THE UNIVERSITY OF CHICAGO

CACHEGEN: KV CACHE COMPRESSION AND STREAMING FOR FAST LANGUAGE MODEL SERVING

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCE
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
YUHAN LIU

CHICAGO, ILLINOIS
MAY 2024

ACKNOWLEDGMENTS

ABSTRACT

As large language models (LLMs) take on complex tasks, their inputs are supplemented with *longer contexts* that incorporate domain knowledge or user-specific information. Yet using long contexts poses a challenge for responsive LLM systems, 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 extra network delays.

CacheGen is a fast context-loading module for LLM systems. First, CacheGen uses a custom tensor encoder, which embraces KV cache's distributional properties, to *encode* a KV cache into more compact bitstream representations with negligible encoding/decoding overhead. This reduces the bandwidth demand to fetch the KV cache. Second, to maintain low context-loading delay and high generation quality, CacheGen *adapts* the streaming strategies to cope with changes in available bandwidth. When available bandwidth drops, CacheGen may raise the compression level for a part of the context or choose to recompute its KV cache on the fly. We test CacheGen on four popular LLMs of various sizes and four datasets (662 contexts in total). Compared to the recent systems that reuse the KV cache, CacheGen reduces the KV cache size by 3.5-4.3× and the *total* delay in fetching and processing contexts by 3.2-3.7× while having negligible impact on the LLM response quality in accuracy or perplexity.

0.1 Introduction

With impressive generative quality, large language models (LLMs) are ubiquitously used [8, 1, 5, 15] in personal assistance, AI healthcare, and marketing. The wide use of LLM APIs (*e.g.,* OpenAI GPT-4 [10]) and the industry-quality open-source models (*e.g.,* Llama [3]), combined with popular application frameworks (*e.g.,* HuggingFace [27], Langchain [12]), further boosts LLMs' popularity.

To perform complex tasks, users or applications often prepend the LLM inputs with *long contexts* containing thousands of tokens or more. In this paper, we regard a long prefix used by different LLM inputs as a context. 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 can still be useful [80, 105], longer inputs often improve response quality and coherence [42, 109, 119, 53, 97, 43, 63, 45], which drives the ongoing race to train LLMs that accept ever longer inputs, from 2K tokens in ChatGPT to 100K in Claude [7].

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 [110, 54, 42, 127, 97]. While some recent works increase the throughput of processing long context [33], the *delay* of processing the context can still be several seconds for long contexts (2 seconds for a 3K context) [33, 60]. 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.,* [72, 37, 60]).

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. 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 [17, 2, 4, 16, 42].

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 [60], or the KV cache is small enough to be sent quickly by a fast interconnection [133]. Yet, as elaborated in §0.3, 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 be 100s milliseconds to over 10 seconds, hurting the interactive user experience [19, 18, 75].

Some emergent systems reduce the *run-time* KV cache size to accommodate the GPU memory-size constraint, but, as elaborated in §0.4, they do not directly optimize for the *network delay* of KV cache. In short, when loading contexts from other machines, solely optimizing computational delay may cause *higher* response latency, as loading the KV cache increases the network delay.

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

KV cache *encoding*: CacheGen encodes a precomputed KV cache into more compact *bitstream* representations, rather than keeping the tensor shapes of the KV cache. This greatly saves the bandwidth and delay for 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. Moreover, the KV cache encoder incurs negligible compute overhead compared to LLM inference, and the encoding is pipelined with network transmission to minimize its impact on the end-to-end delay.

KV cache *streaming*: CacheGen streams the encoded bitstreams of a KV cache in a way that adapts to changes in available bandwidth. Similar to video streaming, before a user query arrives, CacheGen splits a long context into chunks and encodes the KV of each chunk separately at various compression levels. When streaming a context, 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 [101], loading contexts in text form, and state-of-the-art context compression [66, 130], using four popular LLMs of various sizes (from 7B to 70B) and three datasets of long contexts (662 contexts with 1.4 K to 16 K tokens) (Table 1 gives a preview of 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, under lossless compression, CacheGen is 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.
- Compared with applying quantization on context compression methods [130, 66], CacheGen further reduces the bandwidth usage for sending their KV caches by 3.3-4.2×.

Our code is publicly available at: **https://github.com/UChi-JCL/CacheGen**.

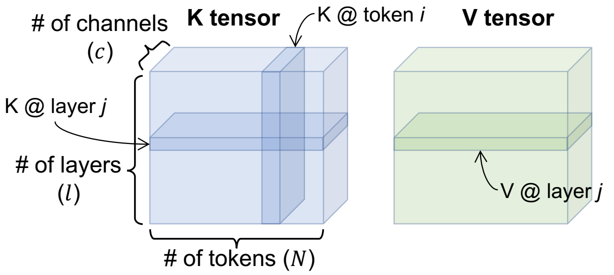


Figure 1: *An illustration of Key & Value Tensors (KV cache).*

0.2 Background and Motivation

0.2.1 Language model basics

Transformers [110, 46, 52] are the de facto models for most language generative 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 three-dimensional tensors, a key (K) tensor and a value (V) tensor (shown in Figure 1). These K and V tensors contain information essential for LLM to utilize the context later. All the KV tensors across different layer 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 [72], 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

[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.

output token, its duration is called *Time-to-First-Token* (*TTFT*). This paper focuses on reducing TTFT during prefilling while not changing the decoding process.

0.2.2 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** [57, 77]. 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 [24, 98, 20], including using latest news to answer fact-checking inquiries [26, 25], using case law or regulation documents to offer legal assistant [99, 106], etc.; *(ii)* code analysis applications retrieve context from a code repository to answer questions or generate a summary about the repository [41, 65, 67], and similarly financial companies use LLMs to generate summaries based on detailed financial documents [89]; *(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 [92, 118, 102]; *(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 [34, 83, 104]; *(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 [69, 38].

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

[2] An example of this process 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.

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 [63] shows that the accuracy increases from 40% to 48% when the context increases from 1K tokens to 10K. Retro [45] similarly shows that the generation quality (perplexity) improves significantly when the context increases from 6K tokens to 24K.

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.

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 [72, 37, 60] and shown its potential with just one caveat, which we discuss in the next section.

0.3 Hidden network bottleneck

Although promising in drastically reducing TTFT, recent studies of reusing the KV cache [72, 37, 60] all rely on the assumption that the KV cache to be re-used is locally available in GPU memory.

In practice, however, the reused KV cache of a long context may need to be fetched from another machine(s). This is because GPU memory is likely not enough to store the KV cache of all repeated contexts. For instance, for the Llama-33B model, the KV cache of a 16K-token[3] context, a common context length in practice [40], is 25 GB, on par with the size of the LLM itself. Storing the KV cache on the GPU for a long time will be even more challenging when the model size increases. Even in chat-based apps that route all requests of a user to the same GPU, the KV cache of the conservation history may not always be in the GPU memory, especially if the next query from the user comes after several hours, during which the KV cache may have been offloaded to make space for fresh chat sessions.

However, the network delay for fetching the KV cache can be substantial, yet it has not received enough attention. Existing LLM systems assume that the KV cache is always in GPU memory or that the KV cache is small enough to be retrieved quickly via a high-speed interconnection. Taking the KV cache size of 25 GB from the previous example, to send it over a 20Gbps link takes 10 seconds whereas running the prefill phase on the text context by the same Llama-34B model takes about the same time. Even with a 100Gbps link, sending the KV cache will take 2 seconds, which damages user experience for its long TTFT [19, 75].

