Cocktail: Chunk-Adaptive Mixed-Precision Quantization for Long-Context LLM Inference

Wei Tao[1,2], Bin Zhang[1,2], Xiaoyang Qu[2], Jiguang Wan[1*], Jianzong Wang[2*]
[1]Huazhong University of Science and Technology, Wuhan, China
[2]Ping An Technology (Shenzhen) Co., Ltd, Shenzhen, China

***Abstract*—Recently, large language models (LLMs) have been able to handle longer and longer contexts. However, a context that is too long may cause intolerant inference latency and GPU memory usage. Existing methods propose mixed-precision quantization to the key-value (KV) cache in LLMs based on token granularity, which is time-consuming in the search process and hardware inefficient during computation. This paper introduces a novel approach called Cocktail, which employs chunk-adaptive mixed-precision quantization to optimize the KV cache. Cocktail consists of two modules: chunk-level quantization search and chunk-level KV cache computation. Chunk-level quantization search determines the optimal bitwidth configuration of the KV cache chunks quickly based on the similarity scores between the corresponding context chunks and the query, maintaining the model accuracy. Furthermore, chunk-level KV cache computation reorders the KV cache chunks before quantization, avoiding the hardware inefficiency caused by mixed-precision quantization in inference computation. Extensive experiments demonstrate that Cocktail outperforms state-of-the-art KV cache quantization methods on various models and datasets. Our code is presented on https://github.com/Sullivan12138/Cocktail.**

***Index Terms*—long-context LLM inference, KV cache, chunk-level quantization search, chunk-level KV cache computation**

I. Introduction

Large language models have found widespread applications across various domains [1]–[4]. To enhance their expressive capabilities, the context length of these models has been steadily increasing [5], [6]. However, extended context also introduces significant challenges. Specifically, longer context lengths lead to a notable increase in inference latency and a substantial rise in memory consumption. The primary reason for the excessive inference latency and high memory consumption in LLMs with long contexts is the KV cache. For instance, in the case of the Llama2 13B model, a context with a length of 128K requires around 100GB of memory to store the KV cache, whereas a single NVIDIA A100 GPU has only 80GB of memory. Besides, during inference, the KV cache must frequently be loaded from the GPU to the GPU's high-speed cache, which is a significant contributor to the high inference latency.

Various methods have been proposed to compress the KV cache, such as attention pruning [7]–[9], removing or quantizing unimportant tokens [10]–[12], and considering optimizations from a system perspective [13]–[15]. Among them, the simplest and most effective approach is quantization. Quantization means converting the KV cache from the original FP16 type to a lower bitwidth type (such as INT8, INT4, or even INT2). However, most existing KV cache quantization efforts [16]–[19] uniformly quantize the KV cache to a single bitwidth. In reality, the importance of tokens within the KV cache is hierarchical, meaning some tokens are more important than others. Therefore, uniform quantization can easily cause accuracy degradation. Some papers [20]–[23] have proposed implementing mixed-precision quantization based on the importance of the tokens to address this issue, but their quantization search strategies are token-level and very time-consuming. Besides, mixed-precision quantization can cause hardware inefficiency during the inference computation.

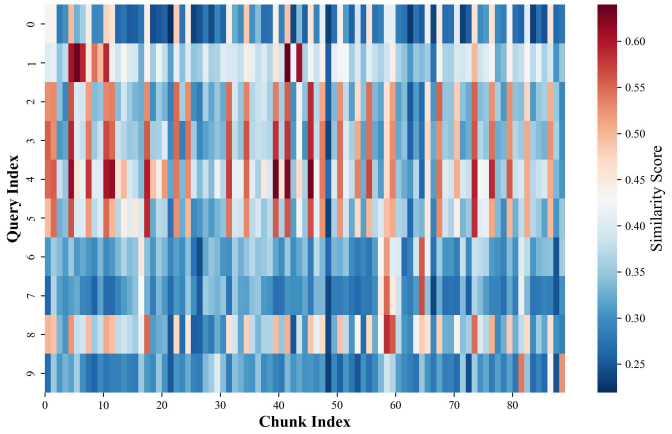

Fig. 1. The similarity heatmap between a long passage and 10 different queries. Most of the passage chunks are irrelevant to the query.

