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.

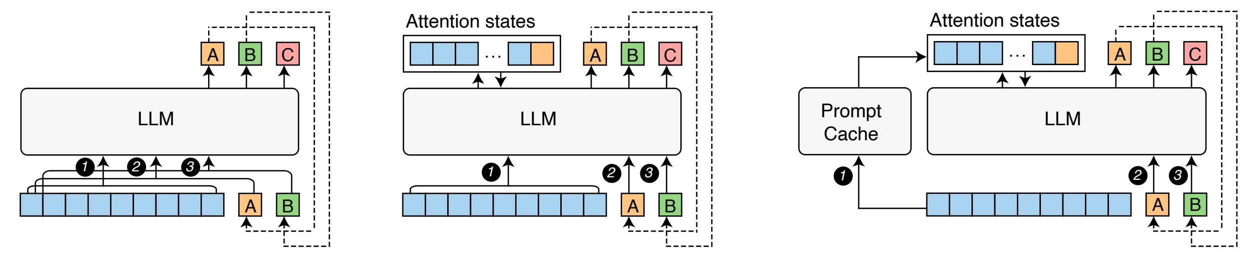
Reusing attention states is a popular strategy for accelerating the service of a single prompt (Pope et al., 2022). The existing approach, often referred to as *Key-Value (KV) Cache*, reuses the key-value attention states of input tokens during the autoregressive token generation. This eliminates the 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 5^{th} *MLSys Conference*, Santa Clara, CA, USA, 2024. Copyright 2024 by the author(s).



(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 ($\boxed{A}$) (❶). It then appends the generated token ($\boxed{A}$) to the prompt to predict the next token ($\boxed{B}$) (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 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: $s_1, \dots, s_n$, and the subsequently generated k tokens as $s_{n+1}, \dots, s_{n+k}$. In naive autoregressive token generation, the attention states $\{(k_1, v_1), \dots, (k_{n+k}, v_{n+k})\}$ 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.

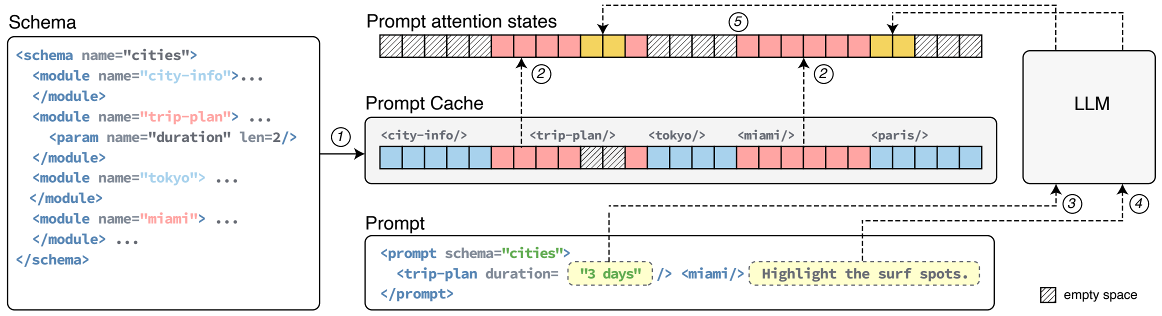
3 Design of Prompt Cache

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-



```
<schema name="cities">
  <module name="city-info">...
  </module>
  <module name="trip-plan"> ...
    <param name="duration" len=2/>
  </module>
  <module name="tokyo"> ...
 </module>
  <module name="miami"> ...
  </module> ...
</schema>
```

```
<prompt schema="cities">
  <trip-plan duration= "3 days" /> <miami/> Highlight the surf spots.
</prompt>
```

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 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.

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.

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.

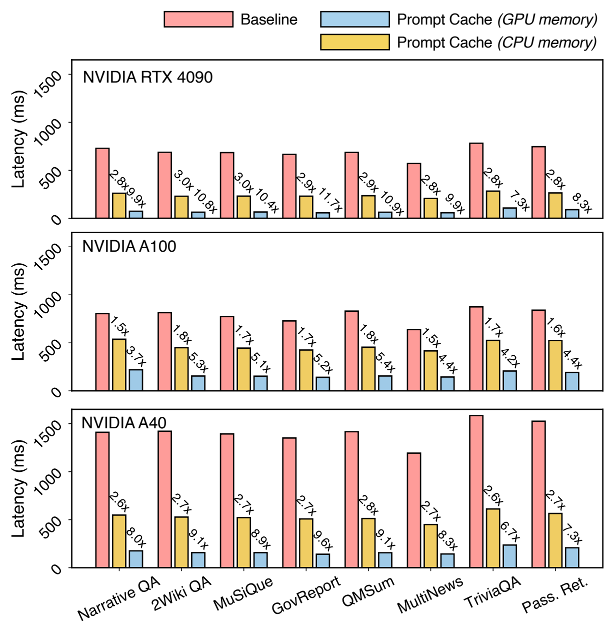


Figure 3. GPU latency measurements: time-to-first-token (TTFT) for eight LongBench datasets across three NVIDIA GPUs.

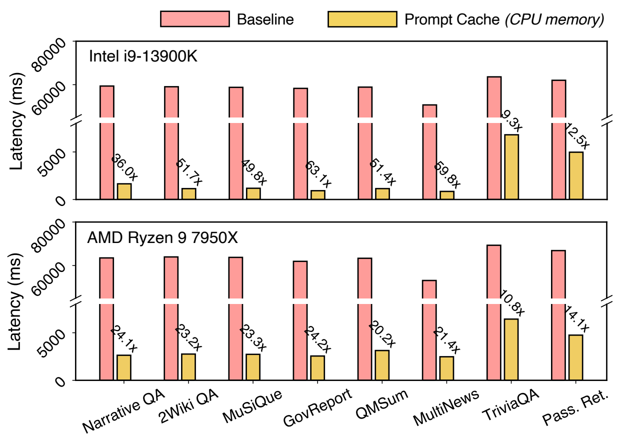


Figure 4. CPU latency measurements: time-to-first-token (TTFT) for eight LongBench datasets across two CPUs.

