.7×. Compared to default quantization, CacheGen is able to reduce TTFT by 3.2-3.7×.

It is important to note that even compared with 8-bit quantization, an almost lossless KV cache compression technique across the four datasets, CacheGen can still reduce the TTFT by 1.67-1.81×. CacheGen's reduction in TTFT is a result of a shorter transmission delay to send the smaller KV caches.

Reduction on KV cache size: Figure 8 show that, across four datasets and three models, CacheGen's KV encoder reduces the KV cache size by 3.5-4.3× compared to default quantization when achieving similar performance for downstream tasks after decoding. Thus, it achieves better quality-size trade-offs across different settings. The degradation caused by lossy compression is marginal—the degradation is no more than 2% in accuracy, less than 0.1% in F1 score, and less than 0.1 in perplexity [65].

Some example text outputs for different baselines are available in §A.

Gains over context compression baselines: We also apply CacheGen to further reduce the size of context compression baselines' KV cache, including H2O and LLMlingua. Note that H2O drops tokens from KV cache which have low attention scores. Specifically, it requires the query tensors of the prompt to compute the attention scores in order to determine which tokens to drop. The query tensors of the prompts are not present in the offline compression stage. In our experiments, we implement an *idealized* version of H2O, where the query tensors of the prompts are used in the offline compression stage.

As shown in Figure 10, compared to the context compression baseline, H2O [153], CacheGen can further reduce compressed KV cache (in floating point). Specifically, CacheGen reduces the size of KV cache by 3.5–4× compared to the H2O's quantized KV caches, and 3.3–4.2× compared to LLMlingu's quantized KV caches, without losing quality. This suggests that even after condensing contexts by H2O and LLMlingua, the resulting KV caches may still have the statistical observations behind CacheGen's KV encoder. Thus, the techniques used in CacheGen's encoder remain beneficial when we encode the KV cache after applying these techniques.

Understanding CacheGen's improvements: CacheGen outperforms various baselines for slightly different reasons. Compared to the text context baseline, CacheGen has lower TTFT, because it reuses KV cache to avoid the long prefill delay for processing long contexts. Compared to the basic quantization baseline, CacheGen compresses KV cache with layer-wise dynamic quantization and further encodes the KV cache tensors into bitstreams, thus able to reduce the transmission delay.

Finally, compared to H2O and LLMlingua, two recent context-condensing techniques, CacheGen can still compress the KV cache produced by H2O. In short, H2O and other context-condensing techniques all prune contexts at the token level and their resulting KV caches are in the form of floating-point tensors, so CacheGen is complementary and can be used to further compress the KV cache into much more compact bitstreams.

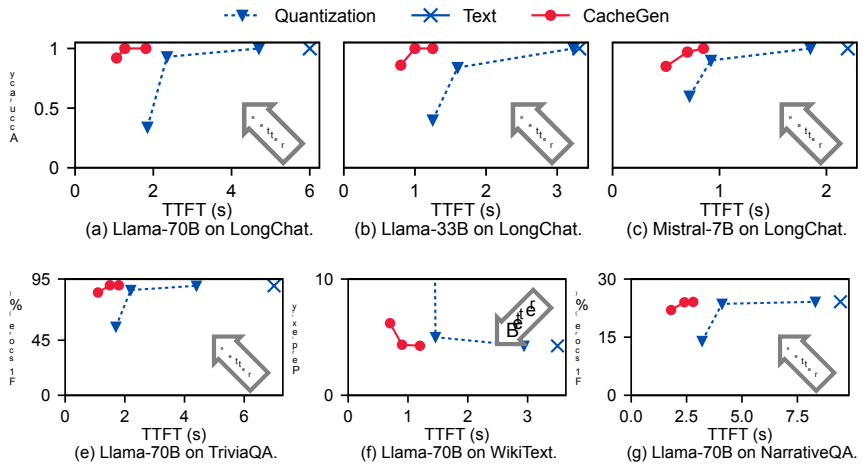


图8：首次令牌时间（TTFT）：在不同模型和不同数据集之间，CacheGen 减少了 TTFT，对质量（准确性、困惑度或 F1 分数）几乎没有负面影响。

- “文本上下文”获取上下文的文本并将其提供给LLM以生成KV缓存。它代表了最小化数据传输的设计，但以高计算开销为代价。我们使用最先进的推理引擎vLLM [82]来进行实验。vLLM的实现已经使用了xFormers [85]，其中包括速度和内存优化的Transformers CUDA内核，并且显示出比HuggingFace Transformers更快的预填充延迟。这是一个非常有竞争力的基线。

- “上下文压缩”要么在文本上下文中丢弃标记(LLMlingua [72]) 或在 KV 缓存中 (H2O [153])。

硬件设置：我们使用一台配备四个GPU的NVIDIA A40 GPU服务器来基准测试我们的结果。该服务器配备384GB内存和两颗默认启用超线程和Turbo Boost的Intel(R) Xeon(R) Gold 6130 CPU。

7.2 整体改善

我们首先展示CacheGen相对于基线的改进，如§7.1所述。

TTFT减少：图8展示了CacheGen在三个模型和四个数据集上减少TTFT的能力。在3 Gbps的带宽下，与文本上下文相比，CacheGen能够将TTFT减少3.1-4.7×。与默认量化相比，CacheGen能够将TTFT减少3.2-3.7×。

值得注意的是，即使与8位量化相比，跨四个数据集的几乎无损KV缓存压缩技术，CacheGen仍然可以将TTFT减少1.67-1.81×。CacheGen在TTFT上的减少是由于发送更小的KV缓存所需的传输延迟更短。

KV缓存大小的减少：图8显示，在四个数据集和三个模型中，CacheGen的KV编码器在解码后实现类似的下游任务性能时，将KV缓存大小减少了3.5-4.3×，与默认量化相比。因此，它在不同设置下实现了更好的质量-大小权衡。由于有损压缩造成的降级是微不足道的——准确率的降级不超过2%，F1分数的降级少于0.1%，困惑度的降级少于0.1[65]。

不同基准的某些示例文本输出可在§A中找到。

在上下文压缩基线上的增益：我们还将CacheGen应用于进一步减少上下文压缩基线的KV缓存的大小，包括H2O和LLMlingua。请注意，H2O会从KV缓存中删除注意力分数低的令牌。具体来说，它需要提示的查询张量来计算注意力分数，以确定要删除哪些令牌。提示的查询张量在离线压缩阶段不存在。在我们的实验中，我们实现了H2O的理想化版本，其中提示的查询张量在离线压缩阶段中使用。

如图10所示，与上下文压缩基线H2O [153]相比，CacheGen可以进一步减少压缩的KV缓存（以浮点数表示）。具体而言，CacheGen将KV缓存的大小减少了3.5–4×，与H2O的量化KV缓存相比，减少了3.3–4.2×，与LLMlingu的量化KV缓存相比，且没有损失质量。这表明，即使在H2O和LLMlingua压缩上下文之后，生成的KV缓存仍可能具有CacheGen的KV编码器背后的统计观察。因此，在应用这些技术后，我们在编码KV缓存时，CacheGen的编码器中使用的技术仍然是有益的。

理解CacheGen的改进：CacheGen在不同的基准测试中表现优异，原因略有不同。与文本上下文基准相比，CacheGen的TTFT更低，因为它重用KV缓存以避免处理长上下文时的长预填充延迟。与基本量化基准相比，CacheGen通过层级动态量化压缩KV缓存，并进一步将KV缓存张量编码为比特流，从而能够减少传输延迟。

最后，与 H2O 和 LLMlingua 这两种最近的上下文压缩技术相比，CacheGen 仍然可以压缩 H2O 产生的 KV 缓存。简而言之，H2O 和其他上下文压缩技术都是在标记级别修剪上下文，它们生成的 KV 缓存以浮点张量的形式存在，因此 CacheGen 是互补的，可以进一步将 KV 缓存压缩成更紧凑的比特流。

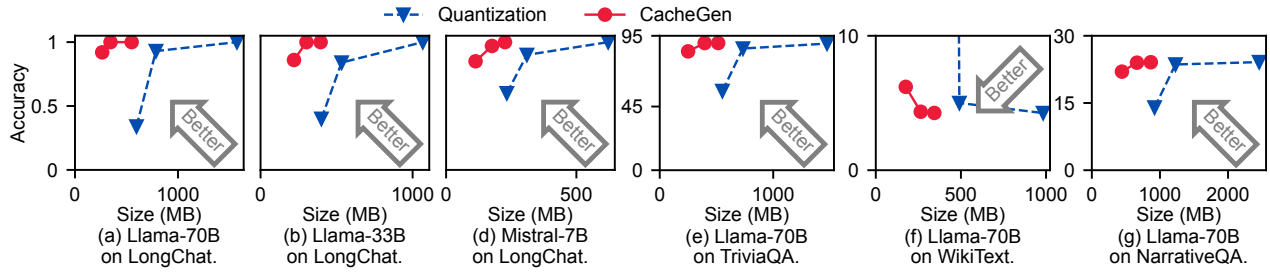


Figure 9: Reducing KV cache size: *Across various models, CacheGen reduces size of KV cache with little accuracy decrease on various datasets.*

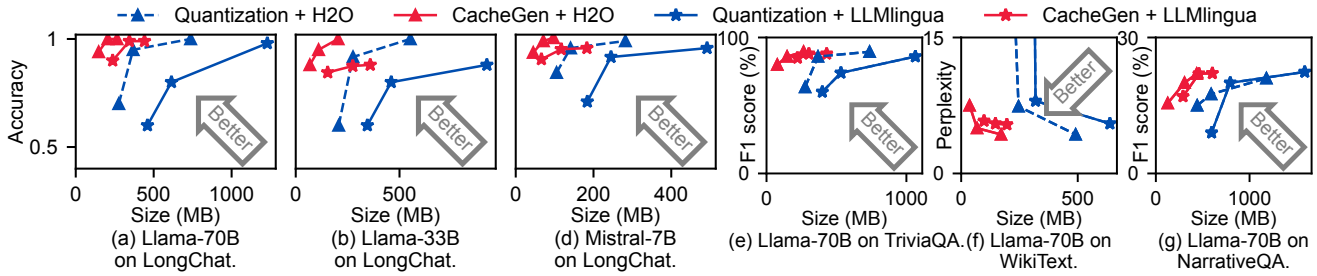


Figure 10: Reducing KV cache size *on top of* H2O [153] and LLMlingu [72]: *Across different models, CacheGen further the size of KV cache, compared to the KV cache shortened by H2O, with little accuracy decrease on different datasets.*

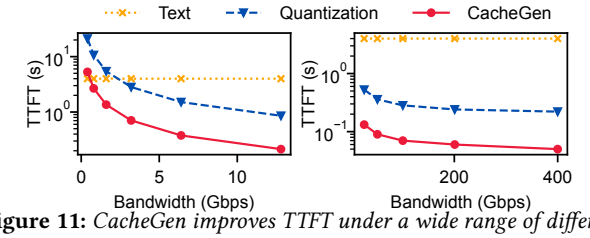


Figure 11: *CacheGen improves TTFT under a wide range of different bandwidths. Plotted with Mistral-7B. y-axis is log scale.*

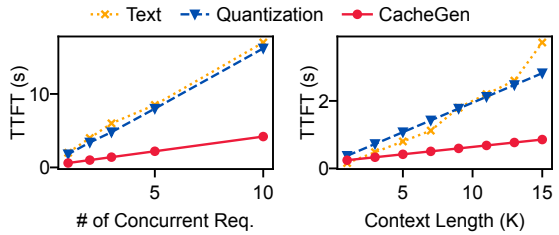


Figure 12: *CacheGen consistently reduces TTFT when there are multiple concurrent requests on one GPU. Plotted with Mistral-7B.*

7.3 Sensitivity analysis

Available bandwidth: The left and right figures in Figure 11 compare the TTFT of CacheGen with baselines under a wide range of bandwidth from 0.4–15 Gbps and 15–400 Gbps, while we fix the context length at 16K tokens. We can see that CacheGen consistently outperforms baselines under almost all bandwidth situations. Arguably, the *absolute reduction* in TTFT becomes smaller under high bandwidth (over 20Gbps), compared to the quantization baseline, since both the quantization baseline and CacheGen can transfer KV caches much faster.

Number of concurrent requests: The left side of Figure 12 shows the TTFT under different numbers of concurrent requests. When the

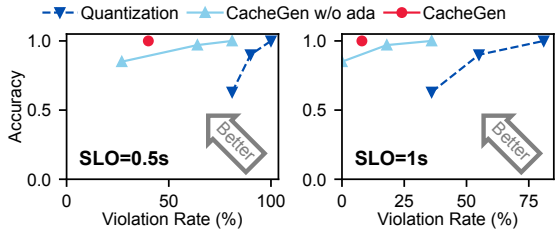


Figure 13: *CacheGen reduces SLO violation rate over CacheGen without adaptation and the quantization baseline. Plotted with Mistral-7B model.*

number of concurrent requests increases (*i.e.,* fewer available GPU cycles for one individual query), CacheGen significantly reduces TTFT than the baselines. This is because the amount of computation required for prefilling on a long input (9.6K in this case) is huge, as discussed in §2.2. §D shows CacheGen's improvement over a complete space of workloads of different bandwidth and GPU resources.

Context lengths: The right side of Figure 12 compares CacheGen's TTFT with the baselines under different input lengths from 0.1K to 15K tokens under a fixed network bandwidth of 3 Gbps. When the context is long, the gain of CacheGen mainly comes from reducing the KV cache sizes. And when the context is short (below 1K), CacheGen will automatically revert to loading the text context as that yields a lower TTFT.