In this paper, we propose to apply the mixed-precision quantization to the KV cache at a larger granularity, i.e., the chunk level. We select a long passage (which can be split into 89 chunks) to simulate a long context and create 10 different queries, calculating the similarity between them. As shown in Figure 1, for each query, there is only a small portion of chunks that are highly relevant, while most are irrelevant. Obviously, for the KV cache corresponding to context chunks that are highly relevant to the query, we need to retain their precision; otherwise, the model would lose critical information. For the KV cache corresponding to context chunks that are irrelevant to the query, quantizing them to very low bits will not significantly affect the model's accuracy. Based on these observations, we propose a chunk-adaptive mixed-precision KV cache quantization method called Cocktail.

Cocktail mainly contains two modules: chunk-level quantization search and chunk-level KV cache computation. Chunk-

*Jigaung Wan (email: jgwan@hust.edu.cn) and Jianzong Wang (email: jzwang@188.com) are the corresponding authors.

978-3-9826741-0-0/DATE25/© 2025 EDAA

level quantization search calculates the similarity scores between the query and each context chunk and determines the bitwidth configuration of the KV cache chunks corresponding to the context chunks according to those scores. This module maintains the accuracy of LLMs quickly and effectively. We further design the chunk-level KV cache computation module to avoid hardware inefficiency. Chunk-level KV cache computation reorders the context KV cache chunks to ensure the same bitwidth configuration in which KV cache chunks are arranged contiguously in physical memory. This module significantly optimizes the inference latency and memory usage during LLM inference.

In summary, our contributions can be outlined as follows:

- We propose a chunk-adaptive mixed-precision KV cache quantization method called Cocktail for long-context LLM inference.
- Cocktail contains the chunk-level quantization search module, which determines the bitwidth configuration of KV cache chunks based on the similarity scores between the corresponding context chunks and the query, maintaining the inference accuracy of the LLM.
- Cocktail contains the chunk-level KV cache computation module, which reorders KV cache chunks in different bitwidth configurations before quantization, avoiding the issue of hardware inefficiency during inference caused by mixed-precision quantization.
- Extensive experiments validate that Cocktail achieves better model accuracy, lower latency, and reduced memory usage compared to state-of-the-art (SOTA) methods.

II. Background

A. LLM Inference

The computation in LLMs inference primarily revolves around the attention module, with the core focus being the calculation of the Q, K, V matrices [24]. LLM inference consists of a prefill phase and several decode phases. During the prefill phase, the LLM processes the input tokens (i.e., the concatenation of context and query) to generate the first output token. Subsequently, in each decode phase, the LLM processes the input token along with all previously generated output tokens to produce the next token. Due to the feature of attention computation, the LLM needs to compute the K and V matrics corresponding to the input tokens and all previously generated output tokens, which is very time-consuming. Therefore, later researchers proposed caching the generated K and V matrices after processing the input tokens and the previously output tokens, known as KV Cache, so that they do not need to be recomputed in subsequent decode phases. However, as the length of the context increases [5], [6], the number of input tokens grows, and the KV cache also becomes larger, leading to severe memory and latency issues.

B. KV Cache Compression

Past researchers have proposed various methods to compress the KV Cache. Some have suggested pruning the attention module within the KV Cache [7]–[9]. For example, Shazeer [7] proposed MQA, where all attention heads share a single set of KV parameters. Some have proposed evicting unimportant tokens from the KV cache [10]–[12]. For instance, Xiao et al. [11] introduced streamingLLM, which retains only the most recent sliding window of KV Cache along with the initial tokens. Zhang et al. [10] proposed Q-Hitter, which evaluates the importance of attention based on the sum of attention scores across each token's column in the attention matrix. Others have approached the problem from a system perspective [13]–[15]. For example, Kwon et al. [14] introduced PageAttention, which applies the memory paging concept from traditional operating systems by mapping logically contiguous KV Cache pages to physically non-contiguous pages through a page table. Dao et al. [13] proposed FlashAttention, which rewrites certain complex operators to be processed as much as possible within the GPU's high-speed cache. However, these methods are not as easy to implement as quantitative methods, and their optimization results are also not as effective as those achieved by quantitative methods.

