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

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

Abstract

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

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

CCS Concepts

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

Keywords

Large Language Models, KV Cache, Compression

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

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

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

1 Introduction

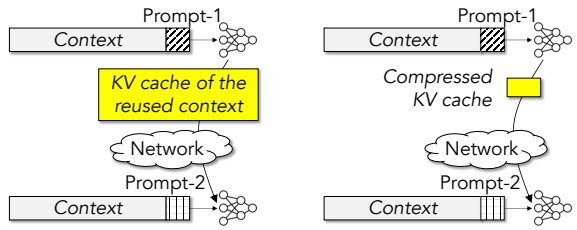
With impressive generative quality, large language models (LLMs) are ubiquitously used [22, 38, 46, 128] in personal assistance, AI healthcare, and marketing. The wide use of LLM APIs (*e.g.,* OpenAI GPT-4 [108]) and the industry-quality open-source models (*e.g.,* Llama [129]), combined with popular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

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

This work does not raise any ethical issues.

2 Background and Motivation

2.1 Large language model basics

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

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

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

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

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

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

2.2 Context in LLM input

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

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

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

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

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

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

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

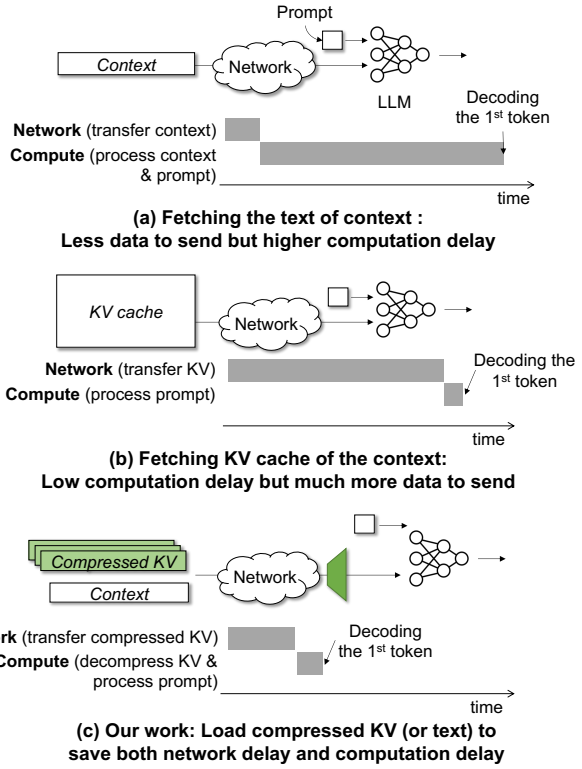


Figure 2: *How different ways of loading context affect the network delay (to transfer context or KV cache) and the computation delay (to run the attention module on the context).*

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

3 The Hidden Network Bottleneck

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

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

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

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

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

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

4 CacheGen: KV Cache Encoding and Streaming

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

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

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

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

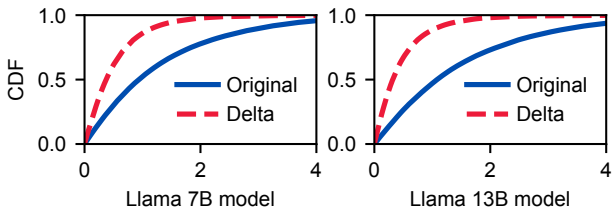


Figure 3: *Contrasting the distribution of the original values and the delta values. We model two Llama models with various long contexts (§5.1). We show absolute values for clarity.*

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

5 CacheGen Design

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

5.1 Empirical insights of KV cache

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

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

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

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

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

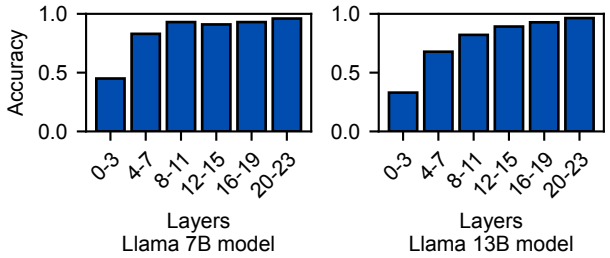


Figure 4: *Applying data loss to different layers of a KV cache has different impact on accuracy. (Same workload as Figure 3).*

