Cahcegen: 通过压缩和传输 KV 缓存实现快速的大模型服务

摘要

随着大型语言模型（LLMs）承担起复杂任务，需要为输入内容补充更长的上下文内容，这些上下文包含了相关领域知识。然而，使用长上下文是具有挑战性的，因为 LLM 需要大量时间处理长上下文。虽然可以通过重用不同输入的上下文的 KV 缓存来减少上下文处理的延迟，但通过网络获取包含大型张量的 KV 缓存可能会导致额外的高网络延迟。CacheGen 是 LLM 系统的一个快速上下文加载模块。首先，CacheGen 使用一个自定义的张量编码器，利用 KV 缓存的分布特性，将 KV 缓存编码成更紧凑的比特流表示，几乎不增加解码开销，从而节省带宽使用。其次，CacheGen 根据可用带宽的变化调整 KV 缓存不同部分的压缩级别，以保持低上下文加载延迟和高生成质量。我们在流行的 LLMs 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。

关键词： LLM；KV 缓存；网络传输

1 引言

凭借令人印象深刻的生成质量，大型语言模型（LLMs）被广泛应用于个人助理、人工智能医疗保健和市场营销等领域。LLM API 和行业级开源模型的广泛使用，结合流行的应用框架，进一步推动了 LLMs 的流行度。为了执行复杂任务，用户或应用程序通常会在 LLM 输入前附加一个包含数千个或更多标记的长上下文。例如，一些上下文补充了用户提示与领域知识文本，以便 LLM 可以使用未嵌入到 LLM 本身的特定知识来生成响应。另一个例子是，用户提示可以补充在用户与 LLM 交互过程中累积的对话历史。尽管简短的输入很有用 [9]，但较长的输入通常可以提高响应质量和连贯性 [3]，这推动了训练接受越来越长输入的 LLMs 的持续竞争，从 ChatGPT 的 2K 标记到 Claude 的 100K。使用长上下文对响应生成延迟提出了挑战，因为在 LLM 加载并处理整个上下文之前无法生成任何响应。处理长上下文所需的计算量随上下文长度超线性增长 [3]。尽管一些近期的工作提高了处理长上下文的吞吐量 [1]，但处理上下文的延迟对于长上下文（3K 上下文为 2 秒）[5] 仍然可能需要几秒钟。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，从而在上下文再次使用时跳过冗余计算（例如 [5]）。然而，当下一个输入到来时，重用的上下文的 KV 缓存可能并不总是在本地 GPU 内存中；相反，可能需要先从另一台机器上检索 KV 缓存，从而导致额

外的网络延迟。例如，背景文档数据库可能位于一个单独的存储服务中，而协助 LLM 推理的文档（即上下文）只有在收到相关查询时才会被选择并检索到 LLM [12]。因为 KV 缓存由大型高维浮点张量组成，其大小随上下文长度和模型大小增长，很容易达到 10s GB。由此产生的网络延迟可能在 100s 毫秒到超过 10 秒之间，损害了交互式用户体验 [7]。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。我们希望减少 KV 缓存的传输时大小以减少网络延迟。因此，我们不需要保持 KV 缓存的张量格式，而是可以将其编码成更紧凑的比特流。CacheGen 是 LLM 系统中的一个快速上下文加载模块，用于减少检索和处理长上下文的网络延迟。它包括两种技术。KV 缓存编码和解码：CacheGen 将预计算的 KV 缓存编码成更紧凑的比特流表示，而不是保持 KV 缓存的张量形状。这在发送 KV 缓存时大大节省了带宽和延迟。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 KV 张量在相邻标记和不同层的 KV 缓存中对量化损失的不同敏感性。此外，通过基于 GPU 的实现加速了 KV 缓存的解码（解压缩），并且解码与传输并行进行，以进一步减少其对整体推理延迟的影响。KV 缓存流式传输：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。在用户查询到来之前，CacheGen 将长上下文分割成块，并以不同的压缩级别分别对每个块的 KV 进行编码（类似于视频流式传输）。在发送上下文的 KV 缓存时，CacheGen 逐个获取块，并调整每个块的压缩级别，以保持高生成质量，同时将网络延迟保持在服务水平目标（SLO）内。当带宽太低时，CacheGen 还可以回退到以文本格式发送块，并让 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。