C. LLM Quantization

The quantization work for LLMs initially focused on quantizing weights and activations [25]–[30]. For example, Xiao et al. [25] proposed SmoothQuant, which scales weights that are easy to quantize and activations that are difficult to quantize separately. However, during inference with long contexts, the KV cache often becomes larger than the weights and activations. Therefore, the quantization work for LLMs has been extended to the KV cache as well [16]–[19]. For instance, Zhao et al. [18] introduced Atom, which performs group quantization of the KV cache to low bits, where each group has independent quantization parameters. Liu et al. [16] proposed KIVI, which applies per-channel quantization to the K cache while keeping the original per-token quantization for the V cache. However, these quantization methods that uniformly quantize tokens to the same bitwidth cannot maintain model accuracy effectively. Other researchers have proposed mixed-precision quantization methods to process important tokens in LLMs [20]–[23]. For example, Kim et al. [20] proposed SqueezeLLM, which divides the weights into dense matrices without outliers and sparse matrices containing outliers, then applies low-bit quantization to the dense matrices while keeping the outliers in FP16 precision. Yang et al. [21] proposed MiKV, which identifies unimportant tokens based on attention scores but quantizes them instead of evicting them. Hooper et al. [22] introduced KVQuant, which uses a custom data type called nuqX to represent the mixed-precision quantized KV Cache. These methods use token-level quantization search, which is very time-consuming, and they have not adequately addressed the hardware inefficiency issues brought by mixed-precision quantization.

III. Method

A. Architecture Overview

The architecture of Cocktail is shown in Figure 2. The long context is first segmented into several equal-length, short chunks (If the length of the context is not divisible by the

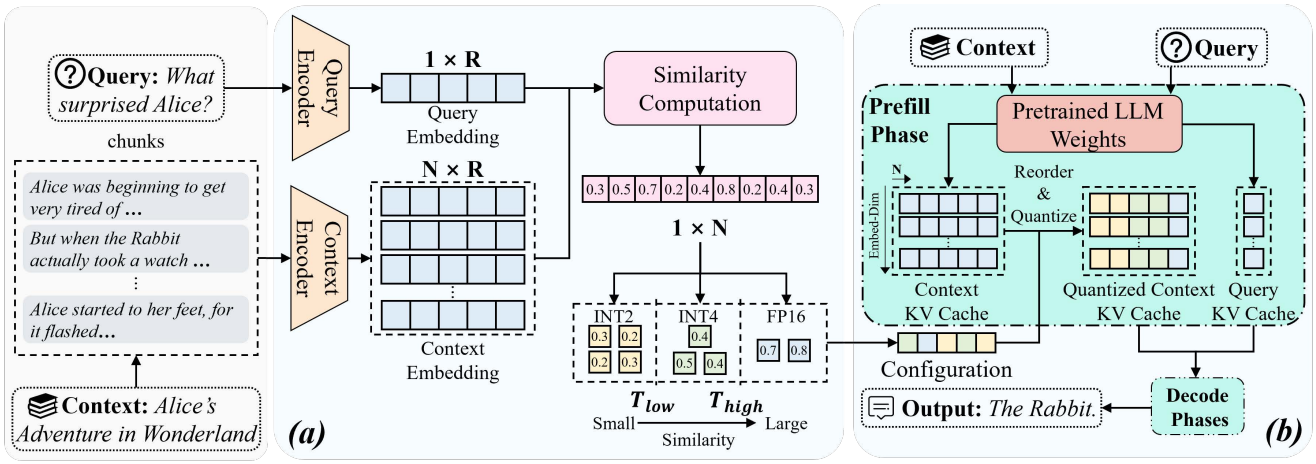

Fig. 2. The architecture overview of Cocktail. (a) The chunk-level quantization search module. (b) The chunk-level KV cache computation module.

chunk size, we truncate the portion at the end of the context that cannot be divided by the chunk size. The KV cache of this portion will be kept in FP16 precision). Then, the chunk-level quantization search receives and processes the context chunks and the query, outputting the quantization bitwidth configuration. Next, the context chunks and the query are fed into the pre-trained LLM for formal inference. During the prefill phase, we reorder the KV cache chunks generated from the context chunks. The reordered KV cache chunks output from this module are subsequently quantized according to the configuration provided by the chunk-level quantization search module. Since the quantized context KV cache in Figure 2 resembles a horizontal three-layered cocktail with distinct colors, we name our method *Cocktail*. Finally, after several decode phases, the pre-trained LLM outputs the answer. To maintain model accuracy, we only quantize the KV cache of the context while retaining FP16 precision for the KV cache of the query and output tokens generated in the decode phases. Given that, in long-context LLM inference datasets, the length of the context is significantly larger than that of the query and output, we believe this strategy will not result in significant memory or inference latency overhead.

