ChunkAttention: Efficient Self-Attention with Prefix-Aware KV Cache and Two-Phase Partition

Anonymous ACL submission

Abstract

Self-attention is an essential component of large language models(LLMs) but a significant source of inference latency for long sequences. In multi-tenant LLMs serving scenarios, the compute and memory operation cost of self-attention can be optimized by using the probability that multiple LLM requests have shared system prompts in prefixes. In this paper, we introduce ChunkAttention, a prefix-aware self-attention module that can detect matching prompt prefixes across multiple requests and share their key/value tensors in memory at runtime to improve the memory utilization of KV cache. This is achieved by breaking monolithic key/value tensors into smaller chunks and structuring them into the auxiliary prefix tree. Consequently, on top of the prefix-tree based KV cache, we design an efficient self-attention kernel, where a two-phase partition algorithm is implemented to improve the data locality during self-attention computation in the presence of shared system prompts. Experiments show that ChunkAttention can speed up the self-attention kernel by 3.2-4.8× compared to the start-of-the-art implementation, with the length of the system prompt ranging from 1024 to 4096. [1]

1 Introduction

Over the last few years, Large Language Models (LLMs) have developed various capabilities, from in-context learning (Dong et al., 2023, 2022) to chain-of-thought reasoning (Chu et al., 2023; Wei et al., 2022), and achieved remarkable success in a wide range of natural language processing related tasks (Chang et al., 2023). Representive models are the GPT (Radford et al., 2018, 2019; Brown et al., 2020; OpenAI, 2023c), LLaMA (Touvron et al., 2023b), PaLM (Anil et al., 2023) and Gemini (Gemini, 2023) series. Following the success of ChatGPT and GPT store, LLM-based applications start to surge, and the demand to optimize LLM's inference cost has been a new area of research interest (Kim et al., 2023; Sheng et al., 2023; Aminabadi et al., 2022).

The self-attention module, as one of the critical components in LLMs, has poor performance during inference (Table 1) since it performs intensive memory operations on key/value tensors of context tokens (KV cache) and is memory-bound (Williams et al., 2009; Jin et al., 2023). The memory complexity grows linearly with context length. As the demand for more context tokens has been a trend (32K for GPT-4), the performance gets worse (OpenAI, 2023c). KV cache additionally restricts the batch size and system throughput. For instance, using FP16, the KV cache for each token in GPT-3(175B) requires 4.5MB of memory. The memory of an inference server with 8*A100 (80G) can only hold 70000 tokens or 35 sequences of 2K context tokens.

On the other hand, the system prompt as a common practice in designing LLM based applications, leads to redundancy in KV cache (Anthropic, 2023). Typically, due to high training and inference costs, LLMs are pre-trained and deployed in a multi-tenant architecture for multiple applications to share. System prompts are essential for LLMs to gain each application's domain knowledge and generate better results (White et al., 2023; Zhou et al., 2023). Since multiple requests share identical system prompts, there is significant overlap in prompt prefixes (§2.1).

An important question is whether we can leverage the sharing characteristic of system prompts to make the self-attention module faster and more memory efficient. To our knowledge, the only related work is a proposal by Kwon et al. (2023), in which the service provider reserves memory for key/value tensors of a set of predefined system prompts from application developers. The

[1] Code will be publicly available on Github after review.

proposal has limitations: i) predefined system prompts are static and inflexible in frequent refreshes for large-scale deployments since both application developers and the service provider are involved in the operation loop; ii) there is memory waste in case of long system prompts and low hit rate; iii) no work has been done to optimize the self-attention kernel in the presence of shared system prompts.

To fill the gap, we propose ChunkAttention, a novel self-attention module featuring the prefix-aware KV cache (PAKV) and two-phase partition (TPP). KV cache in ChunkAttention is a prefix tree built with chunked context tokens and key/value tensors. Thus, the KV cache is prefix-aware and can dynamically detect and remove redundancy at runtime without human involvement. The KV cache only stores key/value tensors of sequences currently in decoding and has zero memory waste. In addition, the prefix-tree structure provides context for ChunkAttention to redesign a highly-optimized self-attention kernel with two-phase partition: chunk-first phase and sequence-first phase. Query tensors from sequences with matching prompt prefixes are batched together to perform attention with key/value tensors.

The main contributions of this paper are as follows: i) we reveal that system prompts can be long (§2.1), providing opportunities for optimizing self-attention; ii) we propose to use prefix tree to implement KV cache, which is out-of-the-box, scalable and robust in terms of redundancy removal; iii) we implement a two-phase partition algorithm to speed up self-attention kernel on prefix-aware KV cache; iv) we prove the feasibility and empirically quantify the gain self-attention can achieve from shared system prompts under various system configurations. Our experiments show that ChunkAttention can be significantly faster as the length of shared system prompts grows and has no performance degradation without shared system prompts, compared to existing highly optimized implementations.

2 Preliminaries

2.1 Shared System Prompt

One paradigm in designing LLM-based applications has been the introduction of system prompt (Anthropic, 2023). It provides instructions, few-shot examples (Dong et al., 2022), and external knowledge as context for LLMs to generate better results. The final prompt to LLMs is a concatenation of system prompt and task-specific input. The system prompt is shared between multiple requests and can be very long. This can be observed in various LLM-based applications, from online chatbots to offline experiments.

Batch Size	Roofline	QKV Projection	Self Attention	MLP
1	FLOPs($\times 10^6$)	100.66	33.57	270.53
	MOPs($\times 10^6$)	100.70	33.85	270.62
	Arithmetic Intensity	1.00	**0.99**	1.00
	Latency(µs)	88.44	**17.82**	160.77
32	FLOPs($\times 10^6$)	3221.23	1074.27	8657.04
	MOPs($\times 10^6$)	101.71	1083.18	273.43
	Arithmetic Intensity	31.67	**0.99**	31.66
	Latency(µs)	90.02	**687.74**	209.82
64	FLOPs($\times 10^6$)	6442.45	2148.53	17314.09
	MOPs($\times 10^6$)	102.76	2166.36	276.33
	Arithmetic Intensity	62.69	**0.99**	62.66
	Latency(µs)	98.04	**1358.40**	217.79

Table 1: Complexity analysis of key modules in each decoder layer when decoding one single token. Llama2 7B, 2048 context tokens, FP16, A100 (80G). The self-attention module has low arithmetic intensity (Williams et al., 2009) and high latency. FLOPs: floating point operations. MOPs: memory operations or memory bytes accessed. Arithmetic Intensity: FLOPs/MOPs.