2 相关工作

获取 KV 缓存的额外网络延迟尚未受到太多关注。以前的系统假设在共享相同上下文的不同请求之间，上下文的 KV 缓存始终保留在同一 GPU 内存中 [5]，或者 KV 缓存足够小，可以通过快速互连快速发送 [13]。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。有些方法从 KV 缓存或上下文文本中丢弃不重要的标记 [14]，还有一些方法对 KV 缓存张量应用智能量化 [10]。

3 本文方法

3.1 本文方法概述

减少 KV 缓存传输延迟的需求促使我们在 LLM 系统中引入了一个新的模块，我们称之为 KV 缓存流式传输器。KV 缓存流式传输器扮演三个角色：（1）将给定的 KV 缓存编码成更紧凑的比特流表示——KV 比特流。这可以在离线状态下完成。（2）通过不同吞吐量的网络连接流式传输编码后的 KV 比特流。（3）将接收到的 KV 比特流解码回 KV 缓存。乍一看，KV 缓存流式传输器可能与最近压缩长上下文的技术（例如 [14]）看起来相似，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：那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，而 KV 缓存流式传输器的目标是减少 KV 缓存的传输时大小，以减少网络延迟。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 缓存可以在运行时直接被 LLM 使用；同时，它们可以利用生成阶段的信息来了解上下文中的哪些标记对正在处理的特定查询更重要。相比之下，KV 缓存流式传输器不需要保持原始张量形状，可以将它们编码成更紧凑的比特流，并根据网络带宽调整它们的表示。同时，KV 缓存流式传输器必须在特定查询处理之前决定使用哪种压缩方案，因此，不能使用生成阶段的信息。本文介绍了 CacheGen，这是 KV 缓存流式传输器的一个具体设计。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，通过采用 KV 缓存张量的几个分布属性。这大大减少了传输 KV 缓存所需的带宽，从而直接减少了首次标记时间（TTFT）。其次，在动态带宽下流式传输 KV 比特流时，CacheGen 会动态切换不同的编码级别或按需计算 KV 缓存，以保持 TTFT 在给定的截止时间内，同时保持高响应质量。KV 编码/解码带来的计算开销可以忽略不计，并且与网络传输并行进行，以最小化对端到端延迟的影响。