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

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

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

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

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

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

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

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

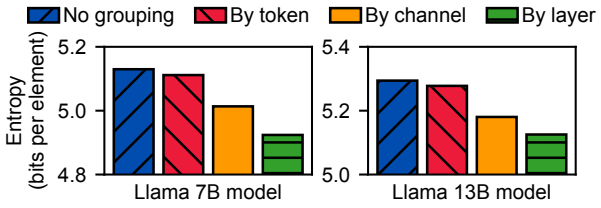


Figure 5: *Entropy (bits per element) when using different grouping strategies (Same workload as Figure 3.)*

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

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

5.2 KV cache encoding

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

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

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

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

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

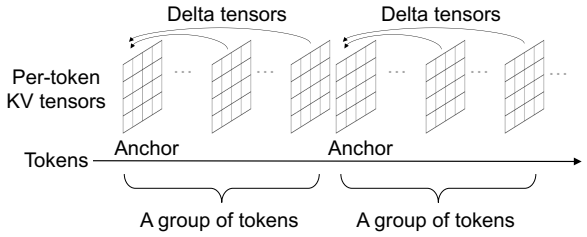


Figure 6: *Within a token group, CacheGen computes delta tensors between KV tensors of the anchor token and those of remaining tokens.*

Next, we explain the details of each step.

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

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

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

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

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

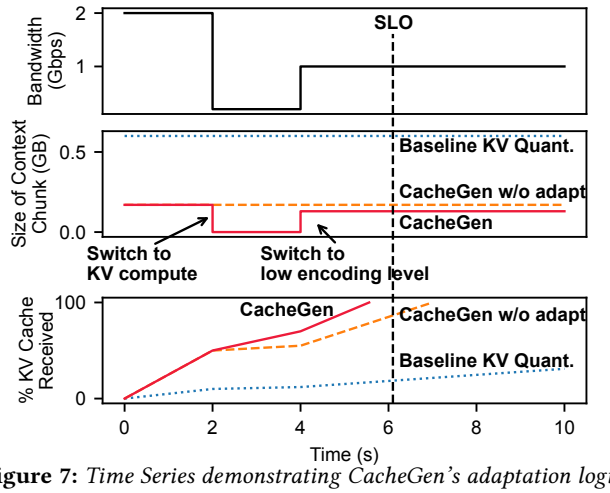


Figure 7: *Time Series demonstrating CacheGen's adaptation logic under bandwidth variation.*

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

5.3 KV cache streaming adaptation

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

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

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

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

However, to adapt efficiently, several questions remain.

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

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

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

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

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

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

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

6 Implementation

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

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

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

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

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

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

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

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

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

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

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

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

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

7 Evaluation

The key takeaways of our evaluation are:

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

7.1 Setup

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

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

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

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

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

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

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

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

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

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

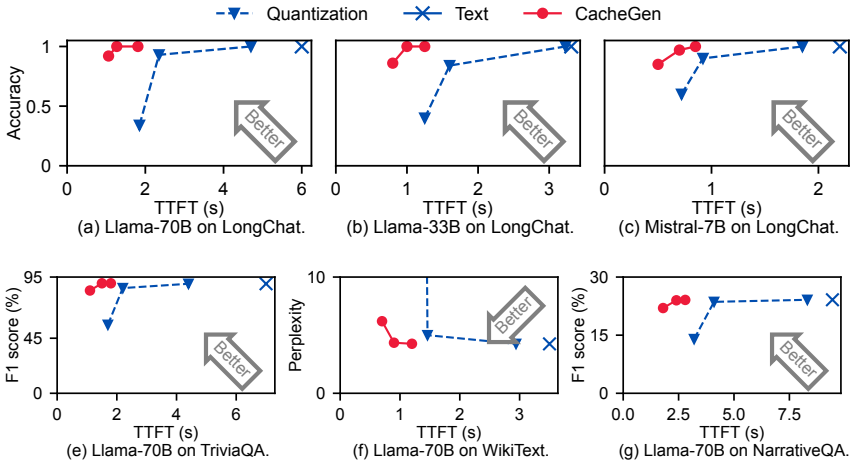


Figure 8: Time-to-first-token (TTFT): *Across different models and different datasets, CacheGen reduces TTFT with little negative impacts on quality (in accuracy, perplexity or F1 score).*