Toolformer or using external tools becomes an essential skill for LLMs to get up-to-date information or perform precise math calculations (Schick et al., 2023; Li et al., 2023). It is implemented by plugins in ChatGPT-like online chatbot applications (OpenAI, 2023a). Equivalent capability is provided by GPT series models through function calling (OpenAI, 2023b). Under the hood, available function specifications are silently injected into the system prompt (OpenAI, 2023d). Experiments indicate that with 6 plugins activated, the token length of the shared system prompt can reach up to 1766 (Appendix A).

Another source of shared system prompts is the offline research-focused experiments conducted on LLMs. In these scenarios, researchers frequently create a large number of templated requests with identical instructions, examples, or external knowledge and issue them to LLMs quickly. Example work includes: i) Chameleon (Lu et al., 2023) reuses policy planning and tool invocation prompts for compositional reasoning on the ScienceQA and TabMWP datasets; ii) CREATOR (Qian et al., 2023) constructs a collection of questions from TabMWP and MATH datasets using a chain-of-thought (CoT) prompt template; iii) PDF-Triage (Saad-Falcon et al., 2023) injects the PDF document metadata into prompt and runs multiple question-answering (QA) tasks over the document;

System	Usage of Prompt	#shared prompt tokens	
		avg	max
Chameleon	Tools definition and examples [1]	1324	2626
CREATOR	CoT examples [2]	879	2492
PDFTriage	PDF document metadata	4257	N.A.
ToolQA	Tools definition and examples [3]	1432	1432

Table 2: Shared prompt tokens in system prompt, tokenized by OpenAI's tiktoken tokenizer library (OpenAI, 2023e).

iv) ToolQA (Zhuang et al., 2023) further releases a QA dateset and reuses the system prompt for evaluations of QA with LLMs. Table 2 shows statistics on shared token counts of system prompts.

2.2 LLM Inferencing

The typical inference process of LLMs consists of two stages: prefilling and decoding (Sheng et al., 2023). After receiving a sequence $S = [t_1, ..., t_{n_p}]$, the server starts to prefill. During prefilling, it feeds all n_p prompt tokens $t_1, ..., t_{n_p}$ into LLMs, computes the attention key/value tensors, and caches them to speed up subsequent computations. Then, the server performs decoding. Decoding is auto-regressive, and the input token to LLMs is the completion token(or output token) generated from the previous decoding iteration. The process continues until the end-of-sequence token or maximum completion tokens are generated.

When the server is decoding b (batch size) sequences $S_1, ..., S_b$ simultaneously, although they are in different iterations, the server can still perform batching at the granularity of iteration and predict the next tokens for all sequences together, rather than separately, which is known as iteration-based batching (Gao et al., 2018; Yu et al., 2022; Silfa et al., 2022). Specifically, iteration-based batching concatenates last input tokens of multiple sequences (one token per sequence) $t^{(1)}, ..., t^{(b)} (t^{(i)} \in S_i)$ into a single input $\boldsymbol{T}$, and computes the QKV projection before self-attention, the output projection and multilayer perceptron after self-attention. The self-attention in the middle has no shared weights and needs to be computed independently for each sequence. During decoding, new sequences can join, and completed sequences can leave, significantly increasing the possibility of forming big batches. Iteration-based batching has been implemented by vLLM (Kwon et al., 2023) and the text-generation-inference server (HuggingFace, 2023). The ChunkAttention in this paper assumes that iteration-based batching is enabled to form batches for its kernel to run efficiently.

3 Our Approach

3.1 Prefix Aware KV Cache (PAKV)

Traditionally, KV cache is stored in dense tensors of size $b \times h \times n \times d$ where b is the batch size, h is the number of heads, n is the sequence length, and d is the head dimension size.

When multiple sequences share common prefix tokens, key/value tensors are the same and thus can be shared in memory. For example, a particular LLM inference server receives sequence $S_i = [t_1, ..., t_{n_s}, t_{n_s+1}, ..., t_{n_p}]$ first, and then receives sequence $S_j = [t_1, ..., t_{n_s}, t'_{n_s+1}, ..., t'_{n_p}]$. KV cache for $t_1, ..., t_{n_s}$ can only have one physical copy in memory.

Given the property, we argue that the KV cache should be made prefix-aware, which is to organize the KV cache of all sequences under decoding into a prefix tree. Precisely, we slice monolithic key/value tensors contiguous in memory along the sequence length dimension. Figure 1 shows the structure of the KV cache stored in a prefix tree. Each node defines a chunk C storing three essential elements: i) a segment of c context tokens shared by sequences $S_i, ..., S_j$ to enable prefix tree operations; ii) a slice of key tensor of size $b \times h \times c \times d$ for the c tokens; iii) the corresponding slice of value tensor. Each path in the prefix tree defines a sequence. Multiple trees (a forest) may exist in the server simultaneously. For instance, application developers design different system prompts.

There are three possible scenarios during inference: i) new sequence joins, ii) completed sequence leaves, and iii) all sequences decode one token together. Each scenario can be translated into prefix tree operations. When a new sequence joins, the prefix tree is searched and updated to insert a new path. When a completed sequence leaves, the prefix tree is updated to delete its path. At each decoding iteration, we append new tokens into leaf chunks or grow a new chunk when the leaf chunk is full.

[1] https://github.com/lupantech/chameleon-llm/blob/main/run_tabmwp/demos/prompt_policy.py

[2] https://github.com/qiancheng0/CREATOR/blob/main/MATH/prompt_lib/prompt_cot.md

[3] https://github.com/night-chen/ToolQA/blob/main/benchmark/chameleon/run_toolqa/demos/prompt_policy.py

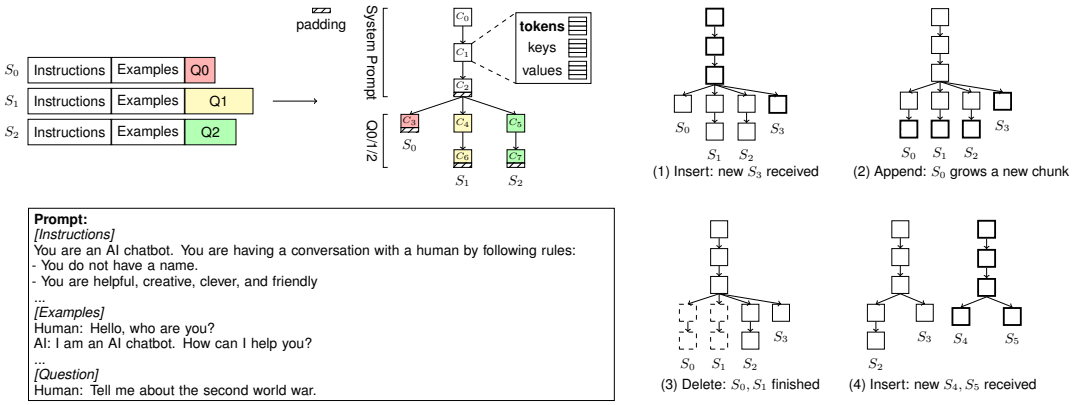


Figure 1: KV cache in prefix tree. The instructions and examples in prompts of S_0, S_1, S_2 are common and sharable. Questions are different and not sharable. Some memory is unused due to alignment.