B. Chunk-level Quantization Search

Chunk-level quantization search draws inspiration from the concept of RAG (Retrieval-Augmented Generation) [31]. RAG retrieves documents similar to the input query from a third-party corpus. These retrieved documents are concatenated with the query to improve the model's understanding. To retrieve the appropriate documents, RAG employs an encoder to encode both the query and the candidate documents, calculates the similarity between the encoded vectors, and selects the top-k documents with the highest similarity. In our chunk-level quantization search, we use the encoder from RAG to distinguish the similarity between different context chunks and the query so that we can quickly determine the quantization bitwidth configuration for their corresponding KV cache chunks. Specifically, we use the Facebook-Contriever [32] model as both the context encoder and the query encoder.

As shown in Figure 2(a), context chunks and query are sent separately to the chunk encoder and query encoder. After encoding, we obtain the context chunk embeddings and query embedding. Next, we calculate the cosine similarity between the query embedding and each context chunk embedding, resulting in a similarity score list. The cosine similarity formula is:

$$sim(q, c_i) = \frac{q \cdot c_i}{||q|| \times ||c_i||} \quad i = 1, 2, ..., N \tag{1}$$

where q is the query embedding, c_i is the i_{th} chunk embedding, N is the number of chunk embeddings. $||\cdot||$ means the L2 norm.

Subsequently, we set two thresholds, T_{low} and T_{high} ($0 < T_{low} < T_{high} < 1$). We compare the similarity score at each index in the similarity score list with these two thresholds. For a similarity score greater than T_{high}, the context chunk at that index is considered highly relevant to the query, and we set the bitwidth configuration of its corresponding KV cache chunk as FP16. For a similarity score less than T_{low}, the context chunk at that index is considered to have little relevance to the query, and we set the bitwidth configuration of its corresponding KV cache chunk as INT2. For a similarity score between T_{low} and T_{high}, we adopt a compromise strategy, setting the bitwidth configuration of its corresponding KV cache chunk as INT4. We use two hyperparameters α and β to control the values of these two thresholds:

$$T_{low} = s_{min} + (s_{max} - s_{min}) \times \alpha \tag{2}$$

$$T_{high} = s_{max} - (s_{max} - s_{min}) \times \beta \tag{3}$$

where s_{min} and s_{max} are the minimal and maximal values in the similarity score list, respectively. We will discuss the influence of different α and β on the model performance in Section IV-C.

C. Chunk-level KV Cache Computation

Directly applying mixed-precision quantization can result in chunks with different bitwidth that are physically contiguous, which can lead to hardware inefficiency during inference computation. For example, modern GPUs use cache lines to optimize data reading, but data with different bitwidths cannot

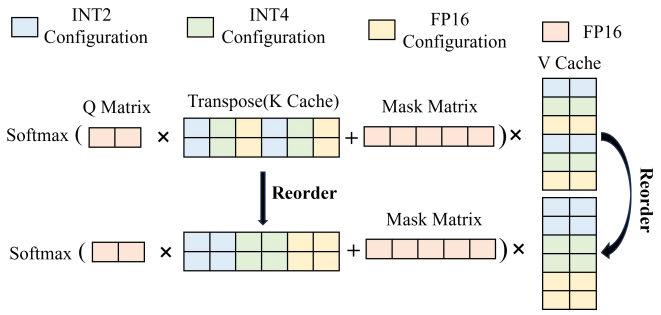

Fig. 3. The process of KV cache chunk reordering.

be aligned properly, potentially spanning multiple cache lines. This requires loading more cache lines, increasing the cache miss rate and memory access frequency, which significantly raises inference latency. Besides, if the hardware is designed to read multi-byte data simultaneously, such as single instruction multiple data (SIMD) architecture, some of the narrower bitwidth data may only use part of the hardware resources, leading to GPU memory waste.

