PROMPT CACHE: MODULAR ATTENTION REUSE FOR LOW-LATENCY INFERENCE

In Gim [1] **Guojun Chen** [1] **Seung-seob Lee** [1] **Nikhil Sarda** [2] **Anurag Khandelwal** [1] **Lin Zhong** [1]

ABSTRACT

We present *Prompt Cache*, an approach for accelerating inference for large language models (LLM) by reusing attention states across different LLM prompts. Many input prompts have overlapping text segments, such as system messages, prompt templates, and documents provided for context. Our key insight is that by precomputing and storing the attention states of these frequently occurring text segments on the inference server, we can efficiently reuse them when these segments appear in user prompts. Prompt Cache employs a *schema* to explicitly define such reusable text segments, called prompt modules. The schema ensures positional accuracy during attention state reuse and provides users with an interface to access cached states in their prompt. Using a prototype implementation, we evaluate Prompt Cache across several LLMs. We show that Prompt Cache significantly reduce latency in time-to-first-token, especially for longer prompts such as document-based question answering and recommendations. The improvements range from **8**× for GPU-based inference to **60**× for CPU-based inference, all while maintaining output accuracy and without the need for model parameter modifications.

1 INTRODUCTION

A substantial fraction of large language model (LLM) prompts are reused frequently. For example, prompts usually commence with identical "system messages" that provide initial guidelines for its functionality. Documents can also overlap in multiple prompts. In a wide range of long-context LLM applications, such as legal analysis (Cui et al., 2023; Nay et al., 2023), healthcare applications (Steinberg et al., 2021; Rasmy et al., 2021), and education (Shen et al., 2021), the prompt includes one or several documents from a pool. Additionally, prompts are often formatted with reusable templates (White et al., 2023) as a result of prompt engineering. Such examples are common in LLM for robotics and tool learning (Huang et al., 2022; Driess et al., 2023; Qin et al., 2023). This further results in a high degree of overlap between prompts using the same template.

We introduce a novel technique termed *Prompt Cache* to reduce the computational overhead in generative LLM inference. Prompt Cache is motivated by the observation that input prompts to LLM often has reusable structures. The key idea is to precompute attention states of the frequently revisited prompt segments in memory, and reuse them when these segments appear in the prompt to reduce latency.

Reusing attention states is a popular strategy for accelerating the service of a single prompt (Pope et al., 2022). The existing approach, often referred to as *Key-Value (KV) Cache*, reuses the key-value attention states of input tokens during the autoregressive token generation. This eliminates the need to compute full attention for every token generation (§ 2.2). By caching the key-value attention computed for the previously generated token, each token generation requires the computation of key-value attention states only once.

Building on top of KV Cache, Prompt Cache extends attention state reuse from a single prompt to multiple prompts by making attention state reuse *modular*. In our approach, frequently reused text segments are individually precomputed and stored in memory. When such "cached" segments appear in the input prompt, the system uses the precomputed key-value attention states from memory instead of recomputing them. As a result, attention computations are only required for uncached text segments. Figure 1 illustrates the difference between full autoregressive generation, KV Cache, and Prompt Cache. We note that the performance advantage becomes more pronounced as the size of cached segments grows since the computation overhead of attention states scales *quadratically* with input sequence size (Keles et al., 2022; Tay et al., 2023) while the space and compute complexity of Prompt Cache scales *linearly* with the size.

Two challenges arise when reusing attention states across

[1]Department of Computer Science, Yale University, USA. {in.gim, guojun.chen, seung-seob.lee, anurag.khandelwal, lin.zhong}@yale.edu [2]Google, Mountain View, California, USA. nikhilsarda@google.com. Correspondence to: Lin Zhong <lin.zhong@yale.edu>.

Proceedings of the 5th MLSys Conference, Santa Clara, CA, USA, 2024.

提示缓存：低延迟推理的模块化注意力重用

在Gim [1] Guojun Chen　李承燮 [1] 尼基尔·萨尔达 [2] 阿努拉格·坎德尔瓦尔 [1]　林中 [1]

摘要

我们提出了 *Prompt Cache*，一种通过在不同的 LLM 提示之间重用注意力状态来加速大型语言模型（LLM）推理的方法。许多输入提示具有重叠的文本片段，例如系统消息、提示模板和提供上下文的文档。我们的关键见解是，通过在推理服务器上预计算并存储这些频繁出现的文本片段的注意力状态，当这些片段出现在用户提示中时，我们可以高效地重用它们。Prompt Cache 使用 *schema* 明确定义这些可重用的文本片段，称为提示模块。该架构确保在重用注意力状态时的位置信息准确性，并为用户提供访问其提示中缓存状态的接口。通过原型实现，我们评估了多个 LLM 上的 Prompt Cache。我们展示了 Prompt Cache 显著减少了首次令牌的延迟，特别是对于基于文档的问题回答和推荐等较长提示。改进范围从 GPU 基础推理的 8× 到 CPU 基础推理的 60×，同时保持输出准确性且无需修改模型参数。

1 引言

大型语言模型（LLM）提示的一个重要部分经常被重复使用。例如，提示通常以相同的“系统消息”开始，这些消息提供了其功能的初步指导。文档在多个提示中也可能重叠。在广泛的长上下文 LLM 应用中，例如法律分析（Cui et al., 2023；Nay et al., 2023）、医疗应用（Steinberg et al., 2021；Rasmy et al., 2021）和教育（Shen et al., 2021），提示包括来自一个池中的一个或多个文档。此外，由于提示工程，提示通常使用可重用的模板格式（White et al., 2023）。这样的例子在用于机器人和工具学习的 LLM 中很常见（Huang et al., 2022；Driess et al., 2023；Qin et al., 2023）。这进一步导致使用相同模板的提示之间存在高度重叠。

我们引入了一种新颖的技术，称为 *Prompt Cache*，以减少生成性 LLM 推理中的计算开销。Prompt Cache 的动机是观察到输入提示到 LLM 通常具有可重用的结构。关键思想是预计算频繁的注意力状态。

[1]Department of Computer Science, Yale University, USA. {in.gim, guojun.chen, seung-seob.lee, anurag.khandelwal, lin.zhong}@yale.edu [2]Google, Mountain View, California, USA. nikhilsarda@google.com. Correspondence to: Lin Zhong <lin.zhong@yale.edu>.

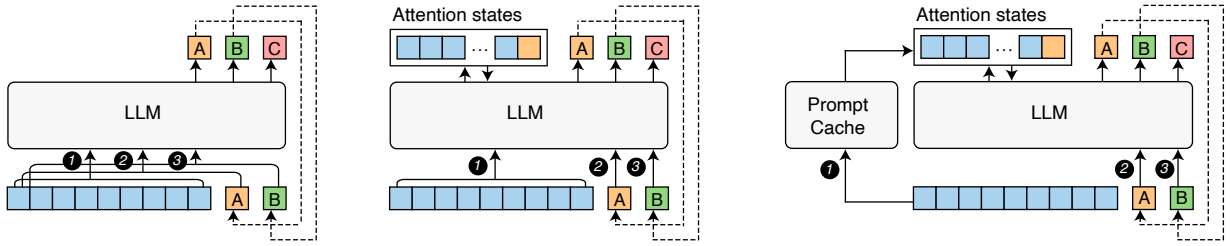
Proceedings of the 5^{th} MLSys Conference, Santa Clara, CA, USA, 2024.

在内存中重新访问提示段，并在这些段出现在提示中时重用它们，以减少延迟。

重用注意力状态是一种加速单个提示服务的流行策略（Pope 等，2022）。现有的方法通常被称为 *Key-Value (KV) Cache*，在自回归令牌生成过程中重用输入令牌的键值注意力状态。这消除了对每个令牌生成计算完整注意力的需要（§ 2.2）。通过缓存为先前生成的令牌计算的键值注意力，每次令牌生成只需计算一次键值注意力状态。

基于KV缓存，Prompt Cache通过使注意力状态重用 *modular*从单个提示扩展到多个提示。我们的方法是，频繁重用的文本片段被单独预计算并存储在内存中。当这些“缓存”的片段出现在输入提示中时，系统使用内存中预计算的键值注意力状态，而不是重新计算它们。因此，仅对未缓存的文本片段需要进行注意力计算。图1说明了完全自回归生成、KV缓存和Prompt Cache之间的区别。我们注意到，随着缓存片段大小的增加，性能优势变得更加明显，因为注意力状态的计算开销随着输入序列大小而增加*quadratically*（Keles等，2022；Tay等，2023），而Prompt Cache的空间和计算复杂度随着大小而增加*linearly*。

在跨越时重用注意力状态时会出现两个挑战



(a) Autoregressive token generation (b) Generation with KV Cache (c) Generation with Prompt Cache

Figure 1. Comparison of LLM token generation methods, each showing three steps (❶ to ❸). Each box indicates a token. Blue boxes represent the prompt. (a) An LLM takes in a prompt (blue tokens) and predicts the next token (A) (❶). It then appends the generated token (A) to the prompt to predict the next token (B) (❷). This process, called autoregressive, continues until a stop condition is met. (b) KV Cache computes time attention states for the prompt only once (❶) and reuses them in the following steps; (c) Prompt Cache reuses the KV state across services to bypass prompt attention computation. Prompt Cache populates its cache when a schema is loaded and reuses the cached states for prompts that are derived from the schema (❶). Figure 2 further elaborates Step ❶.

prompts. First, attention states are position-dependent due to the positional encoding in Transformers. Thus, the attention states of a text segment can only be reused if the segment appears at the same position. Second, the system must be able to efficiently recognize a text segment whose attention states may have been cached in order to reuse.

To tackle these two problems, Prompt Cache combines two ideas. The first is to make the structure of a prompt explicit with a *Prompt Markup Language* (PML). PML makes reusable text segments explicit as modules, *i.e.*, *prompt module*. It not only solves the second problem above but opens the door for solving the first, since each prompt module can be assigned with unique position IDs. Our second idea is our empirical finding that LLMs can operate on attention states with discontinuous position IDs. This means that we can extract different segment of attention states and concatenate them to formulate subset of meanings. We leverage this to enable users to select prompt modules based on their needs, or even update some prompt modules during the runtime.

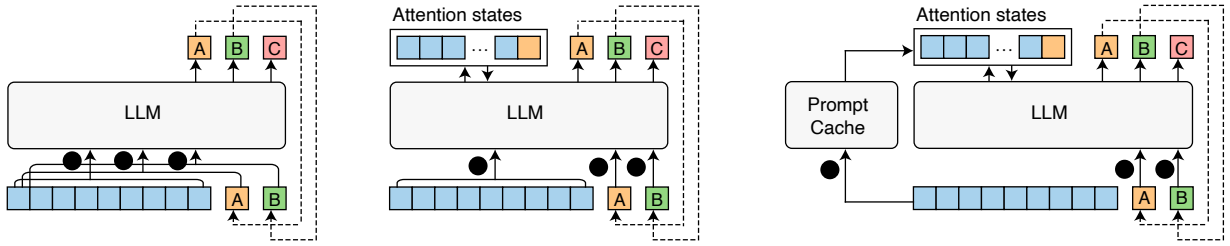
We explain how Prompt Cache works in §3. In summary, an LLM user writes their prompts in PML, with the intention that they may reuse the attention states based on prompt modules. Importantly, they must derive a prompt from a *schema*, which is also written in PML. Figure 2 shows a example prompt based on an example schema. When Prompt Cache receives a prompt, it first processes its schema and computes the attention states for its prompt modules. It reuses these states for the prompt modules in the prompt and other prompts derived from the same schema. In §4, we report a prototype implementation of Prompt Cache on top of the HuggingFace transformers library (Wolf et al., 2020). While Prompt Cache can work with any Transformer architecture compatible with KV Cache, we experiment with three popular Transformer architectures powering the following open-sourced LLMs: Llama2 (Touvron et al., 2023), Falcon (Penedo et al., 2023), and MPT (MosaicML, 2023). We consider two types of memory for storing prompt modules: CPU and GPU memory. While CPU memory can scale to terabyte levels, it brings the overhead of host-to-device memory copying. In contrast, GPU memory does not require coping but has limited capacity.

Using the prototype, we conduct an extensive benchmark evaluation to examine the performance and quantify the accuracy of Prompt Cache across various long-context datasets (§5). We employ the LongBench suite (Bai et al., 2023), which includes recommendation and question-answering (QA) tasks based on multiple documents. In our evaluation, Prompt Cache reduces time-to-first-token (TTFT) latency from $1.5\times$ to $10\times$ for GPU inference with prompt modules on GPU memory and from $20\times$ to $70\times$ for CPU inference, all without any significant accuracy loss. Additionally, we analyze the memory overhead of the precomputed attention states for each model and discuss directions for optimizing the memory footprint of Prompt Cache. We subsequently showcase several generative tasks, including personalization, code generation, and parameterized prompts, to demonstrate the expressiveness of the prompt schema and performance improvement with negligible quality degradation.

In our present study, we mainly focus on techniques for modular attention reuse. However, we foresee Prompt Cache being utilized as a foundational component for future LLM serving systems. Such systems could incorporate enhanced prompt module management and GPU cache replacement strategies, optimizing the advantages of both host DRAM and GPU HBM. Our source code and data used for evaluation are available at github.com/yale-sys/prompt-cache.

2 Background and Related Work

Prompt Cache builds on the ideas of the KV Cache, *i.e.*, key-value attention state reuse during autoregressive decoding in LLMs. This section reviews autoregressive token generation



(a) Autoregressive token generation (b) Generation with KV Cache (c) Generation with Prompt Cache

图 1. LLM 令牌生成方法的比较，每种方法显示三个步骤 (*1* 到 *3*)。每个框表示一个令牌。蓝色框表示提示。(a) LLM 接收一个提示（蓝色令牌）并预测下一个令牌 (*A*) (*1*)。然后，它将生成的令牌 (*A*) 附加到提示中，以预测下一个令牌 (*B*) (*2*)。这个过程称为自回归，持续进行直到满足停止条件。(b) KV 缓存仅计算一次提示的时间注意状态 (*1*)，并在后续步骤中重用它们；(c) 提示缓存跨服务重用 KV 状态，以绕过提示注意计算。当加载模式时，提示缓存填充其缓存，并重用从模式派生的提示的缓存状态 (*1*)。图 2 进一步阐述了步骤 *1* 。

提示。首先，由于变换器中的位置编码，注意状态是位置依赖的。因此，文本段的注意状态只能在该段出现在相同位置时被重用。其次，系统必须能够有效识别可能已被缓存的注意状态的文本段，以便进行重用。

为了解决这两个问题，Prompt Cache 结合了两个想法。第一个是通过 *Prompt Markup Language* (PML) 明确提示的结构。PML 将可重用的文本片段明确为模块，*i.e.*、*prompt mod-*、*ule*。它不仅解决了上述第二个问题，还为解决第一个问题打开了大门，因为每个提示模块都可以分配唯一的位置 ID。我们的第二个想法是我们的经验发现 LLM 可以在具有不连续位置 ID 的注意状态上操作。这意味着我们可以提取不同的注意状态片段并将它们连接起来，以形成意义的子集。我们利用这一点使用户能够根据他们的需求选择提示模块，甚至在运行时更新某些提示模块。

我们在§3中解释了Prompt Cache的工作原理。总之，LLM用户使用PML编写他们的提示，目的是为了基于提示模块重用注意力状态。重要的是，他们必须从*schema*中导出一个提示，该提示也用PML编写。图2展示了一个基于示例模式的示例提示。当Prompt Cache接收到一个提示时，它首先处理其模式并计算其提示模块的注意力状态。它重用这些状态用于提示中的提示模块以及从相同模式导出的其他提示。在§4中，我们报告了基于HuggingFace transformers库（Wolf等，2020）的Prompt Cache原型实现。虽然Prompt Cache可以与任何与KV Cache兼容的Transformer架构一起工作，但我们实验了三种流行的Transformer架构，支持以下开源LLM：Llama2（Touvron等，2023）、Falcon（Penedo等，2023）和MPT（MosaicML，2023）。我们考虑两种类型的内存来存储提示模块：CPU 内存和 GPU 内存。虽然 CPU 内存可以扩展到 TB 级别，但它带来了主机到设备内存复制的开销。相比之下，GPU 内存不需要复制，但容量有限。

使用原型，我们进行广泛的基准评估，以检查 Prompt Cache 在各种长上下文数据集上的性能并量化其准确性 (§5)。我们采用 LongBench 套件（Bai 等，2023），该套件包括基于多个文档的推荐和问答（QA）任务。在我们的评估中，Prompt Cache 将 GPU 推理中使用提示模块的首次令牌时间（TTFT）延迟从 1.5× 降低到 10×，将 CPU 推理中的延迟从 20× 降低到 70×，所有这些都没有显著的准确性损失。此外，我们分析了每个模型的预计算注意状态的内存开销，并讨论了优化 Prompt Cache 内存占用的方向。随后，我们展示了几个生成任务，包括个性化、代码生成和参数化提示，以展示提示模式的表现力和在几乎没有质量下降的情况下的性能提升。

在我们目前的研究中，我们主要关注模块化注意力重用的技术。然而，我们预见到提示缓存将作为未来LLM服务系统的基础组件。这样的系统可以结合增强的提示模块管理和GPU缓存替换策略，优化主机DRAM和GPU HBM的优势。我们用于评估的源代码和数据可在github.com/yale-sys/prompt-cache获取。

2 背景与相关工作

提示缓存基于KV缓存的理念，*i.e.*，在LLMs中自回归解码期间重用键值注意力状态。本节回顾自回归令牌生成。

in LLMs, explains how the incorporation of KV Cache can speed up the token generation process, identifies its approximations, and surveys recent work that leverages the KV Cache for acceleration. We also briefly discuss other existing techniques for accelerating LLM inference.