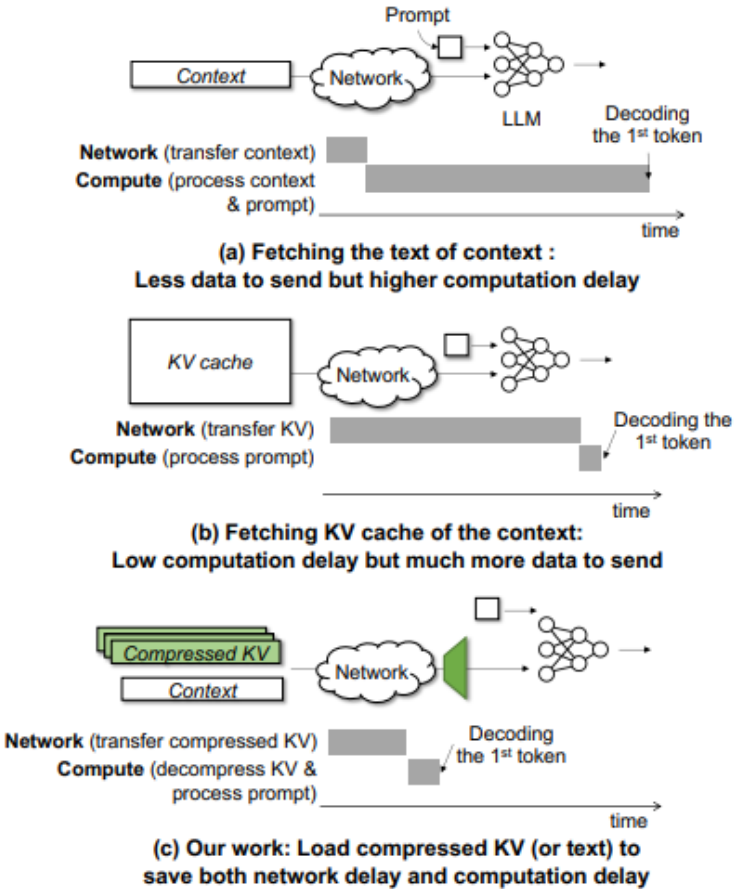

图 1. 方法示意图

3.2 KV 缓存编码器

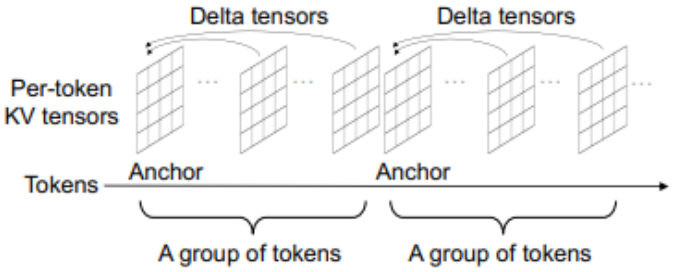

图 2. 在 token 组内，CacheGen 计算锚点令牌的 KV 张量与其余令牌的 KV 张量之间的差分张量

一、计算差分张量

背景：受 token-wise 局部性观察（§5.1.1）的启发，认为 token 之间的差值可能比 KV 张量中的原始值更容易压缩。

操作：首先将上下文分成每组包含十个连续 token 的组。在每个组中，独立地压缩第一个 token（称为锚点 token）的 KV 张量，然后压缩并记录该组中其他每个 token 相对于锚点 token 的差分张量。这类似于视频编码中将帧分成图像组，并在其中进行类似的基于差分的编码，但区别在于不是压缩每对连续 token 之间的差分，而是以同一个锚点 token 为参照来压缩每个 token 的差分张量，这样可以实现压缩和解压缩的并行处理，节省时间。

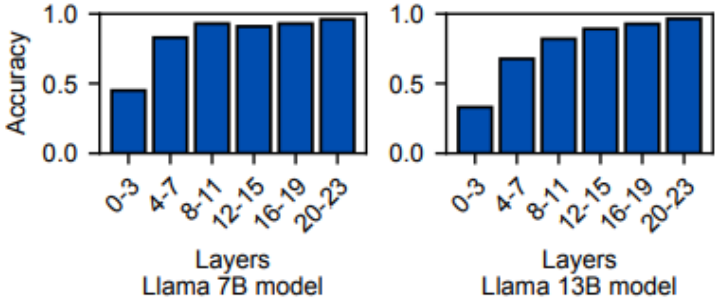

图 3. 对 KV 缓存的不同层应用数据丢失会对准确性产生不同的影响

二、分层量化

背景：为了降低 KV 缓存中元素（浮点数）的精度，以便用更少的位数表示，从而减少存储和传输成本。以往的研究中，元素通常是用相同位数进行均匀量化，没有利用 KV 缓存的独特属性。而根据异质性损失敏感度观察（§5.1.2），对不同层的差分张量采用不同程度的量化。

操作：将 Transformer 层分为三个层组，分别是前 1/3 层、中间 1/3 层和后 1/3 层，并在每个层组的差分张量上分别应用不同大小的量化步长。量化步长从早期层组到后期层组逐渐增大，即后期层组的量化误差更大。采用向量量化方法，这是一种常用于量化模型权重的方法。对于锚点 token 的 KV 缓存，仍使用 8 位量化，这是一种相对较高的精度。因为锚点 token 在整个 token 中占比很小，但其精度会影响该组中剩余 token 所有差分张量的分布，所以需