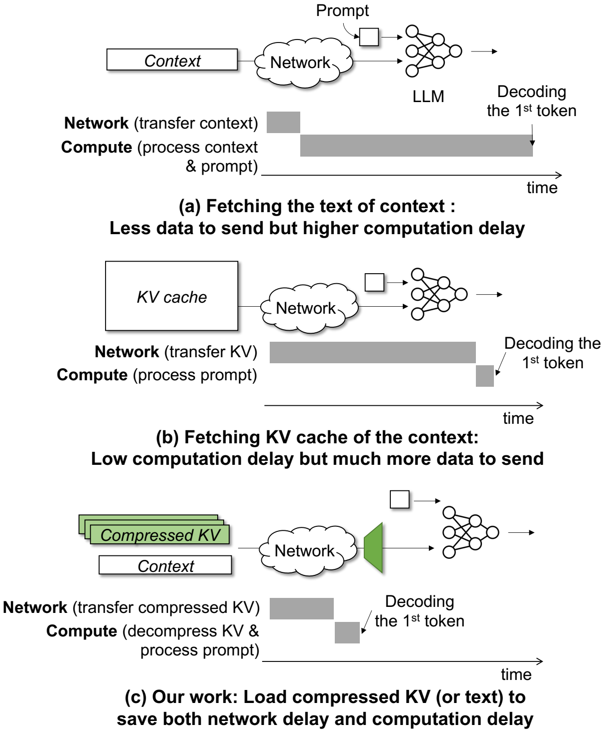
In short, we need a solution to reuse KV cache while reducing the network delay to fetch the KV cache.

0.4 CacheGen: KV Cache 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] About 12,000 English words or merely one SIGCOMM paper!



(a) Fetching the text of context : Less data to send but higher computation delay

(b) Fetching KV cache of the context: Low computation delay but much more data to send

(c) Our work: Load compressed KV (or text) to save both network delay and computation delay

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).

- *(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.,* [66, 130, 81]) 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 cannot use information from the generation phase.

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 (§0.5.1). 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.

0.5 CacheGen Design

We now describe the design of CacheGen, starting with the insights on KV cache (§0.5.1) that inspires KV cache encoder (§0.5.2), followed by how CacheGen adapts to bandwidth (§0.5.3).

0.5.1 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 on 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 [78] (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 §0.7.1.

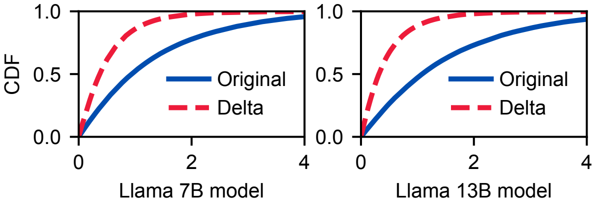


Figure 3: Contrasting the distribution of the original values and the delta values. We model two Llama models with various long contexts (§0.5.1). We show absolute values for clarity.

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 of 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[4]. In both models across the contexts, we can see that the deltas are much more concentrated 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. Though KV cache is computed in a layer-by-layer manner, this 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.

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

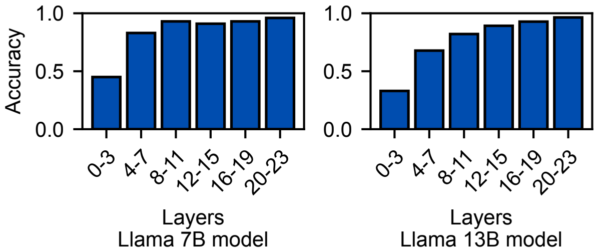


Figure 4: Applying data loss to different layers of a KV cache has different impact on accuracy. (Same workload as Figure 3).

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 LLM is more sensitive to losses in the KV cache values of the shallower layers than to those in 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 §0.7.1) 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 [111, 100]. 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.

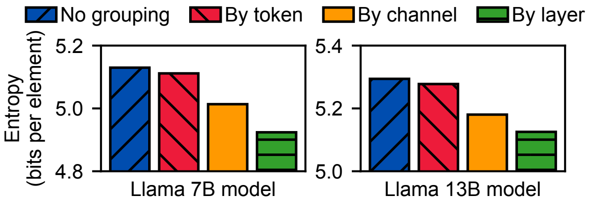


Figure 5: Entropy (bits per element) when using different grouping strategies. (Same workload as Figure 3.) Grouping values by token position (red) reduces entropy much less than grouping by channel (yellow) or by layer (green).

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.*

Intuitively, this can be loosely interpreted as: different KV values in the same channel (or layer) are more similar to each other than different KV values belonging to the same token position. To show the insight empirically, 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 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.

0.5.2 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 (§0.5.1) which suggests deltas

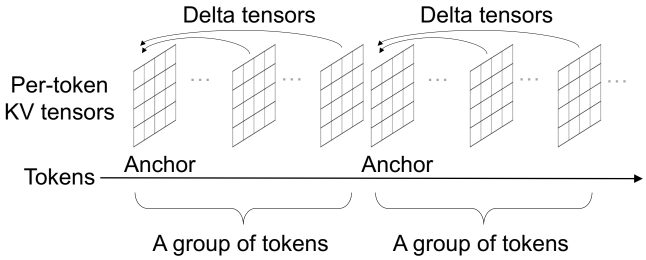


Figure 6: Within a token group, CacheGen computes the delta tensors between the KV tensors of the first anchor token and those of each remaining token.

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 (§0.5.1).

Third, it runs a lossless arithmetic coder to encode the quantized delta tensors into bitstreams. Specifically, inspired by the observation in §0.5.1, the arithmetic coder compresses the values in each layer and channel separately (§0.5.1).

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 [107]. 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 (§0.5.1).

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 [101]. 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 (§0.5.1), 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 [55], 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* [114] (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,

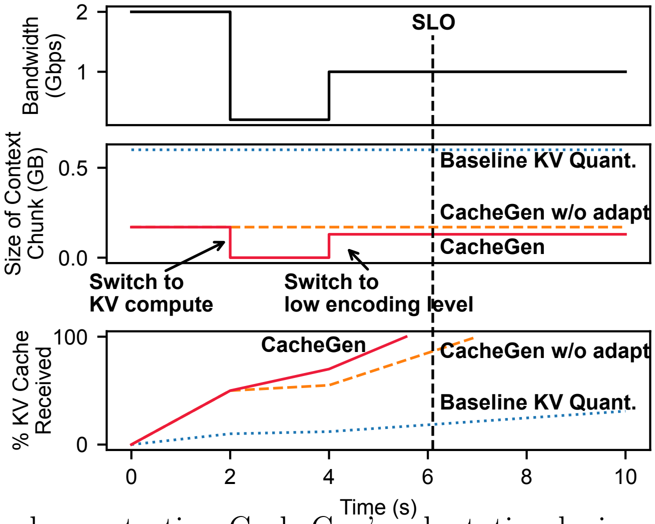


Figure 7: Time Series demonstrating CacheGen's adaptation logic under bandwidth variation.

AC needs *accurate, low-entropy* probability distributions of the elements in the KV cache.

Driven by the observation of the KV value distributions along layers, channels, and token positions (§0.5.1), 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 [85] with CUDA to speed up encoding and decoding (§0.6). In §0.7.5, we empirically show that our method reduces the bitstream size by up to 53% compared to the strawman of using one global symbol distribution.

0.5.3 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) [44] of fetching the KV cache.[5] 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.

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 smaller size. With our adaptation logic (specific algorithm in §0.11.1), 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

[5] 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 [72].

a chunk with different encoding (quantization) levels. Each chunk is encoded *independent* to other chunks *without* affect 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 (§0.5.2). 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.[6]

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 in 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[7], though more optimization may find better chunk lengths.

Finally, *how does CacheGen decide the streaming configuration of the next chunk?* CacheGen chooses the configuration for the next chunk based on the measured throughput when sending the previous chunk. Specifically, CacheGen first assumes the same throughput will remain

[6] A similar concept has been used to split LLM input into prefill chunks for more efficient batching [33].