7.4 KV streamer adaptation

The adaptation logic described in §5.3 allows CacheGen to adapt to bandwidth changes and achieve good quality while meeting the SLO on TTFT. In Figure 13, we generate bandwidth traces where each context chunk's bandwidth is sampled from a random distribution

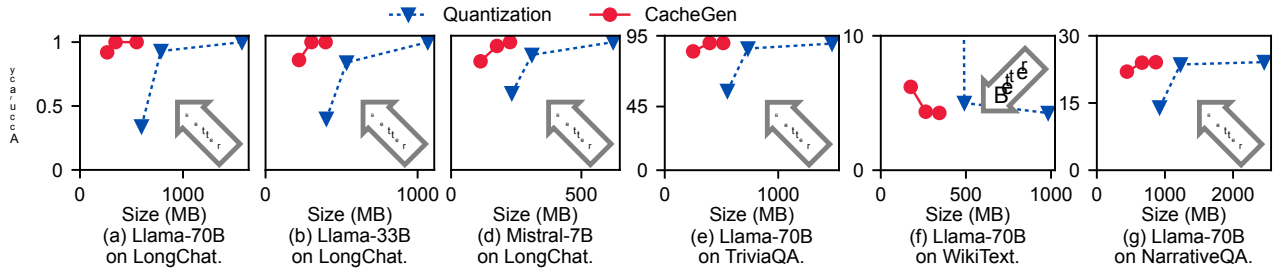


图 9：减少 KV 缓存大小：在各种模型中，CacheGen 在几乎不影响准确性的情况下减少了 KV 缓存数据集上，准确性下降。

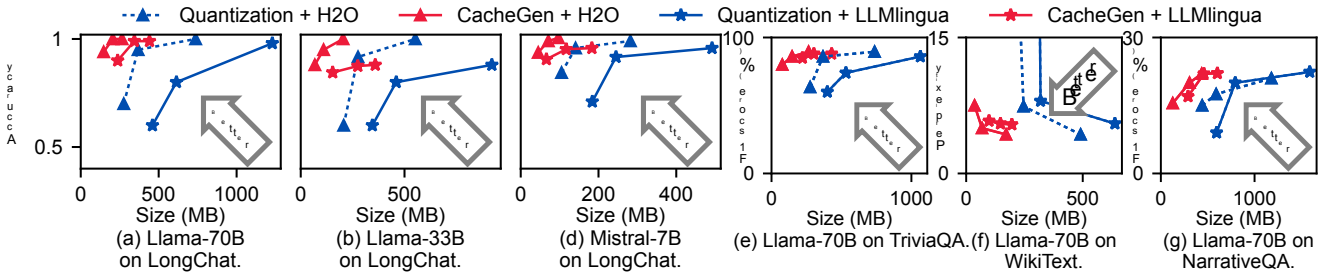


图10：在H2O [153] 和LLMlingu [72] 上减少KV缓存大小：在不同模型中，CacheGen进一步减小了{v*}的大小。KV缓存，与H2O缩短的KV缓存相比，在不同数据集上的准确性下降很小。

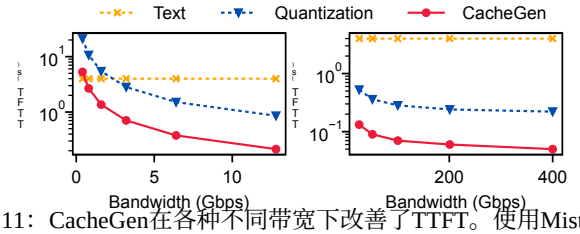


图11：CacheGen在各种不同带宽下改善了TTFT。使用Mistral-7B绘制。y轴为对数刻度。

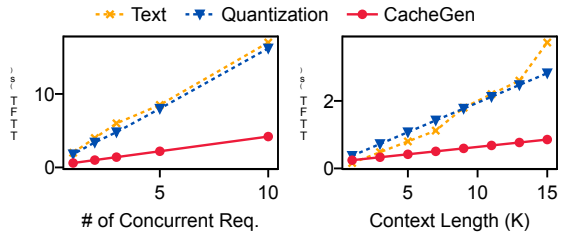


图12：当一个GPU上有多个并发请求时，CacheGen始终减少TTFT。图中使用Mistral-7B绘制。

7.3 敏感性分析

可用带宽：图11中的左侧和右侧图形比较了CacheGen与基线在0.4–15 Gbps和15–400 Gbps的广泛带宽范围内的TTFT，同时我们将上下文长度固定为16K个标记。我们可以看到，CacheGen在几乎所有带宽情况下始终优于基线。可以说，在高带宽（超过20Gbps）下，与量化基线相比，TTFT的绝对减少变得更小，因为量化基线和CacheGen都可以更快地传输KV缓存。

并发请求数：图12的左侧显示了在不同并发请求数下的TTFT。当{v*}

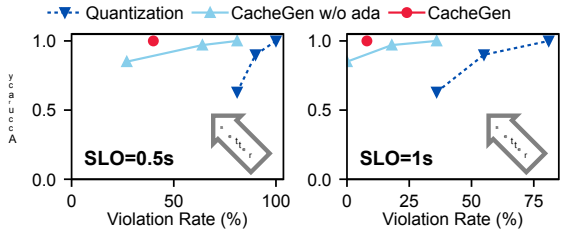


图13：CacheGen在没有适应和量化基线的情况下减少了SLO违例率。使用Mistral-7B模型绘制。

并发请求的数量增加（即，单个查询可用的 GPU 周期减少），CacheGen 显著降低了 TTFT，相比于基线。这是因为在长输入（在这种情况下为 9.6K）上预填充所需的计算量巨大，如 §2.2 中所讨论的。§D 显示了 CacheGen 在不同带宽和 GPU 资源的完整工作负载空间中的改进。

上下文长度：图12的右侧比较了CacheGen的TTFT与不同输入长度（从0.1K到15K个标记）下的基线，网络带宽固定为3 Gbps。当上下文较长时，CacheGen的收益主要来自于减少KV缓存的大小。而当上下文较短（低于1K）时，CacheGen会自动恢复为加载文本上下文，因为这样会产生更低的TTFT。

7.4 KV 放电管适配

§5.3中描述的适应逻辑使CacheGen能够适应带宽变化，并在满足TTFT的SLO的同时实现良好的质量。在图13中，我们生成带宽轨迹，其中每个上下文块的带宽是从随机分布中采样的。

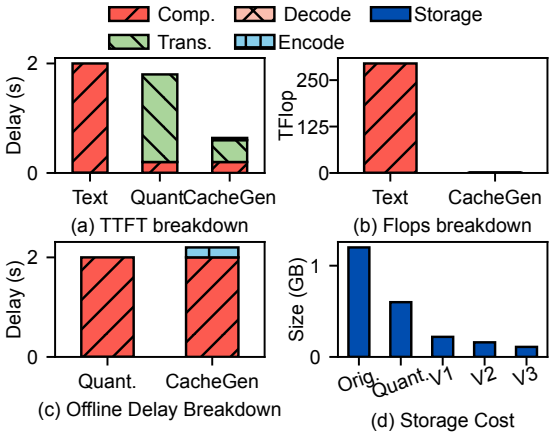


Figure 14: *(a) The breakdown of TTFT for text context, quantization baseline, and CacheGen. (b) Computation overhead of the text baseline and CacheGen. (c) Offline delay breakdown for baseline quantization and CacheGen. (d) The storage cost for CacheGen, quantization baseline and the uncompressed KV cache. Plotted with Mistral-7B.*

of 0.1 – 10 Gbps. Each point is averaged across 20 bandwidth traces on the LongChat dataset. We can see that CacheGen significantly outperforms the quantization baseline and CacheGen without adaptation. Specifically, given an SLO on the TTFT of 0.5s, CacheGen reaches the same quality as the quantization baseline with a 60% lower SLO violation rate. Under an SLO of 1s, CacheGen reaches the same quality as the quantization baseline, while reducing the SLO violation rate from 81% to 8%. The reason why CacheGen has a lower SLO violation rate is that when the bandwidth drops, CacheGen can dynamically reduce the quantization level or fall back to the configuration of computing text from scratch, while the quantization baseline and CacheGen without adaptation cannot.

7.5 Overheads and microbenchmarks

Decoding overhead: While having a better size-quality and TTFT-quality trade-off, CacheGen requires an extra decoding (decompression) step compared to the quantization baseline. CacheGen minimizes the decoding overhead by accelerating it with GPU-based implementation and pipelining the decoding of context chunks with the transmission of the context chunks, so as shown in Figure 14a, the decoding has minimal impact on the end-to-end delay. It is also important to note that although CacheGen's decoding is performed on GPU (see §6), the amount of computation needed by CacheGen's decoding module is negligible compared to the baseline that generates KV cache from text context.

Offline encoding and storage overheads: Unlike prior methods that compress each context only once, CacheGen compresses it into multiple versions (§5.3). CacheGen compresses each context almost as fast as the baselines because the encoding delay is very small (200 ms), as shown in Figure 14c. Figure 14d evaluates the overhead in storage. We can see that despite needing to encode and store multiple bitstream representations, the total storage cost for CacheGen is on par with the quantization baseline.

Ablation Study: To study the impact of individual components in CacheGen's KV encoder, Figure 15 progressively adds each idea

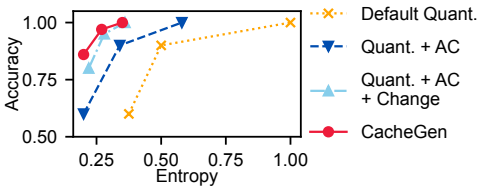


Figure 15: *Contributions of individual ideas behind KV encoder: change-based encoding, layer-wise quantization, and AC based on channel-layer grouping.*

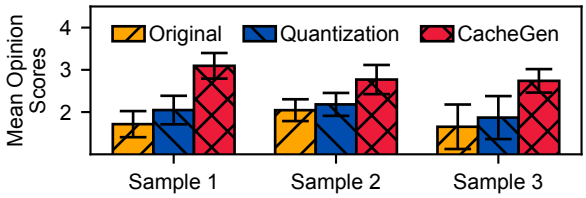


Figure 16: *Real user study shows CacheGen improves QoE significantly over other baselines.*

into the baseline of uniform quantization and default AC, starting with the use of our AC that uses probability distribution for each channel-layer combination, then change-based encoding, and finally layer-wise quantization. As shown in the figure, CacheGen's AC and change-based encoding significantly improve upon the uniform quantization. This indicates that removing the constraint of maintaining the tensor format of KV cache, and encoding them into bitstreams with our change-based encoding and AC can further reduce the size of KV cache after quantization.

Quality of Experience: We performed an IRB-approved user study to validate the effectiveness of CacheGen. We selected three conversation histories from the LongChat dataset used in previous evaluations. For each user, we first present the conversation history with ChatGPT. Then we show the same response but produced by different pipelines by adding different TTFTs and letting users rate the quality of response. With 270 ratings collected from Amazon MTurk [66], we show that CacheGen consistently outperforms other pipelines in QoE with shorter TTFT in Figure 16.

Evaluation results of CacheGen with more baselines are available §B, including using a smaller-sized model to speed up TTFT and Gisting, another context-shrinking technique.

8 Related Work

Faster LLM serving: Most LLM systems research aims to speed up LLM training [114, 122] or make serving systems faster. CacheGen aims at speeding up LLM serving systems by focusing on TTFT reduction. Others explore approximately parallelizing generation [86, 103], accelerating inference on edge devices [148], quantizing LLM weights [21], reducing memory I/O of GPU on-chip SRAM [47] and reducing self-attention computation complexity [116], better scheduling strategies [17, 111, 139, 149, 157], and GPU memory utilization [82]. Another line of work optimizes the communication delay of transmitting KV cache between GPUs, either by smart model parallelism strategies [111, 157] or by implementing a new attention operation [91]. This operation transmits query vectors to the GPUs that host smaller blocks of KV cache during the decoding phase. A common approach for faster inference without modifying

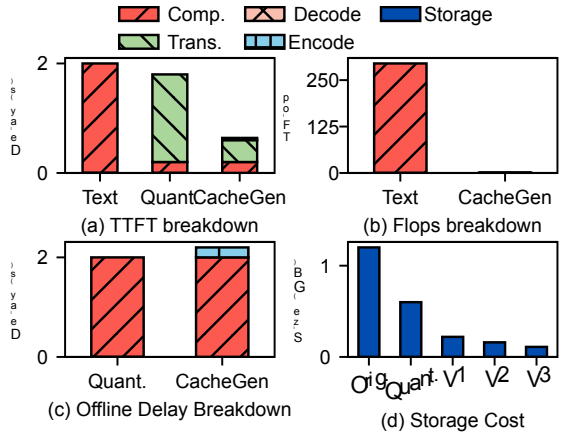


图14：（a）文本上下文、量化基线和CacheGen的TTFT分解。（b）文本基线和CacheGen的计算开销。（c）基线量化和CacheGen的离线延迟分解。（d）CacheGen、量化基线和未压缩KV缓存的存储成本。使用Mistral-7B绘制。

0.1 – 10 Gbps。每个点是基于 LongChat 数据集上的 20 个带宽跟踪的平均值。我们可以看到，CacheGen 显著优于量化基线和没有适应的 CacheGen。具体来说，在 TTFT 为 0.5 秒的 SLO 下，CacheGen 达到了与量化基线相同的质量，同时 SLO 违规率降低了 60%。在 1 秒的 SLO 下，CacheGen 达到了与量化基线相同的质量，同时将 SLO 违规率从 81% 降低到 8%。CacheGen 具有更低 SLO 违规率的原因是，当带宽下降时，CacheGen 可以动态降低量化级别或回退到从头计算文本的配置，而量化基线和没有适应的 CacheGen 则无法做到这一点。

7.5 开销和微基准测试

解码开销：虽然在大小质量和TTFT质量权衡方面表现更好，但与量化基线相比，CacheGen需要额外的解码（解压缩）步骤。CacheGen通过使用基于GPU的实现加速解码，并将上下文块的解码与上下文块的传输进行流水线处理，从而最小化了解码开销，因此如图14a所示，解码对端到端延迟的影响最小。还需要注意的是，尽管CacheGen的解码是在GPU上执行的（见§6），但与从文本上下文生成KV缓存的基线相比，CacheGen的解码模块所需的计算量是微不足道的。

离线编码和存储开销：与之前仅对每个上下文压缩一次的方法不同，CacheGen将其压缩为多个版本（§5.3）。CacheGen几乎与基线一样快地压缩每个上下文，因为编码延迟非常小（200毫秒），如图14c所示。图14d评估了存储开销。我们可以看到，尽管需要编码和存储多个比特流表示，CacheGen的总存储成本与量化基线相当。

消融研究：为了研究CacheGen的KV编码器中各个组件的影响，图15逐步添加每个想法。

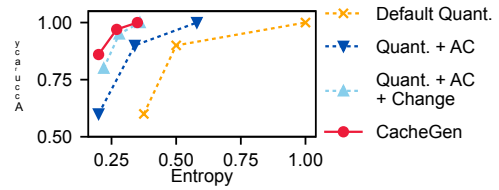


图15：KV编码器背后各个想法的贡献：基于变化的编码、逐层量化和基于通道层分组的AC。

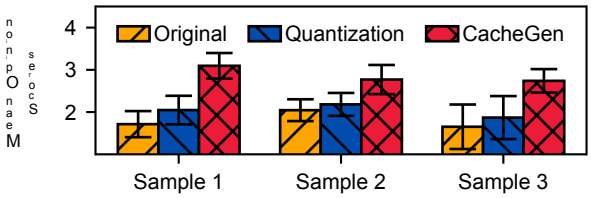


图16：真实用户研究表明，CacheGen在用户体验（QoE）方面显著优于其他基线。

进入均匀量化和默认AC的基线，从使用我们针对每个通道层组合的概率分布的AC开始，然后是基于变化的编码，最后是逐层量化。如图所示，CacheGen的AC和基于变化的编码显著改善了均匀量化。这表明，去除保持KV缓存张量格式的约束，并使用我们的基于变化的编码和AC将其编码为比特流，可以在量化后进一步减少KV缓存的大小。

体验质量：我们进行了一个经过IRB批准的用户研究，以验证CacheGen的有效性。我们从之前评估中使用的LongChat数据集中选择了三个对话历史。对于每个用户，我们首先展示与ChatGPT的对话历史。然后，我们展示相同的响应，但通过不同的管道生成，添加不同的TTFT，并让用户对响应的质量进行评分。通过从Amazon MTurk收集的270个评分[66]，我们展示了CacheGen在QoE方面始终优于其他管道，并且TTFT更短，如图16所示。

CacheGen的评估结果与更多基线可在§B中找到，包括使用更小尺寸的模型来加速TTFT和Gisting，另一种上下文缩减技术。

8 相关工作

更快的 LLM 服务：大多数 LLM 系统研究旨在加速 LLM 训练 [114, 122] 或使服务系统更快。CacheGen 旨在通过关注 TTFT 减少来加速 LLM 服务系统。其他研究探索近似并行生成 [86, 103]、加速边缘设备上的推理 [148]、量化 LLM 权重 [21]、减少 GPU 片上 SRAM 的内存 I/O [47] 和降低自注意力计算复杂性 [116]、更好的调度策略 [17, 111, 139, 149, 157] 以及 GPU 内存利用率 [82]。另一项工作优化了在 GPU 之间传输 KV 缓存的通信延迟，采用智能模型并行策略 [111, 157] 或通过实现新的注意力操作 [91]。该操作在解码阶段将查询向量传输到托管较小 KV 缓存块的 GPU。一个常见的方法是实现更快的推理而不修改 {v*}。

the LLMs is by *caching the KV* of previously used inputs for *one* LLM query [95, 103, 112, 120, 137, 152]. CacheGen works as a module to enable reuse of KV caches *across multiple* LLM queries in these frameworks [17, 35, 58, 82].