Given a fixed chunk size c, memory management is efficient. In ChunkAttention, the pool-based memory allocator is adopted by default (Hill, 1992; Trebino, 2016). It keeps track of both a used and a free chunk list. When a new chunk is requested, the allocator returns a chunk from the free list or allocates fresh memory from the operating system (OS). Unused chunks are returned to the allocator once a sequence is completed, but the allocator does not release memory to the OS, preventing unnecessary memory allocations. Some memory space for alignment is unused. Given that the sequence length is n, the memory loss is bounded by $(c-1)/n$.

By sharing common prefixes, the number of sequences that can be processed simultaneously is increased by approximately $1/(1-r)$. The sharing ratio r is defined by the percentage of shared tokens $n_s/(n_p+n_c)$, and n_c is the completion token count. In memory-limited inference scenarios, this helps increase the batch size and thus improve throughput.

The parent-child relationship defines the subset of sequences each chunk covers. The root node covers all sequences, and the leaf nodes cover only one. A key property of the prefix tree is that sequences covered by each chunk in the prefix tree are contiguous in the sequence index dimension. Therefore, slicing the query tensor in self-attention is particularly efficient during kernel computation, which will be discussed in more detail in the next section.

3.2 Two-phase Partition (TPP)

In this section, we dive into the self-attention kernel implementation on top of the unique prefix-aware KV cache storage.

During prefilling, we perform a prefix lookup to avoid repeated computation of KV projection and position embedding for matched prompt prefixes. For mismatched suffix tokens, KV projection and position embedding are still computed, and the key/value tensors are chunked and inserted into the prefix tree. Then we apply existing highly optimized self-attention kernels, *e.g.*, FlashAttention (Dao, 2023), on the entire key/value tensors.

During iterative decoding, self-attention is divided into chunk-first and sequence-first phases. The two phases focus on different slices of the query tensor, KV cache chunks, and use different parallelization strategies. The process is shown in Figure 2. Since the head dimension is always partitioned, it is omitted and implicit in our discussion.

Chunk-first Phase. In the chunk-first phase, we only process chunks shared by multiple sequences. Since GPUs have more streaming multiprocessors (108 for A100) than the number of heads (32 for Llama 7B), and partitioning by heads under-utilizes hardware resources, we perform additional partition on keys/values. Chunking already provides convenience. The online softmax algorithm is adopted to avoid the synchronization requirement between partitions (Milakov and Gimelshein, 2018; Dao, 2023).

The computation is performed by traversing shared chunks in the prefix tree, executing the partial attention kernel *partial_attn* and saving the partial attention results into memory, as shown in Algorithm 1. The number of sequences (batch size) is denoted by b. $\boldsymbol{Q} \in \mathbb{R}^{b \times d}$ is the queries formed by concatenating the last token of all b sequences in the latest decoding iteration.

The implementation of *partial_attn* is given by

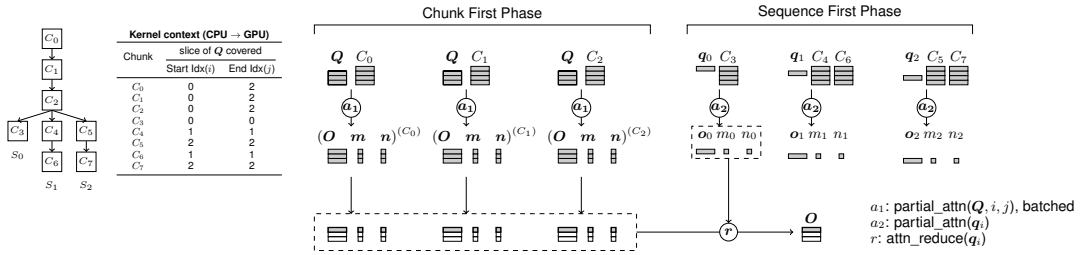


Figure 2: Two-phase partition kernel in ChunkAttention. The server is decoding sequences S_0, S_1, and S_2. They share chunks C_0, C_1 and C_2. In the chunk-first phase, queries $\boldsymbol{q}_0$, $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$ are batched for self-attention with C_0, C_1 and C_2. Partial attention result $\boldsymbol{O}^{(C)}$, $\boldsymbol{m}^{(C)}$ and $\boldsymbol{n}^{(C)}$ are saved into memory. In the sequence-first phase, $\boldsymbol{o}_i$, m_i, and n_i for each sequence are restored, and we continue processing the remaining chunks with respect to $\boldsymbol{q}_i$ only.

Algorithm 1 Self Attention: Chunk First (partition chunks)

Require: $\boldsymbol{Q} \in \mathbb{R}^{b \times d}$ (query), T(prefix tree)
Ensure: $\boldsymbol{O} \in \mathbb{R}^{b \times d}$ (attention output)
  function ATTNCHUNKFIRST($\boldsymbol{Q}, T$)
    Get chunks $C_1, \ldots, C_k$ in T that are shared by multiple sequences
    $\boldsymbol{O}, \boldsymbol{m}, \boldsymbol{n} \leftarrow \boldsymbol{0}, \boldsymbol{0}, \boldsymbol{0}$
    for $C \leftarrow C_1$ to C_k **do**
      $\boldsymbol{K}^{(C)}, \boldsymbol{V}^{(C)} \leftarrow$ key, value cache stored in C
      $i, j \leftarrow$ start index, end index of sequences covered by C
      $\boldsymbol{O}^{(C)}, \boldsymbol{m}^{(C)}, \boldsymbol{n}^{(C)} \leftarrow$ ***partial_attn***$(\boldsymbol{Q}, \boldsymbol{K}^{(C)}, \boldsymbol{V}^{(C)}, i, j)$
      Save partial attention result $\boldsymbol{O}^{(C)}, \boldsymbol{m}^{(C)}, \boldsymbol{n}^{(C)}$ to memory
    end for
  end function

Eqn. (1). It computes the partial attention result $(\boldsymbol{O}, \boldsymbol{m}, \boldsymbol{n})^{(C)}$ with respect to each chunk C independently, thus it can be parallelized. $\boldsymbol{Q}_{i:j,:}$ is a slice of $\boldsymbol{Q}$ for sequences ranging from i to j which share the KV cache stored in chunk C. The maximum attention weights vector $\boldsymbol{M}^{(C)}$ is the row-wise max over the last dimension of attention weights $\boldsymbol{W}^{(C)}$. The softmax normalization term $\boldsymbol{n}^{(C)}$ is the row-wise sum over the last dimension of $\boldsymbol{E}^{(C)}$. $\boldsymbol{M}^{(C)}$ and $\boldsymbol{n}^{(C)}$ are auxiliary variables introduced to further cumulate partial attention results of multiple chunks.