[7] 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.

and calculates the expected delay of each streaming configuration if it is applied to all remaining chunks. The delay of sending an encoded KV bitstream is calculated by dividing its size by the throughput (more details in §0.8). 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).

0.6 Implementation

We implement CacheGen with about 2K lines of code in Python, 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 attention computation (prefill) of the context.

We implement these two interfaces in HuggingFace models using the `transformers` library [11] 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 through the `past_key_values` argument when calling the `generate` function. Similar integrations are also applicable to other LLM libraries, such as FastChat [132], llama.cpp [13], and GGML [9].

We have also integrated CacheGen in LangChain [12], 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 will invoke `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:

- `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 decoding of KV cache in GPU, we pipeline the transmission of context chunk i with the decoding of context chunk $i - 1$. We implemented a GPU-based AC library [85] with CUDA to speed up encoding and decoding. Specifically, each CUDA thread is responsible for decoding the KV cache from bitstreams for one token. The probability distributions are obtained by counting the frequencies of quantized symbols in the KV feature for the corresponding context.

0.7 Evaluation

The key takeaways of our evaluation are:

- Across four datasets and 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 (§0.7.2).
- CacheGen's KV encoder reduces the bandwidth for transferring KV cache by 3.5-4.3× compared to the quantization baseline (§0.7.2).
- CacheGen's reduction in bandwidth usage is still effective when applied to recent context compression baselines [130, 66]. Specifically, CacheGen further reduces the bandwidth usage by 3.3-4.2×, compared to applying quantization on context compression baselines (§0.7.2).
- CacheGen's improvement is significant across various workloads, including different context lengths, network bandwidths, and numbers of concurrent requests (§0.7.3).
- CacheGen's decoding overhead is minimal, in delay and compute, compared with LLM inference itself (§0.7.5).

0.7.1 Setup

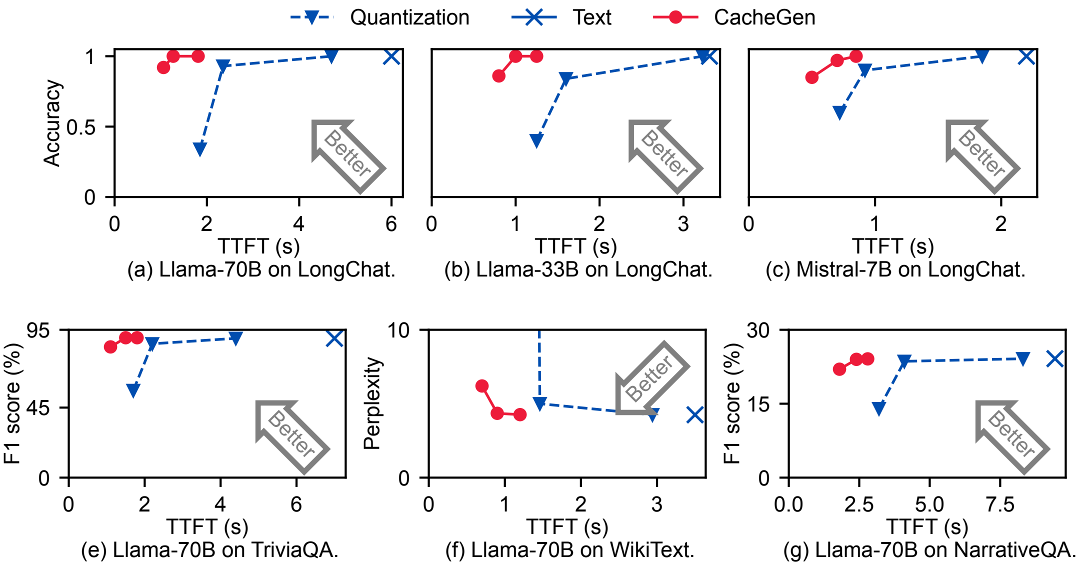
Models: We evaluate CacheGen on four models of different sizes, specifically the fine-tuned versions of Mistral-7B, Llama-34B, Llama-70B. All models are fine-tuned such that they can take long contexts (up to 32K). We did not test CacheGen on other LLMs (*e.g.,* OPT, BLOOM) because there are no public 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 [78] to test LLMs on queries like "What was the

Technique	**KV cache size** (in MB, Lower the better)	**Accuracy** (Higher the better)
CacheGen	176	0.98
8-bit quantization	622	1.00
CacheGen on H2O	71	0.97
H2O [130]	282	0.97
CacheGen on LLMLingua	183	0.94
LLMLingua [66]	492	0.94

Table 1: Performance of CacheGen and the baselines on Mistral-7B with LongChat dataset [78]. Full results are shown in §0.7.



(a) Llama-70B on LongChat.

(b) Llama-33B on LongChat.

(c) Mistral-7B on LongChat.

(e) Llama-70B on TriviaQA.

(f) Llama-70B on WikiText.

(g) Llama-70B on NarrativeQA.

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).

Dataset	Size	Med.	Std.	P95
LongChat [78]	200	9.4K	164	9.6K
TriviaQA [68]	200	9.3K	4497	15K
NarrativeQA [71]	200	14K	1916	15K
WikiText [86]	62	5.9K	4548	14.8K

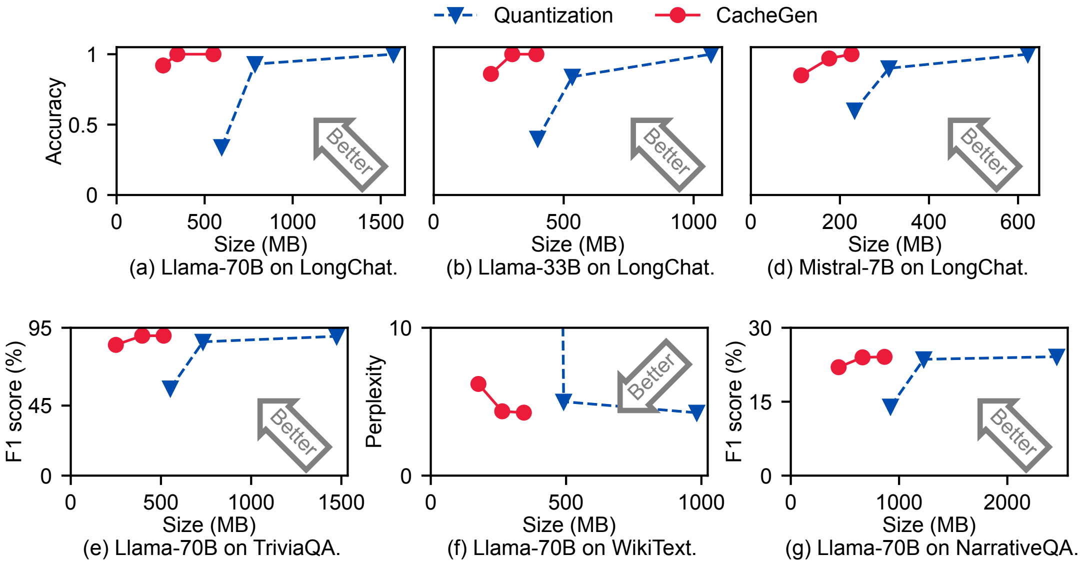
Table 2: Size and context lengths of datasets in the evaluation.

first topic we discussed?" by using all the previous conversations as the context. Most contexts are around 9-9.6K tokens.

- *TriviaQA:* The task tests the reading comprehension ability of the LLMs [40], by giving the LLMs a single document (context), and letting it answer questions based on it. The dataset is part of the LongBench benchmark [40] 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 [86].