要为这些锚点 token 保留更高的精度。

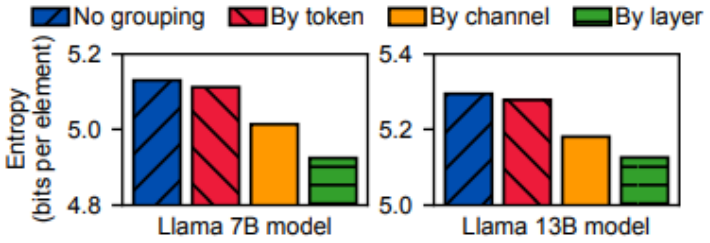

图 4. 使用不同分组策略时的熵（每元素比特数）

三、算术编码

背景：在将 KV 缓存量化为离散符号后，使用算术编码（AC）无损地将上下文的差分张量和锚点张量压缩成比特流。像其他熵编码方案一样，AC 会用更少的位数编码更频繁出现的符号，用更多的位数编码较少出现的符号。为了高效编码，AC 需要准确的、低熵的 KV 缓存元素的概率分布。

操作：根据 KV 值在层、通道和 token 位置上的分布观察，将 KV 值按通道和层进行分组以获得概率分布。具体来说，KV 编码器为 LLM 产生的每个通道-层组合的差分张量以及锚点张量分别离线配置一个概率分布，并且对同一 LLM 产生的所有 KV 缓存都使用相同的分布。CacheGen 使用修改过的 AC 库并结合 CUDA 来加速编码和解码过程。

3.3 KV 流式适配器

由于 KV 缓存的传输可能耗时数百毫秒到几秒，可用带宽可能在传输过程中波动。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 缓存的给定服务水平目标 [4]。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（或简称块），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。这可以在离线状态下完成。当获取上下文时，CacheGen 依次发送这些块，每个块可以选择几种流式传输配置（或简称配置）之一：它可以以其中一个编码级别发送，也可以以文本格式发送，让 LLM 重新计算 K 和 V 张量。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 内。然而，为了高效适配，仍有一些问题需要解决。首先，如何在不同流式传输配置下流式传输多个块而不影响压缩效率？为了离线编码块，CacheGen 首先计算整个上下文的 KV 缓存（即预填充），并将 KV 缓存的 K 和 V 张量沿标记维度分割成子张量，每个子张量包含相同块中的标记的层和通道。然后，它使用 KV 编码器以不同编码（量化）级别编码一个块的 K 或 V 子张量。只要块的长度超过一组标记，每个块就可以独立于其他块进行编码，而不影响压缩效率。这是因为编码一个标记的 KV 张量仅依赖于其自身及其与一组标记的锚点标记的差值。因此，以不同编码级别发送的块可以独立解码，然后连接起来重建 KV 缓存。如果一个块以文本格式发送，LLM 将基于已接收和解码的前一个块的 KV 张量计算其 K 和 V 张量。不同配置下流式传输块会影响生成质量吗？如果一个块由于带宽较低而以比其他块更小尺寸的编码级别发送，那么这个单个块将有较高的压缩损失，但这不会影响其他块的压缩损失。也就是说，我们承认如果带宽太低，无法以高编码级别发送大多数块，质量仍将受到影响。其次，上下文块应该多长？我们