Longer LLM contexts: Recent efforts aim at enabling LLMs to process very long contexts [144]. The challenge is to fit the large attention matrices of longer contexts into limited GPU memory. This is enabled by offloading parts of the attention matrices [120], using external knowledge via KNN [141], approximating via retraining self-attention to only attend to top-k keys [19, 32], mapping long inputs to smaller latent spaces [60] and using local windowed, dilated or sparse [31, 50, 150] attention to scale to inputs of ∼1 billion tokens. Longer contexts inflate the KV cache and CacheGen aims to address this by fast remote loading of the KV cache.

Context shortening: Efforts on shortening long contexts relate well to CacheGen. They aim to select the most important text segments and prune the rest. Using similarity between the user query and the relevant documents [35], only keeping tokens that are less attended to by the prompt (*i.e.,* heavy-hitter tokens) [95, 152] or by hybrid policies including keeping nearby tokens or heavy-hitter tokens [54], using query-aware compression with document reordering to reduce loss-in-the-middle [72, 115] have been explored. All these methods need to know the query, else they risk dropping potentially important tokens and they keep the KV cache intact, to fit into limited GPU memory. Some works retrain LLM models to use contexts rewritten by gisting [104] or auto-encoding [55].

CacheGen differs by compressing the KV cache into bitstreams instead of shortening the context. CacheGen's KV compression does not need to know the query/prompt and doesn't risk quality loss from dropping potentially important tokens. It allows for better compression rates by leveraging distributional properties of KV caches and achieves better delay-quality trade-offs than existing context compressors (§7.5). CacheGen also does not need to retrain the LLM.

Tensor compression: CacheGen's KV cache encoding is essentially a tensor compression technique tailored for LLM's. General tensor compression has been intensively studied [109, 154]. In DNN training, tensor compression has been used to compress gradient updates of DNN weights (*e.g.,* [15, 16, 133]). KV caches and gradients have very different properties. DNN training systems often leverage the sparsity of gradients which occurs due to methods like [42, 43, 151]. However the KV cache is not known to be sparse in general.

Retrieval augmented generation(RAG): RAG [35, 67, 68, 88, 113, 117, 134] focuses on retrieving relevant documents to the query via vector based [40, 106, 145] or DNN-based [79, 88, 143, 146] similarity search algorithms and feeding it as context to generate the answer. We envision RAG as a fitting use case for CacheGen. Many LLM inference platforms support feeding KV caches as retrieved context instead of text [39, 136]. Some works have also attempted to define a systematic way to choose which KV cache to reuse[59]. Another approach is to have LLM applications that cache the query's generated answers to reduce repetitive query costs [100, 127]. While caching answers is useful for reuse, CacheGen provides a more generic way to incorporate context reuse and can generate better-quality answers.

9 Discussion and Limitations

Compatibility with other KV-cache compression work: Emerging techniques like smart quantization [62, 78, 97] are *complementary* with CacheGen. After quantization, CacheGen can still apply delta encoding and arithmetic coding, as shown in Figure 10.

Incremental KV cache streaming: Future work includes extending CacheGen to stream KV caches incrementally, akin to Scalable Video Coding (SVC) [61], by initially sending low-quality KV caches and then incrementally improving quality by sending differences.

Context reuse in real-world LLM applications: In §2.2, we explain why contexts are likely reused across requests using anecdotal evidence, but unfortunately, few industry datasets exist to support it. Future work includes finding or creating such datasets.

Evaluation on higher-end GPUs: In §7, we use NVIDIA A40 GPUs to conduct the experiments. We acknowledge that with very high-power GPUs and relatively low bandwidth, CacheGen might not significantly improve over the text context baseline. Furthermore, due to GPU memory limitations, we have not evaluated our ideas on extra-large models such as OPT-175B. Evaluating CacheGen on more powerful GPUs and larger LLMs is left for future work.

Other system designs: §5 covers CacheGen's encoder and streamer design. Other aspects such as which storage device(s) to store KV cache, caching policies, and locating KV cache quickly are discussed in concurrent works [52, 74, 147]. We leave combining CacheGen with these works to future work.

Other limitations: Task-wise, we did not extensively evaluate CacheGen's performance on "free-text generation" tasks such as story generation because the quality metrics are less well-defined than the tasks in our evaluation. Network-wise, our network model does not include conditions with extremely high bandwidths. Additionally, not all LLM applications can cache KV features. Search-based apps, like Google and Bing, use real-time search results as context, and their volatile contexts will unlikely be reused unless for very popular search results. We expect future work to address these issues.

10 Conclusion

We present CacheGen, a context-loading module to minimize overall delays in fetching and processing contexts for LLMs. CacheGen reduces the bandwidth needed to transmit long contexts' KV cache through an encoder tailored to compress KV cache into compact bitstreams. Experiments across three models of various capacities and four datasets with various context lengths show that CacheGen reduces overall delays while maintaining high task performance.

Acknowledgement

We thank all the anonymous reviewers and our shepherd, Chen Qian, for their insightful feedback and suggestions. The project is funded by NSF CNS-2146496, CNS-2131826, CNS-2313190, CNS-1901466, CNS-1956180, CCF-2119184, UChicago CERES Center, and Marian and Stuart Rice Research Award. The project is also supported by Chameleon Projects [80].

LLMs 通过缓存先前用于一个 LLM 查询的 KV 来实现 [95, 103, 112, 120, 137, 152]。CacheGen 作为一个模块工作，使得在这些框架中可以跨多个 LLM 查询重用 KV 缓存 [17, 35, 58, 82]。

更长的LLM上下文：最近的努力旨在使LLM能够处理非常长的上下文[144]。挑战在于将更长上下文的大型注意力矩阵适配到有限的GPU内存中。这可以通过卸载注意力矩阵的部分[120]、通过KNN使用外部知识[141]、通过重新训练自注意力仅关注前k个键进行近似[19, 32]、将长输入映射到更小的潜在空间[60]以及使用局部窗口、扩张或稀疏[31, 50, 150]注意力来扩展到~10亿个标记的输入来实现。更长的上下文会膨胀KV缓存，而CacheGen旨在通过快速远程加载KV缓存来解决这个问题。

上下文缩短：缩短长上下文的努力与CacheGen密切相关。它们旨在选择最重要的文本片段并修剪其余部分。通过用户查询与相关文档之间的相似性[35]，仅保留提示中关注较少的标记（即重击标记）[95, 152]或通过包括保留附近标记或重击标记的混合策略[54]，使用查询感知压缩和文档重排序以减少中间损失[72, 115]已被探索。所有这些方法都需要知道查询，否则它们可能会丢失潜在的重要标记，并保持KV缓存不变，以适应有限的GPU内存。一些工作重新训练LLM模型，以使用通过摘要[104]或自编码[55]重写的上下文。

CacheGen 的不同之处在于将 KV 缓存压缩为比特流，而不是缩短上下文。CacheGen 的 KV 压缩不需要知道查询/提示，并且不会因丢弃潜在重要的标记而导致质量损失。它通过利用 KV 缓存的分布特性实现更好的压缩率，并比现有的上下文压缩器实现更好的延迟-质量权衡（§7.5）。CacheGen 也不需要重新训练 LLM。

张量压缩：CacheGen 的 KV 缓存编码本质上是一种针对 LLM 的张量压缩技术。一般的张量压缩已经得到了深入研究 [109, 154]。在 DNN 训练中，张量压缩被用于压缩 DNN 权重的梯度更新（例如，[15, 16, 133]）。KV 缓存和梯度具有非常不同的特性。DNN 训练系统通常利用由于 [42, 43, 151] 等方法导致的梯度稀疏性。然而，KV 缓存通常并不被认为是稀疏的。

检索增强生成（RAG）：RAG [35, 67, 68, 88, 113, 117, 134] 专注于通过基于向量的 [40, 106, 145] 或基于 DNN 的 [79, 88, 143, 146] 相似性搜索算法检索与查询相关的文档，并将其作为上下文提供以生成答案。我们设想 RAG 是 CacheGen 的一个合适用例。许多 LLM 推理平台支持将 KV 缓存作为检索的上下文而不是文本 [39, 136]。一些工作也尝试定义一种系统的方法来选择重用哪个 KV 缓存 [59]。另一种方法是让 LLM 应用程序缓存查询生成的答案，以减少重复查询的成本 [100, 127]。虽然缓存答案对重用很有用，但 CacheGen 提供了一种更通用的方法来结合上下文重用，并可以生成更高质量的答案。

9 讨论与局限性

与其他KV-cache压缩工作的兼容性：新兴技术如智能量化[62, 78, 97]与CacheGen是互补的。在量化之后，CacheGen仍然可以应用增量编码和算术编码，如图10所示。

增量KV缓存流：未来的工作包括扩展CacheGen以增量方式流式传输KV缓存，类似于可扩展视频编码（SVC）[61]，通过最初发送低质量的KV缓存，然后通过发送差异逐步提高质量。

在现实世界的 LLM 应用中上下文重用：在 §2.2 中，我们解释了为什么上下文在请求之间可能会被重用，使用的是轶事证据，但不幸的是，几乎没有行业数据集来支持这一点。未来的工作包括寻找或创建这样的数据集。

在高端GPU上的评估：在§7中，我们使用NVIDIA A40 GPU进行实验。我们承认，在非常高功率的GPU和相对较低带宽的情况下，CacheGen可能不会显著改善文本上下文基线。此外，由于GPU内存限制，我们尚未在如OPT-175B等超大模型上评估我们的想法。在更强大的GPU和更大规模的LLM上评估CacheGen将留待未来的工作。

其他系统设计：§5 涉及 CacheGen 的编码器和流媒体设计。其他方面，如存储 KV 缓存的存储设备、缓存策略以及快速定位 KV 缓存等，在并行工作 [52, 74, 147] 中进行了讨论。我们将 CacheGen 与这些工作的结合留待未来的工作。

其他限制：在任务方面，我们没有对CacheGen在“自由文本生成”任务（如故事生成）上的性能进行广泛评估，因为质量指标不如我们评估中的任务明确。在网络方面，我们的网络模型不包括带宽极高的条件。此外，并非所有LLM应用都可以缓存KV特征。基于搜索的应用，如Google和Bing，使用实时搜索结果作为上下文，它们的波动上下文不太可能被重用，除非是非常受欢迎的搜索结果。我们期待未来的工作能解决这些问题。

10 结论

我们介绍了CacheGen，一个上下文加载模块，用于最小化获取和处理LLM上下文的整体延迟。CacheGen通过一个专门设计的编码器来压缩KV缓存为紧凑的比特流，从而减少传输长上下文的KV缓存所需的带宽。在三个不同容量的模型和四个具有不同上下文长度的数据集上的实验表明，CacheGen在保持高任务性能的同时减少了整体延迟。

致谢

我们感谢所有匿名评审和我们的指导者陈倩，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-1956180、CCF-2119184、芝加哥大学CERES中心以及Marian和Stuart Rice研究奖资助。该项目还得到了Chameleon Projects [80]的支持。

References

[1] 2021. How latency affects user engagement. https://pusher.com/blog/how-latency-affects-user-engagement/. (2021). (Accessed on 09/21/2023).

[2] 2023. Best Practices for Deploying Large Language Models (LLMs) in Production. https://medium.com/@_aigeek/best-practices-for-deploying-large-language-models-llms-in-production-fdc5bf240d6a. (2023). (Accessed on 09/21/2023).

[3] 2023. Building RAG-based LLM Applications for Production. https://www.anyscale.com/blog/a-comprehensive-guide-for-building-rag-based-llm-applications-part-1. (2023). Accessed: 2024-01-25.

[4] 2024. Amazon Bedrock Pricing. https://aws.amazon.com/bedrock/pricing/. (2024). Accessed: 2024-01-25.

[5] 2024. Anyscale Pricing. https://docs.endpoints.anyscale.com/pricing. (2024). Accessed: 2024-01-25.

[6] 2024. AWS Pricing examples. https://aws.amazon.com/s3/pricing/. (2024). Accessed: 2024-01-25.

[7] 2024. ChatGPT. https://chat.openai.com/gpts. (2024). Accessed: 2024-01-25.

[8] 2024. pathwaycom/llmapp. https://github.com/pathwaycom/llm-app. (2024). Accessed: 2024-01-25.

[9] 2024. Perplexity. https://www.perplexity.ai/. (2024). Accessed: 2024-01-25.

[10] 2024. RAG-Transform. https://huggingface.co/transformers/v4.3.0/model_doc/rag.html. (2024). Accessed: 2024-01-25.

[11] 2024. Replicate Pricing. https://replicate.com/pricing. (2024). Accessed: 2024-01-25.

[12] 2024. together.pricing. https://www.together.ai/pricing. (2024). Accessed: 2024-01-25.

[13] Daniel Adiwardana, Minh-Thang Luong, David R. So, Jamie Hall, Noah Fiedel, Romal Thoppilan, Zi Yang, Apoorv Kulshreshtha, Gaurav Nemade, Yifeng Lu, and Quoc V. Le. 2020. Towards a Human-like Open-Domain Chatbot. (2020). arXiv:cs.CL/2001.09977

[14] Megha Agarwal, Asfandyar Qureshi, Nikhil Sardana, Linden Li, Julian Quevedo, and Daya Khudia. 2023. LLM Inference Performance Engineering: Best Practices. (Oct. 2023). https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices Accessed: 2024-06-01.

[15] Saurabh Agarwal, Hongyi Wang, Kangwook Lee, Shivaram Venkataraman, and Dimitris Papailiopoulos. 2020. Accordion: Adaptive gradient communication via critical learning regime identification. *arXiv preprint arXiv:2010.16248* (2020).

[16] Saurabh Agarwal, Hongyi Wang, Shivaram Venkataraman, and Dimitris Papailiopoulos. 2022. On the Utility of Gradient Compression in Distributed Training Systems. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 652–672. https://proceedings.mlsys.org/paper_files/paper/2022/file/773862fcc2e29f650d68960ba5bd1101-Paper.pdf

[17] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. 2023. SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills. (2023). arXiv:cs.LG/2308.16369

[18] Toufique Ahmed and Premkumar Devanbu. 2023. Few-shot training LLMs for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 177, 5 pages. https://doi.org/10.1145/3551349.3559555

[19] Joshua Ainslie, Santiago Ontanon, Chris Alberti, Vaclav Cvicek, Zachary Fisher, Philip Pham, Anirudh Ravula, Sumit Sanghai, Qifan Wang, and Li Yang. 2020. ETC: Encoding Long and Structured Inputs in Transformers. (2020). arXiv:cs.LG/2004.08483

[20] Amazon.com Inc. 2023. *2023 Annual Report.* Annual Report. Amazon.com Inc. https://s2.q4cdn.com/299287126/files/doc_financials/2024/ar/Amazon-com-Inc-2023-Annual-Report.pdf

[21] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. 2022. DeepSpeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[22] Zharovskikh Anastasiya. 2023. Applications of Large Language Models - InData Labs. https://indatalabs.com/blog/large-language-model-apps. (June 2023). (Accessed on 09/21/2023).