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 [39, 49]. 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



(a) Llama-70B on LongChat.
(b) Llama-33B on LongChat.
(d) Mistral-7B on LongChat.
(e) Llama-70B on TriviaQA.
(f) Llama-70B on WikiText.
(g) Llama-70B on NarrativeQA.

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

as a proxy [30] to test the impact of pruning or quantizing LLMs on generation performance [55, 120, 82, 97].

Baselines: We compare CacheGen with baselines that do not change the contexts or model (more baselines in §0.7.5).

- *"Default quantization"* uses the uniform quantization of KV cache, specifically the same quantization level for every layer in the LLM (which was used in [101]).
- *"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 [72], to run the experiments. vLLM's implementation already uses xFormers [73], 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 [66]) or in the KV cache (H2O [130]).

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.

0.7.2 Overall improvement

We first show the improvement of CacheGen over the baselines, as described in §0.7.1.

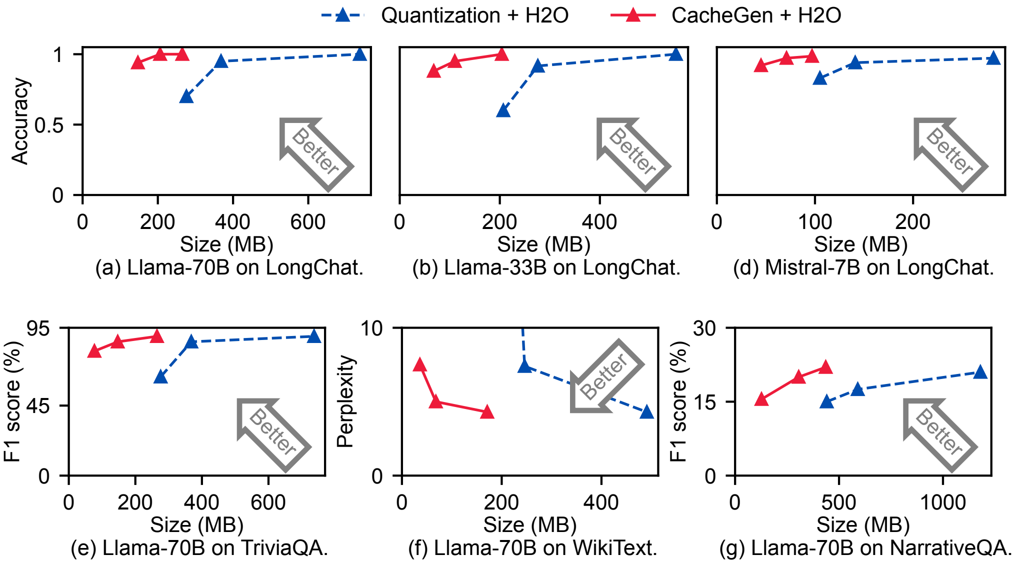
TTFT reduction: Figure 8 demonstrate CacheGen's ability to reduce TTFT, across four models and four datasets. Under bandwidth of 3 Gbps, compared to text context, CacheGen is able to reduce TTFT by 3.1-4.7×. Compared to default quantization, CacheGen is able to reduce TTFT by 3.2-3.7×.

It is important to note that even when the compression is lossless, specifically applying our AC on the KV cache after 8-bit quantization, 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 three datasets and four 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 [14].

Some example text outputs for different baselines are available in §0.8.

Gains over context compression baselines: We also apply CacheGen to further reduce the size of context compression baselines's 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. H2O requires the query tensors of the prompts in order to compress



(a) Llama-70B on LongChat. (b) Llama-33B on LongChat. (d) Mistral-7B on LongChat.

(e) Llama-70B on TriviaQA. (f) Llama-70B on WikiText. (g) Llama-70B on NarrativeQA.

Figure 10: **Reducing KV cache size *on top of* H2O [130]:** 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.

the context's KV cache, which 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 and Figure 11, compared to the context compression baseline, H2O [130], 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.

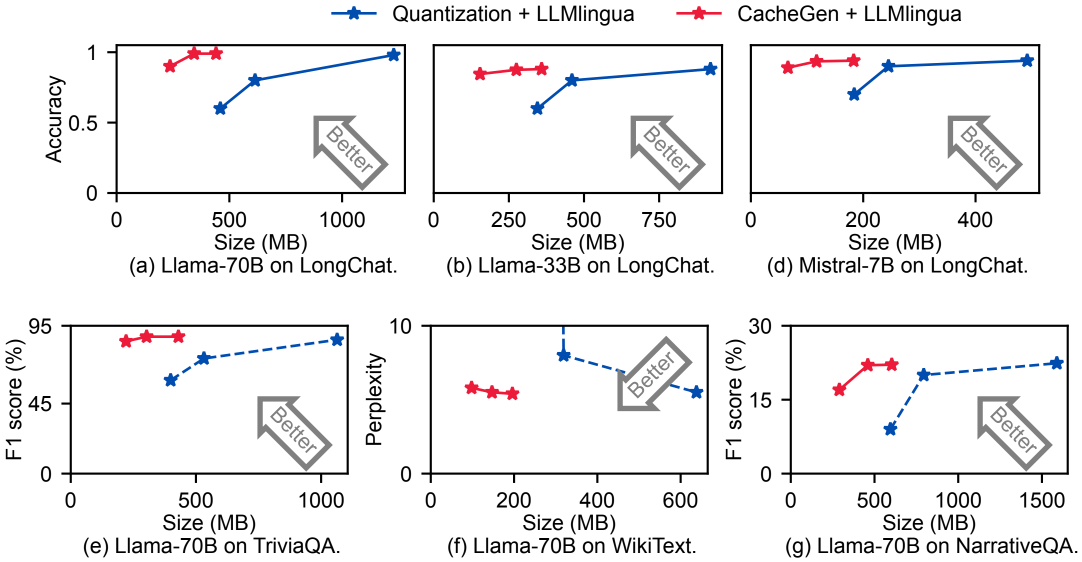


Figure 11: **Reducing KV cache size *on top of* LLMlingua [66]:** Across different models, CacheGen further reduces the size of KV cache, compared to the context shortened by LLMlingua, with little accuracy decrease on different datasets.

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 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.

0.7.3 Sensitivity analysis

Available bandwidth: The left and right figures in Figure 12 compares the TTFT of CacheGen with baselines under a wide range of bandwidth from 0.1–10 Gbps and 10–100 Gbps, while we fix the context length at 9.6K tokens. We can see that CacheGen consistently outperforms baselines under almost all bandwidth situations. Arguably, the absolute

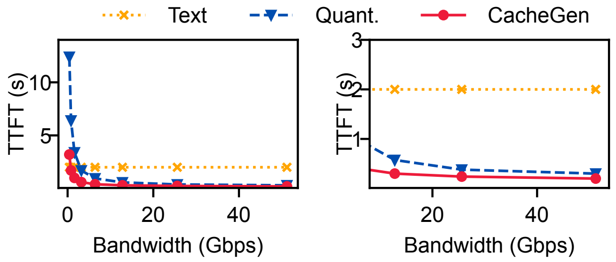


Figure 12: CacheGen improves TTFT under a wide range of different bandwidths. Plotted with Mistral-7B.

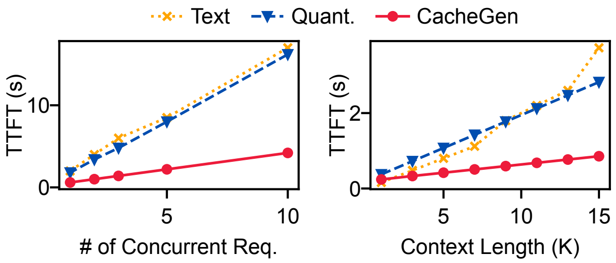


Figure 13: CacheGen consistently reduces TTFT when there are multiple concurrent requests on one GPU. Plotted with Mistral-7B.

reduction in TTFT becomes smaller under high bandwidth (over 20Gbps).