认为块的长度取决于两个考虑因素。
1. 一个块大小的编码 KV 比特流不应太大，否则无法及时响应带宽变化。
2. 块也不应太小，否则如果选择文本格式，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，在实验中经验性地选择了 1.5K 标记作为默认块长度，尽管进一步的优化可能会找到更好的块长度。第三，CacheGen 如何决定下一个块的流式传输配置? CacheGen 通过测量前一个块的吞吐量来估计带宽。它假设这种吞吐量将对剩余块保持不变，并据此计算每种流式传输配置的预期延迟。预期延迟是通过将其尺寸除以吞吐量来计算的。如果有带宽波动，CacheGen 的反应最多会延迟一个块。由于一个块是整个 KV 缓存的一个小子集，这种反应足够快以满足 SLO。然后，它选择压缩损失最小（即文本格式或最低编码级别）且预期延迟仍在 SLO 内的配置，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验知识，CacheGen 将以相同的方式使用它来选择第一个块的配置。否则，CacheGen 从默认的中等编码级别开始。最后，CacheGen 是如何处理多个请求的流式传输的呢？当多个请求在 T 秒内同时到达时，CacheGen 会将它们批量并流式传输在一起。它可以批量处理多达 B 个请求，这是 GPU 服务器能够同时处理的最大数量。每个请求被分成相同大小的块，尽管不同请求的总块数可能不同。对于每个块索引 c，CacheGen 确定包含块 c 的请求数 N。利用前一个块 c−1 测量到的吞吐量，CacheGen 通过将 N 乘以单个请求的延迟来计算每种配置的预期延迟。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。

4 复现细节

4.1 与已有开源代码对比

本次复现工作主体部分采用原论文代码，创新点在于对于 KV 流适配器进行了改进，原工作中 CacheGen 通过测量前一个块的吞吐量来估计带宽。它假设这种吞吐量将对剩余块保持不变，并据此计算每种流式传输配置的预期延迟。这种方法并不能很好的对实际网络带宽进行预测，也不能应对突发的网络环境恶化，为此，我们借鉴了视频流领域的方法 [6]，我们使用两条准则来构建一个更健壮的带宽估计器。首先，我们不使用瞬时吞吐量，而是使用在过去几个块上计算的平滑值。在我们当前的实验中，我们使用最后 20 个样本。其次，我们希望这种平滑对异常值具有鲁棒性。例如，如果一个块的吞吐量非常高或非常低，使用算术平均值会受到异常值的偏见。为此，我们使用最后 20 个样本的调和平均值。采用这种方法的原因有两个。首先，当我们要计算速率的平均值时，调和平均值更为合适，这正是吞吐量估计的情况。其次，它对较大的异常值也更具鲁棒性。

4.2 实验环境搭建

CacheGen 使用约 2K 行的 Python 代码和约 1K 行的 CUDA 内核代码实现，基于 PyTorch v2.0 和 CUDA 12.0。集成到大语言模型推理框架中：CacheGen 通过两个接口操作大语言模型：$calculate\_kv(context) \rightarrow KVCache$: 给定一段上下文，CacheGen 通过此函数调用大语言模型以获取相应的 KV 缓存。$generate\_with\_kv(KVCache) \rightarrow text\ CacheGen$: 将 KV 缓存传递给大语言模型，并让其在跳过上下文预填充的情况下生成标记。使用了 transformers 库

在 HuggingFace 模型中实现了这两个接口，约 500 行 Python 代码。这两个接口都是基于库提供的 generate 函数实现的。对于 $calculate\_kv$，大语言模型仅计算 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 $return\_dict\_in\_generate = True$ 选项。$generate\_with\_kv$ 的实现方式是，在调用 generate 函数时，简单地通过 $past\_key\_values$ 参数传递 KV 缓存。将 CacheGen 集成到了 LangChain，一个流行的大语言模型应用框架中。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。如果存在，CacheGen 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。CacheGen 中的 KV 缓存管理：为了管理 KV 缓存，CacheGen 实现了两个模块：$store\_kv(LLM) \rightarrow chunk\_id : encoded\_KV$: 调用 $caculate\_kv$，将返回的 KV 缓存分割成上下文块，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，其中将 $chunk\_id$ 映射到对应块的 K 和 V 张量的编码比特流。$get\_kv(chunk\_id) \rightarrow encoded\_KV$: 在存储服务器上获取对应 $chunk\_id$ 的编码 KV 张量，并将其传输到推理服务器。每当有新的上下文片段输入时，CacheGen 首先调用 $store\_kv$，它首先生成 KV 缓存，然后在存储服务器上存储编码的比特流。在运行时，CacheGen 调用 get_kv 获取相应的 KV 缓存块并输入到 $generate\_with\_kv$ 中。