2.1 Autoregressive Token Generation

An LLM generates output tokens autoregressively (Radford et al., 2018). It starts with an initial input, often called a prompt, and generates the next token based on the prompt. The model then appends the token to the prompt and uses it to generate the next token. The generation process continues until a stopping condition is met. This could be after a predetermined number of tokens, upon generating a special end-of-sequence token, or when the generated sequence reaches a satisfactory level of coherence or completeness. Importantly, in each step, the model takes the entire prompt and tokens generated so far as the input, and repeat.

2.2 Key-Value Cache

Autoregressive token generation described above incurs substantial computation due to the self-attention mechanism being applied over the entirety of input during each step. To ameliorate this, the Key-Value (KV) Cache mechanism (Pope et al., 2022) is frequently used. This technique computes the key and value embeddings for each token only once throughout the autoregressive token generation. To elaborate, denote a user prompt as a sequence of n tokens: $s_1, \ldots, s_n$, and the subsequently generated k tokens as $s_{n+1}, \ldots, s_{n+k}$. In naive autoregressive token generation, the attention states $\{(k_1, v_1), \ldots, (k_{n+k}, v_{n+k})\}$ are fully recalculated at every step. In contrast, KV Cache initially computes attention states for the input, represented by $S_0 = \{(k_i, v_i) | i \leq n\}$, and caches them in memory. This step is often referred to as the *prefill* phase. For every subsequent step $j \leq k$, the model reuses the cached values $S_j = \{(k_i, v_i) | i < n + j\}$ to compute the attention state (k_{n+j}, v_{n+j}) of the new token s_{n+j}. This approach significantly reduces the computation required for self-attention. Specifically, the computation in each step, measured in FLOPs for matrix operations, is reduced by a factor of $1/n$. The number of operations decreases from approximately $6nd^2 + 4n^2d$ to $6d^2 + 4nd$, where d is a hidden dimension size. After each step, the newly computed (k_{n+j}, v_{n+j}) attention states are appended to the cache for subsequent use. In causal language models, which account for most LLMs, the use of KV Cache does not affect the model's accuracy, since the attention at position i is computed based solely on the tokens at positions located before i-th token.

The KV Cache has catalyzed further exploration into LLM acceleration. Ensuing studies have either centered on refining memory management for KV Cache, as demonstrated in *paged attention* (Kwon et al., 2023), on pruning superfluous KV Cache data (Zhang et al., 2023), or compressing it (Liu et al., 2023b). There are some preliminary works that explore KV Cache reuse across different requests as well. (Feng et al., 2023) reuse memorized attention states based on an embedding similarity metric. Paged attention also demonstrates simple prefix sharing, where different prompts with an identical prefix share KV Cache. However, existing approaches are specific to certain scenarios, while we investigate attention reuse for *general* LLM prompts.

2.3 Other Methods for Low-Latency LLM Inference

Prompt Cache introduces an orthogonal optimization strategy that augments existing systems dedicated to efficient LLM inference. This includes systems that utilize multiple GPUs for inference (Aminabadi et al., 2022) and those with high-performance GPU kernels for softmax attention score computation (Dao et al., 2022). Although our current focus is on achieving low-latency inference in LLMs, Prompt Cache can also benefit systems aiming for high throughput (Sheng et al., 2023) as well via reduced computation.

3 Design of Prompt Cache

The effectiveness of the KV Cache leads us to the next question: *Can attention states be reused across multiple inference requests?* We observe that different prompts often have overlapping text segments. For example, identical "system messages", or metaprompts are frequently inserted at the beginning of a prompt to elicit desired responses from an LLM. For another example, in many legal and medical applications of LLMs (Cui et al., 2023; Steinberg et al., 2021; Rasmy et al., 2021), the same set of documents is often provided as context to different prompts. Finally, reusable prompt formats, *i.e.*, *prompt templates*, are commonly used by LLM applications in robotics and tool learning (Driess et al., 2023; Qin et al., 2023), since most tasks are variations of a few common task. In this section, we describe our approach called *Prompt Cache*, which answers the above question affirmatively. Prompt Cache improves computational efficiency through *inter-request* attention state reuse by leveraging the shared segments in a structured manner.

3.1 Overview

The attention states of a text segment can only be reused if the segment appears at the same position in the LLM input. This is because transformer architectures integrate unique positional embeddings into the (k, v) attention states. This is not a problem for serving a single prompt using KV Cache, because the same prompt text is located at the same position, *i.e.*, the beginning of the input, in all steps.

Shared text segments, on the other hand, can appear in differ-

在LLMs中，解释了KV缓存的引入如何加速令牌生成过程，识别其近似值，并调查了最近利用KV缓存进行加速的工作。我们还简要讨论了其他现有的加速LLM推理的技术。

2.1 自回归令牌生成

一个大型语言模型（LLM）以自回归的方式生成输出标记（Radford et al., 2018）。它从一个初始输入开始，通常称为提示，并根据提示生成下一个标记。然后，模型将该标记附加到提示中，并使用它生成下一个标记。生成过程持续进行，直到满足停止条件。这可能是在预定数量的标记之后，生成一个特殊的序列结束标记，或者当生成的序列达到令人满意的连贯性或完整性水平时。重要的是，在每一步中，模型将整个提示和迄今为止生成的标记作为输入，并重复这一过程。

2.2 键值缓存

自回归令牌生成所描述的过程由于在每一步中对整个输入应用自注意力机制而产生了大量计算。为了改善这一点，通常使用键值（KV）缓存机制（Pope et al., 2022）。该技术在自回归令牌生成过程中仅对每个令牌计算一次键和值嵌入。具体来说，设用户提示为一系列令牌 n：$s_1, \ldots, s_n$，随后生成的 k 令牌为 $s_{n+1}, \ldots, s_{n+k}$。在简单的自回归令牌生成中，注意力状态 $\{(k_1, v_1), \ldots, (k_{n+k}, v_{n+k})\}$ 在每一步都被完全重新计算。相比之下，KV 缓存最初为输入计算注意力状态，表示为 $S_0 = \{(k_i, v_i) | i \leq n\}$，并将其缓存到内存中。这一步通常被称为 *prefill* 阶段。在每个后续步骤 $j \leq k$ 中，模型重用缓存的值 $S_j = \{(k_i, v_i) | i < n + j\}$ 来计算新令牌 s_{n+j} 的注意力状态 (k_{n+j}, v_{n+j})。这种方法显著减少了自注意力所需的计算。具体而言，每一步的计算量（以矩阵操作的 FLOPs 计）减少了 $1/n$ 倍。操作数量从大约 $6nd^2 + 4n^2d$ 减少到 $6d^2 + 4nd$，其中 d 是隐藏维度大小。在每一步之后，新计算的 (k_{n+j}, v_{n+j}) 注意力状态被附加到缓存中以供后续使用。在因果语言模型中，这些模型占大多数 LLM，使用 KV 缓存不会影响模型的准确性，因为在 i 位置的注意力仅基于位于 i 之前位置的令牌计算。

KV缓存催化了对LLM加速的进一步探索。随后的研究要么集中在优化KV缓存的内存管理上，如所示在 *paged attention* (Kwon 等人，2023) 中，关于修剪多余的 KV 缓存数据（Zhang 等人，2023）或对其进行压缩（Liu 等人，2023b）。还有一些初步的工作探索了不同请求之间的 KV 缓存重用。（Feng 等人，2023）基于嵌入相似性度量重用记忆的注意力状态。分页注意力还展示了简单的前缀共享，其中具有相同前缀的不同提示共享 KV 缓存。然而，现有的方法特定于某些场景，而我们研究的是 *general* LLM 提示的注意力重用。

2.3 低延迟 LLM 推理的其他方法

Prompt Cache 引入了一种正交优化策略，增强了现有专注于高效 LLM 推理的系统。这包括利用多个 GPU 进行推理的系统（Aminabadi 等，2022）以及那些具有高性能 GPU 内核用于 softmax 注意力分数计算的系统（Dao 等，2022）。尽管我们当前的重点是实现 LLM 的低延迟推理，Prompt Cache 也可以通过减少计算来使旨在实现高吞吐量的系统（Sheng 等，2023）受益。

3 提示缓存的设计

KV缓存的有效性引导我们进入下一个问题：*Can attention states be reused across multiple inference requests?* 我们观察到，不同的提示通常具有重叠的文本片段。例如，相同的“系统消息”或元提示经常在提示的开头插入，以引导大型语言模型（LLM）产生期望的响应。另一个例子是在许多法律和医疗应用中（Cui et al., 2023; Steinberg et al., 2021; Rasmy et al., 2021），同一组文档通常作为上下文提供给不同的提示。最后，可重用的提示格式，*i.e.*, *prompt templates*，在机器人和工具学习的LLM应用中被广泛使用（Driess et al., 2023; Qin et al., 2023），因为大多数任务是少数常见任务的变体。在本节中，我们描述了我们称为*Prompt Cache*的方法，它肯定地回答了上述问题。提示缓存通过利用结构化方式中的共享片段，通过*inter-request*注意状态重用来提高计算效率。

3.1 概述

文本片段的注意力状态只有在该片段出现在LLM输入的相同位置时才能被重用。这是因为变换器架构将独特的位置嵌入集成到(k, v)注意力状态中。对于使用KV缓存服务单个提示，这不是问题，因为相同的提示文本在所有步骤中位于相同的位置*i.e.*，即输入的开头。

共享文本段落则可以出现在不同的

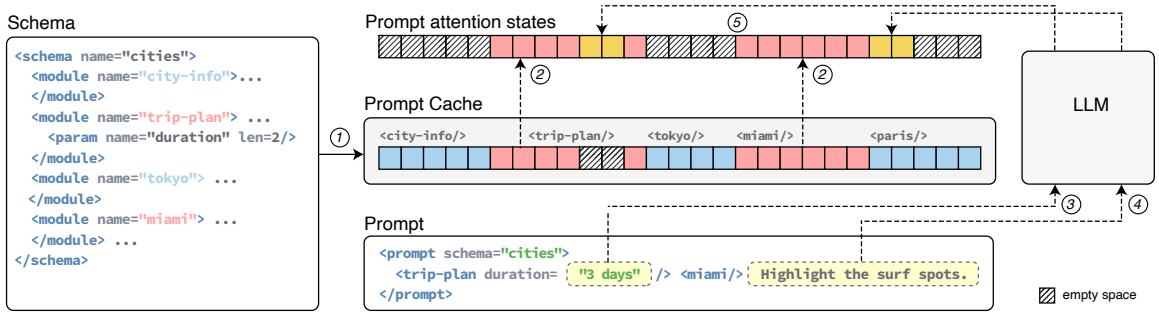


Figure 2. Reuse mechanism in Prompt Cache: (*i*) First, PML (§3.2) makes reusable prompt modules explicit in both Schema and Prompt. A prompt module can have parameters like `trip-plan`. A prompt importing the module supplies a value (`3 days`) to the parameter (`duration`). The prompt can include new text segments in place of excluded modules and parameters and at the end. (*ii*) Second, prompt module encoding (§ 3.3) precomputes attention states (①) for all modules in the schema and caches them for future reuse. (*iii*) Third, when the prompt is served, Prompt Cache employs cached inference (§3.4): it retrieves the attention states cached for imported prompt modules (②), computes them for parameters (③) and new text segments (④), and finally concatenates them to produce the attention states for the entire prompt (⑤). This figure is an elaboration of Step ❶ in Figure 1c.

ent positions in different prompts. To reuse their attention states across prompts, a caching system must tackle two problems. First, it must allow reuse despite a text segment appearing in different positions in different prompts. Second, the system must be able to efficiently recognize a text segment whose attention states may have been cached in order to reuse, when the system receives a new prompt.

To tackle these two problems, we combine two ideas. The first is to make the structure of a prompt explicit with a *Prompt Markup Language* (PML). As illustrated by Figure 2, the PML makes reusable text segments explicit as modules, *i.e.*, *prompt module*. It not only solves the second problem above but opens the door for solving the first, since each prompt module can be assigned with unique position IDs. Our second idea is our empirical finding that LLMs can operate on attention states with discontinuous position IDs. As long as the relative position of tokens is preserved, output quality is not affected. This means that we can extract different segment of attention states and concatenate them to formulate new meanings. We leverage this to enable users to select prompt modules based on their needs, or even replace some meanings during the runtime.

Prompt Cache puts these two ideas together as follows. An LLM user writes their prompts in PML, with the intention that they may reuse the attention states based on prompt modules. Importantly, they must derive a prompt from a *schema*, which is also written in PML. Figure 2 shows a example prompt based on an example schema. When Prompt Cache receives a prompt, it first processes its schema and computes the attention states for its prompt modules. It reuses these states for the prompt modules in the prompt and other prompts derived from the same schema.

We detail the design of PML in §3.2 with a focus on techniques that maximize the opportunity of reusing. We explain how Prompt Cache computes the attention states of prompt modules in a schema in §3.3, and how it may affect the output quality. We explain how Prompt Cache reuse attention states from a schema for the service of a prompt in §3.4.