$$
\begin{aligned}
\boldsymbol{W}^{(C)} &= \boldsymbol{Q}_{i:j,:}\boldsymbol{K}^{(C)} \in \mathbb{R}^{(j-i)\times c} \\
\boldsymbol{m}^{(C)} &= \max\left(\boldsymbol{W}^{(C)}\right) \in \mathbb{R}^{(j-i)} \\
\boldsymbol{E}^{(C)} &= \exp\left(\boldsymbol{W}^{(C)} - \boldsymbol{m}^{(C)} \cdot \boldsymbol{1}^T\right) \in \mathbb{R}^{(j-i)\times c} \\
\boldsymbol{n}^{(C)} &= \mathrm{sum}\left(\boldsymbol{E}^{(C)}\right) \in \mathbb{R}^{(j-i)} \\
\boldsymbol{O}^{(C)} &= \boldsymbol{E}^{(C)}\boldsymbol{V}^{(C)} \in \mathbb{R}^{(j-i)\times d}
\end{aligned} \tag{1}
$$

The *partial_attn* efficiently accesses shared KV cache memory since self-attentions for multiple sequences are batched. The batching happens at a granularity of dot-product between queries $\boldsymbol{Q}_{i,:}, \ldots, \boldsymbol{Q}_{j,:}$ of sequences $S_i, \ldots, S_j$ and shared $\boldsymbol{K}^{(C)}/\boldsymbol{V}^{(C)}$. In addition to improved data locality, another advantage of batching is to turn the query from a vector into a matrix, allowing efficient matrix multiplications with tensor cores (Choquette et al., 2021).

Sequence-first Phase. In the sequence-first phase, we load partial attention results of shared chunks from the chunk-first phase and continue processing chunks related to one specific sequence. We partition sequences, and each $\boldsymbol{q}$ handled by the sequence-first kernel is a vector by slicing the i-th row of $\boldsymbol{Q}$, as shown in Algorithm 2.

Algorithm 2 Self Attention: Sequence First (partition sequences)

Require: $\boldsymbol{Q} \in \mathbb{R}^{b \times d}$ (query), T(prefix tree)
Ensure: $\boldsymbol{O} \in \mathbb{R}^{b \times d}$ (attention output)
  function ATTNSEQFIRST($\boldsymbol{Q}, T$)
    for $\boldsymbol{q} \leftarrow \boldsymbol{q}_1$ to $\boldsymbol{q}_b$ **do**
      $\boldsymbol{o}, m, n \leftarrow \boldsymbol{0}, 0, 0$
      Get partial attn results $(\boldsymbol{O}, \boldsymbol{m}, \boldsymbol{n})^{(C_1)}, \ldots, (\boldsymbol{O}, \boldsymbol{m}, \boldsymbol{n})^{(C_k)}$
      for $(\boldsymbol{O}, \boldsymbol{m}, \boldsymbol{n})^{(C)} \leftarrow (\boldsymbol{O}, \boldsymbol{m}, \boldsymbol{n})^{(C_1)}$ to $(\boldsymbol{O}, \boldsymbol{m}, \boldsymbol{n})^{(C_k)}$ **do**
        Partial attn of $\boldsymbol{q}$: $(\boldsymbol{o}, m, n)^{(C)} \leftarrow$ slicing $(\boldsymbol{O}, \boldsymbol{m}, \boldsymbol{n})^{(C)}$
        attn_reduce$(\boldsymbol{o}^{(C)}, m^{(C)}, n^{(C)}, \boldsymbol{o}, m, n)$
      end for
      Get chunks $C_{k+1}, C_{k+2} \ldots, C_l$ in T with respect to $\boldsymbol{q}$ only
      for $C \leftarrow C_{k+1}$ to C_l **do**
        $\boldsymbol{K}^{(C)}, \boldsymbol{V}^{(C)} \leftarrow$ key, value cache stored in C
        $i \leftarrow$ sequence index of $\boldsymbol{q}$
        $(\boldsymbol{o}, m, n)^{(C)} \leftarrow$ ***partial_attn***$(\boldsymbol{q}, \boldsymbol{K}^{(C)}, \boldsymbol{V}^{(C)}, i, i+1)$
        attn_reduce$(\boldsymbol{o}^{(C)}, m^{(C)}, n^{(C)}, \boldsymbol{o}, m, n)$
      end for
    end for
  end function

The *attn_reduce* repeatly merges partial attention result of one chunk $(\boldsymbol{o}, m, n)^{(C)}$ produced by the *partial_attn* into the cumulative attention result $(\boldsymbol{o}, m, n)$ by scaling them with $x^{(C)}$ and $y^{(C)}$ respectively. Eqn. (2) shows the process. $\boldsymbol{O}_{i,:}$, $\boldsymbol{m}_i$ and $\boldsymbol{n}_i$ are slices for sequence of index i. The final attention output is given by $\boldsymbol{O}/\boldsymbol{n}$ element-wise.

The sequence-first phase is efficient in concurrency since *partial_attn* and *attn_reduce* are performed locally, without communication between thread blocks. However, without the partial attention results generated by the chunk-first phase, it needs to load shared chunks from RAM b times, which adds significant MOPs. The two-phase partition algorithm balances parallelization and cache

locality.

$$
\begin{aligned}
x^{(C)} &= \exp\left(m^{(C)} - \max\left(m^{(C)}, \boldsymbol{m}_i\right)\right) \in \mathbb{R} \\
y^{(C)} &= \exp\left(\boldsymbol{m}_i - \max\left(m^{(C)}, \boldsymbol{m}_i\right)\right) \in \mathbb{R} \\
\boldsymbol{O}_{i,:} &= x^{(C)}\boldsymbol{o}^{(C)} + y^{(C)}\boldsymbol{O}_{i,:} \in \mathbb{R}^d \\
\boldsymbol{n}_i &= x^{(C)}n^{(C)} + y^{(C)}\boldsymbol{n}_i \in \mathbb{R} \\
\boldsymbol{m}_i &= \max\left(m^{(C)}, \boldsymbol{m}_i\right) \in \mathbb{R}
\end{aligned} \tag{2}
$$

3.3 Further Optimizations

The prefix tree structure is maintained in CPU memory. To run the two-phase patation kernel on GPU, we must generate certain context from the prefix tree, including the chunk C, the start index i and end index j of its covered sequences, and copy the context (C, i, j) from CPU to GPU memory. For example, in Figure 2, we need to generate and copy $(C_0/C_1/C_2, 0, 2)$, $(C_3, 0, 0)$, $(C_4/C_6, 1, 1)$, and $(C_5/C_7, 2, 2)$. ChunkAttention manages the overhead in two ways: i) latency hiding. The context generation step on CPU can be overlapped with other kernels on GPU prior to self-attention. ii) lazy context copy. The prefix tree does not change at every decoding iteration. We can cache the context in GPU memory and only trigger memory copy when the tree structure changes. Triggers are chunk full for every c iterations, new sequence joining, and completed sequence leaving. The overhead is amortized.