Number of concurrent requests: Figure 13a shows the TTFT under different numbers of concurrent requests. When the 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 §0.2.2. §0.12 shows CacheGen's improvement over a complete space of workloads of different bandwidth and GPU resources.

Context lengths: Figure 13b 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.

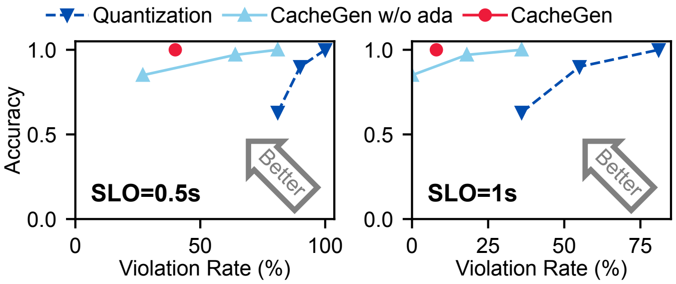


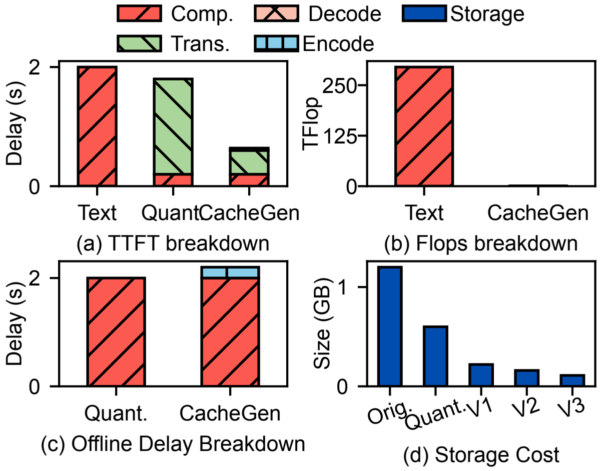
Figure 14: CacheGen reduces SLO violation rate over CacheGen without adaptation and the quantization baseline. Plotted with Mistral-7B model.

0.7.4 KV streamer adaptation

The adaptation logic described in §0.5.3 allows CacheGen to adapt to bandwidth changes and achieve good quality while meeting the SLO on TTFT. In Figure 14, we generate bandwidth traces where each context chunk's bandwidth is sampled from a random distribution 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 an 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.

0.7.5 Overheads and microbenchmarks

Decoding overhead: While having a better size-quality and TTFT-quality trade-off, CacheGen requires an extra decoding step compared to the quantization baseline. CacheGen minimizes the decoding overhead by pipelining the decoding of context chunks with the



(a) TTFT breakdown

(b) Flops breakdown

(c) Offline Delay Breakdown

(d) Storage Cost

Figure 15: *(a) The breakdown of TTFT for text context, quantization baseline, and CacheGen for Llama 13B. (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.*

transmission of the context chunks, so as shown in Figure 15a, 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 §0.6), 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 (§0.5.3). CacheGen compresses each context almost as fast as the baselines because the encoding delay is very small (200 ms), as shown in Figure 15c. Figure 15d 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 16 progressively adds each idea 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. This result underscores the substantial contribution from each idea to the overall reduction

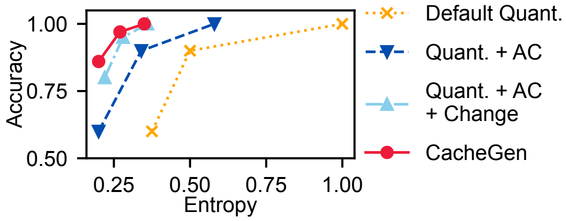


Figure 16: *Contributions of individual ideas behind KV encoder: change-based encoding, layer-wise quantization, and AC based on channel-layer grouping.*

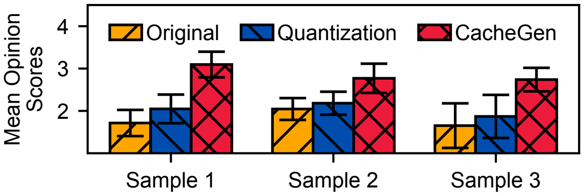


Figure 17: *Real user study shows CacheGen improves QoE significantly over other baselines.*

of KV cache size.

Quality of Experience: We performed an IRB-approved user study to validate the effectiveness of CacheGen. We selected three conversation history 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 TTFT and let users rate the quality of response. With 270 ratings collected from Amazon MTurk [6], we show that CacheGen consistently outperforms other pipelines in QoE with shorter TTFT in Figure 17.

Evaluation results of CacheGen with more baselines is available §0.10, including using a smaller-sized model to speed up TTFT and Gisting, another context-shrinking technique.

0.8 Related work

Faster LLM serving: Most LLM systems research aims to speed up LLM training [95, 103] or make serving systems faster. CacheGen aims at speeding up LLM serving systems by focusing on TTFT reduction. Others explore approximately parallelizing generation [74, 87], accelerating inference on edge devices [125], quantizing LLM weights [36], reducing memory

I/O of GPU on-chip SRAM [54] and reducing self-attention computation complexity [97], better scheduling strategies [117, 126, 133], and GPU memory utilization [72]. Another line of work optimizes the communication delay of transmitting KV cache between GPUs, either by smart model parallelism strategies [133] or by implementing a new attention operation [79]. 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 [101, 129, 81, 93, 87, 116]. CacheGen works as a module to enable reuse of KV caches *across multiple* LLM queries in these frameworks [72, 33, 60, 45].

Longer LLM contexts: Recent efforts aim at enabling LLMs to process very long contexts [122]. 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 [101], using external knowledge via KNN [119], approximating via retraining self-attention to only attend to top-k keys [43, 35], mapping long inputs to smaller latent spaces [62] and using local windowed, dilated or sparse [56, 42, 127] 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 [45], only keeping tokens that are less attended to by the prompt (*i.e.,* heavy-hitter tokens) [81, 129] or by hybrid policies including keeping nearby tokens or heavy-hitter tokens [58], using query-aware compression with document reordering to reduce loss-in-the-middle [66, 96] 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 [88] or auto-encoding [59].

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 (§0.7.5). 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 [131, 91]. In DNN training, tensor compression has been used to compress gradient updates of DNN weights (*e.g.,* [113, 32, 31]). KV caches and gradients have very different properties. DNN training systems often leverage the sparsity of gradients which occurs due to methods like [50, 128, 51]. However the KV cache is not known to be sparse in general.

Retrieval augmented generation(RAG): RAG [45, 64, 63, 98, 76, 112, 94] focuses on retrieving relevant documents to the query via vector based [48, 123, 90] or DNN-based [76, 70, 121, 124] 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 [115, 47]. Some works have also attempted to define a systematic way to choose which KV cache to reuse[61]. Another approach is to have LLM applications that cache the query's generated answers to reduce repetitive query costs [108, 84]. While caching answers is useful for reuse, CacheGen provides a more generic way to incorporate context reuse and can generate better-quality answers.

BIBLIOGRAPHY

[1] 12 Practical Large Language Model (LLM) Applications - Techopedia. https://www.techopedia.com/12-practical-large-language-model-llm-applications. (Accessed on 09/21/2023).

[2] 2112.04426.pdf. https://arxiv.org/pdf/2112.04426.pdf. (Accessed on 09/21/2023).

[3] [2302.13971] llama: Open and efficient foundation language models. https://arxiv.org/abs/2302.13971. (Accessed on 09/21/2023).

[4] [2304.03442] generative agents: Interactive simulacra of human behavior. https://arxiv.org/abs/2304.03442. (Accessed on 09/21/2023).