The modular KV cache construction in Prompt Cache bears resemblance to the approximations observed in *locally masked attention* (Beltagy et al., 2020; Tay et al., 2023), which optimizes computations by setting a limited window for attention score calculations rather than spanning its attention across every token in its input sequence. Consider a scenario within Prompt Cache where each prompt module is encoded independently. Given that attention states are strictly calculated within the confines of the prompt module, this closely mirrors the setup of an attention mask that screens out sequences external to the prompt module. Therefore, the approximation made by Prompt Cache is to limit the attention window to each prompt module. We note that employing such attention masks does not necessarily reduce output quality, as we will discuss in §5. In some contexts, these masks may even introduce beneficial inductive biases by effectively filtering out irrelevant information.

3.2 Prompt Markup Language (PML)

We next describe the key features of PML that is used to define both schemas and schema-derived prompts.

3.2.1 Schema vs. Prompt

A schema is a document that defines prompt modules and delineates their relative positions and hierarchies. Each schema has a unique identifier (via the `name` attribute) and designates prompt modules with the `<module>` tag. Texts not enclosed by `<module>` tags or unspecified identifier

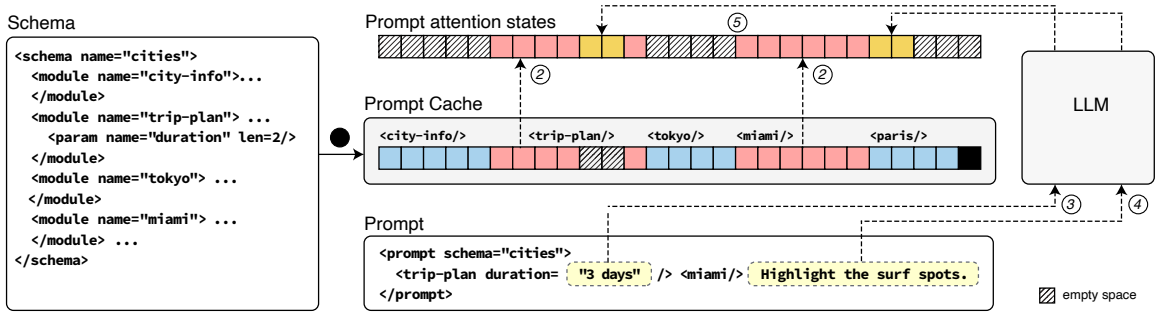


图 2. 提示缓存中的重用机制：(*i*) 首先，PML (§3.2) 在 Schema 和 Prompt 中明确可重用的提示模块。一个提示模块可以有像旅行计划这样的参数。导入该模块的提示为参数（持续时间）提供一个值（3 天）。提示可以在排除的模块和参数的位置以及最后包含新的文本段落。(*ii*) 其次，提示模块编码 (§ 3.3) 预计算 Schema 中所有模块的注意力状态（*1*），并将其缓存以供将来重用。(*iii*) 第三，当提示被提供时，提示缓存使用缓存推理 (§3.4)：它检索为导入的提示模块缓存的注意力状态（*2*），为参数（*3*）和新的文本段落（*4*）计算它们，最后将它们连接以生成整个提示的注意力状态（*5*）。该图是图 1c 中步骤 *1* 的详细说明。

在不同提示中的实体位置。为了在提示之间重用它们的注意状态，缓存系统必须解决两个问题。首先，它必须允许重用，即使文本片段在不同提示中的不同位置出现。其次，当系统接收到新的提示时，系统必须能够有效识别可能已被缓存以便重用的文本片段。

为了解决这两个问题，我们结合了两个想法。第一个是通过*Prompt Markup Language* (PML)明确提示的结构。如图2所示，PML将可重用的文本片段明确为模块，*i.e.*，*prompt module*。它不仅解决了上述第二个问题，还为解决第一个问题打开了大门，因为每个提示模块可以分配唯一的位置ID。我们的第二个想法是我们的实证发现，即LLM可以在具有不连续位置ID的注意状态上操作。只要保留令牌的相对位置，输出质量就不会受到影响。这意味着我们可以提取不同的注意状态片段并将它们连接起来以形成新的含义。我们利用这一点使用户能够根据他们的需求选择提示模块，甚至在运行时替换某些含义。

Prompt Cache 将这两个想法结合如下。LLM 用户用 PML 编写他们的提示，目的是希望能够基于提示模块重用注意力状态。重要的是，他们必须从 *schema* 中派生出一个提示，该提示也用 PML 编写。图 2 显示了基于示例模式的示例提示。当 Prompt Cache 接收到一个提示时，它首先处理其模式并计算其提示模块的注意力状态。它重用这些状态用于提示中的提示模块以及从同一模式派生的其他提示。

我们在§3.2中详细介绍了PML的设计，重点关注技术-最大化重用机会的技术。我们在§3.3中解释了Prompt Cache如何计算模式中提示模块的注意状态，以及它可能如何影响输出质量。我们在§3.4中解释了Prompt Cache如何从模式中重用注意状态以服务于提示。

Prompt Cache中的模块化KV缓存构造与*locally masked attention* (Beltagy等人，2020年；Tay等人，2023年)中观察到的近似情况相似，通过为注意力分数计算设置有限的窗口来优化计算，而不是在输入序列中的每个标记上扩展其注意力。考虑在Prompt Cache中的一个场景，其中每个提示模块独立编码。由于注意力状态严格在提示模块的范围内计算，这与屏蔽掉提示模块外部序列的注意力掩码的设置非常相似。因此，Prompt Cache所做的近似是将注意力窗口限制在每个提示模块上。我们注意到，使用这样的注意力掩码并不一定会降低输出质量，正如我们将在§5中讨论的那样。在某些上下文中，这些掩码甚至可能通过有效过滤掉无关信息而引入有益的归纳偏差。

3.2 提示标记语言 (PML)

我们接下来描述 PML 的关键特性，它用于定义模式和基于模式的提示。

3.2.1 Schema vs. Prompt

模式是一个定义提示模块并划定其相对位置和层次的文档。每个模式都有一个唯一的标识符（通过名称属性）并指定带有 <module> 标签的提示模块。未被 <module> 标签包围或未指定标识符的文本

are treated as anonymous prompt modules and are always included in prompts that are constructed from the schema.

For an LLM user, the schema serves as an interface to create and reuse attention states for prompt modules. The user can construct a prompt from a schema, with the `<prompt>` tag. This tag specifies the schema to use through the `schema` attribute, lists the prompt modules to import, and adds any additional (non-cached) instructions. For example, to import the module `miami` from the schema in Figure 2, one would express it as `<miami/>`. Prompt Cache will only compute the attention states for the text that is not specified in the schema, e.g., `Highlights the surf spots` in Figure 2, and reuse attention states for the imported modules, e.g., `trip-plan` and `miami`, thereby reducing the latency.

3.2.2 *Maximizing Reuse with Parameters*

PML allows a prompt module to be parameterized in order to maximize the reuse opportunities. A parameter is a named placeholder with a specified length that can appear anywhere in a prompt module in a schema. It is defined using the `<param>` tag, with the `name` and `len` attributes indicating its name and the maximum number of tokens for the argument, respectively. When a prompt imports the prompt module, it can supply a value to the parameter. Figure 2 shows an example of a paramterized prompt module (`trip-plan`) and how a prompt would include the prompt module and supply a value (`3 days`) to its argument (`duration`). Augment values are not cached.

There are two important uses of parameterized prompt modules. First, it is common that a prompt module differs from another only in some well-defined places. Parameters allow users to provide specific arguments to customize the module at runtime and still benefit from reusing. Figure 2 illustrates this use case with `trip-plan`. This is especially useful for templated prompts. Second, a parameter can be used to create a "buffer" at the beginning or end of a prompt module in the schema. This buffer allows the user to add an arbitrary text segment in a prompt as long as the segment is no longer than the parameter token length it replaces.

3.2.3 *Other Features*

Union modules: Certain prompt modules exhibit mutually exclusive relationships. That is, within a set of modules, only one should be selected. For instance, consider a prompt that asks the LLM to suggest a book to read based on the reader's profile described by a prompt module. There could be multiple prompt modules each describing a reader profile but the prompt can include only one of them.

```
<union>
  <module name="doc-en-US"> ... </module>
  <module name="doc-zh-CN"> ... </module>
</union>
```

To accommodate these exclusive relationships, we introduce the concept of a *union* for prompt modules. A union of modules is denoted using the `<union>` tag. Prompt modules nested within the same union share the same starting position ID. A union not only streamlines the organization of the layout but also conserves position IDs used to encode prompt modules. Further, the system can utilize this structure for optimizations, such as prefetching.

While parameterized modules and unions appear to be similar, they are different in two aspects. First, as we will show in §3.3, parameters and union modules are encoded in different ways. Second, they serve different purposes: parameters are used for inline modifications to maximize the reuse of a module, while union modules are intended for better prompt structure and more efficient utilization of position IDs.

Nested modules: PML also supports nested modules to express hierarchical prompt modules. That is, a prompt module could include prompt modules or unions as components. In prompts, nested modules are imported as modules within modules as shown in Figure 8.

Compatibility with LLM-specific template: Instruction-tuned LLMs often adhere to specific templates to format conversations. For example, in Llama2, a single interaction between the user and the assistant follows the template: `<s>[INST] user message [/INST] assistant message </s>`. To reduce the effort required to manually format the prompt schema to match such templates for different LLMs, we introduce three dedicated tags: `<system>` for system-level prompts, `<user>` for user-generated prompts, and `<assistant>` for exemplar responses generated by the LLM. Prompt Cache dynamically translates and compiles these specialized tags to align with the designated prompt template of the LLM in use.

3.2.4 *Deriving PML from Prompt Programs*

To simplify PML writing, Prompt Cache can automatically convert prompt programs (Beurer-Kellner et al., 2023; Guidence, 2023) from languages like Python into PML, eliminating the need for manual schema writing. This is primarily achieved using a Python API that transforms Python functions into corresponding PML schemas. The conversion process is straightforward: if statements become `<module>` constructs in PML, encapsulating the conditional prompts within. When a condition evaluates to true, the corresponding module is activated. Choose-one statements, such as if-else or switch statements, are mapped to `<union>` tags. Function calls are translated into nested prompt modules. Additionally, we have implemented a decorator to manage parameters, specifically to restrict the maximum argument length. This corresponds to the len attribute in the `<param>`. This Python-to-PML compilation hides PML complexity from the user provides better maintainability of the prompt.

被视为匿名提示模块，并始终包含在从模式构建的提示中。

对于 LLM 用户，模式作为创建和重用提示模块的注意状态的接口。用户可以从模式构建一个提示，使用 <prompt> 标签。该标签通过模式属性指定要使用的模式，列出要导入的提示模块，并添加任何额外的（非缓存）指令。例如，要从图 2 中的模式导入模块 miami，可以表示为 <miami/>。提示缓存将仅计算未在模式中指定的文本的注意状态，例如，突出显示图 2 中的冲浪点，并重用导入模块的注意状态，例如 trip-plan 和 miami，从而减少延迟。

3.2.2 Maximizing Reuse with Parameters

PML允许对提示模块进行参数化，以最大化重用机会。参数是一个具有指定长度的命名占位符，可以在模式中的提示模块的任何位置出现。它使用<param>标签定义，名称和len属性分别指示其名称和参数的最大令牌数。当提示导入提示模块时，可以为参数提供一个值。图2显示了一个参数化提示模块（旅行计划）的示例，以及提示如何包含提示模块并为其参数（持续时间）提供一个值（3天）。增强值不会被缓存。

参数化提示模块有两个重要用途。首先，提示模块通常只在一些明确定义的地方与另一个模块不同。参数允许用户在运行时提供特定的参数，以自定义模块，同时仍然受益于重用。图2通过旅行计划说明了这一用例。这对于模板化提示尤其有用。其次，参数可以用于在模式中提示模块的开头或结尾创建一个“缓冲区”。这个缓冲区允许用户在提示中添加任意文本段，只要该段的长度不超过它所替换的参数令牌长度。

3.2.3 Other Features

联合模块：某些提示模块表现出相互排斥的关系。也就是说，在一组模块中，只应选择一个。例如，考虑一个提示，要求LLM根据提示模块描述的读者档案建议一本书。可能有多个提示模块各自描述一个读者档案，但提示只能包含其中一个。

```
<union>
  <module name="doc-en-US"> ... </module>
  <module name="doc-zh-CN"> ... </module>
</union>
```

为了适应这些独特的关系，我们引入了提示模块的 *union* 概念。模块的联合使用 <union> 标签表示。嵌套在同一联合中的提示模块共享相同的起始位置 ID。联合不仅简化了布局的组织，还节省了用于编码提示模块的位置 ID。此外，系统可以利用这种结构进行优化，例如预取。

虽然参数化模块和联合看起来相似，但它们在两个方面是不同的。首先，正如我们将在§3.3中展示的，参数和联合模块以不同的方式编码。其次，它们的目的不同：参数用于内联修改，以最大化模块的重用，而联合模块旨在更好的提示结构和更高效地利用位置ID。

嵌套模块：PML 还支持嵌套模块以表达层次化的提示模块。也就是说，一个提示模块可以包含提示模块或联合作为组件。在提示中，嵌套模块作为模块中的模块被导入，如图 8 所示。

与特定于LLM的模板的兼容性：经过指令调优的LLM通常遵循特定模板来格式化对话。例如，在Llama2中，用户与助手之间的单次交互遵循模板：<s>[INST] 用户消息 [/INST] 助手消息 </s>。为了减少手动格式化提示模式以匹配不同LLM的此类模板所需的工作量，我们引入了三个专用标签：<system>用于系统级提示，<user>用于用户生成的提示，以及<assistant>用于LLM生成的示例响应。提示缓存动态翻译并编译这些专用标签，以与所使用的LLM的指定提示模板对齐。

3.2.4 Deriving PML from Prompt Programs

为了简化 PML 编写，Prompt Cache 可以自动将提示程序（Beurer-Kellner 等，2023；Guidence，2023）从 Python 等语言转换为 PML，消除手动编写模式的需要。这主要是通过一个 Python API 实现的，该 API 将 Python 函数转换为相应的 PML 模式。转换过程非常简单：if 语句在 PML 中变成 <module> 构造，将条件提示封装在其中。当条件评估为真时，相应的模块被激活。选择一项语句，如 if-else 或 switch 语句，被映射到 <union> 标签。函数调用被翻译成嵌套的提示模块。此外，我们实现了一个装饰器来管理参数，特别是限制最大参数长度。这对应于 <param> 中的 len 属性。这种 Python 到 PML 的编译隐藏了 PML 的复杂性，为用户提供了更好的提示可维护性。

3.3 Encoding Schema

The first time the attention states of a prompt module are needed, they must be computed and stored in the device memory, which we refer to as *prompt module encoding*. First, Prompt Cache extracts token sequences of a prompt module from the schema. It then assigns position IDs to each token. The starting position ID is determined by the absolute location of the prompt module within the schema. For instance, if two preceding prompt modules have token sequence sizes of 50 and 60 respectively, the prompt module is assigned a starting position ID of 110. An exception exists for the union modules. Since prompt modules within the union start from the same positions, their token sequence size is considered with the size of the largest child.

From the token sequences of the prompt module and the corresponding position IDs, these are then passed to the LLM to compute the (k, v) attention states. We note that the assigned position IDs do not start from zero. This is semantically acceptable since white spaces do not alter the meaning of the precomputed text. However, many existing transformer positional encoding implementations, such as RoPE, often require adaptations to accommodate discontinuous position IDs, which we will discuss in (§ 4.2).