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

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.

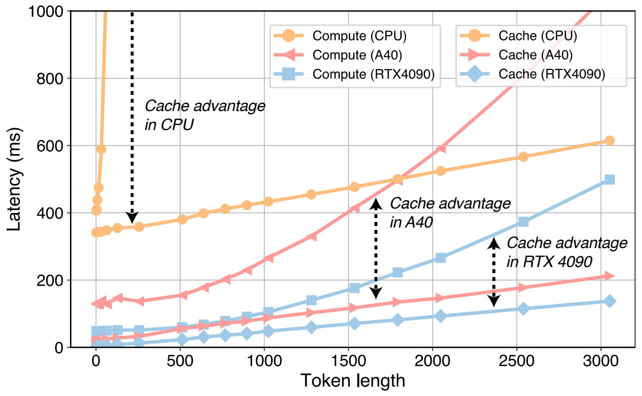


Figure 5. Cache advantage: A comparison of computational and caching overheads in GPUs and CPUs. While attention computation cost increases quadratically, the attention state memory copy overhead (*i.e.*, Prompt Cache) rises linearly. Here, GPUs load prompt modules directly from CPU memory.

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

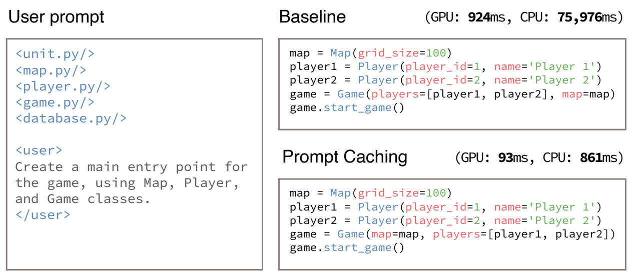
Table 2. Memory overhead of caching a single token

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


User prompt

```
<unit.py/>
<map.py/>
<player.py/>
<game.py/>
<database.py/>

<user>
Create a main entry point for
the game, using Map, Player,
and Game classes.
</user>
```

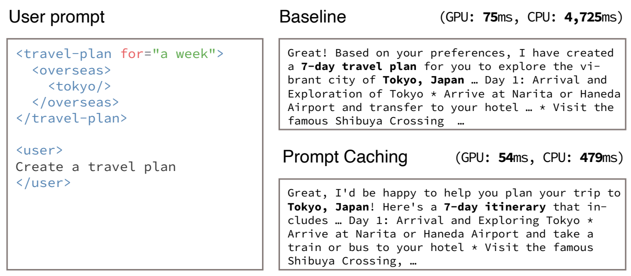
Baseline (GPU: **924**ms, CPU: **75,976**ms)

```
map = Map(grid_size=100)
player1 = Player(player_id=1, name='Player 1')
player2 = Player(player_id=2, name='Player 2')
game = Game(players=[player1, player2], map=map)
game.start_game()
```

Prompt Caching (GPU: **93**ms, CPU: **861**ms)

```
map = Map(grid_size=100)
player1 = Player(player_id=1, name='Player 1')
player2 = Player(player_id=2, name='Player 2')
game = Game(map=map, players=[player1, player2])
game.start_game()
```

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.


User prompt

```
<travel-plan for="a week">
  <overseas>
    <tokyo/>
  </overseas>
</travel-plan>

<user>
Create a travel plan
</user>
```

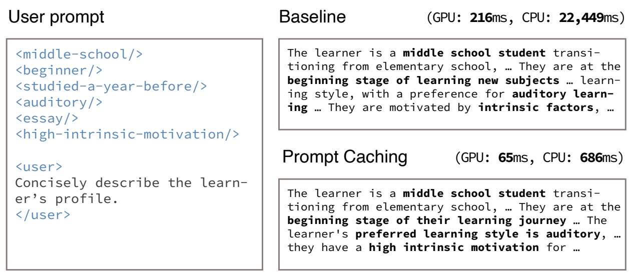
Baseline (GPU: **75**ms, CPU: **4,725**ms)

Great! Based on your preferences, I have created a **7-day travel plan** for you to explore the vibrant city of **Tokyo, Japan** … Day 1: Arrival and Exploration of Tokyo * Arrive at Narita or Haneda Airport and transfer to your hotel … * Visit the famous Shibuya Crossing …

Prompt Caching (GPU: **54**ms, CPU: **479**ms)

Great, I'd be happy to help you plan your trip to **Tokyo, Japan**! Here's a **7-day itinerary** that includes … Day 1: Arrival and Exploring Tokyo * Arrive at Narita or Haneda Airport and take a train or bus to your hotel * Visit the famous Shibuya Crossing, …

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


User prompt

```
<middle-school/>
<beginner/>
<studied-a-year-before/>
<auditory/>
<essay/>
<high-intrinsic-motivation/>

<user>
Concisely describe the learn-
er's profile.
</user>
```

Baseline (GPU: **216**ms, CPU: **22,449**ms)

The learner is a **middle school student** transitioning from elementary school, … They are at the **beginning stage of learning new subjects** … learning style, with a preference for **auditory learning** … They are motivated by **intrinsic factors**, …

Prompt Caching (GPU: **65**ms, CPU: **686**ms)

The learner is a **middle school student** transitioning from elementary school, … They are at the **beginning stage of their learning journey** … The learner's **preferred learning style is auditory**, … they have a **high intrinsic motivation** for …

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.

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, 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.

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, 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., 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.