[5] 7 top large language model use cases and applications. https://www.projectpro.io/article/large-language-model-use-cases-and-applications/887. (Accessed on 09/21/2023).

[6] Amazon Mechanical Turk. https://www.mturk.com/.

[7] Anthropic \ introducing 100k context windows. https://www.anthropic.com/index/100k-context-windows. (Accessed on 09/21/2023).

[8] Applications of large language models - indata labs. https://indatalabs.com/blog/large-language-model-apps. (Accessed on 09/21/2023).

[9] GGML - AI at the edge. https://ggml.ai/.

[10] Gpt-4 api general availability and deprecation of older models in the completions api. https://openai.com/blog/gpt-4-api-general-availability. (Accessed on 09/21/2023).

[11] Huggingface Transformers. https://huggingface.co/docs/transformers/index.

[12] langchain-ai/langchain:building applications with llms through composability. https://github.com/langchain-ai/langchain. (Accessed on 09/21/2023).

[13] llama.cpp. https://github.com/ggerganov/llama.cpp/.

[14] Perplexity in fixed length models. https://huggingface.co/docs/transformers/perplexity.

[15] 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. (Accessed on 09/21/2023).

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

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

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

[19] 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).

[20] 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.

[21] Amazon bedrock pricing. https://aws.amazon.com/bedrock/pricing/, 2024. Accessed: 2024-01-25.

[22] Anyscale pricing. https://docs.endpoints.anyscale.com/pricing, 2024. Accessed: 2024-01-25.

[23] Aws pricing examples. https://aws.amazon.com/s3/pricing/, 2024. Accessed: 2024-01-25.

[24] Chatgpt. https://chat.openai.com/gpts, 2024. Accessed: 2024-01-25.

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

[26] Perplexity. https://www.perplexity.ai/, 2024. Accessed: 2024-01-25.

[27] Rag-transform. https://huggingface.co/transformers/v4.3.0/model_doc/rag.html, 2024. Accessed: 2024-01-25.

[28] Replicate pricing. https://replicate.com/pricing, 2024. Accessed: 2024-01-25.

[29] together.pricing. https://www.together.ai/pricing, 2024. Accessed: 2024-01-25.

[30] 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. Towards a Human-like Open-Domain Chatbot, 2020.

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

[32] Saurabh Agarwal, Hongyi Wang, Shivaram Venkataraman, and Dimitris Papailiopoulos. On the utility of gradient compression in distributed training systems. In D. Mar-

culescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 652–672, 2022.

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

[34] Toufique Ahmed and Premkumar Devanbu. Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

[35] Joshua Ainslie, Santiago Ontanon, Chris Alberti, Vaclav Cvicek, Zachary Fisher, Philip Pham, Anirudh Ravula, Sumit Sanghai, Qifan Wang, and Li Yang. ETC: Encoding Long and Structured Inputs in Transformers, 2020.

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

[37] Anonymous. Chunkattention: Efficient attention on KV cache with chunking sharing and batching, 2024.

[38] AuthorName. 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.

[39] Leif Azzopardi, Mark Girolami, and Keith van Risjbergen. 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, page 369–370, New York, NY, USA, 2003. Association for Computing Machinery.

[40] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*, 2023.

[41] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. Codeplan: Repository-level coding using llms and planning, 2023.

[42] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The Long-Document Transformer, 2020.

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

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

[45] 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. Improving language models by retrieving from trillions of tokens, 2022.

[46] 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. Language Models are Few-Shot Learners, 2020.

[47] Harrison Chase. LangChain, October 2022.

[48] Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. Reading wikipedia to answer open-domain questions, 2017.

[49] Stanley F Chen, Douglas Beeferman, and Roni Rosenfeld. Evaluation Metrics For Language Models. 1 2008.

[50] Tianlong Chen, Zhenyu Zhang, Ajay Jaiswal, Shiwei Liu, and Zhangyang Wang. Sparse moe as the new dropout: Scaling dense and self-slimmable transformers, 2023.

[51] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers, 2019.

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

[53] Zihang Dai*, Zhilin Yang*, Yiming Yang, William W. Cohen, Jaime Carbonell,

Quoc V. Le, and Ruslan Salakhutdinov. Transformer-XL: Language modeling with longer-term dependency, 2019.

[54] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness, 2022.

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

[56] Jiayu Ding, Shuming Ma, Li Dong, Xingxing Zhang, Shaohan Huang, Wenhui Wang, Nanning Zheng, and Furu Wei. LongNet: Scaling Transformers to 1,000,000,000 Tokens, 2023.

[57] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Qianyu Guo, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey, 2024.

[58] Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. Model tells you what to discard: Adaptive kv cache compression for llms, 2023.

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

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

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

[62] 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. General-purpose, long-context autoregressive modeling with perceiver AR. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 8535–8558. PMLR, 17–23 Jul 2022.

[63] Gautier Izacard and Edouard Grave. Leveraging Passage Retrieval with Generative Models for Open Domain Question Answering, 2021.

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

[65] Naman Jain, Tianjun Zhang, Wei-Lin Chiang, Joseph E. Gonzalez, Koushik Sen, and Ion Stoica. Llm-assisted code cleaning for training accurate code generators, 2023.

[66] Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression, 2023.

[67] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2023.

[68] Mandar Joshi, Eunsol Choi, Daniel S. Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension, 2017.

[69] jwatte. 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.

[70] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen tau Yih. Dense passage retrieval for open-domain question answering, 2020.

[71] Tomáš Kočiský, Jonathan Schwarz, Phil Blunsom, Chris Dyer, Karl Moritz Hermann, Gábor Melis, and Edward Grefenstette. The narrativeqa reading comprehension challenge, 2017.

[72] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[73] 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. xformers: A modular and hackable transformer modelling library. https://github.com/facebookresearch/xformers, 2022.

[74] Yaniv Leviathan, Matan Kalman, and Y. Matias. Fast Inference from Transformers via Speculative Decoding. In *International Conference on Machine Learning*, 2022.

[75] Zijian Lew, Joseph B Walther, Augustine Pang, and Wonsun Shin. Interactivity in Online Chat: Conversational Contingency and Response Latency in Computer-mediated Communication. *Journal of Computer-Mediated Communication*, 23(4):201–221, 06 2018.

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

[77] 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. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021.

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

[79] 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. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache, 2024.

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

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

[82] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the Persis-

tence of Importance Hypothesis for LLM KV Cache Compression at Test Time. *arXiv preprint arXiv:2305.17118*, 2023.

[83] Sathiya Kumaran Mani, Yajie Zhou, Kevin Hsieh, Santiago Segarra, Trevor Eberl, Eliran Azulai, Ido Frizler, Ranveer Chandra, and Srikanth Kandula. Enhancing network management using code generated by large language models. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, HotNets '23, page 196–204, New York, NY, USA, 2023. Association for Computing Machinery.

[84] Ignacio Martinez. privategpt. `https://github.com/imartinez/privateGPT`, 2023.

[85] Fabian Mentzer, Eirikur Agustsson, Michael Tschannen, Radu Timofte, and Luc Van Gool. Practical full resolution learned lossless image compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[86] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer Sentinel Mixture Models, 2016.

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

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

[89] Author's Name. Llms in finance: Bloomberggpt and fingpt - what you need to know. Medium, Year of Publication.

[90] Yixin Nie, Songhe Wang, and Mohit Bansal. Revealing the importance of semantic retrieval for machine reading at scale, 2019.

[91] I. V. Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

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

[93] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently Scaling Transformer Inference, 2022.

[94] Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. In-Context Retrieval-Augmented Language Models, 2023.

[95] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 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, page 3505–3506, New York, NY, USA, 2020. Association for Computing Machinery.