For encoding parameterized prompt modules, we use the idea that having white space in a prompt does not affect its semantics. Parameters are replaced by a predetermined number of `<unk>` tokens, equivalent to their `len` attribute value. The position IDs corresponding to these `<unk>` tokens are logged for future replacement. When this module is integrated into a user's prompt and paired with the relevant arguments, the token sequences of these supplied arguments adopt the position IDs previously linked with the `<unk>` tokens. The resulting (k, v) attention states then replace the attention states initially allocated for the `<unk>` tokens. We note that the length of the newly provided tokens can be smaller than the specified parameter length, as trailing white spaces do not change the semantics.

Attention masking effect: Prompt Cache confines attention score computation to the span of each prompt module, masking the attention states across modules. This masking effect can enhance or degrade output quality depending on the semantic independence of the modules. For semantically independent modules, masking reduces noise and improves quality. However, for semantically dependent modules, it can have the opposite effect. Therefore, each prompt module should be self-contained and semantically independent from other modules. One way to remove the masking effect is to use a method we refer to as *scaffolding*. At the cost of additional memory, we allow users to specify "scaffolds", which are sets of prompt modules that are encoded together to share the attention span, in addition to their individual attention states. When all prompt modules in a scaffold are imported in a prompt, the attention states of the scaffold overrides the individual attention states. Scaffolding trades off additional memory for output consistency, which may be useful for applications that need deterministic results.

3.4 Cached Inference

When a prompt is provided to Prompt Cache, Prompt Cache parses it to ensure alignment with the claimed schema. It verifies the validity of the imported modules. Then, as illustrated in Figure 2, Prompt Cache retrieves the (k, v) attention states for the imported prompt modules from the cache (②), computes those for new text segments (③ and ④), and concatenates them to produce the attention states for the entire prompt (⑤), replacing the prefill operation.

To detail the process, Prompt Cache starts by concatenating the KV state tensors corresponding to each imported prompt module in the prompt. For instance, when a user prompt utilizes modules A, B, the concatenated KV tensor is formulated as: $(k_C, v_C) = (\text{concat}(k_A, k_B), (\text{concat}(v_A, v_B))$. It is worth noting that the order of concatenation does not matter due to the permutation invariance of transformers (Dufter et al., 2022). This step solely requires memory copy. Then, Prompt Cache computes the attention states for the segments of the prompt that are not cached, specifically, token sequences not defined in the schema and arguments for parameterized prompt modules. Prompt Cache first identifies the position IDs of uncached texts based on their position relative to other utilized prompt modules. For example, if the text is situated between module A and B, it is assigned the position ID starting from the concluding positions of A, assuming gaps exist between the positions of A and B. Augments for parameterized prompt modules are assigned to the position IDs of `<unk>` tokens. Subsequently, the token sequences and position IDs are aggregated and passed to the LLM *using* (k_C, v_C) *as a KV Cache*, to compute the attention states for the entire prompt. It is important to note that the computational complexity for generating subsequent tokens remains consistent with that of KV Cache, as prompt modules are not employed beyond the initial token. In essence, Prompt Cache diminishes the latency involved in producing the first token, or time-to-first-token (TTFT).

Memory optimization in batch inference: Prompts are usually served in a batch for better GPU utilization. Different prompts derived from the same schema may include the same prompt modules, such as system prompts. This opens up additional optimization opportunities by reducing KV Cache redundancies in a batch. Paged attention (Kwon et al., 2023) can resolve this issue by sharing the *pointer* to the same prompt module across different prompts, instead of duplicating the attention states. Here, the use of Prompt Cache can implicitly improve system throughput by allowing more prompts to be processed in parallel.

3.3 编码方案

第一次需要提示模块的注意状态时，必须在设备内存中计算并存储这些状态，我们称之为 *prompt module encoding*。首先，提示缓存从模式中提取提示模块的令牌序列。然后，它为每个令牌分配位置 ID。起始位置 ID 由提示模块在模式中的绝对位置决定。例如，如果两个前面的提示模块的令牌序列大小分别为 50 和 60，则提示模块的起始位置 ID 为 110。对于联合模块存在一个例外。由于联合中的提示模块从相同的位置开始，因此它们的令牌序列大小与最大子模块的大小一起考虑。

从提示模块的令牌序列和相应的位置 ID 开始，这些然后被传递给 LLM 以计算 (k, v) 注意力状态。我们注意到分配的位置 ID 并不是从零开始的。这在语义上是可以接受的，因为空格不会改变预计算文本的含义。然而，许多现有的变换器位置编码实现，例如 RoPE，通常需要进行调整以适应不连续的位置 ID，我们将在 (§ 4.2) 中讨论。

对于编码参数化提示模块，我们使用的理念是提示中的空格不会影响其语义。参数被预定数量的 <unk> 令牌替换，这些令牌的数量等于它们的 len 属性值。与这些 <unk> 令牌对应的位置 ID 被记录以便将来替换。当该模块集成到用户的提示中并与相关参数配对时，这些提供的参数的令牌序列采用之前与 <unk> 令牌链接的位置 ID。生成的 (k, v) 注意状态随后替换最初分配给 <unk> 令牌的注意状态。我们注意到，新提供的令牌的长度可以小于指定的参数长度，因为尾随空格不会改变语义。

注意掩蔽效应：提示缓存将注意力分数计算限制在每个提示模块的范围内，掩蔽跨模块的注意力状态。这个掩蔽效应可以根据模块的语义独立性增强或降低输出质量。对于语义独立的模块，掩蔽减少噪声并提高质量。然而，对于语义依赖的模块，它可能产生相反的效果。因此，每个提示模块应该是自包含的，并且在语义上独立于其他模块。消除掩蔽效应的一种方法是使用我们称之为 *scaffolding* 的方法。以额外的内存为代价，我们允许用户指定“支架”，这些支架是一组一起编码的提示模块，以共享注意力范围，除了它们各自的注意力状态。当支架中的所有提示模块都是在提示中导入时，支架的注意状态会覆盖个体注意状态。支架在输出一致性上权衡了额外的内存，这对于需要确定性结果的应用可能是有用的。

3.4 缓存推理

当提示提供给提示缓存时，提示缓存会解析它以确保与声明的模式对齐。它验证导入模块的有效性。然后，如图2所示，提示缓存从缓存中检索导入提示模块的（k, v）注意状态（*2*），计算新文本段的注意状态（*3*和*4*），并将它们连接以生成整个提示的注意状态（*5*），替代预填充操作。

为了详细说明这个过程，Prompt Cache 首先通过连接与每个导入的提示模块对应的 KV 状态张量来开始处理。例如，当用户提示使用模块 A, B 时，连接的 KV 张量被公式化为：$(k_C, v_C) = (\text{concat}(k_A, k_B), (\text{concat}(v_A, v_B))$。值得注意的是，由于变换器的置换不变性（Dufter 等，2022），连接的顺序并不重要。这一步仅需要内存复制。然后，Prompt Cache 计算未缓存的提示段的注意状态，具体来说，是未在模式中定义的令牌序列和参数化提示模块的参数。Prompt Cache 首先根据未缓存文本相对于其他使用的提示模块的位置识别位置 ID。例如，如果文本位于模块 A 和 B 之间，则它被分配一个从 A 的结束位置开始的位置信息 ID，假设 A 和 B 的位置之间存在间隙。参数化提示模块的增强被分配给 <unk> 令牌的位置 ID。随后，令牌序列和位置 ID 被聚合并传递给 LLM *using* (k_C, v_C) *as a KV Cache*，以计算整个提示的注意状态。重要的是要注意，生成后续令牌的计算复杂性与 KV Cache 的复杂性保持一致，因为提示模块在初始令牌之后不再被使用。总之，Prompt Cache 减少了生成第一个令牌的延迟，或称为首次令牌时间（TTFT）。

批量推理中的内存优化：通常会将提示以批量形式提供，以更好地利用GPU。来自同一模式的不同提示可能包含相同的提示模块，例如系统提示。这通过减少批量中的KV缓存冗余，打开了额外的优化机会。分页注意力（Kwon等，2023）可以通过在不同提示之间共享*pointer*到相同的提示模块来解决此问题，而不是重复注意状态。在这里，使用提示缓存可以通过允许更多提示并行处理，隐式提高系统吞吐量。

4 IMPLEMENTATION

We build a Prompt Cache prototype using the HuggingFace transformers library (Wolf et al., 2020) in PyTorch and comprises 3K lines of Python code. We aim to seamlessly integrate with an existing LLM codebase and reuse its weights. We implement Prompt Cache to use both CPU and GPU memory to accommodate prompt modules and evaluate it on both platforms.

4.1 Storing Prompt Modules in Memory

We store encoded prompt modules in two types of memory: CPU memory (host DRAM) and GPU memory (HBM). To manage tensors across both memory types, we employ the PyTorch (Paszke et al., 2019) memory allocator. Beyond simply pairing CPUs with prompt modules in CPU memory and GPUs with GPU memory, we also enable GPUs to access prompt modules stored in CPU memory. This is done by copying the prompt modules from the host to the device as needed. This process incurs a host-to-device memory copy overhead. Nonetheless, it allows the GPU to leverage the abundant CPU memory, which can scale up to terabyte levels. As we will show in §5, the computational savings from Prompt Cache more than compensate for the latencies caused by memory copy operations. Using GPUs exposes trade-offs between memory capacity and latency: GPU memory is faster but limited in capacity, while CPU memory can scale easily yet incurs additional memory copy overhead. It appears feasible to contemplate a caching mechanism that leverages both CPU and GPU memory. We leave the development of a system that incorporates cache replacement and prefetching strategies to future research.

4.2 Adapting Transformer Architectures

Implementing Prompt Cache requires support for discontinuous position IDs (§3.2). Although the Transformers library currently does not offer these features, they can be integrated with minor modifications. For instance, approximately 20 lines of additional code are needed for each LLM. We outline the required adjustments:

Embedding tables: Early models like BERT (Vaswani et al., 2023) and GPT-2 (Radford et al., 2018) use lookup tables for mapping position IDs to learned embeddings or fixed bias, requiring no alterations.

RoPE: LLMs such as Llama2 (Touvron et al., 2023) and Falcon (Penedo et al., 2023) adopt RoPE (Su et al., 2021), which employs rotation matrices for positional encoding in attention computations. We create a lookup table for each rotation matrix, enabling retrieval based on position IDs.

ALiBi: Utilized in models like MPT (MosaicML, 2023) and Bloom (Scao et al., 2022), ALiBi (Press et al., 2022) integrates a static bias during softmax score calculations. Analogous to RoPE, we design a lookup table to adjust the bias matrix according to the provided position IDs.

We also override PyTorch's concatenation operator for more efficient memory allocation. PyTorch only supports contiguous tensors, and therefore, concatenation of two tensors always results in a new memory allocation. Prompt Cache needs to concatenate attention states of prompt modules, and the default behavior would lead to redundant memory allocations. We implement a buffered concatenation operator that reuses memory when concatenating tensors. This optimization improves the memory footprint of Prompt Cache and reduces the overhead of memory allocation.

5 EVALUATION

Our evaluation of Prompt Cache focuses on answering the following three research questions: (i) What is the impact of Prompt Cache on time-to-first-token (TTFT) latency and output quality (§5.2 – §5.4), (ii) What is the memory storage overhead (§5.5), and (iii) What applications are a good fit for Prompt Cache (§5.6). We use the regular KV Cache (Pope et al., 2022) as our baseline. Prompt Cache and KV Cache share the exact same inference pipeline except for attention state computation. We use TTFT latency for comparison, which measures the time to generate the first token, as Prompt Cache and KV Cache have the same decoding latency after the first token.

5.1 Evaluation Environment

We evaluate Prompt Cache on two CPU configurations: an Intel i9-13900K accompanied by 128 GB DDR5 RAM at 5600 MT/s and an AMD Ryzen 9 7950X paired with 128 GB DDR4 RAM at 3600 MT/s. For our GPU benchmarks, we deploy three NVIDIA GPUs: the RTX 4090, which is paired with the Intel i9-13900K, and the A40 and A100, both virtual nodes hosted on NCSA Delta, each provisioned with a 16-core AMD EPIC 7763 and 224 GB RAM. We employ several open-source LLMs, including Llama2, CodeLlama, MPT, and Falcon. We use LLMs that fit within the memory capacity of a single GPU (40 GB). We utilize the LongBench suite (Bai et al., 2023) to assess TTFT improvements and output quality changes. LongBench encompasses a curated subsample of elongated data, ranging from 4K to 10K context length, excerpts from 21 datasets across 6 categories, including tasks like multi-document question answering (Yang et al., 2018; Ho et al., 2020; Trivedi et al., 2022; Kočiskỳ et al., 2018; Joshi et al., 2017), summarization (Huang et al., 2021; Zhong et al., 2021; Fabbri et al., 2019), and code completion (Guo et al., 2023; Liu et al., 2023a). We defined the documents in the LongBench datasets, such as wiki pages and news articles, as prompt modules. We kept the task-specific directives as uncached user text.

4 实施

我们使用Hugging-Face transformers库（Wolf等，2020）在PyTorch中构建了一个Prompt Cache原型，包含3000行Python代码。我们的目标是与现有的LLM代码库无缝集成并重用其权重。我们实现了Prompt Cache，以同时使用CPU和GPU内存来容纳提示模块，并在这两个平台上进行评估。

4.1 在内存中存储提示模块

我们将编码的提示模块存储在两种类型的内存中：CPU 内存（主机 DRAM）和 GPU 内存（HBM）。为了管理这两种内存类型中的张量，我们使用 PyTorch（Paszke 等，2019）内存分配器。除了简单地将 CPU 与 CPU 内存中的提示模块配对，以及将 GPU 与 GPU 内存配对外，我们还允许 GPU 访问存储在 CPU 内存中的提示模块。这是通过根据需要将提示模块从主机复制到设备来实现的。这个过程会产生主机到设备的内存复制开销。尽管如此，它允许 GPU 利用丰富的 CPU 内存，容量可以扩展到 TB 级别。正如我们将在 §5 中展示的，来自提示缓存的计算节省足以弥补内存复制操作造成的延迟。使用 GPU 暴露了内存容量和延迟之间的权衡：GPU 内存速度更快但容量有限，而 CPU 内存可以轻松扩展，但会产生额外的内存复制开销。考虑利用 CPU 和 GPU 内存的缓存机制似乎是可行的。我们将缓存替换和预取策略的系统开发留给未来的研究。

4.2 适应变压器架构

实现提示缓存需要对不连续位置ID的支持（§3.2）。尽管Transformers库目前不提供这些功能，但可以通过少量修改进行集成。例如，每个LLM大约需要20行额外的代码。我们概述了所需的调整：

嵌入表：早期模型如 BERT (Vaswani et al., 2023) 和 GPT-2 (Radford et al., 2018) 使用查找表将位置 ID 映射到学习的嵌入或固定偏置，无需进行更改。

RoPE：像Llama2（Touvron等，2023）和Falcon（Penedo等，2023）这样的LLM采用RoPE（Su等，2021），该方法在注意力计算中使用旋转矩阵进行位置编码。我们为每个旋转矩阵创建一个查找表，使得可以根据位置ID进行检索。

ALiBi：在 MPT（MosaicML，2023）和 Bloom（Scao 等，2022）等模型中使用，ALiBi（Press 等，2022）在softmax得分计算中集成了静态偏差。类似于RoPE，我们设计了一个查找表，以根据提供的位置ID调整偏差矩阵。

我们还重写了 PyTorch 的连接操作符，以实现更高效的内存分配。PyTorch 仅支持连续张量，因此，两个张量的连接总是会导致新的内存分配。Prompt Cache 需要连接提示模块的注意状态，而默认行为会导致冗余的内存分配。我们实现了一个缓冲连接操作符，在连接张量时重用内存。此优化改善了 Prompt Cache 的内存占用，并减少了内存分配的开销。