Therefore, we design the chunk-level KV cache computation module to address the hardware inefficiency problem. The algorithm pseudocode is shown in Algorithm 1. Specifically, during the prefill phase, we first adopt KV cache chunk reordering. As shown in Figure 3, the KV cache chunks with the same bitwidth configuration are arranged together, making them contiguous in physical memory. Next, these chunks are quantized following the bitwidth configuration determined by the KV cache quantization search module. During the decode phase, we multiply the Q matrix with the transposed three blocks of the K matrix with three different bitwidths to obtain three attention matrix blocks. These blocks are then concatenated along the last dimension to form the attention matrix. The attention matrix is then added to the mask matrix and processed with the softmax function. The processed attention matrix is divided into three blocks again, and each is multiplied by the V matrix blocks with the corresponding bitwidth to produce three small output matrices. The final output matrix is obtained by summing these three small output matrices.

We prove that the output obtained in this way is the same as the output from the traditional computation method: In traditional computation method, assuming there are N context chunks, then K matrix can be divided into a block matrix of the form $[K_1|K_2|...|K_N]^T$, and V matrix can be divided into a block matrix of the form $[V_1|V_2|...|V_N]^T$, both of which are partitioned along the token length dimension. Obviously, attention matrix A can also be divided into $[A_1|A_2|...|A_N]$. According to block matrix multiplication, the final output matrix O is:

$$O = [A_1V_1 + A_2V_2 + ... + A_NV_N] \tag{4}$$

Now, if we perform KV cache chunk reordering, it is equivalent to changing the order of the sub-blocks within the K and V matrices. Suppose after reordering, K becomes $[K_{x_1}|K_{x_2}|...|K_{x_N}]^T$, and V becomes $[V_{x_1}|V_{x_2}|...|V_{x_N}]^T$, where $\{x_1, x_2, ..., x_N\}$ is a permutation of $\{1, 2, ..., N\}$. Since the softmax function is not affected by the order of the matrix

Algorithm 1: Pseudocode of Chunk-level KV Cache Computation in a Pytorch-like Style

```
# s: the similarity score list
# N: the number of context chunks
# K, V: the K, V cache of the context
# T_low, T_high: the two thresholds
# During the prefill phase:
for i in range(N):
   if s[i] < T_low:
      K_int2.append(K[i])
      V_int2.append(V[i])
   elif s[i] > T_high:
      K_fp16.append(K[i])
      V_fp16.append(V[i])
   else:
      K_int4.append(K[i])
      V_int4.append(V[i])
K_q2, K_q4 = quant(K_int2), quant(K_int4)
V_q2, V_q4 = quant(V_int2), quant(V_int4)
# Q: the Q vector of the current token
# mask: attention mask matrix
# len_2, len_4 = len(K_q2), len(K_q4)
# During the decode phase:
att = fqm(Q,transpose(K_q2),2)
att = cat(att,fqm(Q,transpose(K_q4),4),-1)
att = cat(att,mm(Q,transpose(K_fp16)),-1)
att = softmax(att + mask)
output = fqm(att[:len_2)],V_q2)
output += fqm(att[len_2:len_2+len_4],V_q4)
output += mm(att[len_2+len_4:],V_fp16)
```

quant: the quantization function; fqm: FP16 matrix and quantized matrix multiply; mm: FP16 matrix multiply; cat: concatenation.

sub-blocks, the sub-blocks of matrix A will eventually be arranged in the same order as the sub-blocks within K, which means A is $[A_{x_1}|A_{x_2}|...|A_{x_N}]$. The final output O' will be:

$$O' = [A_{x_1}V_{x_1} + A_{x_2}V_{x_2} + ... + A_{x_N}V_{x_N}] \tag{5}$$

which is obviously equal to the matrix O due to the commutative invariance of matrix addition.

TABLE I
EVALUATION DATASET AND METRICS.

Dataset	Task	Evaluation Metric
Qasper	Single-Document QA	F1-score
QMSum	Summarization	ROUGE score
MultiNews	Summarization	ROUGE score
TREC	Few-shot Learning	Classification score
TriviaQA	Few-shot Learning	F1 score
SAMSum	Few-shot Learning	ROUGE score
LCC	Code Completion	Similarity score
RepoBench-P	Code Completion	Similarity score

IV. EVALUATION

A. Experiment Setup

Models. We evaluate Cocktail on four famous LLM models: Llama2-7B [33], Llama2-13B [33], Mistral-7B [34] and Longchat-7B [35], where Longchat-7B is a specified fine-tuned model for chat tasks. The maximum context length of the first two models is 4K, while this number of the other two models is 32K. The output length is set as 128.