The temporary memory allocated for partial attention results in the chunk-first phase can be eliminated by executing *attn_reduce* right after *partial_attn* to directly merge partial attention results into the final result. Since multiple shared chunks with a parent-child relationship in the prefix tree write into the same slice of $(\boldsymbol{O}, \boldsymbol{m}, \boldsymbol{n})$, *attn_reduce* needs to be serialized. On GPU devices, atomic operations are heavy, and we do not use this approach. However, on CPU devices, the overhead of serializing is insignificant, and reduction can be implemented using spin locks.

4 Experiments

The evaluations are conducted at both the self-attention microkernel level and the end-to-end GPT-style model level. The microkernel level evaluations only capture time spent in the self-attention CUDA kernel. The side effects of PAKV and TPP, *e.g.*, prefix tree operations, are captured in end-to-end evaluations. We run all experiments with NVIDIA A100 GPU (80G) and CUDA 11.8.

4.1 Microkernel Evaluation

Baselines. We select four self-attention implementations as baselines: Naive PyTorch implementation by the formula $\text{softmax}(\boldsymbol{QK}^T/\sqrt{d})\boldsymbol{V}$, the memory-efficient self-attention implemented in xformers (Lefaudeux et al., 2022), FlashAttention integrated in PyTorch (Dao et al., 2022), and PagedAttention in vLLM (Kwon et al., 2023).

Since Naive, xformers, and FlashAttn are all built on monolithic KV tensors, they cannot be prefix-aware by partially sharing KV cache of prompt prefixes. PagedAttn does not implement PAKV either. However, its paging design enables us to manually create a fixed page table, mapping virtually non-shared memory to the same physical memory. It simulates the KV cache sharing scenario and helps us observe the performance of PagedAttn's CUDA kernel, which is denoted as PagedAttn*. None of the kernels support the TPP algorithm.

Workload. Sequences are processed in batch mode, and the batch size is b. All sequences within the same batch start and finish simultaneously. Each sequence is prefilled with n_p prompt tokens, and the leading n_s tokens are common prefixes. The task is to decode the next n_c completion tokens iteratively. We measure the decoding latency t and the throughput defined by token rate(tokens per second or TPS, $n_c * b/t$). For all experiments, the head dimension d is 128, the number of heads h is 32, and the chunk size c is 64. All tensors are in FP16.

Results. We run experiments to observe the performance gain brought by PAKV and TPP by varying the following system hyperparameters: prompt and shared token count, completion token count, and batch size.

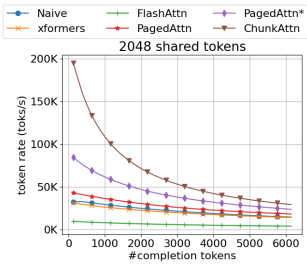
Table 3 shows the latency of self-attention implementations given various prompt and shared token counts. ChunkAttn and PagedAttn* outperform Naive, xformers, FlashAttn, and PagedAttn, which are agnostic to shared token count. The Naive is 6.6× and 2.1× slower than ChunkAttn and PagedAttn*, respectively (n_s=4096). By comparing PagedAttn* and PagedAttn, we observe the performance gain brought by sharing KV cache memory physically. Although PagedAttn* does not implement PAKV or TPP, the hardware cache helps reduce its latency by up to 52% compared to PagedAttn (n_s=4096): repeatedly accessing the same physical memory blocks provides sig-

n_p	n_s	Latency (ms)					
		Naive	xformers	FlashAttn	PagedAttn	PagedAttn*	ChunkAttn
1024	0	363.35	378.19	1586.73	356.17	355.82	332.50
1024	512	364.73	385.79	1587.14	355.88	257.74	**198.87**
1024	768	362.43	378.50	1591.61	356.02	215.18	**131.21**
1024	1024	361.76	379.36	1586.90	355.44	154.46	**56.00**
2048	0	686.40	816.44	3175.25	702.98	703.50	655.44
2048	1024	687.52	828.76	3173.53	703.35	505.32	**384.37**
2048	1536	685.78	820.19	3174.96	702.90	421.25	**247.14**
2048	2048	688.41	823.60	3152.25	703.72	338.41	**110.48**
4096	0	1369.52	1720.00	6289.55	1400.61	1400.17	1301.78
4096	2048	1370.47	1722.42	6303.21	1400.99	998.78	**747.56**
4096	3072	1369.74	1725.57	6301.41	1400.30	828.98	**477.66**
4096	4096	1370.41	1713.13	6300.65	1399.51	663.84	**206.22**

Table 3: Latency (ms) of self-attention kernel given n_p context tokens and n_s prefix tokens are shared. Chunk size c=64, batch size b=32.

nificant performance gain. The benefit of TPP can be further seen by comparing PagedAttn* and ChunkAttn. ChunkAttn outperforms PagedAttn* by 2.8-3.2×, with a range of n_s from 1024 to 4096. TPP does not cause performance regression when no token is shared (n_s=0, ChunkAttn vs. PagedAttn* in Table 3). As a result, TPP should always be enabled.

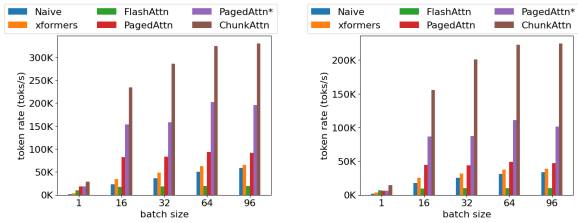
As the decoding proceeds, sequences start to diverge, and the performance gain of ChunkAttn gradually decreases, as shown in Figure 3. Given 2048 shared tokens, ChunkAttn yields 3.6× token rate improvement compared to PagedAttn when n_c reaches 512, and the speedup drops to 2.3× when n_c reaches 2048. However, it is still a significant improvement. The improvement of ChunkAttn over PagedAttn* is lower since PagedAttn* benefits from physically shared KV cache memory, and only TPP makes a difference here. However, given n_s=2048, ChunkAttn is still 2.0× (145K against 73K) and 1.5× (70K against 46K) faster than PagedAttn* when n_c reaches 512 and 2048 respectively.



n_s	n_c	Token Rate($\times 10^3$) (toks/s)		
		Paged	Chunk	Speedup
1024	256	76.35	241.93	**3.2** ×
	512	69.15	186.44	2.7 ×
	1024	58.12	127.85	2.2 ×
2048	512	39.85	145.41	**3.6** ×
	1024	36.18	107.37	3.0 ×
	2048	30.17	70.33	2.3 ×
4096	512	21.04	101.69	**4.8** ×
	1024	19.85	81.69	4.1 ×
	2048	17.98	58.33	3.2 ×
	4096	15.12	37.05	2.4 ×

Figure 3: Throughput in token rate when generating up to n_c completion tokens, given n_s prefix tokens are shared. Chunk size c=64, batch size b=32.