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

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

7.2 Overall improvement

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

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

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

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

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

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

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

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

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

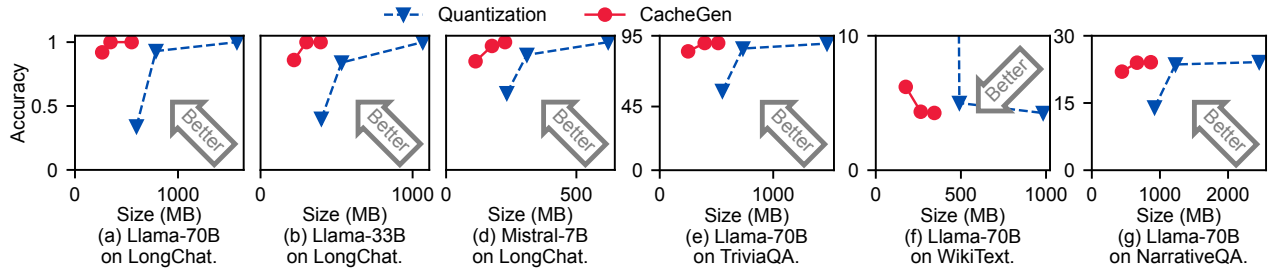


Figure 9: Reducing KV cache size: *Across various models, CacheGen reduces size of KV cache with little accuracy decrease on various datasets.*

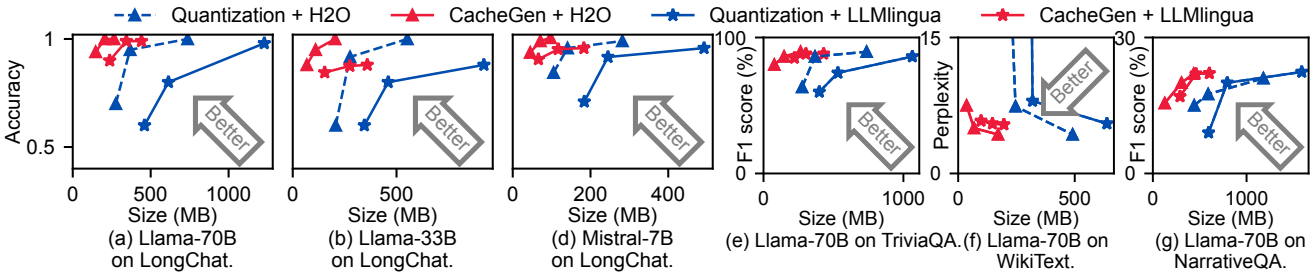


Figure 10: Reducing KV cache size *on top of* H2O [153] and LLMlingu [72]: *Across different models, CacheGen further the size of KV cache, compared to the KV cache shortened by H2O, with little accuracy decrease on different datasets.*

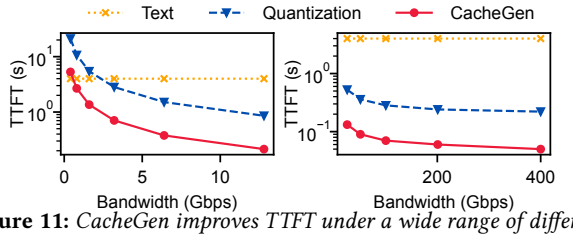


Figure 11: *CacheGen improves TTFT under a wide range of different bandwidths. Plotted with Mistral-7B. y-axis is log scale.*