5 实验结果分析

模型：在四种不同大小的模型上评估 CacheGen，具体为 Mistral-7B、Llama-34B 和 Llama-70B 的微调版本。所有模型都进行了微调，以便它们能够处理长上下文（长达 32K）。由于没有公开的长上下文微调版本，没有在其他大语言模型（例如 OPT、BLOOM）上测试 CacheGen。
数据集：在来自四个不同数据集的 662 个上下文中评估 CacheGen，这些数据集有不同的任务：LongChat：该任务最近发布 [8]，通过使用所有之前的对话作为上下文，测试大语言模型对诸如 “我们首先讨论的主题是什么?” 这类问题的回答能力。大多数上下文包含大约 9-9.6K 个标记。TriviaQA：该任务测试大语言模型的阅读理解能力 [2]，通过给大语言模型提供一个单独的文档（上下文），并让其根据文档回答问题。该数据集是 LongBench 基准测试套件 [2] 的一部分。NarrativeQA：该任务让大语言模型根据提供的故事或剧本（作为单个文档的上下文）回答问题。该数据集也是 LongBench 的一部分。Wikitext：该任务是根据由特定维基页面相关文档组成的上下文，预测序列中下一个标记的概率 [11]。

质量指标：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 LongChat 数据集上模型的输出。该任务预测用户与大语言模型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 TriviaQA 和 NarrativeQA 数据集上模型的回答。它衡量生成的答案与问答任务的真实答案相匹配的概率。
困惑度用于评估模型在 Wikitext 数据集上的表现。困惑度定义为下一个标记的平均负对数似然的指数。低困惑度意味着模型很可能正确生成下一个标记。尽管困惑度并不等同于文本生成质量，但它被广泛用作代理，以测试修剪或量化大语言模型对生成性能的影响。

基线：将 CacheGen 与不改变上下文或模型的基线进行比较。“默认量化” 使用键值缓存的均匀量化，具体为大语言模型中每一层使用相同的量化级别。“文本上下文” 获取上下

文的文本并将其输入大语言模型以生成其键值缓存。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 运行实验。vLLM 的实现已经使用了 xFormers，其中包括速度和内存优化的 Transformer CUDA 内核，并已显示出比 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），要么在键值缓存中丢弃标记（H2O）。

硬件设置：使用配备四个 GPU 的 NVIDIA A40 GPU 服务器来基准测试我们的结果。服务器配备了 384GB 的内存和两个启用了超线程和 Turbo Boost 的 Intel(R) Xeon(R) Gold 6130 CPU。