Figure 4 focuses on varying the batch size. For all implementations except ChunkAttn and PagedAttn*, the throughput peaks when the batch size reaches 16 due to memory-bound. Given n_s is 2048, ChunkAttn's throughput continues to grow from 155k to 224k toks/s for the batch size ranging from 16 to 96 due to better data locality and improved arithmetic intensity.



(a) n_s=1024 (b) n_s=2048

Figure 4: Token rate when decoding up to n_c=64 completion tokens given various batch sizes. Chunk size c=64.

4.2 End-to-end Evaluation

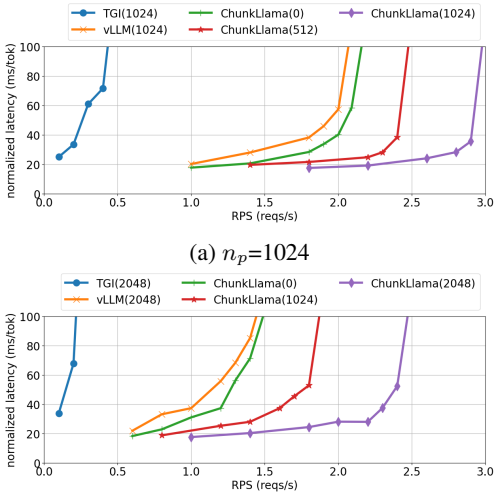
ChunkLlama is built on top of Huggingface Llama and vLLM's optimized kernels (layer normalization and rotary embedding) under Apache-2.0 license, but the attention module is substituted by ChunkAttn. We run all experiments on the Open Llama2 7B model in FP16 (Geng and Liu, 2023; Computer, 2023; Touvron et al., 2023a).

Baselines. We select two widely used and optimized LLMs serving toolkits with proven production usages: the start-of-the-art vLLM 0.2.7 (Kwon et al., 2023) and Huggingface's Text Generation Inference (TGI) 1.3.4 (HuggingFace, 2023).

Workload. Requests arrive at the server randomly following the Poisson arrival process (Hill, 1992) parameterized by λ, which is the average requests per second (RPS). The actual batch size is adjusted dynamically by each system during decoding, and we configure its maximum to 32 equally. Application developers provide no information about the shared prompt prefix for the service provider to pre-configure. We measure the normalized latency (ms/tok or 1/TPS) as in vLLM, which is the mean of each request's end-to-end latency t (including queuing time) divided by the completion token count n_c, and the peak memory bytes used by KV cache.

Results. ChunkLlama yields the fastest inference speed, as shown in Figure 5. ChunkLlama can achieve 1.6× (2.9 against 1.8) and 2.3× (2.3 against 1.0) higher throughput compared to vLLM when 1024 and 2048 prefix tokens are shared while maintaining a normalized latency of less

than 40 ms/token. Table 4 compares the latency and KV cache memory usage of our ChunkLlama to vLLM. No performance regression is observed in ChunkLlama without shared prefix tokens. The KV cache memory usage is reduced by 70%-90% with long shared prefixes. The peak batch size is also reduced by 20%-40% since ChunkLlama can decode faster.



(a) n_p=1024

(b) n_p=2048

Figure 5: Normalized latency given different request arrival rates (RPS). Each line is marked by the system and shared prompt token count: system(n_s).

n_p	n_s	n_c	RPS	Latency (ms/tok)		Peak KV Cache (GB)		Peak Batch Size	
				vLLM	ChunkLlama	vLLM	ChunkLlama	vLLM	ChunkLlama
1024	0	512	1.0	19.92	19.11	14.73	11.90	23	18
1024	1024	512	1.0	20.80	**14.07**	14.79	**3.28**	23	14
2048	0	512	0.6	21.90	19.43	21.70	22.41	19	20
2048	2048	512	0.6	21.61	**15.20**	21.09	**3.40**	19	12
4096	0	512	0.4	26.23	26.88	34.59	35.13	16	16
4096	4096	512	0.4	27.62	**17.16**	35.42	**4.00**	16	11

Table 4: Normalized latency, peak KV cache memory, and batch size reached during decoding.

5 Related Work

The most relevant work on optimizing the memory utilization of KV cache is PagedAttention in vLLM (Kwon et al., 2023). It introduces the paging technique in OS to solve the problem of memory waste caused by dynamic and unknown sequence lengths during decoding. However, only a proposal on service providers to pre-configure shared prompts is mentioned, and it is not implemented in vLLM releases (up to 0.2.7). Our solution, which differs from the paging one, uses the prefix tree to manage memory and aims to discover redundancy in KV cache across user requests at runtime automatically. The solution is more practical for multi-tenant deployment scenarios where service providers centrally host models and have requirements on scalability. According to vLLM, the shared KV cache is similar to the dynamic link library shared by multiple processes. vLLM's strategy is to compile before publishing (AoT). We expect to compile in real-time (JIT). Based on the context captured in the prefix tree, our work further proposes a two-phase partition algorithm to explore the optimization opportunities shared system prompts bring to the self-attention kernel, which is another difference between our work and the existing work.

Partition strategies in ChunkAttention are built on online softmax (Milakov and Gimelshein, 2018) and inspired by FlashAttention (Dao et al., 2022; Dao, 2023), which adopted the same algorithm. FlashAttention thoroughly researched and implemented various tiling techniques, accelerating self-attention by 2-4× while cutting memory operations by 10-20×. FlashAttention-2 altered tiling strategies and additionally doubled the speed. However, FlashAttention is inflexible regarding non-contiguous memory or variable sequence lengths, making it more suitable for model training than inference. There is little gain when the query token count is always one during decoding. ChunkAttention handles variable sequence lengths during decoding and batches attention operations of several sequences to reduce memory operations. As a result, our work and FlashAttention are complementary.