[23] Anonymous. 2024. ChunkAttention: Efficient Attention on KV Cache with Chunking Sharing and Batching. (2024). https://openreview.net/forum?id=9k27IITeAZ

[24] Anthropic. 2023. Anthropic \ Introducing 100K Context Windows. https://www.anthropic.com/index/100k-context-windows. (May 2023). (Accessed on 09/21/2023).

[25] Anyscale Team. 2023. Comparing LLM Performance: Introducing the Open Source Leaderboard for LLM APIs. (Dec. 2023). https://www.anyscale.com/blog/comparing-llm-performance-introducing-the-open-source-leaderboard-for-llm Accessed: 2024-06-01.

[26] AuthorName. Year. Can ChatGPT understand context and keep track of conversation history. https://www.quora.com/Can-ChatGPT-understand-context-and-keep-track-of-conversation-history. (Year). Quora question.

[27] AutoGPT. 2023. Significant-Gravitas/Auto-GPT: An experimental open-source attempt to make GPT-4 fully autonomous. https://github.com/Significant-Gravitas/Auto-GPT. (September 2023). (Accessed on 09/21/2023).

[28] Leif Azzopardi, Mark Girolami, and Keith van Risjbergen. 2003. Investigating the Relationship between Language Model Perplexity and IR Precision-Recall Measures. In *Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Informaion Retrieval (SIGIR '03)*. Association for Computing Machinery, New York, NY, USA, 369–370. https://doi.org/10.1145/860435.860505

[29] 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. 2023. LongBench: A Bilingual, Multitask Benchmark for Long Context Understanding. *arXiv preprint arXiv:2308.14508* (2023).

[30] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. 2023. CodePlan: Repository-level Coding using LLMs and Planning. (2023). arXiv:cs.SE/2309.12499

[31] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. (2020). arXiv:cs.CL/2004.05150

[32] Amanda Bertsch, Uri Alon, Graham Neubig, and Matthew R Gormley. 2023. Unlimiformer: Long-range transformers with unlimited length input. *arXiv preprint arXiv:2305.01625* (2023).

[33] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. *Site Reliability Engineering: How Google Runs Production Systems* (1st ed.). O'Reilly Media, Inc.

[34] Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. 2019. PIQA: Reasoning about Physical Commonsense in Natural Language. (2019). arXiv:cs.CL/1911.11641

[35] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. 2022. Improving language models by retrieving from trillions of tokens. (2022). arXiv:cs.CL/2112.04426

[36] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. 2022. Improving language models by retrieving from trillions of tokens. (2022). arXiv:cs.CL/2112.04426 https://arxiv.org/abs/2112.04426

[37] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. (2020). arXiv:cs.CL/2005.14165

[38] CellStrat. 2023. 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. (April 2023). (Accessed on 09/21/2023).

[39] Harrison Chase. 2022. LangChain. (Oct. 2022). https://github.com/langchain-ai/langchain

[40] Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. 2017. Reading Wikipedia to Answer Open-Domain Questions. (2017). arXiv:cs.CL/1704.00051

[41] Stanley F Chen, Douglas Beeferman, and Roni Rosenfeld. 2008. Evaluation Metrics For Language Models. (1 2008). https://doi.org/10.1184/R1/6605324.v1

[42] Tianlong Chen, Zhenyu Zhang, Ajay Jaiswal, Shiwei Liu, and Zhangyang Wang. 2023. Sparse MoE as the New Dropout: Scaling Dense and Self-Slimmable Transformers. (2023). arXiv:cs.LG/2303.01610

[43] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating Long Sequences with Sparse Transformers. (2019). arXiv:cs.LG/1904.10509

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

[45] Zihang Dai*, Zhilin Yang*, Yiming Yang, William W. Cohen, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. 2019. Transformer-XL: Language Modeling with Longer-Term Dependency. (2019). https://openreview.net/forum?id=HJePno0cYm

[46] Daivi. 21. 7 Top Large Language Model Use Cases And Applications. https://www.projectpro.io/article/large-language-model-use-cases-and-applications/887. (March 21). (Accessed on 09/21/2023).

参考文献

[1] 2021. 延迟如何影响用户参与度。 https://pusher.com/blog/how-latency-affects-user-engagement/. (2021). (访问于 2023年09月21日)。 [2] 2023. 在生产中部署大型语言模型（LLMs）的最佳实践。 https://medium.com/@_aigeek/best-practices-for-deploying-large-language-models-llms-in-production-fdc5bf240d6a. (2023). (访问于 2023年09月21日)。 [3] 2023. 为生产构建基于RAG的LLM应用程序。 https://www.anyscale.com/blog/a-comprehensive-guide-for-building-rag-based-llm-applications-part-1. (2023). 访问时间：2024年01月25日。 [4] 2024. 亚马逊Bedrock定价。 https://aws.amazon.com/bedrock/pricing/. (2024). 访问时间：2024年01月25日。 [5] 2024. Anyscale定价。 https://docs.endpoints.anyscale.com/pricing. (2024). 访问时间：2024年01月25日。 [6] 2024. AWS定价示例。 https://aws.amazon.com/s3/pricing/. (2024). 访问时间：2024年01月25日。 [7] 2024. ChatGPT。 https://chat.openai.com/gpts. (2024). 访问时间：2024年01月25日。 [8] 2024. pathwaycom/llmapp。 https://github.com/pathwaycom/llm-app. (2024). 访问时间：2024年01月25日。 [9] 2024. Perplexity。 https://www.perplexity.ai/. (2024). 访问时间：2024年01月25日。 [10] 2024. RAG-Transform。 https://huggingface.co/transformers/v4.3.0/model_doc/rag.html. (2024). 访问时间：2024年01月25日。 [11] 2024. Replicate定价。 https://replicate.com/pricing. (2024). 访问时间：2024年01月25日。 [12] 2024. together.pricing。 https://www.together.ai/pricing. (2024). 访问时间：2024年01月25日。 [13] Daniel Adiwardana, Minh-Thang Luong, David R. So, Jamie Hall, Noah Fiedel, Romal Thoppilan, Zi Yang, Apoorv Kulshreshtha, Gaurav Nemade, Yifeng Lu, 和 Quoc V. Le. 2020. 朝着类人开放域聊天机器人迈进。 (2020). arXiv:cs.CL/2001.09977 [14] Megha Agarwal, Asfandyar Qureshi, Nikhil Sardana, Linden Li, Julian Quevedo, 和 Daya Khudia. 2023. LLM推理性能工程：最佳实践。 (2023年10月). https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices 访问时间：2024年06月01日。 [15] Saurabh Agarwal, Hongyi Wang, Kangwook Lee, Shivaram Venkataraman, 和 Dimitris Papailiopoulos. 2020. Accordion：通过关键学习机制识别进行自适应梯度通信。 arXiv预印本 arXiv:2010.16248 (2020)。 [16] Saurabh Agarwal, Hongyi Wang, Shivaram Venkataraman, 和 Dimitris Papailiopoulos. 2022. 在分布式训练系统中梯度压缩的效用。 在机器学习与系统会议论文集中，D. Marculescu, Y. Chi, 和 C. Wu (编)，第4卷。652–672。 https://proceedings.mlsys.org/paper_files/paper/2022/file/773862fcc2e29f650d68960ba5bd1101-Paper.pdf [17] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, 和 Ramachandran Ramjee. 2023. SARATHI：通过搭载解码与分块预填充实现高效LLM推理。 (2023). arXiv:cs.LG/2308.16369 [18] Toufique Ahmed 和 Premkumar Devanbu. 2023. 为项目特定代码摘要进行少量训练的LLMs。 在第37届IEEE/ACM国际自动化软件工程会议（ASE '22）论文集中。计算机协会，纽约，NY，美国，第177篇，5页。 https://doi.org/10.1145/3551349.3559555 [19] Joshua Ainslie, Santiago Ontanon, Chris Alberti, Vaclav Cvicek, Zachary Fisher, Philip Pham, Anirudh Ravula, Sumit Sanghai, Qifan Wang, 和 Li Yang. 2020. ETC：在变换器中编码长且结构化的输入。 (2020). arXiv:cs.LG/2004.08483 [20] 亚马逊公司。 2023. 2023年年度报告。 年度报告。 亚马逊公司。 https://s2.q4cdn.com/299287126/files/doc_financials/2024/ar/Amazon-com-Inc-2023-Annual-Report.pdf [21] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley 等。 2022. DeepSpeed推理：在前所未有的规模上实现变换器模型的高效推理。 在SC22：高性能计算、网络、存储和分析国际会议上。 IEEE，1–15。 [22] Zharovskikh Anastasiya. 2023. 大型语言模型的应用 - In Data Labs。 https://indatalabs.com/blog/large-language-model-apps. (2023年6月)。 (访问于 2023年09月21日)。 [23] 匿名。 2024. ChunkAttention：通过分块共享和批处理实现KV缓存上的高效注意力。 (2024). https://openreview.net/forum?id=9k27IITeAZ [24] Anthropic. 2023. Anthropic \ 引入100K上下文窗口。 https://www.anthropic.com/index/100k-context-windows. (2023年5月)。 (访问于 2023年09月21日)。 [25] Anyscale团队。 2023. 比较LLM性能：推出LLM API的开源排行榜。 (2023年12月). https://www.anyscale.com/blog/comparing-llm-performance-introducing-the-open-source-leaderboard-for-llm 访问时间：2024年06月01日。 [26] AuthorName. 年份。 ChatGPT能否理解上下文并跟踪对话历史。 https://www.quora.com/Can-ChatGPT-understand-context-and-keep-track-of-conversation-history. (年份). Quora问题。

[27] AutoGPT. 2023. Significant-Gravitas/Auto-GPT：一个使GPT-4完全自主的实验性开源尝试。https://github.com/Significant-Gravitas/Auto-GPT。（2023年9月）。（访问于2023年9月21日）。 [28] Leif Azzopardi, Mark Girolami, 和 Keith van Risjbergen. 2003. 研究语言模型困惑度与信息检索精确度-召回率度量之间的关系。在第26届国际ACM SIGIR信息检索研究与开发年会论文集（SIGIR '03）。计算机协会，纽约，NY，美国，369–370。https://doi.org/10.1145/860435.860505 [29] 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. 2023. 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）。 [30] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, 和 Shashank Shet. 2023. CodePlan：使用LLMs和规划进行库级编码。（2023）。arXiv:cs.SE/2309.12499 [31] Iz Beltagy, Matthew E. Peters, 和 Arman Cohan. 2020. Longformer：长文档变换器。（2020）。arXiv:cs.CL/2004.05150 [32] Amanda Bertsch, Uri Alon, Graham Neubig, 和 Matthew R Gormley. 2023. Unlimiformer：具有无限长度输入的长范围变换器。arXiv预印本 arXiv:2305.01625（2023）。 [33] Betsy Beyer, Chris Jones, Jennifer Petoff, 和 Niall Richard Murphy. 2016. 站点可靠性工程：谷歌如何运行生产系统（第1版）。O'Reilly Media, Inc. [34] Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, 和 Yejin Choi. 2019. PIQA：在自然语言中推理物理常识。（2019）。arXiv:cs.CL/1911.11641 [35] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, 和 Laurent Sifre. 2022. 通过从万亿个标记中检索来改进语言模型。（2022）。arXiv:cs.CL/2112.04426 [36] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, 和 Laurent Sifre. 2022. 通过从万亿个标记中检索来改进语言模型。（2022）。arXiv:cs.CL/2112.04426 https://arxiv.org/abs/2112.04426 [37] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, 和 Dario Amodei. 2020. 语言模型是少量学习者。（2020）。arXiv:cs.CL/2005.14165 [38] CellStrat. 2023. 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。（2023年4月）。（访问于2023年9月21日）。 [39] Harrison Chase. 2022. LangChain。（2022年10月）。https://github.com/langchain-ai/langchain [40] Danqi Chen, Adam Fisch, Jason Weston, 和 Antoine Bordes. 2017. 阅读维基百科以回答开放领域问题。（2017）。arXiv:cs.CL/1704.00051 [41] Stanley F Chen, Douglas Beeferman, 和 Roni Rosenfeld. 2008. 语言模型的评估指标。（2008年1月）。https://doi.org/10.1184/R1/6605324.v1 [42] Tianlong Chen, Zhenyu Zhang, Ajay Jaiswal, Shiwei Liu, 和 Zhangyang Wang. 2023. 稀疏MoE作为新的丢弃：扩展密集和自适应变换器。（2023）。arXiv:cs.LG/2303.01610 [43] Rewon Child, Scott Gray, Alec Radford, 和 Ilya Sutskever. 2019. 使用稀疏变换器生成长序列。（2019）。arXiv:cs.LG/1904.10509 [44] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, 等. 2022. Palm：通过路径扩展语言建模。arXiv预印本 arXiv:2204.02311（2022）。 [45] Zihang Dai*, Zhilin Yang*, Yiming Yang, William W. Cohen, Jaime Carbonell, Quoc V. Le, 和 Ruslan Salakhutdinov. 2019. Transformer-XL：具有长期依赖的语言建模。（2019）。https://openreview.net/forum?id= HJePno0cYm [46] Daivi. 21. 7个大型语言模型的顶级用例和应用。https://www.projectpro.io/article/large-language-model-use-cases-and-applications/887。（3月21日）。（访问于2023年9月21日）。

[47] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. (2022). arXiv:cs.LG/2205.14135

[48] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339* (2022).

[49] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[50] Jiayu Ding, Shuming Ma, Li Dong, Xingxing Zhang, Shaohan Huang, Wenhui Wang, Nanning Zheng, and Furu Wei. 2023. LongNet: Scaling Transformers to 1,000,000,000 Tokens. (2023). arXiv:cs.CL/2307.02486

[51] Yiran Ding, Li Lyna Zhang, Chengruidong Zhang, Yuanyuan Xu, Ning Shang, Jiahang Xu, Fan Yang, and Mao Yang. 2024. LongRoPE: Extending LLM Context Window Beyond 2 Million Tokens. *arXiv preprint arXiv:2402.13753* (2024).

[52] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. AttentionStore: Cost-effective Attention Reuse across Multi-turn Conversations in Large Language Model Serving. *arXiv preprint arXiv:2403.19708* (2024).

[53] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Qianyu Guo, Meng Wang, and Haofen Wang. 2024. Retrieval-Augmented Generation for Large Language Models: A Survey. (2024). arXiv:cs.CL/2312.10997

[54] Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. 2023. Model Tells You What to Discard: Adaptive KV Cache Compression for LLMs. (2023). arXiv:cs.CL/2310.01801

[55] Tao Ge, Jing Hu, Xun Wang, Si-Qing Chen, and Furu Wei. 2023. In-context Autoencoder for Context Compression in a Large Language Model. *arXiv preprint arXiv:2307.06945* (2023).

[56] Tao Ge, Hu Jing, Lei Wang, Xun Wang, Si-Qing Chen, and Furu Wei. 2024. In-context Autoencoder for Context Compression in a Large Language Model. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=uREj4ZuGJE

[57] GGML. [n. d.]. GGML - AI at the edge. https://ggml.ai/. ([n. d.]).