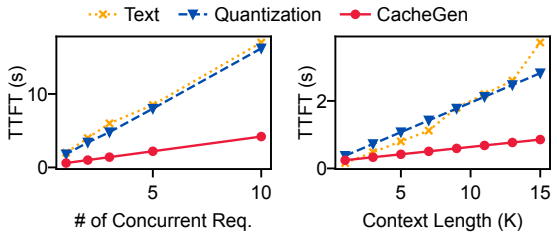


Figure 12: *CacheGen consistently reduces TTFT when there are multiple concurrent requests on one GPU. Plotted with Mistral-7B.*

7.3 Sensitivity analysis

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

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

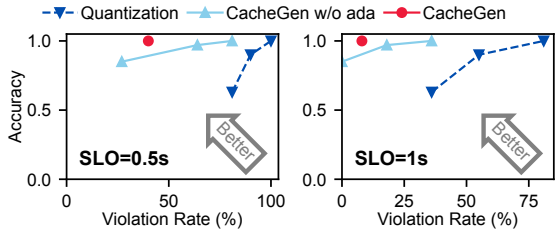


Figure 13: *CacheGen reduces SLO violation rate over CacheGen without adaptation and the quantization baseline. Plotted with Mistral-7B model.*

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

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

7.4 KV streamer adaptation

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

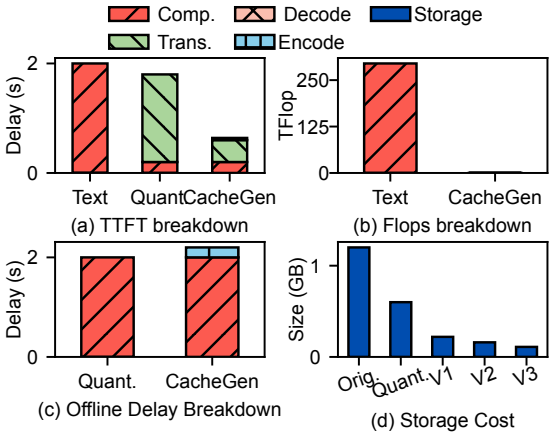


Figure 14: *(a) The breakdown of TTFT for text context, quantization baseline, and CacheGen. (b) Computation overhead of the text baseline and CacheGen. (c) Offline delay breakdown for baseline quantization and CacheGen. (d) The storage cost for CacheGen, quantization baseline and the uncompressed KV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

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

7.5 Overheads and microbenchmarks

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

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

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

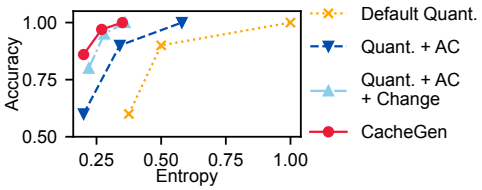


Figure 15: *Contributions of individual ideas behind KV encoder: change-based encoding, layer-wise quantization, and AC based on channel-layer grouping.*

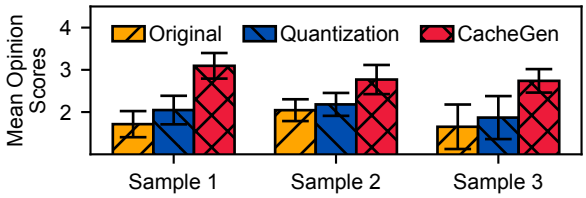


Figure 16: *Real user study shows CacheGen improves QoE significantly over other baselines.*

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

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

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

8 Related Work

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

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

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

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

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

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

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

9 Discussion and Limitations

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

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

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

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

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

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

10 Conclusion

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

Acknowledgement

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[29] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. 2023. LongBench: A Bilingual, Multitask Benchmark for Long Context Understanding. *arXiv preprint arXiv:2308.14508* (2023).