6 Conclusion

In this paper, we propose ChunkAttention, a novel self-attention module, to efficiently manage KV cache and accelerate the self-attention kernel for LLMs inference. We successfully adopt the prefix tree to create a prefix-aware KV cache. It addresses the challenge of detecting and removing redundant KV cache at runtime. We evaluate ChunkAttention in various configurations and at different levels, proving its feasibility and the side effects can be managed. Experiments show that the ChunkAttention kernel can achieve comparable throughput with SOTA PagedAttention kernel without shared system prompts and can outperform it by 3.2-4.8× with a shared system prompt of 1024 to 4096 tokens on A100 (80G) by applying prefix-aware KV cache and two-phase partition.

7 Limitations

The Position of System Prompt. To share the key/value tensors in memory, the shared system prompt must appear at the beginning of the sequence. Although this is the most common practice in many works and systems (Lu et al., 2023; Qian et al., 2023; Saad-Falcon et al., 2023; Zhuang et al., 2023), it is not mandatory. Liu et al. (2023) reveals that language model performance degrades significantly when changing the position of relevant information, indicating that models struggle to access and use information in long input contexts robustly. In particular, performance is often lowest when models must use information in the middle of long input contexts. As a result, when application developers do not put the system prompt at the beginning for performance concerns after evaluations or unintentional mistakes, KV caches of the entire sequences are different, and PAKV cannot save memory in this case, although they have a large number of tokens in common.

Fine Tuning. In addition to using system prompts, fine-tuning is another promising way to infuse domain knowledge into LLMs (Houlsby et al., 2019; Hu et al., 2023). Due to the high training and deployment cost, LLMs are typically pre-trained and centrally hosted for multiple applications to share. It is not cost-efficient for each application to fine-tune models and deploy private instances. However, fine-tuning may become more practical and popular as hardware and software environments evolve. In this case, we no longer need to design long system prompts for each application, and the sharing opportunities of system prompts are reduced. As of today, we have not seen promising and cost-efficient fine-tuning and hosting solutions in this direction than using system prompts.

Model and Hardware Compatibility. To achieve the best performance, ChunkAttention implements the two-phase partition kernel with the low-level CUDA programming instead of leveraging high-level primitives in cuDNN (oneDNN Contributors, 2023) or PyTorch. We tune its performance for the most common LLM configurations, *e.g.*, 128 head dimension size, and hardware, *e.g.*, NVIDIA A100, GeForce RTX 4090, and Intel Xeon CPU. For other configurations and hardware, we need to tune and verify the performance case by case, which adds significant development costs. We believe community efforts are needed to generalize the two-phase partition algorithm and make it compatible with more model configurations and hardware.

References

Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. 2022. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15.

Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. 2023. Palm 2 technical report. *arXiv e-prints*, pages arXiv–2305.

Anthropic. 2023. How to use system prompts. `https://docs.anthropic.com/claude/docs/how-to-use-system-prompts`.

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

Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie. 2023. A survey on evaluation of large language models.

Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29–35.

Zheng Chu, Jingchang Chen, Qianglong Chen, Weijiang Yu, Tao He, Haotian Wang, Weihua Peng, Ming Liu, Bing Qin, and Ting Liu. 2023. A survey of chain of thought reasoning: Advances, frontiers and future.

Together Computer. 2023. Redpajama-data: An open source recipe to reproduce llama training dataset.

Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, Lei Li, and Zhifang Sui. 2023. A survey on in-context learning.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A survey for in-context learning. *arXiv preprint arXiv:2301.00234*.

Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. 2018. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15.

Gemini. 2023. Gemini: A family of highly capable multimodal models.

Xinyang Geng and Hao Liu. 2023. Openllama: An open reproduction of llama.

Steve Hill. 1992. A simple fast memory allocator. In DAVID KIRK, editor, *Graphics Gems III (IBM Version)*, pages 49–50. Morgan Kaufmann, San Francisco.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pages 2790–2799. PMLR.

Zhiqiang Hu, Lei Wang, Yihuai Lan, Wanyu Xu, Ee-Peng Lim, Lidong Bing, Xing Xu, Soujanya Poria, and Roy Ka-Wei Lee. 2023. Llm-adapters: An adapter family for parameter-efficient fine-tuning of large language models.

HuggingFace. 2023. huggingface/text-generation-inference: Large language model text generation inference. https://github.com/huggingface/text-generation-inference.

Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. 2023. S3: Increasing gpu utilization during generative inference for higher throughput. *arXiv preprint arXiv:2306.06000*.

Sehoon Kim, Coleman Hooper, Thanakul Wattanawong, Minwoo Kang, Ruohan Yan, Hasan Genc, Grace Dinh, Qijing Huang, Kurt Keutzer, Michael W. Mahoney, Yakun Sophia Shao, and Amir Gholami. 2023. Full stack optimization of transformer inference: a survey.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention.

Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, and Daniel Haziza. 2022. xformers: A modular and hackable transformer modelling library.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. Api-bank: A comprehensive benchmark for tool-augmented llms.

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

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. 2023. Chameleon: Plug-and-play compositional reasoning with large language models.

Maxim Milakov and Natalia Gimelshein. 2018. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*.

oneDNN Contributors. 2023. oneapi deep neural network library (onednn). https://github.com/oneapi-src/oneDNN.

OpenAI. 2023a. Chatgpt plugins. https://platform.openai.com/docs/plugins/introduction.

OpenAI. 2023b. Function calling - openai api. https://platform.openai.com/docs/guides/function-calling.

OpenAI. 2023c. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

OpenAI. 2023d. How to call functions with chat models. https://cookbook.openai.com/examples/how_to_call_functions_with_chat_models.

OpenAI. 2023e. openai/tiktoken: tiktoken is a fast bpe tokeniser for use with openai's models. https://github.com/openai/tiktoken.

Cheng Qian, Chi Han, Yi R. Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2023. Creator: Tool creation for disentangling abstract and concrete reasoning of large language models.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Jon Saad-Falcon, Joe Barrow, Alexa Siu, Ani Nenkova, David Seunghyun Yoon, Ryan A. Rossi, and Franck Dernoncourt. 2023. Pdftriage: Question answering over long, structured documents.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools.

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

Franyell Silfa, Jose Maria Arnau, and Antonio González. 2022. E-batch: Energy-efficient and high-throughput rnn batching. *ACM Trans. Archit. Code Optim.*, 19(1).

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023a. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023b. Llama: Open and efficient foundation language models.

Mariano Trebino. 2016. mtrebi/memory-allocators: Custom memory allocators in c++ to improve the performance of dynamic memory allocation. `https://github.com/mtrebi/memory-allocators#pool-allocator`.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt.

Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76.

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

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2023. Large language models are human-level prompt engineers.

Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. 2023. Toolqa: A dataset for llm question answering with external tools.