5 评估

我们对 Prompt Cache 的评估集中在回答以下三个研究问题：（i）Prompt Cache 对首次令牌时间（TTFT）延迟和输出质量的影响是什么（§5.2 – §5.4），（ii）内存存储开销是多少（§5.5），以及（iii）哪些应用程序适合 Prompt Cache（§5.6）。我们使用常规 KV Cache（Pope 等，2022）作为我们的基线。Prompt Cache 和 KV Cache 共享完全相同的推理管道，除了注意力状态计算。我们使用 TTFT 延迟进行比较，它测量生成第一个令牌的时间，因为 Prompt Cache 和 KV Cache 在第一个令牌之后具有相同的解码延迟。

5.1 评估环境

我们在两种 CPU 配置上评估 Prompt Cache：一台配备 128 GB DDR5 RAM、运行速度为 5600 MT/s 的 Intel i9-13900K，以及一台配备 128 GB DDR4 RAM、运行速度为 3600 MT/s 的 AMD Ryzen 9 7950X。对于我们的 GPU 基准测试，我们部署了三款 NVIDIA GPU：RTX 4090，与 Intel i9-13900K 配对，以及 A40 和 A100，这两者都是托管在 NCSA Delta 上的虚拟节点，每个节点配备 16 核 AMD EPIC 7763 和 224 GB RAM。我们使用了多个开源 LLM，包括 Llama2、CodeLlama、MPT 和 Falcon。我们使用的 LLM 适合单个 GPU 的内存容量（40 GB）。我们利用 LongBench 套件（Bai 等，2023）来评估 TTFT 改进和输出质量变化。LongBench 包含一个经过精心挑选的延长数据子样本，范围从 4K 到 10K 的上下文长度，摘录自 21 个数据集，涵盖 6 个类别，包括多文档问答任务（Yang 等，2018；Ho 等，2020；Trivedi 等，2022；Kočiskỳ 等，2018；Joshi 等，2017）、摘要（Huang 等，2021；Zhong 等，2021；Fabbri 等，2019）和代码补全（Guo 等，2023；Liu 等，2023a）。我们将 LongBench 数据集中的文档，如维基页面和新闻文章，定义为提示模块。我们将任务特定的指令保留为未缓存的用户文本。

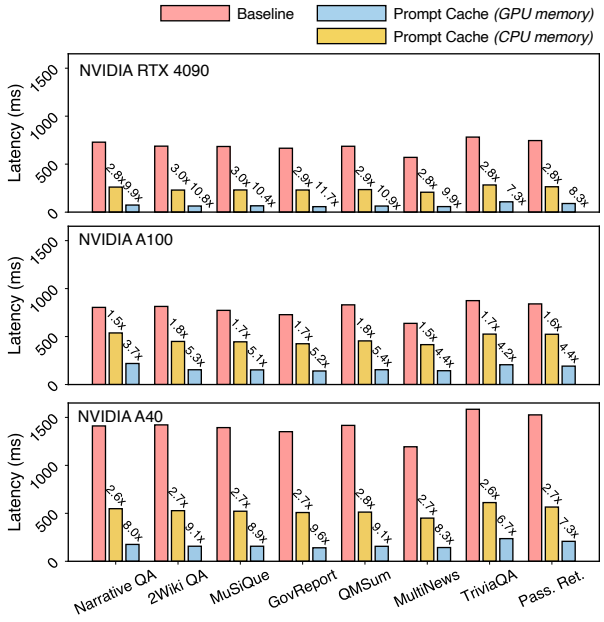


Figure 3. GPU latency measurements: time-to-first-token (TTFT) for eight LongBench datasets across three NVIDIA GPUs.

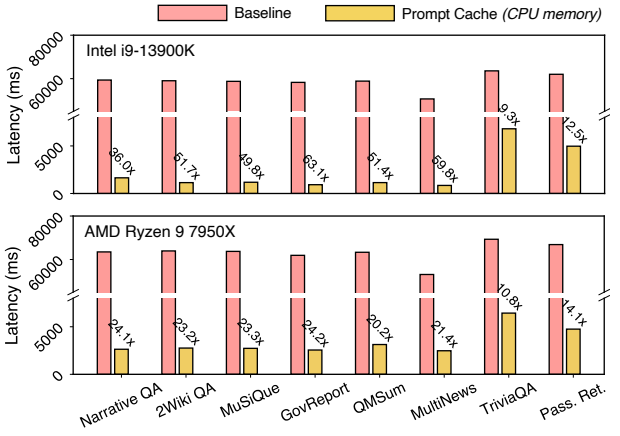


Figure 4. CPU latency measurements: time-to-first-token (TTFT) for eight LongBench datasets across two CPUs.

5.2 Latency Improvements on Benchmark Datasets

We measured the TTFT latency on both GPU and CPU using Llama 7B, as shown in Figure 3 and Figure 4. In our GPU evaluation, we used two memory setups: storing prompt modules in either CPU or GPU memory. For CPU experiments, we used CPU memory. Due to space constraints, we present only 8 benchmarks. The complete benchmark from 21 datasets can be found in the Appendix.

5.2.1 GPU Inference Latency

We summarize our findings in Figure 3, evaluated on three NVIDIA GPUs: RTX 4090, A40, and A100. Yellow bars represent loading prompt modules from CPU memory, while blue bars represent the case in GPU memory. There is a consistent latency trend across the datasets since the LongBench samples have comparable lengths, averaging 5K tokens. We observe significant TTFT latency reductions across all datasets and GPUs, ranging from 1.5× to 3× when using CPU memory, and from 5× to 10× when employing GPU memory. These results delineate the upper and lower bounds of latency reductions possible with Prompt Cache. The actual latency reduction in practice will fall between these bounds, based on how much of each memory type is used.

5.2.2 CPU Inference Latency

Figure 4 shows that Prompt Cache achieves up to a 70× and 20× latency reduction on the Intel and AMD CPUs, respectively. We surmise that this disparity is influenced by the difference in memory bandwidth in system setups (5600MT/s DDR5 RAM on the Intel CPU versus 3600MT/s DDR4 RAM on the AMD CPU). As expected, the latency is higher for the datasets with a larger proportion of uncached prompts, such as `TriviaQA`. Interestingly, CPU inference benefits more significantly from Prompt Cache than GPU inference does. This is attributed to the much greater latency of attention computation in the CPU, especially as the sequences become longer (*e.g.*, lower FP16/FP32 FLOPs compared to GPU). This indicates that Prompt Cache is particularly beneficial for optimizing inference in resource-constrained environments, such as edge devices or cloud servers with limited GPU resources.

5.3 Accuracy with Prompt Cache

To verify the impact of Prompt Cache on the quality of LLM response, without scaffolding, we measure accuracy scores with the LongBench suite. To demonstrate general applicability, we apply Prompt Cache to the three LLMs having different transformer architectures (§4.2): Llama2, MPT, and Falcon. The accuracy benchmark results shown in Table 1 demonstrate Prompt Cache preserves the precision of the output. We use deterministic sampling where the token with the highest probability is chosen at every step so that the results with and without Prompt Cache are comparable. Across all datasets, the accuracy of output with Prompt Cache is comparable to the baseline.

5.4 Understanding Latency Improvements

Theoretically, Prompt Cache should offer quadratic TTFT latency reduction over regular KV Cache. This is because, while Prompt Cache's memcpy overhead grows linearly with sequence length, computing self-attention has quadratic computational complexity with respect to sequence length. To validate this, we tested Prompt Cache on a synthetic

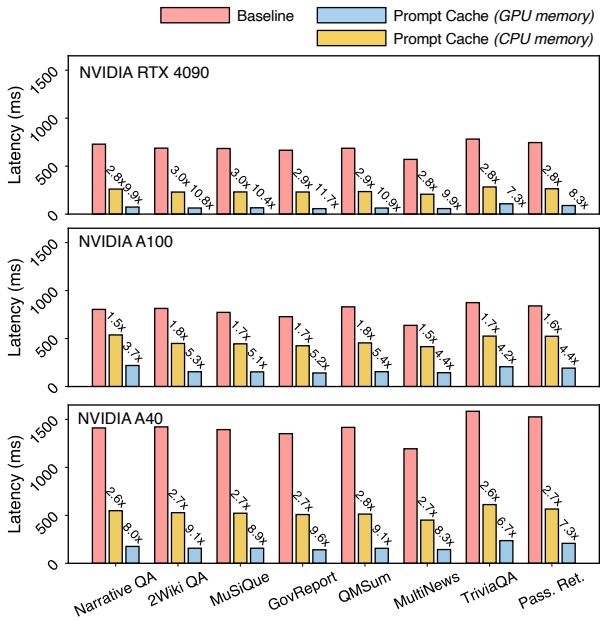


图3. GPU延迟测量：在三款NVIDIA GPU上，八个LongBench数据集的首次令牌时间（TTFT）。

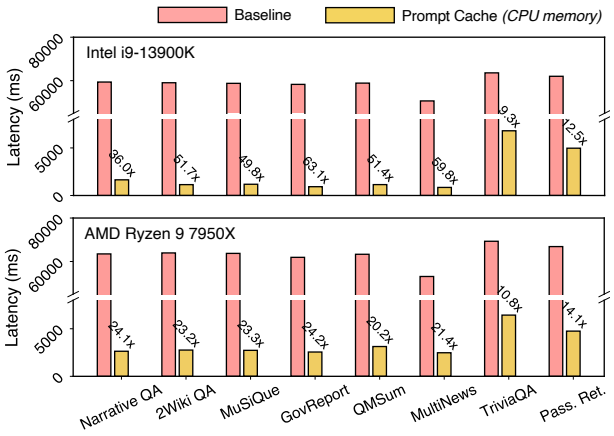


图4. CPU延迟测量：两个CPU上八个LongBench数据集的首次令牌时间（TTFT）。

5.2 基准数据集上的延迟改进

我们使用 Llama 7B 在 GPU 和 CPU 上测量了 TTFT 延迟，如图 3 和图 4 所示。在我们的 GPU 评估中，我们使用了两种内存设置：将提示模块存储在 CPU 或 GPU 内存中。对于 CPU 实验，我们使用了 CPU 内存。由于空间限制，我们仅展示了 8 个基准测试。完整的 21 个数据集的基准测试可以在附录中找到。

5.2.1 GPU Inference Latency

我们在图3中总结了我们的发现，评估了三款NVIDIA GPU：RTX 4090、A40和A100。黄色条表示从CPU内存加载提示模块，而蓝色条表示在GPU内存中的情况。由于LongBench样本的长度相当，平均为5K个标记，因此在各个数据集之间存在一致的延迟趋势。我们观察到在所有数据集和GPU上，TTFT延迟显著减少，使用CPU内存时范围从1.5×到3×，使用GPU内存时范围从5×到10×。这些结果描绘了使用Prompt Cache时延迟减少的上下限。实际的延迟减少将在这些范围之间，具体取决于每种内存类型的使用量。

5.2.2 CPU Inference Latency

图4显示，Prompt Cache在Intel和AMD CPU上分别实现了高达70×和20×的延迟减少。我们推测这种差异受到影响。

通过系统配置中内存带宽的差异（英特尔 CPU 上的 5600MT/s DDR5 RAM 与 AMD CPU 上的 3600MT/s DDR4 RAM）。正如预期的那样，对于具有更大比例未缓存提示的数据集（例如 TriviaQA），延迟更高。有趣的是，CPU 推理比 GPU 推理更显著地受益于提示缓存。这归因于 CPU 中注意力计算的延迟要大得多，尤其是当序列变得更长时（*e.g.*，与 GPU 相比，FP16/FP32 FLOPs 较低）。这表明，提示缓存对于优化资源受限环境中的推理特别有利，例如边缘设备或 GPU 资源有限的云服务器。

5.3 使用提示缓存的准确性

为了验证提示缓存对LLM响应质量的影响，在没有支架的情况下，我们使用LongBench套件测量准确性分数。为了展示通用适用性，我们将提示缓存应用于三种具有不同变换器架构的LLM（§4.2）：Llama2、MPT和Falcon。表1中显示的准确性基准结果表明，提示缓存保持了输出的精度。我们使用确定性采样，在每一步选择概率最高的标记，以便有和没有提示缓存的结果可以进行比较。在所有数据集中，使用提示缓存的输出准确性与基线相当。

5.4 理解延迟改进

理论上，Prompt Cache 应该在常规 KV Cache 上提供二次 TTFT 延迟减少。这是因为，虽然 Prompt Cache 的 memcpy 开销随着序列长度线性增长，但计算自注意力的计算复杂度与序列长度呈二次关系。为了验证这一点，我们在一个合成数据集上测试了 Prompt Cache。

Dataset	Metric	Llama2 7B		Llama2 13B		MPT 7B		Falcon 7B	
		Baseline	Cached	Baseline	Cached	Baseline	Cached	Baseline	Cached
Narrative QA	F1	19.93	19.38	20.37	19.94	10.43	11.33	7.14	8.87
2 Wiki Multi-Hop QA	F1	**16.63**	13.95	14.59	**17.69**	10.44	**13.70**	14.42	15.07
MuSiQue	F1	7.31	8.57	10.03	12.14	7.38	7.32	4.81	5.86
GovReport	Rouge L	24.67	25.37	28.13	28.18	26.96	27.49	22.39	23.40
QMSum	Rouge L	19.24	19.46	18.80	18.82	15.19	15.51	12.84	12.96
MultiNews	Rouge L	24.33	24.22	25.43	26.23	25.42	25.66	20.91	21.19
TriviaQA	F1	13.04	12.33	23.19	22.38	10.57	9.17	13.31	11.42
Passage Retrieval	Acc	**7.50**	4.25	**9.08**	6.50	3.03	3.85	3.00	3.45

Table 1. Accuracy benchmarks on LongBench datasets. We mark the outliers as **bold**, of which the performance is higher than 2.5 compared to the counterpart.

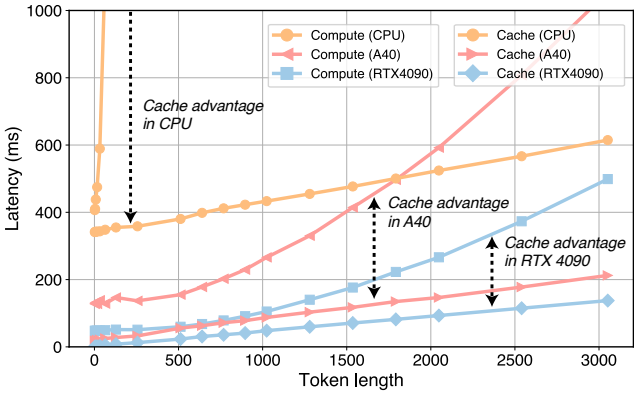


Figure 5. Cache advantage: A comparison of computational and caching overheads in GPUs and CPUs. While attention computation cost increases quadratically, the attention state memory copy overhead (*i.e.*, Prompt Cache) rises linearly. Here, GPUs load prompt modules directly from CPU memory.

LLM	BERT	Falcon 1B	Llama 7B	Llama 13B
MB/token	0.03	0.18	0.50	0.78
LLM	MPT 30B	Falcon 40B	Llama 70B	Falcon 180B
MB/token	1.31	1.87	2.5	4.53

Table 2. Memory overhead of caching a single token

dataset with varied sequence lengths, assuming all prompts were cached. We compared the TTFT latency of Prompt Cache to that of regular KV Cache using an Intel i9-13900K CPU and two GPUs (NVIDIA RTX 4090 and A40) with the Llama2 7B model. For both CPU and GPU, CPU memory is used for prompt module storage.