TABLE II
PERFORMANCE COMPARISON OF COCKTAIL AND DIFFERENT BASELINE KV CACHE QUANTIZATION METHODS.

Model	Method	Qasper	QMSum	MultiNews	TREC	TriviaQA	SAMSum	LCC	RepoBench-P	Average
Llama2-7B	FP16	9.63	21.32	3.47	66	87.74	41.81	66.62	59.77	44.55
	Atom	9.03	20.14	2.8	65.2	87.46	41.25	66.78	59.02	43.96
	KIVI	9.18	20.66	1.38	65.7	87.42	41.53	66.45	59.43	43.97
	KVQuant	9.24	21.02	3.08	65.8	87.48	41.52	66.94	58.93	44.25
	Cocktail	**9.67**	**21.21**	**3.21**	**66**	**87.66**	**42.08**	**67.58**	59.1	**44.56**
Llama2-13B	FP16	9.44	21.25	3.74	70.3	88.02	43.89	66.64	56.82	45.01
	Atom	8.63	20.34	4.56	69.4	86.75	43.18	64.22	55.35	44.05
	KIVI	8.58	20.69	4.39	69.5	87.03	43.31	65.08	55.46	44.26
	KVQuant	8.66	20.87	4.55	70	87.85	43.34	65.38	55.76	44.55
	Cocktail	**8.79**	**21.03**	**4.82**	70	87.37	**43.97**	**66.15**	**56.65**	**44.85**
Mistral-7B	FP16	32.99	24.24	27.1	71	86.23	42.96	54.02	51.92	48.8
	Atom	30.54	23.66	26.35	70.32	85.79	42.18	53.27	51.64	47.97
	KIVI	30.72	23.98	26.93	67.8	86	43.34	53.59	51.73	48.39
	KVQuant	31.32	23.85	26.73	70.44	86.07	42.23	53.85	51.67	48.27
	Cocktail	**32.67**	**24.11**	**27.07**	**71**	**86.36**	**43.62**	53.16	**51.94**	**48.74**
Longchat-7B	FP16	29.41	22.77	26.61	66.5	83.99	40.9	52.94	56.78	47.49
	Atom	28.44	21.56	25.75	64.8	83.62	40.77	50.4	54.58	46.24
	KIVI	28.69	22.59	26.28	66.5	83.19	40.28	52.4	55.13	46.88
	KVQuant	29.6	22.48	26.38	66.52	83.59	40.82	51.46	55.33	47.02
	Cocktail	**31.88**	**22.65**	**26.43**	66.5	**83.88**	**41.1**	51.95	**55.42**	**47.48**

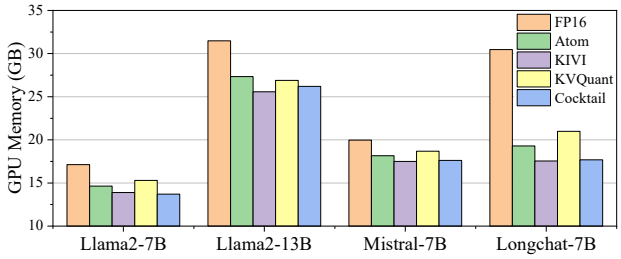

Fig. 4. GPU memory of different models.

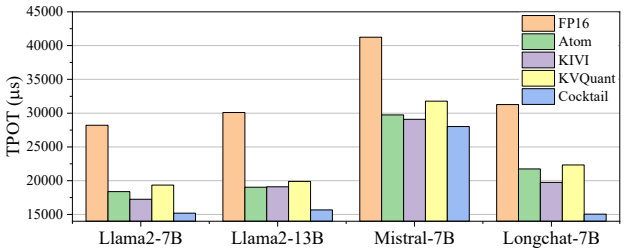

Fig. 5. Time per output token (TPOT) of different models.

Datasets. We adopt LongBench [36] benchmark for long context evaluation, the datasets in which are shown in Table I.

Baselines. We only select three representative SOTA methods for comparison due to page limitations. They are Atom [18], representing trivial uniform quantization; KIVI [16], representing per-channel key quantization and per-token value quantization; and KVQuant [22], representing token-level mixed-precision quantization. The KV Cache is quantized to INT2 and INT4 uniformly in Atom and KIVI, respectively. As for KVQuant, a small portion (1%) of the KV Cache retains FP16 precision, while the rest is quantized to INT4. Atom also includes functionality for quantizing weights and activations. However, to ensure a fair comparison, we do not use this functionality, focusing solely on the methods' ability to quantize the KV cache.