[58] In Gim, Guojun Chen, Seung seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2023. Prompt Cache: Modular Attention Reuse for Low-Latency Inference. (2023). arXiv:cs.CL/2311.04934

[59] In Gim, Guojun Chen, Seung seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2023. Prompt Cache: Modular Attention Reuse for Low-Latency Inference. (2023). arXiv:cs.CL/2311.04934

[60] Curtis Hawthorne, Andrew Jaegle, Cătălina Cangea, Sebastian Borgeaud, Charlie Nash, Mateusz Malinowski, Sander Dieleman, Oriol Vinyals, Matthew Botvinick, Ian Simon, Hannah Sheahan, Neil Zeghidour, Jean-Baptiste Alayrac, Joao Carreira, and Jesse Engel. 2022. General-purpose, long-context autoregressive modeling with Perceiver AR. In *Proceedings of the 39th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (Eds.), Vol. 162. PMLR, 8535–8558. https://proceedings.mlr.press/v162/hawthorne22a.html

[61] Hermann Hellwagner, Ingo Kofler, Michael Eberhard, Robert Kuschnig, Michael Ransburg, and Michael Sablatschan. 2011. *Scalable Video Coding: Techniques and Applications for Adaptive Streaming*. 1–23. https://doi.org/10.4018/978-1-61692-831-5

[62] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. 2024. KVQuant: Towards 10 Million Context Length LLM Inference with KV Cache Quantization. *arXiv preprint arXiv:2401.18079* (2024).

[63] Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, and Boris Ginsburg. 2024. RULER: What's the Real Context Size of Your Long-Context Language Models? *arXiv preprint arXiv:2404.06654* (2024).

[64] Huggingface. [n. d.]. Huggingface Transformers. https://huggingface.co/docs/transformers/index. ([n. d.]).

[65] Huggingface. [n. d.]. Perplexity in fixed length models. https://huggingface.co/docs/transformers/perplexity. ([n. d.]).

[66] Amazon Inc. [n. d.]. Amazon Mechanical Turk. https://www.mturk.com/. ([n. d.]).

[67] Gautier Izacard and Edouard Grave. 2021. Leveraging Passage Retrieval with Generative Models for Open Domain Question Answering. (2021). arXiv:cs.CL/2007.01282

[68] Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. 2022. Few-shot learning with retrieval augmented language models. *arXiv preprint arXiv:2208.03299* (2022).

[69] Naman Jain, Tianjun Zhang, Wei-Lin Chiang, Joseph E. Gonzalez, Koushik Sen, and Ion Stoica. 2023. LLM-Assisted Code Cleaning For Training Accurate Code Generators. (2023). arXiv:cs.LG/2311.14904

[70] Paras Jain, Sam Kumar, Sarah Wooders, Shishir G. Patil, Joseph E. Gonzalez, and Ion Stoica. 2023. Skyplane: Optimizing Transfer Cost and Throughput Using Cloud-Aware Overlays. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1375–1389. https://www.usenix.org/conference/nsdi23/presentation/jain

[71] Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023. LLMLingua: Compressing Prompts for Accelerated Inference of Large Language Models. (2023). arXiv:cs.CL/2310.05736

[72] Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023. LongLLMLingua: Accelerating and Enhancing LLMs in Long Context Scenarios via Prompt Compression. (2023). arXiv:cs.CL/2310.06839

[73] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? (2023). arXiv:cs.CL/2310.06770

[74] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. 2024. RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation. *arXiv preprint arXiv:2404.12457* (2024).

[75] Mandar Joshi, Eunsol Choi, Daniel S. Weld, and Luke Zettlemoyer. 2017. TriviaQA: A Large Scale Distantly Supervised Challenge Dataset for Reading Comprehension. (2017). arXiv:cs.CL/1705.03551

[76] jwatte. 2023. How does ChatGPT store history of chat. https://community.openai.com/t/how-does-chatgpt-store-history-of-chat/319608/2. (Aug 2023). OpenAI Community Forum.

[77] Waleed Kadous, Kyle Huang, Wendi Ding, Liguang Xie, Avnish Narayan, and Ricky Xu. 2023. Reproducible Performance Metrics for LLM Inference. (Nov. 2023). https://www.anyscale.com/blog/reproducible-performance-metrics-for-llm-inference Accessed: 2024-06-01.

[78] Hao Kang, Qingru Zhang, Souvik Kundu, Geonhwa Jeong, Zaoxing Liu, Tushar Krishna, and Tuo Zhao. 2024. Gear: An efficient kv cache compression recipefor near-lossless generative inference of llm. *arXiv preprint arXiv:2403.05527* (2024).

[79] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. (2020). arXiv:cs.CL/2004.04906

[80] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbah, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 219–233. https://www.usenix.org/conference/atc20/presentation/keahey

[81] Tomáš Kočiský, Jonathan Schwarz, Phil Blunsom, Chris Dyer, Karl Moritz Hermann, Gábor Melis, and Edward Grefenstette. 2017. The NarrativeQA Reading Comprehension Challenge. (2017). arXiv:cs.CL/1712.07040

[82] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[83] LangChain. 2024. langchain-ai/langchain:Building applications with LLMs through composability. https://github.com/langchain-ai/langchain. (Feburary 2024). (Accessed on 09/21/2023).

[84] LangChain. 2024. Store and reference chat history | Langchain. https://python.langchain.com/docs/use_cases/question_answering/how_to/chat_vector_db. (Feburary 2024). (Accessed on 09/21/2023).

[85] Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, Daniel Haziza, Luca Wehrstedt, Jeremy Reizenstein, and Grigory Sizov. 2022. xFormers: A modular and hackable Transformer modelling library. https://github.com/facebookresearch/xformers. (2022).

[86] Yaniv Leviathan, Matan Kalman, and Y. Matias. 2022. Fast Inference from Transformers via Speculative Decoding. In *International Conference on Machine Learning*. https://api.semanticscholar.org/CorpusID:254096365

[87] Zijian Lew, Joseph B Walther, Augustine Pang, and Wonsun Shin. 2018. Interactivity in Online Chat: Conversational Contingency and Response Latency in Computer-mediated Communication. *Journal of Computer-Mediated Communication* 23, 4 (06 2018), 201–221. https://doi.org/10.1093/jcmc/zmy009 arXiv:https://academic.oup.com/jcmc/article-pdf/23/4/201/25113924/zmy009.pdf

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

[89] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2021. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. (2021). arXiv:cs.CL/2005.11401

[47] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, 和 Christopher Ré. 2022. FlashAttention: 快速且内存高效的精确注意力与 IO 感知. (2022). arXiv:cs.LG/2205.14135 [48] Tim Dettmers, Mike Lewis, Younes Belkada, 和 Luke Zettlemoyer. 2022. Llm. int8 (): 大规模变换器的 8 位矩阵乘法. arXiv 预印本 arXiv:2208.07339 (2022). [49] Jacob Devlin, Ming-Wei Chang, Kenton Lee, 和 Kristina Toutanova. 2018. Bert: 深度双向变换器的预训练用于语言理解. arXiv 预印本 arXiv:1810.04805 (2018). [50] Jiayu Ding, Shuming Ma, Li Dong, Xingxing Zhang, Shaohan Huang, Wenhui Wang, Nanning Zheng, 和 Furu Wei. 2023. LongNet: 将变换器扩展到 1,000,000,000 个标记. (2023). arXiv:cs.CL/2307.02486 [51] Yiran Ding, Li Lyna Zhang, Chengruidong Zhang, Yuanyuan Xu, Ning Shang, Jiahang Xu, Fan Yang, 和 Mao Yang. 2024. LongRoPE: 将 LLM 上下文窗口扩展到超过 200 万个标记. arXiv 预印本 arXiv:2402.13753 (2024). [52] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, 和 Pengfei Zuo. 2024. AttentionStore: 在大型语言模型服务中跨多轮对话的成本有效注意力重用. arXiv 预印本 arXiv:2403.19708 (2024). [53] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Qianyu Guo, Meng Wang, 和 Haofen Wang. 2024. 针对大型语言模型的检索增强生成: 一项调查. (2024). arXiv:cs.CL/2312.10997 [54] Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, 和 Jianfeng Gao. 2023. 模型告诉你该丢弃什么: LLM 的自适应 KV 缓存压缩. (2023). arXiv:cs.CL/2310.01801 [55] Tao Ge, Jing Hu, Xun Wang, Si-Qing Chen, 和 Furu Wei. 2023. 大型语言模型中的上下文压缩的上下文自编码器. arXiv 预印本 arXiv:2307.06945 (2023). [56] Tao Ge, Hu Jing, Lei Wang, Xun Wang, Si-Qing Chen, 和 Furu Wei. 2024. 大型语言模型中的上下文压缩的上下文自编码器. 在第十二届国际学习表征会议上. https: //openreview.net/forum?id=uREj4ZuGJE [57] GGML. [n. d.]. GGML - 边缘 AI. https://ggml.ai/. ([n. d.]). [58] In Gim, Guojun Chen, Seung seob Lee, Nikhil Sarda, Anurag Khandelwal, 和 Lin Zhong. 2023. 提示缓存: 低延迟推理的模块化注意力重用. (2023). arXiv:cs.CL/2311.04934 [59] In Gim, Guojun Chen, Seung seob Lee, Nikhil Sarda, Anurag Khandelwal, 和 Lin Zhong. 2023. 提示缓存: 低延迟推理的模块化注意力重用. (2023). arXiv:cs.CL/2311.04934 [60] Curtis Hawthorne, Andrew Jaegle, Cătălina Cangea, Sebastian Borgeaud, Charlie Nash, Mateusz Malinowski, Sander Dieleman, Oriol Vinyals, Matthew Botvinick, Ian Simon, Hannah Sheahan, Neil Zeghidour, Jean-Baptiste Alayrac, Joao Car- reira, 和 Jesse Engel. 2022. 通用的长上下文自回归建模与 Perceiver AR. 在第 39 届国际机器学习会议论文集中 (机器学习研究论文集), Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, 和 Sivan Sabato (编辑), 第 162 卷. PMLR, 8535–8558. https://proceedings.mlr.press/v162/hawthorne22a.html [61] Hermann Hellwagner, Ingo Kofler, Michael Eberhard, Robert Kuschnig, Michael Ransburg, 和 Michael Sablatschan. 2011. 可扩展视频编码: 自适应流媒体的技术与应用. 1–23. https://doi.org/10.4018/978-1- 61692-831-5 [62] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun Sophia Shao, Kurt Keutzer, 和 Amir Gholami. 2024. 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). [63] Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, 和 Boris Ginsburg. 2024. RULER: 你的长上下文语言模型的真实上下文大小是多少? arXiv 预印本 arXiv:2404.06654 (2024). [64] Huggingface. [n. d.]. Huggingface Transformers. https://huggingface.co/docs/ transformers/index. ([n. d.]). [65] Huggingface. [n. d.]. 固定长度模型中的困惑度. https://huggingface.co /docs/transformers/perplexity. ([n. d.]). [66] Amazon Inc. [n. d.]. 亚马逊机械土耳其. https://www.mturk.com/. ([n. d.]). [67] Gautier Izacard 和 Edouard Grave. 2021. 利用生成模型进行开放域问答的段落检索. (2021). arXiv:cs.CL/2007.01282 [68] Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, 和 Edouard Grave. 2022. 通过检索增强语言模型进行少量学习. arXiv 预印本 arXiv:2208.03299 (2022). [69] Naman Jain, Tianjun Zhang, Wei-Lin Chiang, Joseph E. Gonzalez, Koushik Sen, 和 Ion Stoica. 2023. LLM 辅助代码清理以训练准确的代码生成器. (2023). arXiv:cs.LG/2311.14904

[70] Paras Jain, Sam Kumar, Sarah Wooders, Shishir G. Patil, Joseph E. Gonzalez, 和 Ion Stoica. 2023. Skyplane: 使用云感知覆盖层优化传输成本和吞吐量. 在第20届USENIX网络系统设计与实施研讨会（NSDI 23）. USENIX协会, 波士顿, MA, 1375–1389. https://www.usenix.org/conference/nsdi23/presentation/jain [71] Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, 和 Lili Qiu. 2023. LLMLingua: 压缩提示以加速大型语言模型的推理. (2023). arXiv:cs.CL/2310.05736 [72] Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, 和 Lili Qiu. 2023. LongLLMLingua: 通过提示压缩加速和增强长上下文场景中的 LLM. (2023). arXiv:cs.CL/2310.06839 [73] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, 和 Karthik Narasimhan. 2023. SWE-bench: 语言模型能否解决现实世界的GitHub问题? (2023). arXiv:cs.CL/2310.06770 [74] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, 和 Xin Jin. 2024. RAGCache: 用于检索增强生成的高效知识缓存. arXiv预印本 arXiv:2404.12457 (2024). [75] Mandar Joshi, Eunsol Choi, Daniel S. Weld, 和 Luke Zettlemoyer. 2017. Trivi- aQA: 一个大规模远程监督的阅读理解挑战数据集. (2017). arXiv:cs.CL/1705.03551 [76] jwatte. 2023. ChatGPT如何存储聊天历史. https://community.openai.com/t/how-does-chatgpt-store-history-of-chat/319608/2. (2023年8月). OpenAI社区论坛. [77] Waleed Kadous, Kyle Huang, Wendi Ding, Liguang Xie, Avnish Narayan, 和 Ricky Xu. 2023. LLM推理的可重复性能指标. (2023年11月). https://www.anyscale.com/blog/reproducible-performance-metrics-for- llm-inference 访问时间: 2024-06-01. [78] Hao Kang, Qingru Zhang, Souvik Kundu, Geonhwa Jeong, Zaoxing Liu, Tushar Krishna, 和 Tuo Zhao. 2024. Gear: 一种高效的kv缓存压缩配方，用于近无损的LLM生成推理. arXiv预印本 arXiv:2403.05527 (2024). [79] Vladimir Karpukhin, Barlas O uz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, 和 Wen tau Yih. 2020. 开放域问答的密集段落检索. (2020). arXiv:cs.CL/2004.04906 [80] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbah, Alex Rocha, 和 Joe Stubbs. 2020. 从变色龙测试平台中获得的经验教训. 在2020年USENIX年度技术会议（USENIX ATC 20）. USENIX协会, 219–233. https://www.usenix.org/conference/atc20/presentation/keahey [81] Tomá Ko iský, Jonathan Schwarz, Phil Blunsom, Chris Dyer, Karl Moritz Her- mann, Gábor Melis, 和 Edward Grefenstette. 2017. NarrativeQA阅读理解挑战. (2017). arXiv:cs.CL/1712.07040 [82] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, 和 Ion Stoica. 2023. 大型语言模型服务的高效内存管理与分页注意力. 在ACM SIGOPS第29届操作系统原理研讨会论文集. [83] LangChain. 2024. langchain-ai/langchain: 通过可组合性构建LLM应用. https://github.com/langchain-ai/langchain. (2024年2月). (访问时间: 2023年9月21日). [84] LangChain. 2024. 存储和引用聊天历史 | Langchain. https://python.l angchain.com/docs/use_cases/question_answering/how_to/chat_vector_db. (2024年2月). (访问时间: 2023年9月21日). [85] Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, Daniel Haziza, Luca Wehrstedt, Jeremy Reizenstein, 和 Grigory Sizov. 2022. xFormers: 一个模块化和可黑客的Transformer建模库. https: //github.com/facebookresearch/xformers. (2022). [86] Yaniv Leviathan, Matan Kalman, 和 Y. Matias. 2022. 通过推测解码实现快速推理. 在国际机器学习会议上. https://api.semanticscholar.org/CorpusID:254096365 [87] Zijian Lew, Joseph B Walther, Augustine Pang, 和 Wonsun Shin. 2018. 在线聊天中的互动性: 计算机媒介沟通中的对话偶然性和响应延迟. 计算机媒介沟通杂志 23, 4 (2018年6月), 201–221. https://doi.org/10.1093/jcmc/zmy009 arXiv:https://academic.oup.com/jcmc/article- pdf/23/4/201/25113924/zmy009.pdf [88] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rock- täschel, 等. 2020. 知识密集型NLP任务的检索增强生成. 神经信息处理系统进展 33 (2020), 9459–9474. [89] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, 和 Douwe Kiela. 2021. 知识密集型NLP任务的检索增强生成. (2021). arXiv:cs.CL/2005.11401

[90] Dacheng Li*, Rulin Shao*, Anze Xie, Lianmin Zheng Ying Sheng, Joseph E. Gonzalez, Ion Stoica, Xuezhe Ma, and Hao Zhang. 2023. How Long Can Open-Source LLMs Truly Promise on Context Length? (June 2023). https://lmsys.org/blog/2023-06-29-longchat

[91] Bin Lin, Tao Peng, Chen Zhang, Minmin Sun, Lanbo Li, Hanyu Zhao, Wencong Xiao, Qi Xu, Xiafei Qiu, Shen Li, Zhigang Ji, Yong Li, and Wei Lin. 2024. Infinite-LLM: Efficient LLM Service for Long Context with DistAttention and Distributed KVCache. (2024). arXiv:cs.DC/2401.02669

[92] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. 2021. A Survey of Transformers. (2021). arXiv:cs.LG/2106.04554

[93] Jiachen Liu, Zhiyu Wu, Jae-Won Chung, Fan Lai, Myungjin Lee, and Mosharaf Chowdhury. 2024. Andes: Defining and Enhancing Quality-of-Experience in LLM-Based Text Streaming Services. *arXiv preprint arXiv:2404.16283* (2024).