[96] Luka Ribar, Ivan Chelombiev, Luke Hudlass-Galley, Charlie Blake, Carlo Luschi, and Douglas Orr. Sparq attention: Bandwidth-efficient llm inference, 2023.

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

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

[99] Ayesha Saleem. Llm for lawyers, enrich your precedents with the use of ai. Data Science Dojo, July 2023.

[100] Hang Shao, Bei Liu, and Yanmin Qian. One-shot sensitivity-aware mixed sparsity pruning for large language models, 2024.

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

[102] Zijing Shi, Meng Fang, Shunfeng Zheng, Shilong Deng, Ling Chen, and Yali Du. Cooperation on the fly: Exploring language agents for ad hoc teamwork in the avalon game, 2023.

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

[104] Yisheng Song, Ting Wang, Puyu Cai, Subrota K. Mondal, and Jyoti Prakash Sahoo. A comprehensive survey of few-shot learning: Evolution, applications, challenges, and opportunities. *ACM Comput. Surv.*, 55(13s), jul 2023.

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

[106] Pavlo Sydorenko. Top 5 applications of large language models (llms) in legal practice. Medium, 2023.

[107] Vivienne Sze and Madhukar Budagavi. High Throughput CABAC Entropy Coding in HEVC. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1778–1791, 2012.

[108] Zilliz Technology. Gptcache. `https://github.com/zilliztech/GPTCache`, 2023.

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

[110] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[111] Yiding Wang, Decang Sun, Kai Chen, Fan Lai, and Mosharaf Chowdhury. Egeria: Efficient dnn training with knowledge-guided layer freezing. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 851–866, New York, NY, USA, 2023. Association for Computing Machinery.

[112] Zhenhailong Wang, Xiaoman Pan, Dian Yu, Dong Yu, Jianshu Chen, and Heng Ji. Zemi: Learning zero-shot semi-parametric language models from multiple tasks, 2023.

[113] Zhuang Wang, Haibin Lin, Yibo Zhu, and T. S. Eugene Ng. 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, page 867–882, New York, NY, USA, 2023. Association for Computing Machinery.

[114] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540, jun 1987.

[115] 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. Transformers: State-of-the-Art Natural Language Processing. pages 38–45. Association for Computational Linguistics, October 2020.

[116] 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. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[117] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast Distributed Inference Serving for Large Language Models, 2023.

[118] Dekun Wu, Haochen Shi, Zhiyuan Sun, and Bang Liu. Deciphering digital detectives: Understanding llm behaviors and capabilities in multi-agent mystery games, 2023.

[119] Yuhuai Wu, Markus Norman Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing Transformers. In *International Conference on Learning Representations*, 2022.

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

[121] Wenhan Xiong, Hong Wang, and William Yang Wang. 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*, pages 2803–2815, Online, April 2021. Association for Computational Linguistics.

[122] Peng Xu, Wei Ping, Xianchao Wu, Lawrence McAfee, Chen Zhu, Zihan Liu, Sandeep

Subramanian, Evelina Bakhturina, Mohammad Shoeybi, and Bryan Catanzaro. Retrieval meets long context large language models, 2024.

[123] Wei Yang, Yuqing Xie, Aileen Lin, Xingyu Li, Luchen Tan, Kun Xiong, Ming Li, and Jimmy Lin. End-to-end open-domain question answering with. In *Proceedings of the 2019 Conference of the North*. Association for Computational Linguistics, 2019.

[124] Yingrui Yang, Yifan Qiao, Jinjin Shao, Xifeng Yan, and Tao Yang. 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, page 1234–1244, New York, NY, USA, 2022. Association for Computing Machinery.

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

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

[127] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big Bird: Transformers for Longer Sequences, 2021.

[128] Lin Zehui, Pengfei Liu, Luyao Huang, Junkun Chen, Xipeng Qiu, and Xuanjing Huang. Dropattention: A regularization method for fully-connected self-attention networks, 2019.

[129] 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. H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. In *Workshop on Efficient Systems for Foundation Models @ ICML2023*, 2023.

[130] 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. H_2o: Heavy-hitter oracle for efficient generative inference of large language models, 2023.

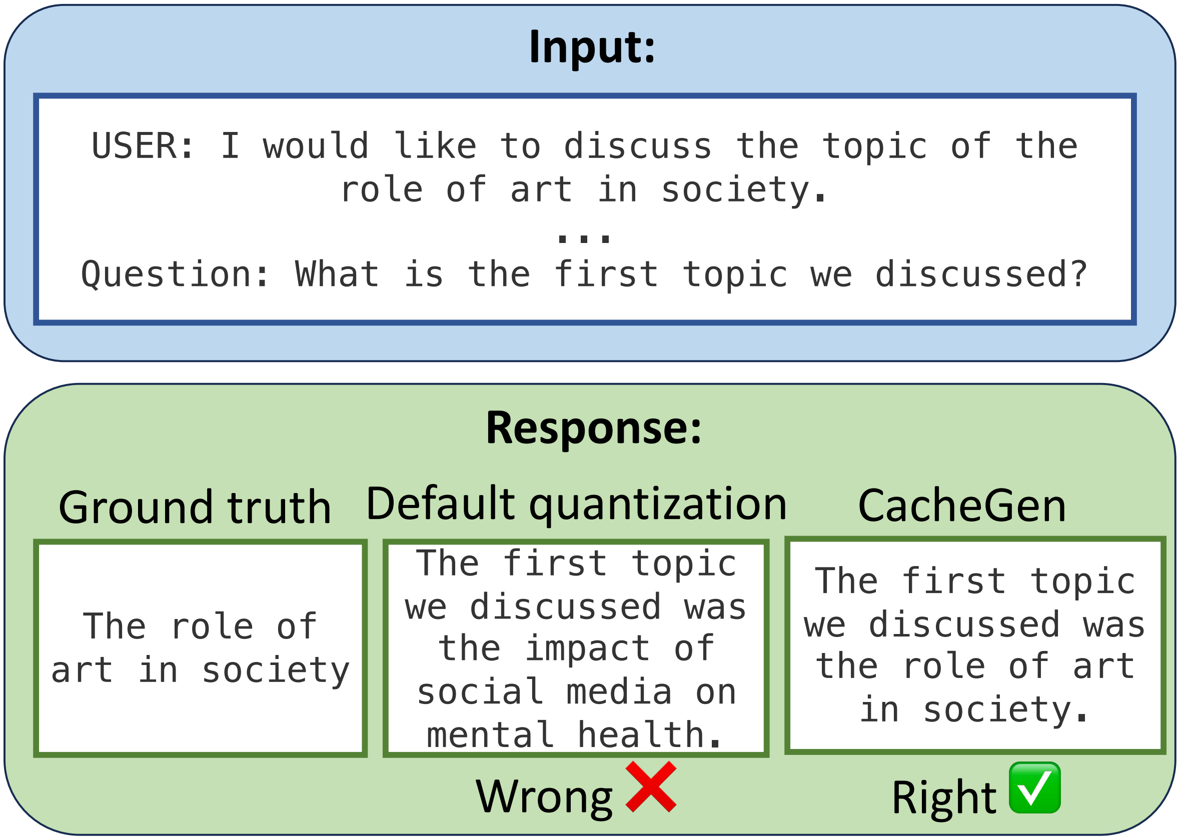
[131] Qibin Zhao, Guoxu Zhou, Shengli Xie, Liqing Zhang, and Andrzej Cichocki. Tensor Ring Decomposition, 2016.

[132] 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. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena, 2023.

[133] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024.

0.9 Text Output Examples of CacheGen

Figure 18 visualizes an example from the LongChat dataset [78] used in §0.7.2. 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.