A System Prompt for Chatbot Applications with Plugins

The following system prompt teaches the chatbot to use Bing Search, Expedia, OpenTable, and Spotify APIs to answer user queries. The token count is 1766.

System Prompt and User Query

Instructions: Given the following list of API specifications and user query, you will choose the most appropriate API to invoke and try to parse the corresponding parameters from the user query.

- If none of the API descriptions match the user query intent, you will return `not_found()`.
- If a parameter is required but not included in the user query, then return `not_found()`.
- Your response must strictly follow the syntax of: `api_chosen(param1=PARSED_PARAM1, ...)`.

Following are the list of API definitions and their parameters:

- `bing_web_search(count, offset, q, safe_search, set_lang)`: The Web search API lets you send a search query to Bing and get back search results that include links to webpages, images, and more. If the user explicitly or implicitly wants to find the latest information from the web, you must use this API.
  Parameters:
  - `count`: [optional] The number of search results to return in the response. The default is 10 and the maximum value is 50.
  - `offset`: [optional] The zero-based offset that indicates the number of search results to skip before returning results.
  - `q`: [required] The user search query term. The term may not be empty.
  - `safe_search`: [optional] A filter used to filter results for adult content. "Off": Return webpages with adult text, images, or videos. "Moderate": Return webpages with adult text, but not adult images or videos. "Strict": Do not return webpages with adult text, images, or videos. The default is "Moderate".
  - `set_lang`: [optional] The language to use for user interface strings. You may specify the language using either a 2-letter or 4-letter code. Using 4-letter codes is preferred.

- `bing_images_search(count, offset, q, safe_search, set_lang)`: The Image Search API lets you send a search query to Bing and get back a list of relevant images.
  Parameters:
  - `count`: [optional] The number of image results to return in the response. The actual number delivered may be less than requested. The default is 35. The maximum is 150.
  - `offset`: [optional] The zero-based offset that indicates the number of image results to skip before returning results.
  - `q`: [required] The user's search query term. The term may not be empty. The term may contain Bing Advanced Operators. For example, to limit images to a specific domain, use the "site:" operator.
  - `safe_search`: [optional] A filter used to filter results for adult content. "Off": Return webpages with adult text, images, or videos. "Moderate": Return webpages with adult text, but not adult images or videos. "Strict": Do not return webpages with adult text, images, or videos. The default is "Moderate".
  - `set_lang`: [optional] The language to use for user interface strings. You may specify the language using either a 2-letter or 4-letter code. Using 4-letter codes is preferred.

- `expedia_search_hotel(city, hotel_name, price_buckets, star_ratings, guest_ratings)`: Search for a hotel based on the user query.
  Parameters:
  - `city`: [required] A string to identify the city to search for.
  - `hotel_name`: [optional] Hotel name used to filter the search results.
  - `price_buckets`: [optional] Used to define custom price filter buckets. If not provided then the default price filter buckets for the POS will be used.
  - `star_ratings`: [optional] Used to filter by star rating. Must be in increments of 5 and separated by commas (minStarRating=0 and maxStarRating=50). Ex. "0,5,10" means 0, 0.5 and 1 star hotels.
  - `guest_ratings`: [optional] Used to filter by amenities. Must be as a list of amenity ids separated by commas. Please note that there is no way at this time to validate the ids are actually valid.

- `expedia_search_flights(departure_date, return_date, departure_airport, arrival_airport, number_of_adult_travelers, child_traveler_age, non_stop_flight, airline_preference)`: Flight search for one-way and round-trip scenarios.
  Parameters:
  - `departure_date`: [required] Date the customer wants to leave for their flight on, in ISO format.
  - `return_date`: [optional] Date the customer wants to return on. If present, indicates a round trip search. If not supplied, then it's a one-way search.
  - `departure_airport`: [required] The three-letter airport code for where the customer is leaving from.

- `arrival_airport`: [optional] The three-letter airport code to where the customer is going.
- `number_of_adult_travelers`: [optional] Number of Adult Travelers (Default: 1).
- `child_traveler_age`: [optional] "childTravelerAge" represents the age of a single child traveler. You are required to specify the age of all child travelers. That means you must specify this parameter for each child that will be flying. Valid values are 0-17 (0 for an infant under the age of one year). If you would like to specify 3 child travelers with the ages of 1, 3 and 8 years, then your query string should contain: "childTravelerAge=1&childTravelerAge=3&childTravelerAge=8"
- `non_stop_flight`: [optional] Set to true to return only nonstop flights in the search response (Default: False).
- `airline_preference`: [optional] Optional parameter to get specific airline carrier information. By default, the preference is all.

• `opentable_search_restaurants(name, category, city, day)`: Returns a list of restaurants.
Parameters:

- `name`: [optional] Name of the restaurant to search for.
- `category`: [optional] Category of the restaurant to search for.
- `city`: [optional] City to search in.
- `day`: [optional] Date to search for.

• `spotify_search_catalog(q, type, limit, offset)`: Get Spotify Catalog information about albums, artists, playlists, tracks, shows or episodes that match a keyword string.
Parameters:

- `q`: [required] Search query. Keywords and optional field filters and operators.
- `type`: [required] A comma-separated list of item types to search across. Valid types are: album, artist, playlist, track, show and episode. Search results include hits from all the specified item types. For example: "q=name:abacab&type=album,track" returns both albums and tracks with "abacab" included in their name.
- `limit`: [optional] Maximum number of results to return. Default: 20 Minimum: 1 Maximum: 50. Note: The limit is applied within each type, not on the total response. For example, if the limit value is 3 and the type is "artist,album", the response contains 3 artists and 3 albums.
- `offset`: [optional] The index of the first result to return. Default: 0 (the first result). Maximum offset (including limit): 1,000. Use with limit to get the next page of search results.

Below are some examples of choosing the API that matches the user query:

datetime_now=2023-11-17T10:45:07+08:00
user_query=Do you believe in God?
api_call=`not_found()`

datetime_now=2023-11-17T10:50:00+08:00
user_query=What is the price of the iPhone 15 Pro Max?
api_call=`bing_web_search(q="price of iPhone 15 Pro Max", set_lang="en")`

datetime_now=2023-11-17T11:09:10+08:00
user_query=OpenAI's logo
api_call=`bing_images_search(q="OpenAI logo", set_lang="zh")`

datetime_now=2023-11-17T13:21:30+08:00
user_query=What is Taylor Swift's latest album?
api_call=`spotify_search_catalog(q="Taylor Swift", type="album", limit=1)`

datetime_now=2023-11-17T11:21:42+08:00

user_query=Looking to eat vegan food in San Francisco this weekend, could you get me one great restaurant suggestion for Saturday?

api_call=

868