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

[95] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2023. Scissorhands: Exploiting the Persistence of Importance Hypothesis for LLM KV Cache Compression at Test Time. *arXiv preprint arXiv:2305.17118* (2023).

[96] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2023. Scissorhands: Exploiting the Persistence of Importance Hypothesis for LLM KV Cache Compression at Test Time. *arXiv preprint arXiv:2305.17118* (2023).

[97] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2024. KIVI: A Tuning-Free Asymmetric 2bit Quantization for KV Cache. *arXiv preprint arXiv:2402.02750* (2024).

[98] llama.cpp. [n. d.]. llama.cpp. https://github.com/ggerganov/llama.cpp/. ([n. d.]).

[99] Sathiya Kumaran Mani, Yajie Zhou, Kevin Hsieh, Santiago Segarra, Trevor Eberl, Eliran Azulai, Ido Frizler, Ranveer Chandra, and Srikanth Kandula. 2023. Enhancing Network Management Using Code Generated by Large Language Models. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks (HotNets '23)*. Association for Computing Machinery, New York, NY, USA, 196–204. https://doi.org/10.1145/3626111.3628183

[100] Ignacio Martinez. 2023. privateGPT. https://github.com/imartinez/privateGPT. (2023).

[101] Fabian Mentzer, Eirikur Agustsson, Michael Tschannen, Radu Timofte, and Luc Van Gool. 2019. Practical Full Resolution Learned Lossless Image Compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[102] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer Sentinel Mixture Models. (2016). arXiv:cs.CL/1609.07843

[103] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2023. SpecInfer: Accelerating Generative LLM Serving with Speculative Inference and Token Tree Verification. *arXiv preprint arXiv:2305.09781* (2023).

[104] Jesse Mu, Xiang Lisa Li, and Noah Goodman. 2023. Learning to compress prompts with gist tokens. *arXiv preprint arXiv:2304.08467* (2023).

[105] Author's Name. Year of Publication. LLMs in Finance: BloombergGPT and FinGPT - What You Need to Know. Medium. (Year of Publication). https://12gunika.medium.com/llms-in-finance-bloomberggpt-and-fingpt-what-you-need-to-know-2fdf3af29217

[106] Yixin Nie, Songhe Wang, and Mohit Bansal. 2019. Revealing the Importance of Semantic Retrieval for Machine Reading at Scale. (2019). arXiv:cs.CL/1909.08041

[107] Antonio Nucci. 2024. Large Language Models in Financial Services & Banking. (2024). https://aisera.com/blog/large-language-models-in-financial-services-banking/

[108] OpenAI. 2024. GPT-4 API general availability and deprecation of older models in the Completions API. https://openai.com/blog/gpt-4-api-general-availability. (April 2024). (Accessed on 09/21/2023).

[109] I. V. Oseledets. 2011. Tensor-Train Decomposition. *SIAM Journal on Scientific Computing* 33, 5 (2011), 2295–2317. https://doi.org/10.1137/090752286 arXiv:https://doi.org/10.1137/090752286

[110] Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative Agents: Interactive Simulacra of Human Behavior. (2023). arXiv:cs.HC/2304.03442

[111] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. 2023. Splitwise: Efficient generative llm inference using phase splitting. *arXiv preprint arXiv:2311.18677* (2023).

[112] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2022. Efficiently Scaling Transformer Inference. (2022). arXiv:cs.LG/2211.05102

[113] Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. 2023. In-Context Retrieval-Augmented Language Models. (2023). arXiv:cs.CL/2302.00083

[114] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. https://doi.org/10.1145/3394486.3406703

[115] Luka Ribar, Ivan Chelombiev, Luke Hudlass-Galley, Charlie Blake, Carlo Luschi, and Douglas Orr. 2023. SparQ Attention: Bandwidth-Efficient LLM Inference. (2023). arXiv:cs.LG/2312.04985

[116] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. 2021. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics* 9 (2021), 53–68.

[117] Ohad Rubin and Jonathan Berant. 2023. Long-range Language Modeling with Self-retrieval. *arXiv preprint arXiv:2306.13421* (2023).

[118] Ayesha Saleem. 2023. LLM for Lawyers, Enrich Your Precedents with the Use of AI. Data Science Dojo. (25 July 2023). https://datasciencedojo.com/blog/llm-for-lawyers/

[119] Hang Shao, Bei Liu, and Yanmin Qian. 2024. One-Shot Sensitivity-Aware Mixed Sparsity Pruning for Large Language Models. (2024). arXiv:cs.CL/2310.09499

[120] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E Gonzalez, et al. 2023. High-throughput generative inference of large language models with a single gpu. *arXiv preprint arXiv:2303.06865* (2023).

[121] Zijing Shi, Meng Fang, Shunfeng Zheng, Shilong Deng, Ling Chen, and Yali Du. 2023. Cooperation on the Fly: Exploring Language Agents for Ad Hoc Teamwork in the Avalon Game. (2023). arXiv:cs.CL/2312.17515

[122] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[123] Yisheng Song, Ting Wang, Puyu Cai, Subrota K. Mondal, and Jyoti Prakash Sahoo. 2023. A Comprehensive Survey of Few-shot Learning: Evolution, Applications, Challenges, and Opportunities. *ACM Comput. Surv.* 55, 13s, Article 271 (jul 2023), 40 pages. https://doi.org/10.1145/3582688

[124] Simeng Sun, Kalpesh Krishna, Andrew Mattarella-Micke, and Mohit Iyyer. 2021. Do long-range language models actually use long-range context? *arXiv preprint arXiv:2109.09115* (2021).

[125] Pavlo Sydorenko. 2023. Top 5 Applications of Large Language Models (LLMs) in Legal Practice. Medium. (2023). https://medium.com/jurdep/top-5-applications-of-large-language-models-llms-in-legal-practice-d29cde9c38ef

[126] Vivienne Sze and Madhukar Budagavi. 2012. High Throughput CABAC Entropy Coding in HEVC. *IEEE Transactions on Circuits and Systems for Video Technology* 22, 12 (2012), 1778–1791. https://doi.org/10.1109/TCSVT.2012.2221526

[127] Zilliz Technology. 2023. GPTCache. https://github.com/zilliztech/GPTCache. (2023).

[128] Keary Tim. 2024. 12 Practical Large Language Model (LLM) Applications - Techopedia. https://www.techopedia.com/12-practical-large-language-model-llm-applications. (January 2024). (Accessed on 09/21/2023).

[129] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. (2023). arXiv:cs.CL/2302.13971 https://arxiv.org/abs/2302.13971

[130] Szymon Tworkowski, Konrad Staniszewski, Mikołaj Pacek, Yuhuai Wu, Henryk Michalewski, and Piotr Miłoś. 2023. Focused transformer: Contrastive training for context scaling. *arXiv preprint arXiv:2307.03170* (2023).

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

[132] Yiding Wang, Decang Sun, Kai Chen, Fan Lai, and Mosharaf Chowdhury. 2023. Egeria: Efficient DNN Training with Knowledge-Guided Layer Freezing. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 851–866. https://doi.org/10.1145/3552326.3587451

[133] Zhuang Wang, Haibin Lin, Yibo Zhu, and T. S. Eugene Ng. 2023. Hi-Speed DNN Training with Espresso: Unleashing the Full Potential of Gradient Compression with Near-Optimal Usage Strategies. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 867–882. https://doi.org/10.1145/3552326.3567505

[134] Zhenhailong Wang, Xiaoman Pan, Dian Yu, Dong Yu, Jianshu Chen, and Heng Ji. 2023. Zemi: Learning Zero-Shot Semi-Parametric Language Models from Multiple Tasks. (2023). arXiv:cs.CL/2210.00185

[135] Ian H. Witten, Radford M. Neal, and John G. Cleary. 1987. Arithmetic Coding for Data Compression. *Commun. ACM* 30, 6 (jun 1987), 520–540. https://doi.org/10.1145/214762.214771

[136] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Perric Cistac, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest,

[90] Dacheng Li*, Rulin Shao*, Anze Xie, Lianmin Zheng Ying Sheng, Joseph E. Gonzalez, Ion Stoica, Xuezhe Ma, 和 Hao Zhang. 2023. 开源 LLM 在上下文长度上能真正承诺多长时间？（2023年6月）。https://lmsys.org/blog/2023-06-29-longchat [91] Bin Lin, Tao Peng, Chen Zhang, Minmin Sun, Lanbo Li, Hanyu Zhao, Wencong Xiao, Qi Xu, Xiafei Qiu, Shen Li, Zhigang Ji, Yong Li, 和 Wei Lin. 2024. Infinite-LLM：用于长上下文的高效 LLM 服务，结合 DistAttention 和分布式 KVCache。（2024年）。arXiv:cs.DC/2401.02669 [92] Tianyang Lin, Yuxin Wang, Xiangyang Liu, 和 Xipeng Qiu. 2021. 变压器的调查。（2021年）。arXiv:cs.LG/2106.04554 [93] Jiachen Liu, Zhiyu Wu, Jae-Won Chung, Fan Lai, Myungjin Lee, 和 Mosharaf Chowdhury. 2024. Andes：在基于 LLM 的文本流服务中定义和增强用户体验质量。arXiv 预印本 arXiv:2404.16283（2024年）。 [94] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, 和 Percy Liang. 2023. 在中间迷失：语言模型如何使用长上下文。arXiv 预印本 arXiv:2307.03172（2023年）。 [95] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, 和 Anshumali Shrivastava. 2023. Scissorhands：利用重要性假设的持久性进行 LLM KV 缓存压缩。arXiv 预印本 arXiv:2305.17118（2023年）。 [96] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, 和 Anshumali Shrivastava. 2023. Scissorhands：利用重要性假设的持久性进行 LLM KV 缓存压缩。arXiv 预印本 arXiv:2305.17118（2023年）。 [97] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, 和 Xia Hu. 2024. KIVI：一种无调优的非对称 2bit 量化用于 KV 缓存。arXiv 预印本 arXiv:2402.02750（2024年）。 [98] llama.cpp. [n. d.]. llama.cpp. https://github.com/ggerganov/llama.cpp/.（[n. d.]）。 [99] Sathiya Kumaran Mani, Yajie Zhou, Kevin Hsieh, Santiago Segarra, Trevor Eberl, Eliran Azulai, Ido Frizler, Ranveer Chandra, 和 Srikanth Kandula. 2023. 使用大型语言模型生成的代码增强网络管理。在第22届 ACM 热点网络研讨会（HotNets '23）论文集中。计算机协会，纽约，NY，美国，196–204。https://doi.org/10.1145/3626111.3628183 [100] Ignacio Martinez. 2023. privateGPT. https://github.com/imartinez/privateGPT.（2023年）。 [101] Fabian Mentzer, Eirikur Agustsson, Michael Tschannen, Radu Timofte, 和 Luc Van Gool. 2019. 实用的全分辨率学习无损图像压缩。在 IEEE 计算机视觉与模式识别会议（CVPR）论文集中。 [102] Stephen Merity, Caiming Xiong, James Bradbury, 和 Richard Socher. 2016. 指针哨兵混合模型。（2016年）。arXiv:cs.CL/1609.07843 [103] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, 和 Zhihao Jia. 2023. SpecInfer：通过推测推理和令牌树验证加速生成 LLM 服务。arXiv 预印本 arXiv:2305.09781（2023年）。 [104] Jesse Mu, Xiang Lisa Li, 和 Noah Goodman. 2023. 学习使用要点令牌压缩提示。arXiv 预印本 arXiv:2304.08467（2023年）。 [105] 作者姓名。出版年份。金融中的 LLM：BloombergGPT 和 FinGPT - 您需要知道的事项。Medium。（出版年份）。https://12gunika.medium.com/llms-in-finance-bloomberggpt-and-fingpt-what-you-need-to-know-2fdf3af29217 [106] Yixin Nie, Songhe Wang, 和 Mohit Bansal. 2019. 揭示语义检索在大规模机器阅读中的重要性。（2019年）。arXiv:cs.CL/1909.08041 [107] Antonio Nucci. 2024. 金融服务与银行中的大型语言模型。（2024年）。https://aisera.com/blog/large-language-models-in-financial-services-banking/ [108] OpenAI. 2024. GPT-4 API 的普遍可用性和旧模型在 Completions API 中的弃用。https://openai.com/blog/gpt-4-api-general-availability.（2024年4月）。（访问日期：2023年9月21日）。 [109] I. V. Oseledets. 2011. 张量列车分解。SIAM 科学计算杂志 33, 5（2011年），2295–2317。https://doi.org/10.1137/090752286 arXiv:https://doi.org/10.1137/090752286 [110] Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, 和 Michael S. Bernstein. 2023. 生成代理：人类行为的互动模拟。（2023年）。arXiv:cs.HC/2304.03442 [111] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, 和 Ricardo Bianchini. 2023. Splitwise：使用相位分割的高效生成 LLM 推理。arXiv 预印本 arXiv:2311.18677（2023年）。 [112] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, 和 Jeff Dean. 2022. 高效扩展变压器推理。（2022年）。arXiv:cs.LG/2211.05102 [113] Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, 和 Yoav Shoham. 2023. 上下文检索增强语言模型。（2023年）。arXiv:cs.CL/2302.00083 [114] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, 和 Yuxiong He. 2020. DeepSpeed: 系统优化使得训练超过 1000 亿参数的深度学习模型成为可能. 在第 26 届 ACM SIGKDD 国际知识发现与数据挖掘会议 (KDD '20) 论文集中. 计算机协会, 纽约, NY, 美国, 3505–3506. https://doi.org/10.1145/3394486.3406703 [115] Luka Ribar, Ivan Chelombiev, Luke Hudlass-Galley, Charlie Blake, Carlo Luschi, 和 Douglas Orr. 2023. SparQ Attention: 带宽高效的 LLM 推理. (2023). arXiv:cs.LG/2312.04985 [116] Aurko Roy, Mohammad Saffar, Ashish Vaswani, 和 David Grangier. 2021. 基于内容的高效稀疏注意力与路由变换器. 计算语言学协会会刊 9 (2021), 53–68. [117] Ohad Rubin 和 Jonathan Berant. 2023. 自我检索的长程语言建模. arXiv 预印本 arXiv:2306.13421 (2023). [118] Ayesha Saleem. 2023. 律师的 LLM，利用 AI 丰富您的先例. 数据科学道场. (2023年7月25日). https://datasciencedojo.com/blog/llm-for-lawyers/ [119] Hang Shao, Bei Liu, 和 Yanmin Qian. 2024. 针对大型语言模型的一次性敏感性感知混合稀疏修剪. (2024). arXiv:cs.CL/2310.09499 [120] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E Gonzalez, 等. 2023. 使用单个 GPU 的大型语言模型的高通量生成推理. arXiv 预印本 arXiv:2303.06865 (2023). [121] Zijing Shi, Meng Fang, Shunfeng Zheng, Shilong Deng, Ling Chen, 和 Yali Du. 2023. 随时合作：探索阿瓦隆游戏中的语言代理以进行临时团队合作. (2023). arXiv:cs.CL/2312.17515 [122] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, 和 Bryan Catanzaro. 2019. Megatron-lm: 使用模型并行训练数十亿参数的语言模型. arXiv 预印本 arXiv:1909.08053 (2019). [123] Yisheng Song, Ting Wang, Puyu Cai, Subrota K. Mondal, 和 Jyoti Prakash Sahoo. 2023. 少样本学习的综合调查：演变、应用、挑战与机遇. ACM 计算机调查 55, 13s, 文章 271 (2023年7月), 40 页. https://doi.org/10.1145/3582688 [124] Simeng Sun, Kalpesh Krishna, Andrew Mattarella-Micke, 和 Mohit Iyyer. 2021. 长程语言模型是否真的使用长程上下文？arXiv 预印本 arXiv:2109.09115 (2021). [125] Pavlo Sydorenko. 2023. 大型语言模型 (LLMs) 在法律实践中的前 5 大应用. Medium. (2023). https://medium.com/jurdep/top-5-applications-of-large-language-models-llms-in-legal-practice-d29cde9c38ef [126] Vivienne Sze 和 Madhukar Budagavi. 2012. HEVC 中的高通量 CABAC 熵编码. IEEE 视频技术电路与系统交易 22, 12 (2012), 1778–1791. https://doi.org/10.1109/TCSVT.2012.2221526 [127] Zilliz Technology. 2023. GPTCache. https://github.com/zilliztech/GPTCache. (2023). [128] Keary Tim. 2024. 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. (2024年1月). (访问于 2023年9月21日). [129] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, 和 Guillaume Lample. 2023. LLaMA: 开放且高效的基础语言模型. (2023). arXiv:cs.CL/2302.13971 https://arxiv.org/abs/2302.13971 [130] Szymon Tworkowski, Konrad Staniszewski, Mikoaj Pacek, Yuhuai Wu, Henryk Michalewski, 和 Piotr Mio . 2023. 聚焦变换器：用于上下文缩放的对比训练. arXiv 预印本 arXiv:2307.03170 (2023). [131] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, 和 Illia Polosukhin. 2023. 注意力就是你所需要的一切. (2023). arXiv:cs.CL/1706.03762 [132] Yiding Wang, Decang Sun, Kai Chen, Fan Lai, 和 Mosharaf Chowdhury. 2023. Egeria: 通过知识引导的层冻结实现高效 DNN 训练. 在第十八届欧洲计算机系统会议 (EuroSys '23) 论文集中. 计算机协会, 纽约, NY, 美国, 851–866. https://doi.org/10.1145/3552326.3587451 [133] Zhuang Wang, Haibin Lin, Yibo Zhu, 和 T. S. Eugene Ng. 2023. 使用 Espresso 的高速 DNN 训练：释放梯度压缩的全部潜力与近乎最佳的使用策略. 在第十八届欧洲计算机系统会议 (EuroSys '23) 论文集中. 计算机协会, 纽约, NY, 美国, 867–882. https://doi.org/10.1145/3552326.3567505 [134] Zhenhailong Wang, Xiaoman Pan, Dian Yu, Dong Yu, Jianshu Chen, 和 Heng Ji. 2023. Zemi：从多个任务中学习零样本半参数语言模型. (2023). arXiv:cs.CL/2210.00185 [135] Ian H. Witten, Radford M. Neal, 和 John G. Cleary. 1987. 数据压缩的算术编码. ACM 通信 30, 6 (1987年6月), 520–540. https://doi.org/10.1145/214762.214771 [136] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Perric Cistac, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest,