Quadratic improvement: Our findings, presented in Figure 5, show that KV Cache's latency increases quadratically with sequence length, while Prompt Cache's memory copy cost grows linearly. This means that the latency advantage of Prompt Cache (the gap between the two curves) expands quadratically with sequence length. This difference is more pronounced on CPUs than GPUs since CPUs experience higher attention computation latencies, whereas the disparity between Prompt Cache's overhead, *i.e.*, host-to-device memcpy in GPUs and host-to-host memcpy in CPUs is not significant. With attention states with 5K tokens, latency for host-to-host, host-to-device, and device-to-device memcpy are respectively 3.79 ms, 5.34 ms, and 0.23 ms.

Effect of model size: Furthermore, as the model's parameter size grows, so does the computational overhead for KV Cache. For example, moving from a 7B to 13B model at a token length of 3K added 220 ms latency, whereas Prompt Cache added only 30 ms. This difference stems from the fact that LLM complexity also scales quadratically with hidden dimension size. For example, the FLOPS of attention is $6nd^2 + 4n^2d$, for prefill operation. This suggests that Prompt Cache's advantage over KV Cache also quadratically increases with model size (*i.e.*, hidden dimension).

End-to-end latency: Since Prompt Cache reduces only TTFT, its impact on the time needed to receive the complete LLM response diminishes as the number of generated tokens increases. For instance, on the RTX 4090 with Llama 7B for 3K context, Prompt Cache enhances TTFT from 900 ms to 90 ms, while the token generation time or the time-to-subsequent-token (TTST) remains consistent between KV Cache and Prompt Cache at an average of 32 ms per token, regardless of the token length. Nonetheless, a quicker response time contributes positively to the user experience and the overall end-to-end latency (Lew et al., 2018; Liu et al., 2023b), For instance, Given that Prompt Cache enhances TTFT from 900 ms to 90 ms, this equates to the generation of 25 more tokens within the same timeframe. Another factor is that Prompt Cache enables sharing attention states within the same batch, as we discussed in §3.4. Depending on the workload characteristics, Prompt Cache can improve overall throughput by utilizing the larger batch size enabled by the reduced memory footprint. For example, suppose there are 100 requests, each with a 2K token prompt. If all prompts share the same 1K token module, Prompt Cache can reduce the memory footprint by 50% when combined with methods like paged attention, allowing for a larger

Dataset	Metric	Llama2 7B		Llama2 13B		MPT 7B		Falcon 7B	
		Baseline	Cached	Baseline	Cached	Baseline	Cached	Baseline	Cached
Narrative QA	F1	19.93	19.38	20.37	19.94	10.43	11.33	7.14	8.87
2 Wiki Multi-Hop QA	F1	**16.63**	13.95	14.59	**17.69**	10.44	**13.70**	14.42	15.07
MuSiQue	F1	7.31	8.57	10.03	12.14	7.38	7.32	4.81	5.86
GovReport	Rouge L	24.67	25.37	28.13	28.18	26.96	27.49	22.39	23.40
QMSum	Rouge L	19.24	19.46	18.80	18.82	15.19	15.51	12.84	12.96
MultiNews	Rouge L	24.33	24.22	25.43	26.23	25.42	25.66	20.91	21.19
TriviaQA	F1	13.04	12.33	23.19	22.38	10.57	9.17	13.31	11.42
Passage Retrieval	Acc	**7.50**	4.25	**9.08**	6.50	3.03	3.85	3.00	3.45

表1. LongBench数据集上的准确性基准。我们将异常值标记为粗体，其性能高于与之对应的2.5。

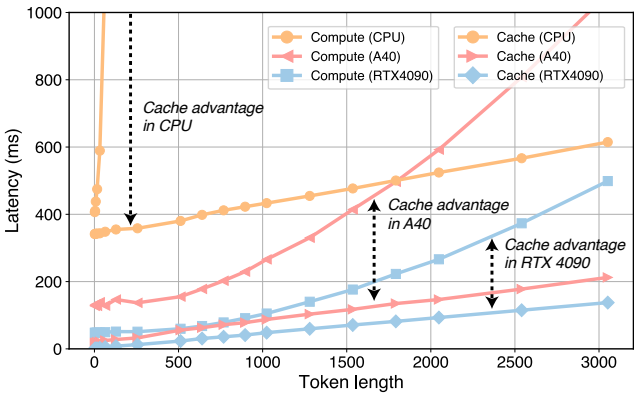


图5. 缓存优势：GPU和CPU中计算和缓存开销的比较。虽然注意力计算成本呈平方增长，但注意力状态内存复制开销（*i.e.*，提示缓存）呈线性上升。在这里，GPU直接从CPU内存加载提示模块。

LLM	BERT	Falcon 1B	Llama 7B	Llama 13B
MB/token	0.03	0.18	0.50	0.78
LLM	MPT 30B	Falcon 40B	Llama 70B	Falcon 180B
MB/token	1.31	1.87	2.5	4.53

表 2. 缓存单个令牌的内存开销

数据集具有不同的序列长度，假设所有提示都已缓存。我们比较了使用Intel i9-13900K CPU和两块GPU（NVIDIA RTX 4090和A40）以及Llama2 7B模型的Prompt Cache与常规KV Cache的TTFT延迟。对于CPU和GPU，CPU内存用于提示模块存储。

二次改进：我们在图5中展示的发现表明，KV Cache的延迟随着序列长度的增加而呈二次增长，而Prompt Cache的内存复制成本则线性增长。这意味着Prompt Cache的延迟优势（两条曲线之间的差距）随着序列长度的增加而呈二次扩展。这个差异在CPU上比在GPU上更为明显，因为CPU经历了更高的注意力计算延迟，而Prompt Cache的开销*i.e.*、GPU中的主机到设备的内存拷贝和CPU中的主机到主机的内存拷贝之间的差异并不显著。在具有5K标记的注意力状态下，主机到主机、主机到设备和设备到设备的内存拷贝延迟分别为3.79毫秒、5.34毫秒和0.23毫秒。

模型大小的影响：此外，随着模型参数大小的增长，KV缓存的计算开销也随之增加。例如，从7B模型迁移到13B模型，在3K的令牌长度下增加了220毫秒的延迟，而Prompt Cache仅增加了30毫秒。这一差异源于LLM复杂性也随着隐藏维度大小呈二次方增长。例如，注意力的FLOPS为$6nd^2 + 4n^2d$，用于预填充操作。这表明，Prompt Cache相对于KV Cache的优势也随着模型大小（*i.e.*，隐藏维度）呈二次方增加。

端到端延迟：由于 Prompt Cache 仅减少 TTFT，因此其对接收完整 LLM 响应所需时间的影响随着生成的 token 数量增加而减小。例如，在 RTX 4090 上使用 Llama 7B 进行 3K 上下文时，Prompt Cache 将 TTFT 从 900 毫秒提升至 90 毫秒，而 token 生成时间或后续 token 的时间 (TTST) 在 KV Cache 和 Prompt Cache 之间保持一致，平均为每个 token 32 毫秒，无论 token 长度如何。尽管如此，更快的响应时间对用户体验和整体端到端延迟有积极贡献 (Lew et al., 2018; Liu et al., 2023b)。例如，考虑到 Prompt Cache 将 TTFT 从 900 毫秒提升至 90 毫秒，这相当于在相同时间内生成 25 个额外的 token。另一个因素是，Prompt Cache 允许在同一批次内共享注意力状态，正如我们在 §3.4 中讨论的那样。根据工作负载特征，Prompt Cache 可以通过利用减少内存占用所带来的更大批量大小来提高整体吞吐量。例如，假设有 100 个请求，每个请求都有一个 2K token 提示。如果所有提示共享相同的 1K token 模块，结合分页注意力等方法，Prompt Cache 可以将内存占用减少 50%。

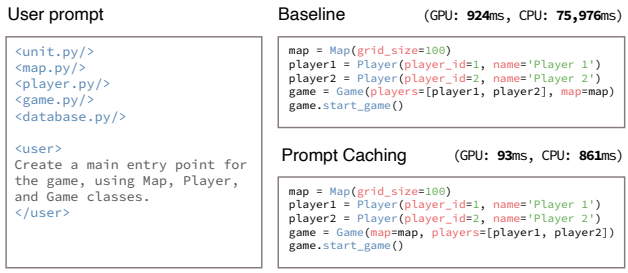


Figure 6. Code generation using Prompt Cache: Each source file becomes a prompt module, allowing users to "import" files in their prompt context with minimal overhead.

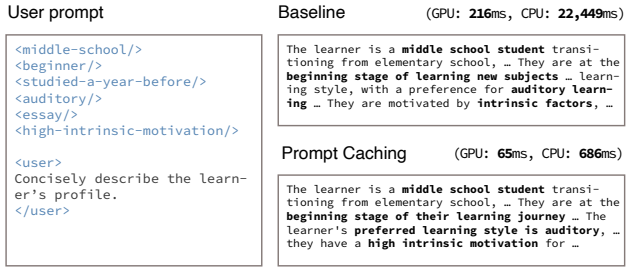


Figure 7. Personalization example: Six categories each have five traits. Traits in the same category are grouped in `<union>`.

working batch size and thus higher throughput.

5.5 Memory Overhead

The memory overhead associated with Prompt Cache is proportional to the aggregated number of tokens cached. This overhead can be determined by referencing both the prompt schema and the target LLM. In Table 2, we elucidate the memory overhead on a per-token basis, under the assumption of utilizing a 16-bit precision for floating points. For compact models, such as Falcon 1B, caching a document containing 1K tokens would require approximately 180 MB of memory. If there are hundreds of prompt modules, the combined memory consumption would range in the tens of gigabytes—a quantity within the memory confines of server-grade GPUs. Conversely, for larger models like Llama 70B, caching a 1K length module would command a substantial 2.5 GB of memory per document, which leaves CPU memory as the only option for prompt module storage. Given these considerations, compression methods for attention states (Zhang et al., 2023) remain an avenue for future research in prompt caching techniques.

5.6 Applications of Prompt Cache

We demonstrate the expressiveness of PML with example use cases that require more complicated prompt structures and advanced features (§3.2) than the LongBench benchmarks: (*i*) multiple modules in a query, (*ii*) union, and (*iii*) parameterization. Furthermore, these tasks underscore the notable latency reduction as the number of cached tokens increases in such complicated use cases. Across use cases, we provide a qualitative assessment of the output by juxtaposing cached and non-cached generation, showcasing that Prompt Cache maintains output quality, along with the latency reductions achieved by Prompt Cache. We use Llama2 7B and store prompt modules in the local memory (*i.e.*, GPU memory for GPU inference). The full schema for these tasks is available in Appendix B.

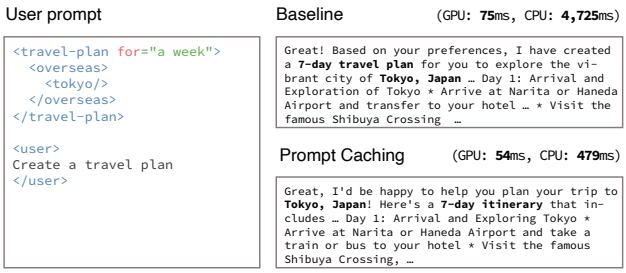


Figure 8. Parameterized prompts: The `<travel-plan>` is reconfigured at runtime while maintaining caching efficiency, offering flexible prompt structuring.

5.6.1 *Code Generation*

LLMs are commonly used for code generation (Guo et al., 2023; Liu et al., 2023a), aiding programmers in either assisting with or directly generating code. Currently available methods, such as Copilot (GitHub, 2023), typically focus on individual source files. Prompt Cache, however, can extend this to multiple files leveraging a modular nature of source code. For instance, each class or function could be a distinct prompt module. Figure 6 illustrates multi-source code generation using CodeLlama 7B (Rozière et al., 2023). We treat individual classes like `Unit`, `Map`, `Game`, and `Player` as prompt modules in our schema for game programming. Users can then include these prompt modules whenever they need them in the code. There is a $4\times$ improvement in TTFT latency on GPUs while the output remains identical.

5.6.2 *Personalization*

Figure 7 shows the latency benefits and the output quality of Prompt Cache in a personalization use case. Personalization is integral to many recommender systems (Wu et al., 2023), finding prominent applications in LLM contexts such as education, content recommendations, and targeted marketing. We highlight the efficacy of feature-based personalization through Prompt Cache. Here, personalization hinges on a defined set of features. Each feature is represented as a distinct prompt module, with relationships between features denoted using union tags such as grade level, proficiency,

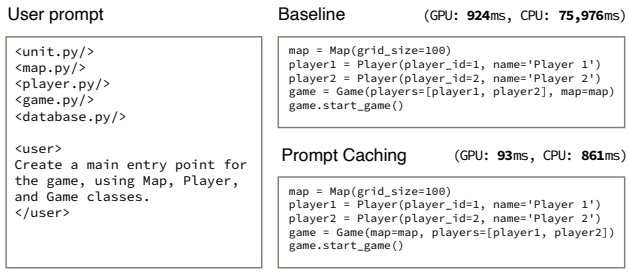


图6. 使用提示缓存的代码生成：每个源文件成为一个提示模块，允许用户在其提示上下文中“导入”文件，开销最小。

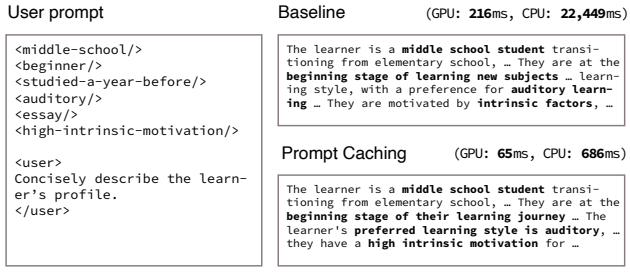


图7. 个性化示例：六个类别每个都有五个特征。同一类别中的特征被分组在 <union> 中。

工作批量大小，从而提高吞吐量。

5.5 内存开销

与提示缓存相关的内存开销与缓存的令牌总数成正比。这个开销可以通过参考提示模式和目标 LLM 来确定。在表 2 中，我们阐明了在假设使用 16 位浮点精度的情况下，每个令牌的内存开销。对于紧凑型模型，如 Falcon 1B，缓存一个包含 1K 令牌的文档大约需要 180 MB 的内存。如果有数百个提示模块，综合内存消耗将达到数十 GB——这是服务器级 GPU 的内存限制范围内的数量。相反，对于像 Llama 70B 这样的大型模型，缓存一个 1K 长度的模块每个文档将需要高达 2.5 GB 的内存，这使得 CPU 内存成为提示模块存储的唯一选择。考虑到这些因素，注意状态的压缩方法（Zhang et al., 2023）仍然是提示缓存技术未来研究的一个方向。

5.6 提示缓存的应用

我们通过示例用例展示了PML的表现力，这些用例需要比LongBench基准测试更复杂的提示结构和高级功能（§3.2）。

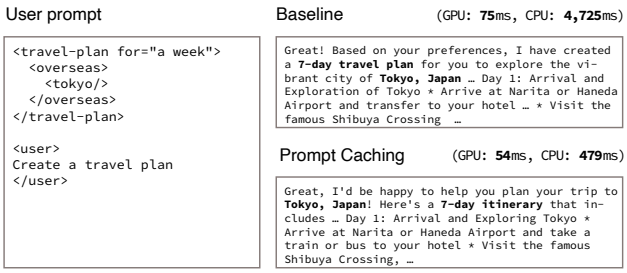


图8. 参数化提示：<旅行计划>在运行时重新配置，同时保持缓存效率，提供灵活的提示结构。

标记: (*i*) 查询中的多个模块, (*ii*) 联合, 和 (*iii*) 参数化。此外，这些任务强调了在如此复杂的用例中，随着缓存令牌数量的增加，显著降低延迟。在用例中，我们通过对比缓存生成和非缓存生成提供输出的定性评估，展示了 Prompt Cache 维持输出质量，以及通过 Prompt Cache 实现的延迟减少。我们使用 Llama2 7B，并将提示模块存储在本地内存中 (*i.e.*, 用于 GPU 推理的 GPU 内存)。这些任务的完整方案可在附录 B 中找到。

5.6.1 Code Generation

LLM 通常用于代码生成（Guo 等，2023；Liu 等，2023a），帮助程序员协助或直接生成代码。目前可用的方法，如 Copilot（GitHub，2023），通常专注于单个源文件。然而，Prompt Cache 可以利用源代码的模块化特性将其扩展到多个文件。例如，每个类或函数都可以是一个独立的提示模块。图 6 说明了使用 CodeLlama 7B（Rozière 等，2023）进行多源代码生成。我们将 Unit、Map、Game 和 Player 等单独类视为我们游戏编程方案中的提示模块。用户可以在代码中需要时包含这些提示模块。在 GPU 上，TTFT 延迟提高了 4×，而输出保持不变。

5.6.2 Personalization

图7展示了在个性化用例中Prompt Cache的延迟优势和输出质量。个性化是许多推荐系统的核心（Wu et al., 2023），在教育、内容推荐和目标营销等LLM背景下找到了显著的应用。我们通过Prompt Cache强调基于特征的个性化的有效性。在这里，个性化依赖于一组定义好的特征。每个特征被表示为一个独特的提示模块，特征之间的关系使用联合标签表示，例如年级水平、熟练度，

learning history, learning style, and assessment type.