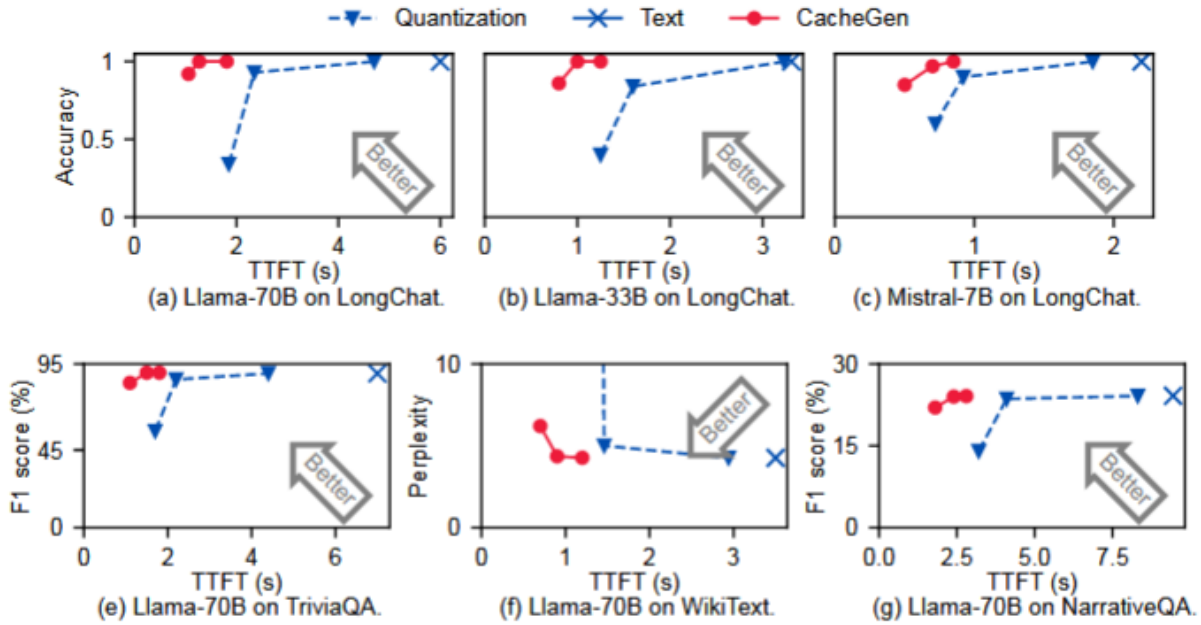

图 5. 原文中，在不同的模型和不同的数据集上，CacheGen 减少了首次训练时间（TTFT），对质量（在准确性、困惑度或 F1 分数方面）几乎没有负面影响。

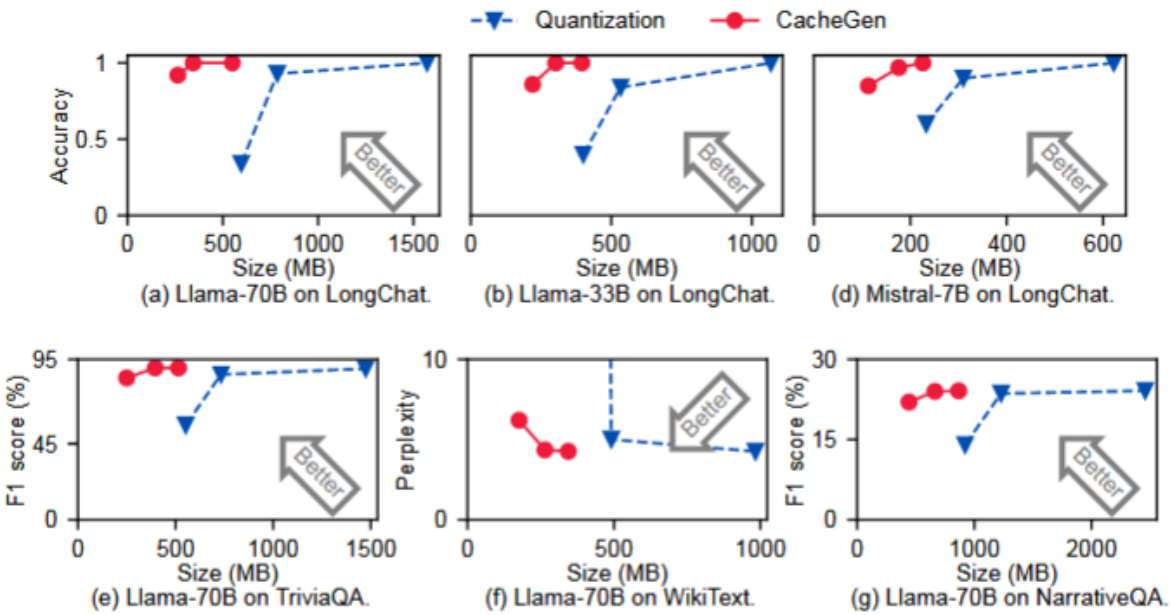

图 6. 原文中，在不同的模型中，CacheGen 在不同数据集上减少了 KV 缓存的大小，同时对准确性的影响很小。。

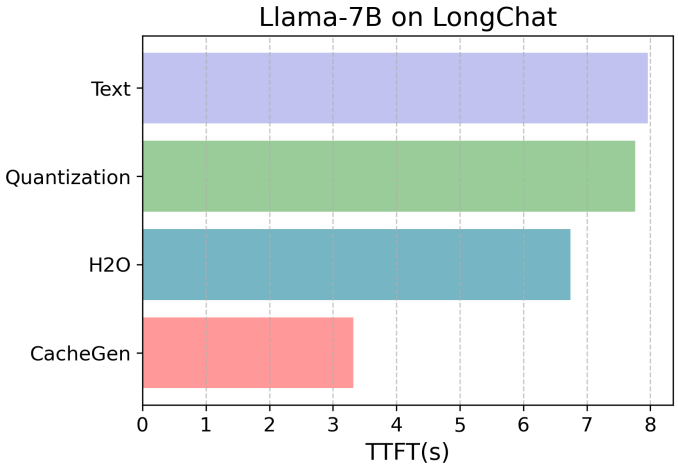

图 7. 在复现实验中，在不同的模型和不同的数据集上，改进的 CacheGen 减少了首次训练时间（TTFT），达到了接近原文的实验结果

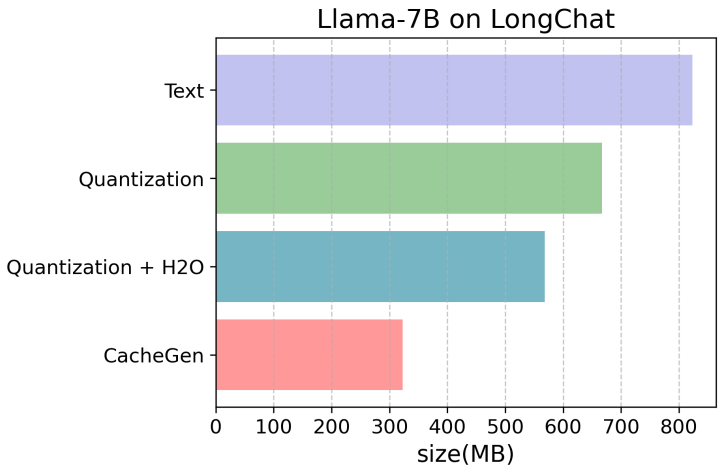

图 8. 在复现实验中，在不同的模型中，改进的 CacheGen 在不同数据集上减少了 KV 缓存的大小，达到了接近原文的实验结果。

6 总结与展望

CacheGen 是一个上下文加载模块，旨在最小化为大语言模型（LLMs）获取和处理上下文的总体延迟。CacheGen 通过一个专门的编码器减少传输长上下文的 KV 缓存所需的带宽，该编码器可将 KV 缓存压缩成紧凑的比特流。在三种不同容量的模型和三种不同上下文长度的数据集上的实验表明，CacheGen 可以在保持高任务性能的同时减少总体延迟。在复现 CacheGen 时，我们改进对预测网络带宽的模型，以应对更复杂的网络环境，实现了接近原文的结果。未来可能的工作是当大模型被部署在边缘服务器时，KV cache 的替换策略。

参考文献

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

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

[3] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020.

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

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

[6] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. *IEEE/ACM Transactions on Networking*, 22(1):326–340, 2014.

[7] Zijian Lew, Joseph B. Walther, Augustine Pang, and Wonsun Shin. Interactivity in online chat: Conversational contingency and response latency in computer-mediated communication. *Journal of Computer-Mediated Communication*, 23(4):201–221, 06 2018.

[8] Dacheng Li, Rulin Shao, Anze Xie, Lianmin Zheng, Ying Sheng, Joseph E. Gonzalez, Ion Stoica, Xuezhe Ma, and Hao Zhang. How long can open-source llms truly promise on context length?, June 2023. arXiv preprint.

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

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

[11] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016. arXiv preprint arXiv:1609.07843.

[12] Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior, 2023.

[13] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting, 2023.

[14] 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. H2o: Heavy-hitter oracle for efficient generative inference of large language models, 2023.