and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. Association for Computational Linguistics, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[137] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[138] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. LoongServe: Efficiently Serving Long-context Large Language Models with Elastic Sequence Parallelism. *arXiv preprint arXiv:2404.09526* (2024).

[139] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Fast Distributed Inference Serving for Large Language Models. (2023). arXiv:cs.LG/2305.05920

[140] Dekun Wu, Haochen Shi, Zhiyuan Sun, and Bang Liu. 2023. Deciphering Digital Detectives: Understanding LLM Behaviors and Capabilities in Multi-Agent Mystery Games. (2023). arXiv:cs.AI/2312.00746

[141] Yuhuai Wu, Markus Norman Rabe, DeLesley Hutchins, and Christian Szegedy. 2022. Memorizing Transformers. In *International Conference on Learning Representations*. https://openreview.net/forum?id=TrjbxzRcnf-

[142] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*. PMLR, 38087–38099.

[143] Wenhan Xiong, Hong Wang, and William Yang Wang. 2021. Progressively Pretrained Dense Corpus Index for Open-Domain Question Answering. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*. Association for Computational Linguistics, Online, 2803–2815. https://doi.org/10.18653/v1/2021.eacl-main.244

[144] Peng Xu, Wei Ping, Xianchao Wu, Lawrence McAfee, Chen Zhu, Zihan Liu, Sandeep Subramanian, Evelina Bakhturina, Mohammad Shoeybi, and Bryan Catanzaro. 2024. Retrieval meets Long Context Large Language Models. (2024). arXiv:cs.CL/2310.03025

[145] Wei Yang, Yuqing Xie, Aileen Lin, Xingyu Li, Luchen Tan, Kun Xiong, Ming Li, and Jimmy Lin. 2019. End-to-End Open-Domain Question Answering with. In *Proceedings of the 2019 Conference of the North*. Association for Computational Linguistics. https://doi.org/10.18653/v1/n19-4013

[146] Yingrui Yang, Yifan Qiao, Jinjin Shao, Xifeng Yan, and Tao Yang. 2022. Lightweight Composite Re-Ranking for Efficient Keyword Search with BERT. In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining (WSDM '22)*. Association for Computing Machinery, New York, NY, USA, 1234–1244. https://doi.org/10.1145/3488560.3498495

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

[148] Rongjie Yi, Liwei Guo, Shiyun Wei, Ao Zhou, Shangguang Wang, and Mengwei Xu. 2023. EdgeMoE: Fast On-Device Inference of MoE-based Large Language Models. *arXiv preprint arXiv:2308.14352* (2023).

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

[150] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. 2021. Big Bird: Transformers for Longer Sequences. (2021). arXiv:cs.LG/2007.14062

[151] Lin Zehui, Pengfei Liu, Luyao Huang, Junkun Chen, Xipeng Qiu, and Xuanjing Huang. 2019. DropAttention: A Regularization Method for Fully-Connected Self-Attention Networks. (2019). arXiv:cs.CL/1907.11065

[152] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Re, Clark Barrett, Zhangyang Wang, and Beidi Chen. 2023. H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. In *Workshop on Efficient Systems for Foundation Models @ ICML2023*. https://openreview.net/forum?id=ctPizehA9D

[153] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. 2023. H_2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. (2023). arXiv:cs.LG/2306.14048

[154] Qibin Zhao, Guoxu Zhou, Shengli Xie, Liqing Zhang, and Andrzej Cichocki. 2016. Tensor Ring Decomposition. (2016). arXiv:cs.NA/1606.05535

[155] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. (2023). arXiv:cs.CL/2306.05685

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

[157] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. (2024). arXiv:cs.DC/2401.09670

亚历山大·M拉什。2020年。《变压器：最先进的自然语言处理》。计算语言学协会，38–45。https://www.aclweb.org/anthology/2020.emnlp-demos.6 [137] 托马斯·沃尔夫，利桑德尔·德比，维克多·桑赫，朱利安·肖蒙，克莱门特·德朗，安东尼·莫伊，皮埃里克·西斯塔克，蒂姆·劳特，雷米·卢夫，摩根·芬托维茨，乔·戴维森，萨姆·施莱弗，帕特里克·冯·普拉特，克拉拉·马，雅辛·杰尔尼特，朱利安·普鲁，徐灿文，特文·勒·斯卡奥，西尔万·古格，玛丽亚马·德拉梅，昆廷·洛赫斯特，亚历山大·M拉什。2020年。《变压器：最先进的自然语言处理》。在2020年自然语言处理经验方法会议：系统演示的论文集中。计算语言学协会，在线，38–45。https://www.aclweb.org/anthology/2020.emnlp-demos.6 [138] 吴秉扬，刘胜宇，钟银敏，孙鹏，刘轩哲，金鑫。2024年。《LoongServe：高效服务长上下文大型语言模型的弹性序列并行》。arXiv预印本arXiv:2404.09526（2024）。[139] 吴秉扬，钟银敏，张子力，黄刚，刘轩哲，金鑫。2023年。《大型语言模型的快速分布式推理服务》。（2023）。arXiv:cs.LG/2305.05920 [140] 吴德坤，施浩辰，孙志远，刘邦。2023年。《解读数字侦探：理解多代理神秘游戏中的LLM行为和能力》。（2023）。arXiv:cs.AI/2312.00746 [141] 吴宇怀，马库斯·诺曼·拉贝，德莱斯利·哈钦斯，克里斯蒂安·塞格迪。2022年。《记忆变压器》。在国际学习表征会议上。https://openreview.net/forum?id=TrjbxzRcnf- [142] 肖光轩，林骥，米卡埃尔·塞兹内克，吴浩，朱利安·德穆斯，韩松。2023年。《Smoothquant：针对大型语言模型的准确高效后训练量化》。在国际机器学习会议上。PMLR，38087–38099。[143] 熊文汉，王洪，王威廉·杨。2021年。《逐步预训练的稠密语料索引用于开放域问答》。在第十六届欧洲计算语言学协会会议论文集：主卷中。计算语言学协会，在线，2803–2815。https://doi.org/10.18653/v1/2021.eacl-main.244 [144] 许鹏，平伟，吴先超，劳伦斯·麦卡菲，朱晨，刘子涵，桑迪普·苏布拉马尼安，埃维琳娜·巴赫图里娜，穆罕默德·肖耶比，布莱恩·卡坦扎罗。2024年。《检索与长上下文大型语言模型相遇》。（2024）。arXiv:cs.CL/2310.03025 [145] 杨伟，谢宇清，林艾琳，李星宇，谭璐晨，熊坤，李明，林吉米。2019年。《端到端开放域问答》。在2019年北方计算语言学协会会议论文集中。https://doi.org/10.18653/v1/n19-4013 [146] 杨英瑞，乔逸凡，邵金金，严西峰，杨涛。2022年。《轻量级复合重排序用于高效关键词搜索与BERT》。在第十五届ACM国际网络搜索与数据挖掘会议（WSDM '22）论文集中。计算机协会，纽约，NY，美国，1234–1244。https://doi.org/10.1145/3488560.3498495 [147] 姚佳怡，李汉辰，刘宇涵，西达特·雷，程义华，张启正，杜坤泰，卢珊，蒋俊辰。2024年。《CacheBlend：快速大型语言模型服务与缓存知识融合》。arXiv预印本arXiv:2405.16444（2024）。[148] 易荣杰，郭立伟，魏诗云，周傲，王尚广，徐梦伟。2023年。《EdgeMoE：基于MoE的大型语言模型的快速设备推理》。arXiv预印本arXiv:2308.14352（2023）。[149] 俞庆仁，郑周成，金建宇，金秀正，春炳根。2022年。《Orca：一个用于{基于变压器}生成模型的分布式服务系统》。在第十六届USENIX操作系统设计与实现研讨会（OSDI 22）上。521–538。[150] 曼齐尔·扎希尔，古鲁·古鲁甘什，阿维纳瓦·杜贝，约书亚·艾因斯利，克里斯·阿尔伯蒂，圣地亚哥·昂塔农，菲利普·范，阿尼鲁德·拉武拉，齐凡·王，李扬，阿姆尔·艾哈迈德。2021年。《大鸟：用于更长序列的变压器》。（2021）。arXiv:cs.LG/2007.14062 [151] 林泽辉，刘鹏飞，黄璐瑶，陈俊琨，邱希鹏，黄轩晶。2019年。《DropAttention：一种用于全连接自注意力网络的正则化方法》。（2019）。arXiv:cs.CL/1907.11065 [152] 张振宇，英盛，周天怡，陈天龙，郑连敏，蔡瑞思，宋兆，田远东，克里斯托弗·雷，克拉克·巴雷特，张扬·王，陈贝迪。2023年。《H2O：高效生成大型语言模型推理的重击者预言机》。在ICML2023基础模型高效系统研讨会上。https://openreview.net/forum?id=ctPizehA9D [153] 张振宇，英盛，周天怡，陈天龙，郑连敏，蔡瑞思，宋兆，田远东，克里斯托弗·雷，克拉克·巴雷特，张扬·王，陈贝迪。2023年。《H_2O：高效生成大型语言模型推理的重击者预言机》。（2023）。arXiv:cs.LG/2306.14048 [154] 赵启斌，周国旭，谢胜利，张丽青，安杰伊·奇霍基。2016年。《张量环分解》。（2016）。arXiv:cs.NA/1606.05535 [155] 郑连敏，魏林·蒋，英盛，庄思源，吴张浩，庄永浩，林子，李卓涵，李大成，埃里克·P·辛，张浩，约瑟夫·E·冈萨雷斯和伊昂·斯托伊卡。2023年。使用MT-Bench和聊天机器人竞技场评估LLM作为法官的表现。（2023年）。arXiv:cs.CL/2306.05685 [156] 郑连敏、尹良生、谢志强、黄杰、孙楚悦、余浩、曹世怡、克里斯托斯·科齐拉基斯、伊昂·斯托伊卡、约瑟夫·E·冈萨雷斯等。2023年。使用sglang高效编程大型语言模型。arXiv预印本arXiv:2312.07104（2023年）。[157] 钟寅敏、刘胜宇、陈俊达、胡建波、朱怡博、刘轩哲、金鑫和张浩。2024年。DistServe：为优化良率的大型语言模型服务分解预填充和解码。（2024年）。arXiv:cs.DC/2401.09670

Note: Appendices are supporting material that has not been peer-reviewed.

A Text Output Examples of CacheGen

Figure 17 visualizes an example from the LongChat dataset [90] used in §7.2. The context fed into the LLM is a long, multi-round conversation history between the LLM and the user. An abridged context is shown in the upper box, where the first topic is about the role of art in society. The prompt to the LLM asks "What is the first topic we discussed? " CacheGen correctly generates the answer, whereas the default quantization baseline, which has a similar compressed KV cache size as CacheGen, generates the wrong answer.