5.6.3 *Parameterized Prompts*

In Figure 8, we show a trip planning use case leveraging parameterization (§3.2). The schema used in this use case encompasses one adjustable parameter to specify the trip duration along with two union modules to select the destination. Users can reuse the templated prompt with custom parameters, enjoying lower TTFT latency and the same quality of LLM response enabled by Prompt Cache.

6 Conclusions and Future Work

We introduce Prompt Cache, an acceleration technique based on the insight that attention states can be reused across LLM prompts. Prompt Cache utilizes a prompt schema to delineate such reused text segments, formulating them into a modular and positionally coherent structure termed "prompt modules". This allows LLM users to incorporate these modules seamlessly into their prompts, thereby leveraging them for context with negligible latency implications. Our evaluations on benchmark data sets indicate TTFT latency reductions of up to 8× on GPUs and 60× on CPUs.

For future work, we plan on using Prompt Cache as a building block for future LLM serving systems. Such a system could be equipped with GPU cache replacement strategies optimized to achieve the latency lower bound made possible by Prompt Cache. Different strategies for reducing host-to-device memory overhead can also be beneficial, such as the integration of compression techniques in the KV cache, or utilization of grouped query attention. Another promising exploration GPU primitives for sharing attention states across concurrent requests, as we breifly discussed in §3.4. This can not only reduce the TTFT latency but also time-per-output-token (TPOT) latency by packing more requests into a single batch. Finally, Prompt Cache can directly accelerate in-context retrieval augmented generation (RAG) methods, where the information retrieval system basically serves as a database of prompt modules. Prompt Cache can be particularly useful for latency-sensitive RAG applications in real-time question answering and dialogue systems.

Acknowledgements

This work is supported in part by NSF Athena AI Institute (Award #2112562), NSF Award #2047220, and Yale University. This work used the Delta system at the National Center for Supercomputing Applications (NCSA) through allocation CIS230289 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants (Award #2138259, #2138286, #2138307, #2137603, and #2138296).

References

Aminabadi, R. Y., Rajbhandari, S., Awan, A. A., Li, C., Li, D., Zheng, E., Ruwase, O., Smith, S., Zhang, M., Rasley, J., and He, Y. Deepspeed- inference: Enabling efficient inference of transformer models at unprecedented scale. In Wolf, F., Shende, S., Culhane, C., Alam, S. R., and Jagode, H. (eds.), *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, November 13-18, 2022*, pp. 46:1–46:15. IEEE, 2022. doi: 10.1109/SC41404.2022.00051. URL https://doi.org/10.1109/SC41404.2022.00051.

Bai, Y., Lv, X., Zhang, J., Lyu, H., Tang, J., Huang, Z., Du, Z., Liu, X., Zeng, A., Hou, L., Dong, Y., Tang, J., and Li, J. Longbench: A bilingual, multitask benchmark for long context understanding. *CoRR*, abs/2308.14508, 2023. doi: 10.48550/ARXIV.2308.14508. URL https://doi.org/10.48550/arXiv.2308.14508.

Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The long-document transformer. *CoRR*, abs/2004.05150, 2020. URL https://arxiv.org/abs/2004.05150.

Beurer-Kellner, L., Fischer, M., and Vechev, M. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1946–1969, 2023.

Cui, J., Li, Z., Yan, Y., Chen, B., and Yuan, L. Chatlaw: Open-source legal large language model with integrated external knowledge bases, 2023.

Dao, T., Fu, D., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

Driess, D., Xia, F., Sajjadi, M. S. M., Lynch, C., Chowdhery, A., Ichter, B., Wahid, A., Tompson, J., Vuong, Q., Yu, T., Huang, W., Chebotar, Y., Sermanet, P., Duckworth, D., Levine, S., Vanhoucke, V., Hausman, K., Toussaint, M., Greff, K., Zeng, A., Mordatch, I., and Florence, P. Palm-e: An embodied multimodal language model, 2023.

Dufter, P., Schmitt, M., and Schütze, H. Position information in transformers: An overview. *Computational Linguistics*, 48(3):733–763, 2022.

Fabbri, A. R., Li, I., She, T., Li, S., and Radev, D. Multi-news: A large-scale multi-document summarization dataset and abstractive hierarchical model. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 1074–1084, 2019.

学习历史、学习风格和评估类型。

5.6.3 Parameterized Prompts

在图8中，我们展示了一个利用参数化（§3.2）的旅行规划用例。该用例中使用的模式包含一个可调参数，用于指定旅行持续时间，以及两个联合模块用于选择目的地。用户可以使用自定义参数重用模板提示，享受更低的TTFT延迟和由提示缓存提供的相同质量的LLM响应。

6 结论与未来工作

我们介绍了提示缓存（Prompt Cache），这是一种加速技术，基于注意力状态可以在大型语言模型（LLM）提示之间重用的洞察。提示缓存利用提示模式来划分这些重用的文本片段，将它们构造成一种称为“提示模块”的模块化和位置一致的结构。这使得LLM用户能够将这些模块无缝地融入他们的提示中，从而在几乎没有延迟影响的情况下利用它们作为上下文。我们在基准数据集上的评估表明，GPU上的TTFT延迟减少高达8×，CPU上减少高达60×。

对于未来的工作，我们计划将 Prompt Cache 作为未来 LLM 服务系统的构建模块。这样的系统可以配备优化的 GPU 缓存替换策略，以实现由 Prompt Cache 提供的延迟下限。减少主机到设备内存开销的不同策略也可能是有益的，例如在 KV 缓存中集成压缩技术，或利用分组查询注意力。另一个有前景的探索是 GPU 原语，用于在并发请求之间共享注意力状态，正如我们在 §3.4 中简要讨论的。这不仅可以减少 TTFT 延迟，还可以通过将更多请求打包到单个批次中来降低每个输出令牌 (TPOT) 延迟。最后，Prompt Cache 可以直接加速上下文检索增强生成 (RAG) 方法，其中信息检索系统基本上充当提示模块的数据库。Prompt Cache 对于实时问答和对话系统中的延迟敏感 RAG 应用特别有用。

致谢

本工作部分得到了国家科学基金会雅典娜人工智能研究所（奖项编号 #2112562）、国家科学基金会奖项 #2047220 和耶鲁大学的支持。本工作使用了国家超级计算应用中心（NCSA）的 Delta 系统，通过高级网络基础设施协调生态系统：服务与支持（ACCESS）计划的分配 CIS230289，该计划得到了国家科学基金会的资助（奖项编号 #2138259、#2138286、#2138307、#2137603 和 #2138296）。

参考文献

Aminabadi, R. Y., Rajbhandari, S., Awan, A. A., Li, C., Li, D., Zheng, E., Ruwase, O., Smith, S., Zhang, M., Rasley, J., 和 He, Y. Deepspeed-推理：在前所未有的规模上实现高效的变换器模型推理。在 Wolf, F., Shende, S., Culhane, C., Alam, S. R., 和 Jagode, H. (编辑), *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, November 13-18, 2022*，第 46:1–46:15 页。IEEE，2022。doi: 10.1109/SC41404.2022.00051。网址 https://doi.org/10.1109/SC41404.2022.00051。Bai, Y., Lv, X., Zhang, J., Lyu, H., Tang, J., Huang, Z., Du, Z., Liu, X., Zeng, A., Hou, L., Dong, Y., Tang, J., 和 Li, J. Longbench：一个用于长上下文理解的双语多任务基准。*CoRR*，abs/2308.14508，2023。doi: 10.48550/ARXIV.2308.14508。网址 https://doi.org/10.48550/arXiv.2308.14508。Beltagy, I., Peters, M. E., 和 Cohan, A. Longformer：长文档变换器。*CoRR*，abs/2004.05150，2020。网址 https://arxiv.org/abs/2004.05150。Beurer-Kellner, L., Fischer, M., 和 Vechev, M. 提示即编程：大型语言模型的查询语言。*Proceedings of the ACM on Programming Lan- guages*，7 (PLDI)：1946–1969，2023。Cui, J., Li, Z., Yan, Y., Chen, B., 和 Yuan, L. Chatlaw：集成外部知识库的开源法律大型语言模型，2023。Dao, T., Fu, D., Ermon, S., Rudra, A., 和 Ré, C. Flashat- tention：快速且内存高效的精确注意力与 IO 感知。*Advances in Neural Information Process- ing Systems*，35:16344–16359，2022。Driess, D., Xia, F., Sajjadi, M. S. M., Lynch, C., Chowdhery, A., Ichter, B., Wahid, A., Tompson, J., Vuong, Q., Yu, T., Huang, W., Chebotar, Y., Sermanet, P., Duckworth, D., Levine, S., Vanhoucke, V., Hausman, K., Toussaint, M., Greff, K., Zeng, A., Mordatch, I., 和 Florence, P. Palm-e：一个具身的多模态语言模型，2023。Dufter, P., Schmitt, M., 和 Schütze, H. 变换器中的位置信息：概述。*Computational Linguistics*，48(3)：733–763，2022。Fabbri, A. R., Li, I., She, T., Li, S., 和 Radev, D. Multi-news：一个大规模多文档摘要数据集和抽象层次模型。在 *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*，第 1074–1084 页，2019。

Feng, Y., Jeon, H., Blagojevic, F., Guyot, C., Li, Q., and Li, D. Attmemo : Accelerating transformers with memoization on big memory systems, 2023.

GitHub. Github copilot · your ai pair programmer, 2023. URL https://github.com/features/copilot.

Guidence. A guidance language for controlling large language models. https://github.com/guidance-ai/guidance, 2023.

Guo, D., Xu, C., Duan, N., Yin, J., and McAuley, J. Longcoder: A long-range pre-trained language model for code completion. *arXiv preprint arXiv:2306.14893*, 2023.

Ho, X., Nguyen, A.-K. D., Sugawara, S., and Aizawa, A. Constructing a multi-hop qa dataset for comprehensive evaluation of reasoning steps. In *Proceedings of the 28th International Conference on Computational Linguistics*, pp. 6609–6625, 2020.

Huang, L., Cao, S., Parulian, N., Ji, H., and Wang, L. Efficient attentions for long document summarization. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 1419–1436, 2021.

Huang, W., Abbeel, P., Pathak, D., and Mordatch, I. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. *arXiv preprint arXiv:2201.07207*, 2022.

Joshi, M., Choi, E., Weld, D. S., and Zettlemoyer, L. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1601–1611, 2017.

Keles, F. D., Wijewardena, P. M., and Hegde, C. On the computational complexity of self-attention, 2022.

Kočiskỳ, T., Schwarz, J., Blunsom, P., Dyer, C., Hermann, K. M., Melis, G., and Grefenstette, E. The narrativeqa reading comprehension challenge. *Transactions of the Association for Computational Linguistics*, 6:317–328, 2018.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. *arXiv preprint arXiv:2309.06180*, 2023.

Lew, Z., Walther, J. B., Pang, A., and Shin, W. Interactivity in online chat: Conversational contingency and response latency in computer-mediated communication. *Journal of Computer-Mediated Communication*, 23(4):201–221, 2018.

Liu, T., Xu, C., and McAuley, J. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*, 2023a.

Liu, Y., Li, H., Du, K., Yao, J., Cheng, Y., Huang, Y., Lu, S., Maire, M., Hoffmann, H., Holtzman, A., Ananthanarayanan, G., and Jiang, J. Cachegen: Fast context loading for language model applications, 2023b.

MosaicML. Introducing mpt-7b: A new standard for open-source, commercially usable llms, 2023. URL www.mosaicml.com/blog/mpt-7b. Accessed: 2023-05-05.

Nay, J. J., Karamardian, D., Lawsky, S. B., Tao, W., Bhat, M., Jain, R., Lee, A. T., Choi, J. H., and Kasai, J. Large language models as tax attorneys: A case study in legal capabilities emergence, 2023.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Penedo, G., Malartic, Q., Hesslow, D., Cojocaru, R., Cappelli, A., Alobeidli, H., Pannier, B., Almazrouei, E., and Launay, J. The refinedweb dataset for falcon llm: outperforming curated corpora with web data, and web data only. *arXiv preprint arXiv:2306.01116*, 2023.

Pope, R., Douglas, S., Chowdhery, A., Devlin, J., Bradbury, J., Levskaya, A., Heek, J., Xiao, K., Agrawal, S., and Dean, J. Efficiently scaling transformer inference, 2022.

Press, O., Smith, N. A., and Lewis, M. Train short, test long: Attention with linear biases enables input length extrapolation. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=R8sQPpGCv0.

Qin, Y., Hu, S., Lin, Y., Chen, W., Ding, N., Cui, G., Zeng, Z., Huang, Y., Xiao, C., Han, C., Fung, Y. R., Su, Y., Wang, H., Qian, C., Tian, R., Zhu, K., Liang, S., Shen, X., Xu, B., Zhang, Z., Ye, Y., Li, B., Tang, Z., Yi, J., Zhu, Y., Dai, Z., Yan, L., Cong, X., Lu, Y., Zhao, W., Huang, Y., Yan, J., Han, X., Sun, X., Li, D., Phang, J., Yang,

Feng, Y., Jeon, H., Blagojevic, F., Guyot, C., Li, Q., 和 Li, D. Attmemo : 在大内存系统上通过记忆化加速变换器, 2023.

GitHub。GitHub Copilot · 你的 AI 编程伙伴，2023。网址 https://github.com/features/copilot。

指导。用于控制大型语言模型的指导语言。https://github.com/guidance-ai/guidance，2023。

郭, D., 许, C., 段, N., 尹, J., 和 麦考利, J. Long-coder: 一种用于代码补全的长距离预训练语言模型。*arXiv preprint arXiv:2306.14893*, 2023.