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

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

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

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

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

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

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

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

[38] CellStrat. 2023. Real-World Use Cases for Large Language Models (LLMs) | by CellStrat | Medium. https://cellstrat.medium.com/real-world-use-cases-for-large-language-models-llms-d71c3a577bf2. (April 2023). (Accessed on 09/21/2023).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A Text Output Examples of CacheGen

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

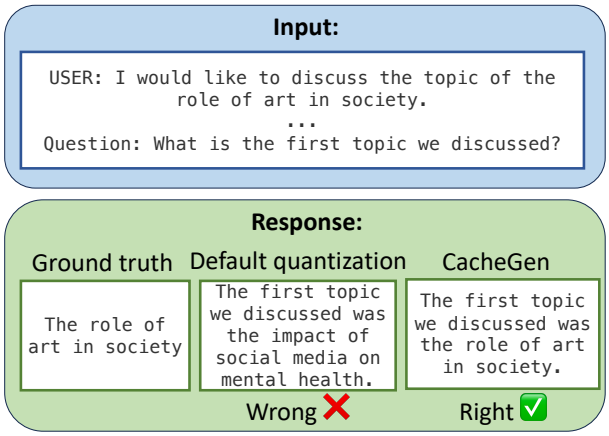


Figure 17: *An example of CacheGen's output on the LongChat dataset with LongChat-7b-16k model.*

B CacheGen vs. more intrusive methods

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

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

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

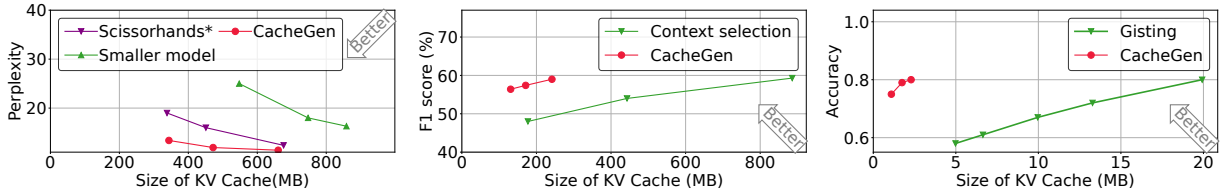


Figure 18: *Comparing CacheGen and more intrusive methods, including smaller models, token dropping (left), context selection (middle), and gisting (right).*

C CacheGen System Settings

C.1 KV Streamer Adaptation Logic

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

Algorithm 1: CacheGen Streaming Adapter Logic

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

C.2 Default Encoding Level

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

D CacheGen's improvement under various workloads

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

E Cost of storing KV cache

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

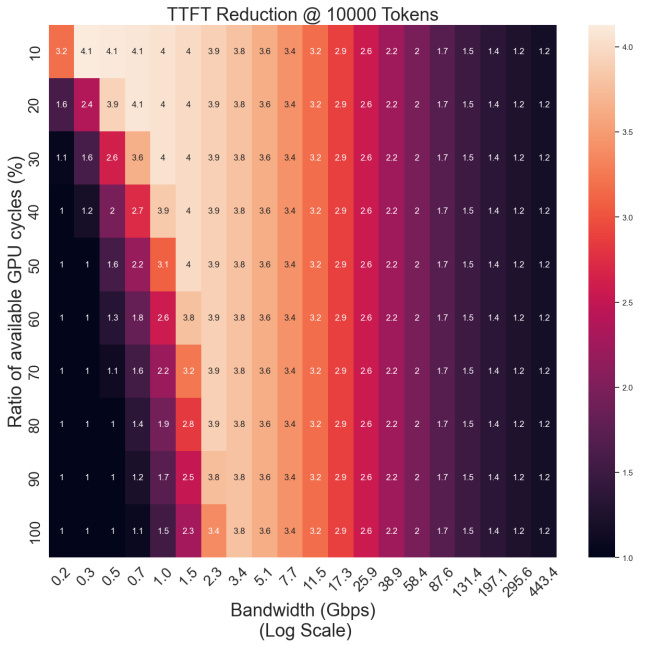


Figure 19: *Heatmap showing CacheGen's improvement over the best baseline over a complete space of workloads. Brighter cells means more TTFT reduction.*