USER: I would like to discuss the topic of the role of art in society.
...
Question: What is the first topic we discussed?

The role of art in society

The first topic we discussed was the impact of social media on mental health.

The first topic we discussed was the role of art in society.

Figure 18: *An example of CacheGen's output on the LongChat dataset with LongChat-7b-16k model.*

0.10 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 19 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 19a replaces the Llama-7B model with a smaller Llama-3B and applies different quantization levels.
- *Token selection:* Figure 19b uses Scissorhands as an example of *removing tokens* with low self-attention scores from the LLM input [82]. Since the self-attention scores are only available during the actual generation, it cannot reduce TTFT, but we make an effort to

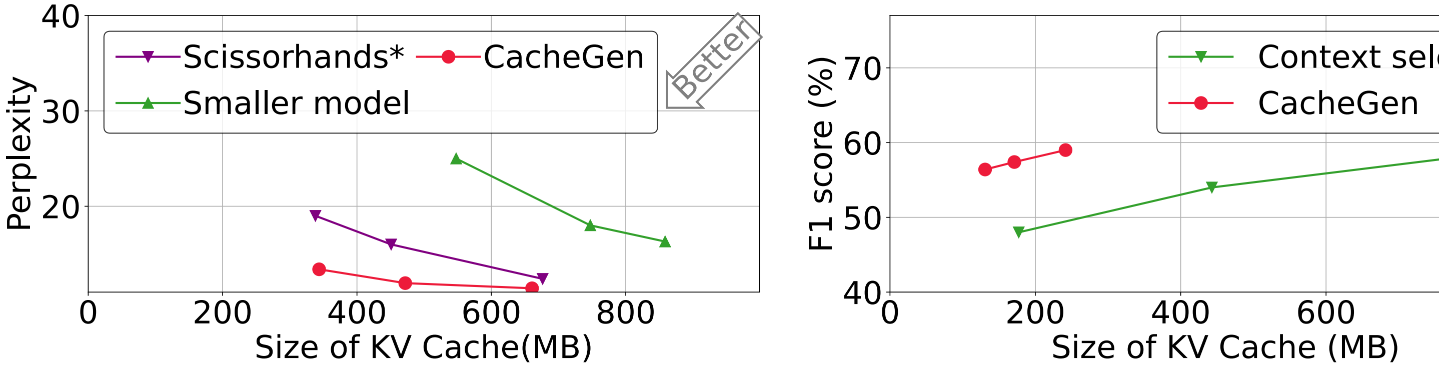


Figure 19: *Comparing CacheGen and more intrusive methods, including smaller models, token dropping (left), context selection (middle), and gisting (right).*

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 [88].

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.

0.11 CacheGen System Settings

0.11.1 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

```
$chunks\_to\_send \leftarrow$ context chunks
while $chunks\_to\_send \neq empty$ do
  get $chunk\_data$
  $throughput \leftarrow$ network throughput
  $remaining\_time \leftarrow SLO - time\_elapsed$
  if $time\_recompute \leq remaining\_time$ then
    $cur\_chunk \leftarrow$ text of $chunk\_data$
  else
    $level \leftarrow max(level|size(chunks\_to\_send, level) \div throughput \leq remaining\_time$
    $cur\_chunk \leftarrow encode(chunk\_data, level)$
  end if
  send $cur\_chunk$
  $chunks\_to\_send \leftarrow chunks\_to\_send \setminus chunk\_data$
end while
```

0.11.2 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.

0.12 CacheGen's improvement under various workloads

Figure 20 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 12 and Figure 13 can be seen as horizontal/vertical cross-sections of this figure.

0.13 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 [23]. On the other hand, recomputing the KV cache from text costs at least \$0.00085 (input only) every time [29, 22, 21, 28]. 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.

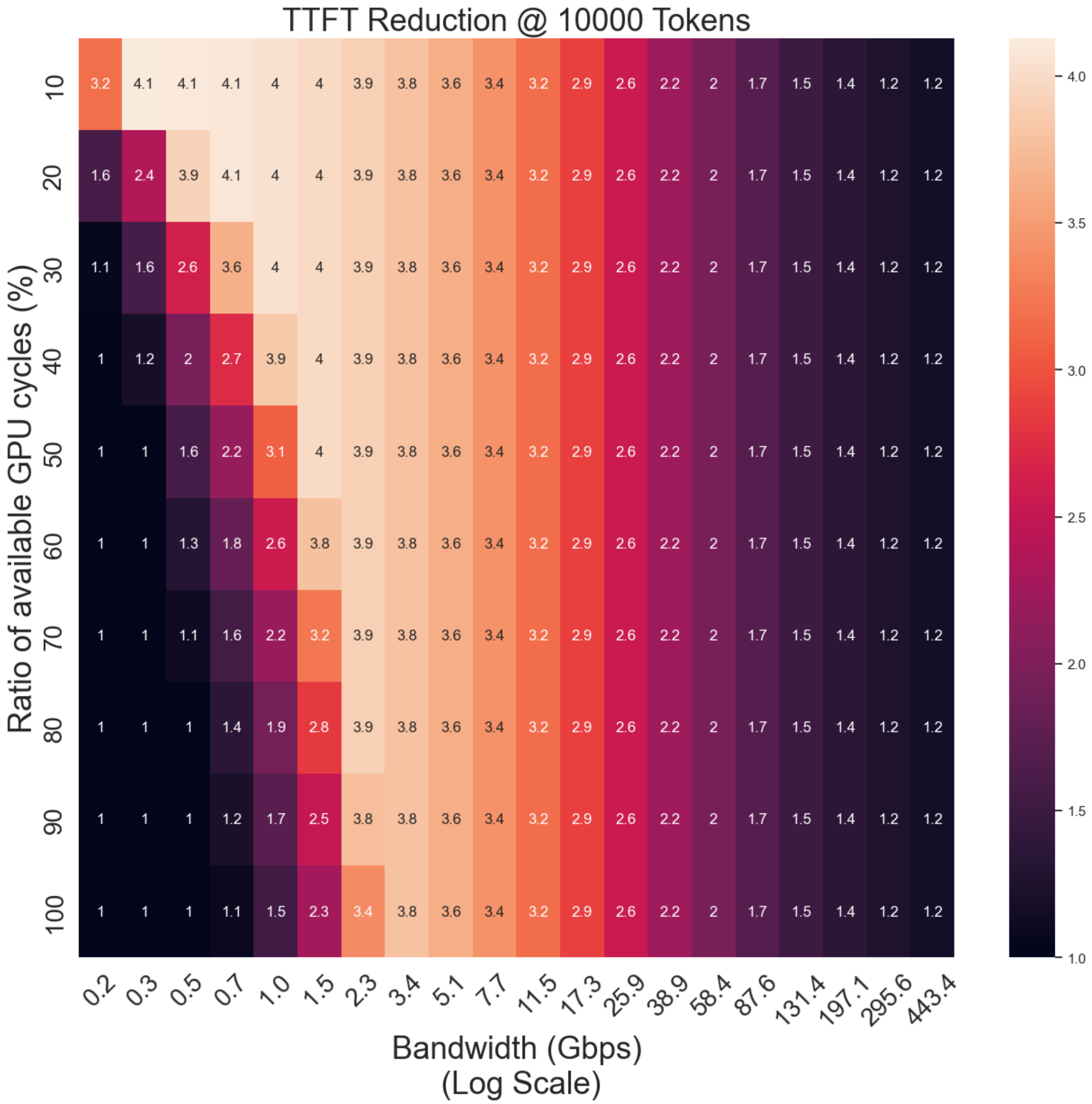


Figure 20: *Heatmap showing CacheGen's improvement over the best baseline over a complete space of workloads. Brighter cells means more TTFT reduction.*