Hardware. We conduct all the experiments on an NVIDIA L40S GPU containing 48GB GPU memory, with a 25-core AMD EPYC 7T83 CPU and 100GB memory.

B. Performance Comparison

We compare the performance of Cocktail and the baseline methods on four models over eight different long-context datasets with α and β set as 0.6 and 0.1, chunk size set as 32.

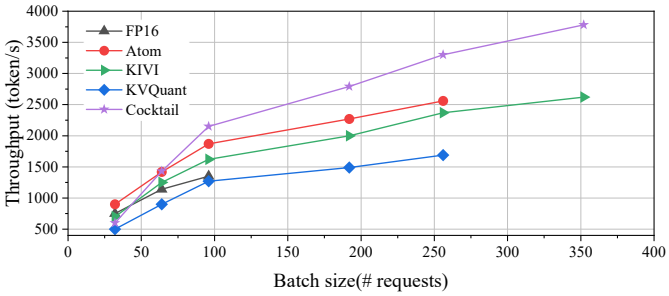

Fig. 6. The throughput of different models with different batch sizes. The interrupted lines in the figure are due to OOM (Out of Memory).

The results are shown in Table II. Cocktail achieves the best performance over most of the datasets, and the best average performance on all four models. Additionally, Cocktail has the least performance loss compared to the original FP16 precision model, with its performance score dropping by an average of only 0.055 compared to the original model. This is because we use chunk-level mixed-precision quantization to successfully maintain the precision of those important context parts.

Furthermore, we compare the GPU memory usage and time per output token (TPOT) of Cocktail and other methods on

TABLE III
THE IMPACT OF DIFFERENT CHUNK SIZE ON MODEL PERFORMANCE.

Chunk Size	8	16	32	64	128	256
Rouge Score	21.24	21.19	21.21	20.15	19.82	16.35

the QMSum dataset over four models. As shown in Figure 4, Cocktail can reduce the GPU memory usage by 12%-42% compared to the original FP16 precision model. Figure 5 shows that Cocktail has the lowest TPOT, reducing by 32%-52% compared to the FP16 precision model. The improvements of Cocktail in GPU memory usage and TPOT are due to the use of chunk-level KV cache computation.

Besides, we evaluate the throughput of Cocktail and the baseline methods with different batch sizes on the QMSum dataset. As shown in Figure 6, with the increase of batch size, the throughput of Cocktail is initially lower than the uniform quantization methods, but it gradually surpasses them. This is because when batch size is small, the latency of the chunk-level quantization search process limits the throughput of Cocktail. However, when the batch size is large, this latency is negligible compared with the LLM inference process. Since Cocktail's TPOT is smaller than that of the uniform quantization methods, its throughput is naturally higher. The throughput of Cocktail is always higher than KVQuant because our chunk-level quantization search is faster than its token-level quantization search.

C. Analysis

We evaluate the impact of α and β on the Llama2-7B model on QMSum dataset. As shown in Figure 7, the model's accuracy worsens as α increases and improves as β increases. When β increases to a certain extent, further changes in the model's accuracy become less noticeable. This is because a larger α means more context is quantized to INT2. Conversely, a larger β means more context is retained in FP16 precision. However, when β becomes sufficiently large, since only a few chunks of the context are closely related to the query, retaining more chunks in FP16 precision doesn't contribute much to improving model accuracy. We also evaluate the impact of different chunk sizes. As shown in Table III, when the chunk size is smaller than 32, the model performance stays steady. But when the chunk size is larger than 32, the model accuracy drops quickly. This is because the important context parts within a chunk of large size can be surrounded by many unimportant parts, leading to incorrect quantization to lower bits.

Furthermore, we explore different context and query encoder architectures in the chunk-level quantization search module. The results are shown in Table IV. We select four prevalent encoder: ADA-002 [37], BM25 [38], LLM Embedder [39], and Facebook-Contriever [32]. Over four different datasets, the Facebook-Contriever encoder has the best performance. Therefore, we choose it as our context and query encoder.