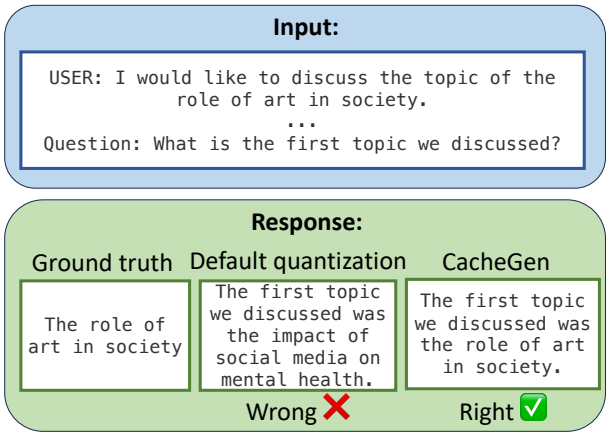


Figure 17: *An example of CacheGen's output on the LongChat dataset with LongChat-7b-16k model.*

B CacheGen vs. more intrusive methods

So far, all methods we have evaluated, including CacheGen, do not modify the LLM or context. As a complement, Figure 18 tests CacheGen against recent methods that *change* the context or LLM.

- *Smaller models:* Replacing the LLM with *smaller models* may speed up the computation. Figure 18a replaces the Llama-7B model with a smaller Llama-3B and applies different quantization levels.
- *Token selection:* Figure 18b uses Scissorhands as an example of *removing tokens* with low self-attention scores from the LLM input [96]. Since the self-attention scores are only available during the actual generation, it cannot reduce TTFT, but we make an effort to create an idealized version of Scissorhands (Scissorhands*) by running the self-attention offline to determine which tokens to drop and provide this information to Scissorhands* online.
- *Gisting* Finally, we test Gisting as an example of a more advanced method that shortens contexts into gist tokens and changes the LLM to accept the gist tokens [104]. In Figure 18c, we test the pre-trained gisting model, which is based on Llama-7B. The gisting model retrains the LLM's attention model in order to run inference on a compressed version of the input prompts. Since the gisting model can compress arbitrary long contexts into *one token*, we vary the compression ratio of the gisting model to obtain a trade-off in size and accuracy. This is done by adapting the fraction of input tokens that are compressed into one token. We apply CacheGen on the original Llama-7B model on the PIQA [34] dataset, which is one of the most popular question-answering datasets. We did not apply CacheGen on other datasets in our evaluation because the public pre-trained gisting model can only take up to 512 tokens, and truncating the dataset into smaller will not be able to preserve the information in the context.

We can see that CacheGen outperforms these baselines, reducing TTFT or KV cache size while achieving similar or better LLM's performance on the respective tasks. In particular, CacheGen is faster than smaller models (which are slowed down by transformer operations), and can reduce KV cache better than context selection or gisting because it compresses the KV features to more compact bitstream representations. We want to stress that even though CacheGen is compared head-to-head with these methods, it makes no assumption about the context and the model, so one can combine CacheGen with these methods to potentially further improve the performance.

注意：附录是未经过同行评审的支持材料。

缓存生成的文本输出示例

图17可视化了在§7.2中使用的LongChat数据集[90]中的一个示例。输入到LLM的上下文是LLM与用户之间的长时间多轮对话历史。上方框中显示了简化的上下文，其中第一个主题是艺术在社会中的作用。对LLM的提示是“我们讨论的第一个主题是什么？” CacheGen正确生成了答案，而默认量化基线（其压缩的KV缓存大小与CacheGen相似）生成了错误的答案。

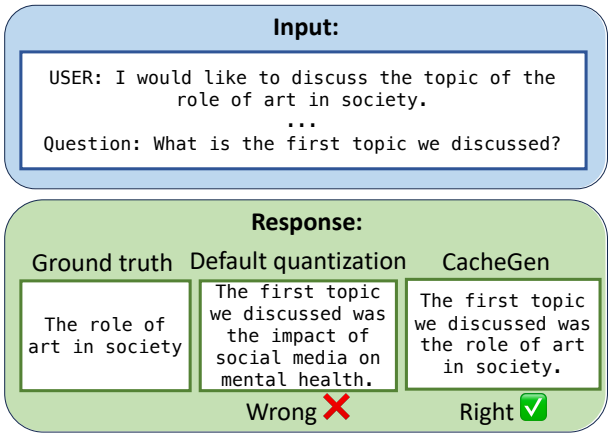


图17：CacheGen在LongChat数据集上使用LongChat-7b-16k模型的输出示例。

B CacheGen 与更具侵入性的方法

到目前为止，我们评估的所有方法，包括CacheGen，都没有修改LLM或上下文。作为补充，图18测试了CacheGen与最近改变上下文或LLM的方法。

- 较小的模型：用较小的模型替换LLM可能会加快计算速度。图18a将Llama-7B模型替换为较小的Llama-3B，并应用不同的量化级别。
- 令牌选择：图18b以剪刀手为例，说明如何从LLM输入中移除自注意力得分低的令牌[96]。由于自注意力得分仅在实际生成过程中可用，因此无法减少TTFT，但我们努力通过离线运行自注意力来创建剪刀手的理想化版本（Scissorhands*），以确定要丢弃的令牌，并将此信息在线提供给Scissorhands*。
- 摘要 最后，我们测试摘要作为一种更高级的方法的示例，该方法将上下文缩短为摘要令牌，并改变LLM以接受摘要令牌[104]。在图18c中，我们测试基于Llama-7B的预训练摘要模型。摘要模型重新训练LLM的注意力模型，以便在输入提示的压缩版本上进行推理。由于摘要模型可以将任意长的上下文压缩为一个令牌，我们改变摘要模型的压缩比，以获得大小和准确性之间的权衡。这是通过调整{v*}完成的。

输入标记被压缩为一个标记的比例。我们在 PIQA [34] 数据集上对原始的 Llama-7B 模型应用了 CacheGen，该数据集是最受欢迎的问题回答数据集之一。我们在评估中没有对其他数据集应用 CacheGen，因为公共预训练的摘要模型最多只能处理 512 个标记，而将数据集截断为更小的部分将无法保留上下文中的信息。

我们可以看到，CacheGen 的性能优于这些基线，减少了 TTFT 或 KV 缓存的大小，同时在各自的任务上实现了相似或更好的 LLM 性能。特别是，CacheGen 比较小的模型更快（这些模型因变换器操作而变慢），并且能够比上下文选择或摘要更好地减少 KV 缓存，因为它将 KV 特征压缩为更紧凑的比特流表示。我们想强调的是，尽管 CacheGen 与这些方法进行了正面比较，但它并不对上下文和模型做出任何假设，因此可以将 CacheGen 与这些方法结合，以潜在地进一步提高性能。

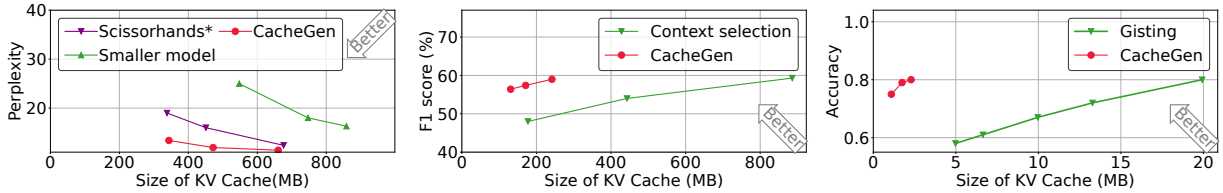


Figure 18: *Comparing CacheGen and more intrusive methods, including smaller models, token dropping (left), context selection (middle), and gisting (right).*

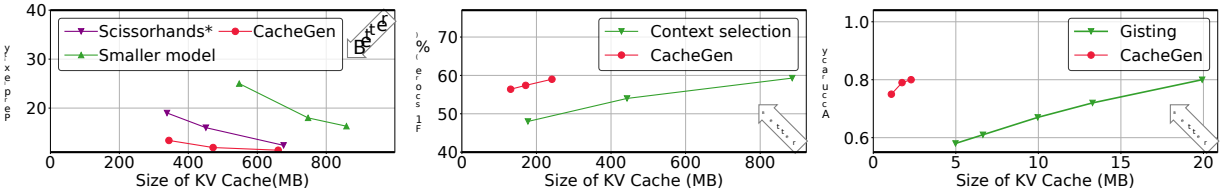


图18：比较CacheGen和更具侵入性的方法，包括更小的模型、令牌丢弃（左）、上下文选择（中）和摘要（右）。

C CacheGen System Settings

C.1 KV Streamer Adaptation Logic

We present the pseudo-code for the KV streamer logic that adapts to bandwidth here.

Algorithm 1: CacheGen Streaming Adapter Logic

```
chunks_to_send ← context chunks
while chunks_to_send ≠ empty do
    get chunk_data
    throughput ← network throughput
    remaining_time ← SLO − time_elapsed
    if time_recompute ≤ remaining_time then
        cur_chunk ← text of chunk_data
    else
        level ← max(level|size(chunks_to_send, level) ÷
        throughput ≤ remaining_time
        cur_chunk ← encode(chunk_data, level)
    end if
    send cur_chunk
    chunks_to_send ← chunks_to_send \ chunk_data
end while
```

C.2 Default Encoding Level

By default, CacheGen encoding is done with the following parameters: we partition the layers in the LLM into three groups with equal distance, and set quantization bins to be 0.5, 1, 1.5 respectively.

D CacheGen's improvement under various workloads

Figure 19 shows CacheGen's improvement over the best baseline (between quantization and text context) over a complete space of workloads characterized along the two dimensions of GPU available cycles (i.e., $1/n$ with n being the number of concurrent requests) and available bandwidth (in log scale). Figure 11 and Figure 12 can be seen as horizontal/vertical cross-sections of this figure.

E Cost of storing KV cache

Our main focus in this paper is to reduce TTFT to achieve service SLO with minimal impact on the generation quality of LLM. However, context loading systems, especially CacheGen, could be an economical choice for LLM service providers as well. For example, one piece of 8.5K-token context in Llama-13B takes roughly 5GB to store different versions compressed with CacheGen. It costs $0.05 per month to store this data on AWS [6]. On the other hand, recomputing the KV cache from text costs at least $0.00085 (input only) every time [4, 5, 11, 12]. If there are more than 150 requests reusing this piece of context every month, CacheGen will also reduce the inference cost. The calculation here only serves as a rough estimation to highlight CacheGen's potential. We leave the design of such a context loading system targeting cost-saving to future work.

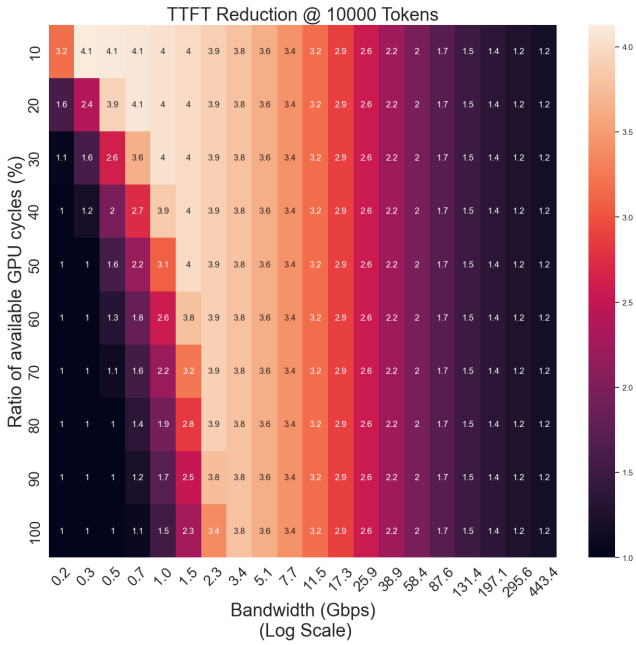


Figure 19: *Heatmap showing CacheGen's improvement over the best baseline over a complete space of workloads. Brighter cells means more TTFT reduction.*

C CacheGen 系统设置 C.1 KV 流媒体适配逻辑

我们在这里展示了适应带宽的KV流媒体逻辑的伪代码。

Algorithm 1: CacheGen Streaming Adapter Logic

```
chunks_to_send ← context chunks
while chunks_to_send ≠ empty do
    get chunk_data
    throughput ← network throughput
    remaining_time ← SLO − time_elapsed
    if time_recompute ≤ remaining_time then
        cur_chunk ← text of chunk_data
    else
        level ← max(level|size(chunks_to_send, level) ÷
        throughput ≤ remaining_time
        cur_chunk ← encode(chunk_data, level)
    end if
    send cur_chunk
    chunks_to_send ← chunks_to_send \ chunk_data
end while
```

C.2 默认编码级别

默认情况下，CacheGen 编码使用以下参数：我们将 LLM 中的层分成三个等距的组，并将量化区间分别设置为 0.5、1、1.5。

D CacheGen 在各种工作负载下的改进

图19显示了CacheGen在最佳基线（量化与文本上下文之间）上的改进，涵盖了在GPU可用周期（即1/n，其中n为并发请求数量）和可用带宽（以对数刻度表示）这两个维度上特征化的完整工作负载空间。图11和图12可以视为该图的水平/垂直截面。

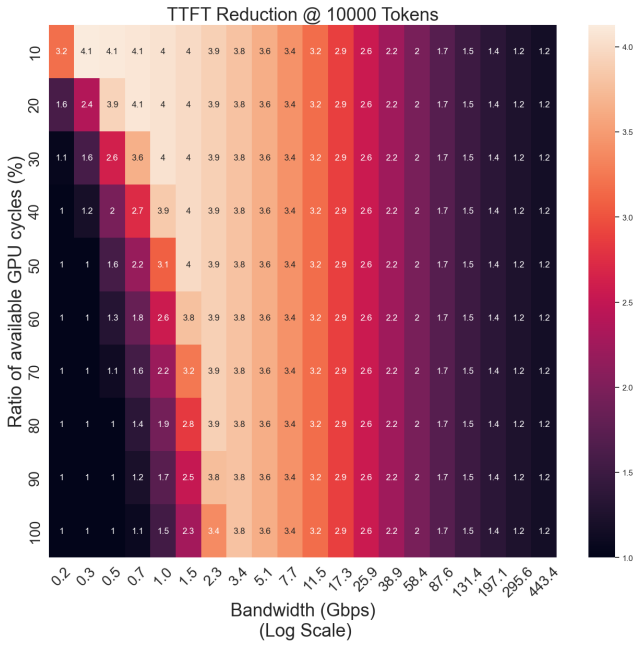


图19：热图显示了CacheGen在完整工作负载空间中相对于最佳基线的改进。更亮的单元格意味着更大的TTFT减少。

E 存储 KV 缓存的成本

我们在本文中的主要关注点是减少 TTFT，以实现服务 SLO，同时对 LLM 的生成质量影响最小。然而，上下文加载系统，特别是 CacheGen，对于 LLM 服务提供商来说也可能是一个经济的选择。例如，在 Llama-13B 中，一段 8.5K 令牌的上下文大约需要 5GB 来存储使用 CacheGen 压缩的不同版本。费用为 $0.05。

每月在AWS上存储这些数据的费用为[6]。另一方面，从文本重新计算KV缓存每次至少需要$0.00085（仅输入）[4, 5, 11, 12]。如果每月有超过150个请求重用这段上下文，CacheGen也将降低推理成本。这里的计算仅作为粗略估计，以突出CacheGen的潜力。我们将设计这样一个针对节省成本的上下文加载系统的工作留给未来。