Ho, X., Nguyen, A.-K. D., Sugawara, S., 和 Aizawa, A. 构建一个多跳问答数据集以全面评估推理步骤。在 *Proceedings of the 28th International Conference on Computational Linguistics*，第 6609–6625 页，2020 年。

黄, L., 曹, S., Parulian, N., 吉, H., 和王, L. 长文档摘要的高效注意力机制. 在 *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 第1419–1436页, 2021年.

黄, W., Abbeel, P., Pathak, D., 和 Mordatch, I. 语言模型作为零-shot 规划者：为具身智能体提取可操作知识。*arXiv preprint arXiv:2201.07207*, 2022.

Joshi, M., Choi, E., Weld, D. S., 和 Zettlemoyer, L. Trivi-a qa：一个大规模远程监督的阅读理解挑战数据集。在 *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*，第 1601–1611 页，2017 年。

Keles, F. D., Wijewardena, P. M. 和 Hegde, C. 关于自注意力的计算复杂性，2022。

Kočiskỳ, T., Schwarz, J., Blunsom, P., Dyer, C., Hermann, K. M., Melis, G., 和 Grefenstette, E. 叙事问答阅读理解挑战。*Transactions of the Association for Computational Linguistics*, 6:317–328, 2018.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., 和 Stoica, I. 大语言模型服务的高效内存管理与分页注意力。*arXiv preprint arXiv:2309.06180*, 2023.

Lew, Z., Walther, J. B., Pang, A., 和 Shin, W. 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。 *Journal of Computer-Mediated Communication*, 23(4):201–221, 2018.

刘, T., 许, C., 和 麦考利, J. Repobench: 基于仓库级代码自动补全系统的基准测试。 *arXiv preprint arXiv:2306.03091*, 2023a.

刘, Y., 李, H., 杜, K., 姚, J., 程, Y., 黄, Y., 陆, S., 马尔, M., 霍夫曼, H., 霍尔茨曼, A., 阿南塔-纳拉扬, G., 和姜, J. Cachegen: 语言模型应用的快速上下文加载, 2023b.

MosaicML。介绍 mpt-7b：开放源代码、可商业使用的 llms 的新标准，2023。网址 www.mosaicml.com/blog/mpt-7b。访问时间：2023-05-05。

奈，J. J.，卡拉马尔迪安，D.，劳斯基，S. B.，陶，W.，巴特，M.，贾因，R.，李，A. T.，崔，J. H.，和笠井，J. 大型语言模型作为税务律师：法律能力出现的案例研究，2023。

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., 和 Chintala, S. Pytorch：一种命令式风格的高性能深度学习库。在 *Advances in Neural Information Processing Systems 32*，第 8024–8035 页。Curran Associates, Inc.，2019。网址 http://papers.neurips.cc/paper/9015-pytorch-a n-imperative-style-high-performance-d eep-learning-library.pdf。

Penedo, G., Malartic, Q., Hesslow, D., Cojocaru, R., Cappelli, A., Alobeidli, H., Pannier, B., Almazrouei, E., 和 Launay, J. 精炼的网络数据集用于猎鹰 LLM：超越仅使用网络数据的策划语料库。*arXiv preprint arXiv:2306.01116*，2023。

教皇, R., 道格拉斯, S., Chowdhery, A., 德夫林, J., 布拉德伯里, J., 列夫斯卡亚, A., 希克, J., 肖, K., 阿格拉瓦尔, S., 和 迪恩, J. 高效扩展变换器推理, 2022.

Press, O.，Smith, N. A. 和 Lewis, M. 训练短，测试长：带有线性偏差的注意力使输入长度外推成为可能。在 *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*。OpenReview.net，2022。网址 https://openreview.net/forum?id=R8sQPpGCv0。

秦, Y., 胡, S., 林, Y., 陈, W., 丁, N., 崔, G., 曾, Z., 黄, Y., 肖, C., 韩, C., 冯, Y. R., 苏, Y., 王, H., 钱, C., 田, R., 朱, K., 梁, S., 沈, X., 许, B., 张, Z., 叶, Y., 李, B., 唐, Z., 易, J., 朱, Y., 戴, Z., 闫, L., 丛, X., 陆, Y., 赵, W., 黄, Y., 闫, J., 韩, X., 孙, X., 李, D., 方, J., 杨,

C., Wu, T., Ji, H., Liu, Z., and Sun, M. Tool learning with foundation models. *CoRR*, abs/2304.08354, 2023. doi: 10.48550/ARXIV.2304.08354. URL https://doi.org/10.48550/arXiv.2304.08354.

Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al. Improving language understanding by generative pre-training. 2018.

Rasmy, L., Xiang, Y., Xie, Z., Tao, C., and Zhi, D. Med-bert: pretrained contextualized embeddings on large-scale structured electronic health records for disease prediction. *npj Digit. Medicine*, 4, 2021. doi: 10.1038/S41746-021-00455-Y. URL https://doi.org/10.1038/s41746-021-00455-y.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Canton-Ferrer, C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950, 2023. doi: 10.48550/ARXIV.2308.12950. URL https://doi.org/10.48550/arXiv.2308.12950.

Scao, T. L., Fan, A., Akiki, C., Pavlick, E., Ilic, S., Hesslow, D., Castagné, R., Luccioni, A. S., Yvon, F., Gallé, M., Tow, J., Rush, A. M., Biderman, S., Webson, A., Ammanamanchi, P. S., Wang, T., Sagot, B., Muennighoff, N., del Moral, A. V., Ruwase, O., Bawden, R., Bekman, S., McMillan-Major, A., Beltagy, I., Nguyen, H., Saulnier, L., Tan, S., Suarez, P. O., Sanh, V., Laurençon, H., Jernite, Y., Launay, J., Mitchell, M., Raffel, C., Gokaslan, A., Simhi, A., Soroa, A., Aji, A. F., Alfassy, A., Rogers, A., Nitzav, A. K., Xu, C., Mou, C., Emezue, C., Klamm, C., Leong, C., van Strien, D., Adelani, D. I., and et al. BLOOM: A 176b-parameter open-access multilingual language model. *CoRR*, abs/2211.05100, 2022. doi: 10.48550/ARXIV.2211.05100. URL https://doi.org/10.48550/arXiv.2211.05100.

Shen, J. T., Yamashita, M., Prihar, E., Heffernan, N. T., Wu, X., and Lee, D. Mathbert: A pre-trained language model for general NLP tasks in mathematics education. *CoRR*, abs/2106.07340, 2021. URL https://arxiv.org/abs/2106.07340.

Sheng, Y., Zheng, L., Yuan, B., Li, Z., Ryabinin, M., Chen, B., Liang, P., Ré, C., Stoica, I., and Zhang, C. Flexgen: High-throughput generative inference of large language models with a single GPU. In Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., and Scarlett, J. (eds.), *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pp. 31094–31116. PMLR, 2023. URL https://proceedings.mlr.press/v202/sheng23a.html.

Steinberg, E., Jung, K., Fries, J. A., Corbin, C. K., Pfohl, S. R., and Shah, N. H. Language models are an effective representation learning technique for electronic health record data. *J. Biomed. Informatics*, 113:103637, 2021. doi: 10.1016/J.JBI.2020.103637. URL https://doi.org/10.1016/j.jbi.2020.103637.

Su, J., Lu, Y., Pan, S., Wen, B., and Liu, Y. Roformer: Enhanced transformer with rotary position embedding. *CoRR*, abs/2104.09864, 2021. URL https://arxiv.org/abs/2104.09864.

Tay, Y., Dehghani, M., Bahri, D., and Metzler, D. Efficient transformers: A survey. *ACM Comput. Surv.*, 55(6):109:1–109:28, 2023. doi: 10.1145/3530811. URL https://doi.org/10.1145/3530811.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M.-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models, 2023.

Trivedi, H., Balasubramanian, N., Khot, T., and Sabharwal, A. Musique: Multihop questions via single-hop question composition. *Transactions of the Association for Computational Linguistics*, 10:539–554, 2022.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need, 2023.

White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., and Schmidt, D. C. A prompt pattern catalog to enhance prompt engineering with chatgpt. *CoRR*, abs/2302.11382, 2023. doi: 10.48550/ARXIV.2302.11382. URL https://doi.org/10.48550/arXiv.2302.11382.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., Drame, M.,

C., Wu, T., Ji, H., Liu, Z., 和 Sun, M. 基础模型的工具学习。 *CoRR*, abs/2304.08354, 2023. doi: 10.48550/ARXIV.2304.08354. URL https: //doi.org/10.48550/arXiv.2304.08354.

拉德福德，A.，纳拉西曼，K.，萨利曼斯，T.，苏茨克弗，I.，等。通过生成预训练提高语言理解。2018。

Rasmy, L., Xiang, Y., Xie, Z., Tao, C., 和 Zhi, D. Med-bert：在大规模结构化电子健康记录上预训练的上下文嵌入用于疾病预测。*npj Digit. Medicine*, 4, 2021. doi: 10.1038/S41746-021 -00455-Y. URL https://doi.org/10.1038/s4 1746-021-00455-y.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Canton-Ferrer, C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., 和 Synnaeve, G. 代码骆驼：开源代码基础模型。*CoRR*，abs/2308.12950，2023。doi: 10.48550/ARX IV.2308.12950。网址 https://doi.org/10.485 50/arXiv.2308.12950。

Scao, T. L., Fan, A., Akiki, C., Pavlick, E., Ilic, S., Hesslow, D., Castagné, R., Luccioni, A. S., Yvon, F., Gallé, M., Tow, J., Rush, A. M., Biderman, S., Webson, A., Am- mana manchi, P. S., Wang, T., Sagot, B., Muennighoff, N., del Moral, A. V., Ruwase, O., Bawden, R., Bek- man, S., McMillan-Major, A., Beltagy, I., Nguyen, H., Saulnier, L., Tan, S., Suarez, P. O., Sanh, V., Laurençon, H., Jernite, Y., Launay, J., Mitchell, M., Raffel, C., Gokaslan, A., Simhi, A., Soroa, A., Aji, A. F., Alfassy, A., Rogers, A., Nitzav, A. K., Xu, C., Mou, C., Emezue, C., Klamm, C., Leong, C., van Strien, D., Adelani, D. I., 等. BLOOM：一个176b参数的开放获取多语言模型。*CoRR*，abs/2211.05100，2022。doi：10.48550/ARXIV.2211.05100。网址 https://doi.org/10.48550/arXiv.2211.05100。

沈, J. T., 山下, M., Prihar, E., Heffernan, N. T., 吴, X., 和 李, D. Mathbert: 一种用于数学教育中一般自然语言处理任务的预训练语言模型。 *CoRR*, abs/2106.07340, 2021. URL https://arxiv.org/ abs/2106.07340.

Sheng, Y., Zheng, L., Yuan, B., Li, Z., Ryabinin, M., Chen, B., Liang, P., Ré, C., Stoica, I., 和 Zhang, C. Flexgen: 使用单个 GPU 的大语言模型的高通量生成推理。在 Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., 和 Scarlett, J. (编辑), *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, *Proceedings of Machine Learning Research* 第 202 卷, 页码。31094–31116. PMLR, 2023. URL https://procee dings.mlr.press/v202/sheng23a.html.

Steinberg, E., Jung, K., Fries, J. A., Corbin, C. K., Pfohl, S. R., 和 Shah, N. H. 语言模型是电子健康记录数据的有效表示学习技术。*J. Biomed. Informatics*, 113:103637, 2021. doi: 10.1016/J.JBI.2020.103637. URL https://doi.org/10.1016/j.jbi.2020.103637.

苏, J., 陆, Y., 潘, S., 温, B., 和 刘, Y. Roformer: 带有旋转位置嵌入的增强型变换器. *CoRR*, abs/2104.09864, 2021. URL https://arxiv. org/abs/2104.09864.

Tay, Y., Dehghani, M., Bahri, D., 和 Metzler, D. 高效变换器：一项调查。 *ACM Comput. Surv.*, 55(6):109:1– 109:28, 2023. doi: 10.1145/3530811. URL https: //doi.org/10.1145/3530811.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M.-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., 和 Scialom, T. Llama 2：开放基础和微调聊天模型，2023。

Trivedi, H., Balasubramanian, N., Khot, T., 和 Sabharwal, A. Musique: 通过单跳问题组合的多跳问题。 *Transactions of the Association for Computational Linguistics*, 10:539–554, 2022.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., 和 Polosukhin, I. 注意力就是你所需要的一切，2023。

白, J., 傅, Q., 海斯, S., 桑德伯恩, M., 奥利亚, C., 吉尔伯特, H., 埃尔纳沙尔, A., 斯宾塞-史密斯, J., 和施密特, D. C. 一个提示模式目录，以增强与chatgpt的提示工程。 *CoRR*, abs/2302.11382, 2023. doi: 10.48550/ARXIV.2302.11382. URL https://doi.org/10.48550/arXiv.2302.11382.

狼, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., Drame, M.,

Lhoest, Q., and Rush, A. M. Huggingface's transformers: State-of-the-art natural language processing, 2020.

Wu, L., Zheng, Z., Qiu, Z., Wang, H., Gu, H., Shen, T., Qin, C., Zhu, C., Zhu, H., Liu, Q., Xiong, H., and Chen, E. A survey on large language models for recommendation. *CoRR*, abs/2305.19860, 2023. doi: 10.48550/ARXIV.2305.19860. URL https://doi.org/10.48550/arXiv.2305.19860.

Yang, Z., Qi, P., Zhang, S., Bengio, Y., Cohen, W., Salakhutdinov, R., and Manning, C. D. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380, 2018.

Zhang, Z., Sheng, Y., Zhou, T., Chen, T., Zheng, L., Cai, R., Song, Z., Tian, Y., Ré, C., Barrett, C., Wang, Z., and Chen, B. H_2o: Heavy-hitter oracle for efficient generative inference of large language models, 2023.

Zhong, M., Yin, D., Yu, T., Zaidi, A., Mutuma, M., Jha, R., Hassan, A., Celikyilmaz, A., Liu, Y., Qiu, X., et al. Qmsum: A new benchmark for query-based multi-domain meeting summarization. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 5905–5921, 2021.

Lhoest, Q. 和 Rush, A. M. Huggingface 的变换器：最先进的自然语言处理，2020。Wu, L., Zheng, Z., Qiu, Z., Wang, H., Gu, H., Shen, T., Qin, C., Zhu, C., Zhu, H., Liu, Q., Xiong, H. 和 Chen, E. 关于推荐的大型语言模型的调查。*CoRR*，abs/2305.19860，2023。doi: 10.48550/ARXIV.2305.19860。URL https://doi.org/10.48550/arXiv.2305.19860。Yang, Z., Qi, P., Zhang, S., Bengio, Y., Cohen, W., Salakhut-dinov, R. 和 Manning, C. D. Hotpotqa：一个用于多样化、可解释的多跳问答的数据集。在 *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*，第 2369–2380 页，2018。Zhang, Z., Sheng, Y., Zhou, T., Chen, T., Zheng, L., Cai, R., Song, Z., Tian, Y., Ré, C., Barrett, C., Wang, Z. 和 Chen, B. H_2o：用于大型语言模型高效生成推理的重击预言机，2023。Zhong, M., Yin, D., Yu, T., Zaidi, A., Mutuma, M., Jha, R., Hassan, A., Celikyilmaz, A., Liu, Y., Qiu, X. 等。Qm-sum：一个基于查询的多领域会议摘要的新基准。在 *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*，第 5905–5921 页，2021。