D. Ablation Study

We conduct ablation studies to prove the effect of the two modules in Cocktail. The experiments are conducted on the QMSum dataset on Llama2-7B model, with results shown in Table V (The two modules are referred to as module I and II, respectively). Without chunk-level quantization search, the model accuracy can drop drastically. On the other hand, without chunk-level KV cache computation, GPU memory usage and TPOT can increase significantly. This experiment illustrates the effect of chunk-level mixed-precision quantization on maintaining model accuracy and the effect of chunk-level KV cache computation on reducing GPU memory usage and latency.

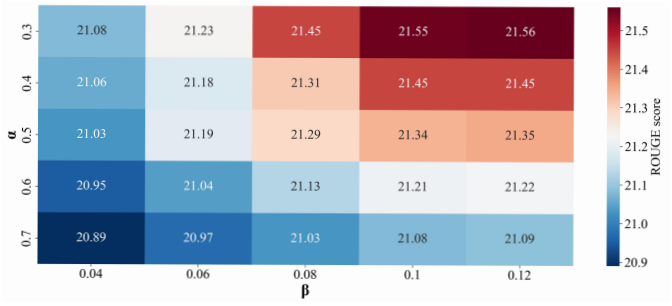

Fig. 7. The impact of α and β to model performance.

TABLE IV
PERFORMANCE COMPARISON OF DIFFERENT CONTEXT ENCODER AND QUERY ENCODER ON LLAMA2-7B OVER FOUR DATASETS.

Method	Qasper	SAMSum	TriviaQA	RepoBench-P
Baseline	9.52	41.69	87.72	59.82
ADA-002 [37]	8.89	41.34	86.69	58.32
BM25 [38]	7.36	35.67	84.22	55.35
LLM Embedder [39]	8.98	41.56	87.85	59.77
Facebook-Contriever [32]	9.67	42.08	88.06	60.1

TABLE V
THE IMPACT OF CHUNK-LEVEL QUANTIZATION SEARCH AND CHUNK-LEVEL KV CACHE COMPUTATION.

Method	F1 score ↑	GPU Memory(GB) ↓	TPOT(μs) ↓
Baseline	21.32	17.13	28214
w/o Module I	19.33	13.88	15258
w/o Module II	21.27	20.6	29653
Cocktail	21.21	13.7	15201

V. CONCLUSION

This paper introduces Cocktail, a chunk-adaptive mixed-precision KV cache quantization method for long-context LLM inference. Cocktail uses a chunk-level quantization search module to determine the bitwidth configuration of context KV cache chunks. It also contains a chunk-level KV cache computation module, reordering these KV cache chunks to avoid hardware inefficiency. Extensive experiments on multiple datasets and models demonstrate that Cocktail outperforms state-of-the-art KV cache quantization methods.

VI. ACKNOWLEDGEMENTS

This work was sponsored by the Key Research and Development Program of Guangdong Province under grant No. 2021B0101400003, the National Key Research and Development Program of China under Grant No.2023YFB4502701.

REFERENCES

[1] Chao Xiao, Yifei Deng, Zhijie Yang, Renzhi Chen, Hong Wang, Jingyue Zhao, Huadong Dai, Lei Wang, Yuhua Tang, and Weixia Xu. Llm-based processor verification: A case study for neuromorphic processor. In *2024 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, 2024.

[2] Jiayi Yuan, Ruixiang Tang, Xiaoqian Jiang, and Xia Hu. Large language models for healthcare data augmentation: An example on patient-trial matching. In *AMIA Annual Symposium Proceedings*, volume 2023, page 1324. American Medical Informatics Association, 2023.

[3] Chenxi Sun, Hongyan Li, Yaliang Li, and Shenda Hong. Test: Text prototype aligned embedding to activate llm's ability for time series. *arXiv preprint arXiv:2308.08241*, 2023.

[4] Chao Xiao, Yifei Deng, Zhijie Yang, Renzhi Chen, Hong Wang, Jingyue Zhao, Huadong Dai, Lei Wang, Yuhua Tang, and Weixia Xu. Llm-based processor verification: A case study for neuronnorphic processor. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2024.

[5] Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*, 2023.

[6] Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. Yarn: Efficient context window extension of large language models. *arXiv preprint arXiv:2309.00071*, 2023.

[7] Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

[8] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

[9] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D Lee, Deming Chen, and Tri Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774*, 2024.

[10] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[11] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*, 2023.

[12] Chi Han, Qifan Wang, Wenhan Xiong, Yu Chen, Heng Ji, and Sinong Wang. Lm-infinite: Simple on-the-fly length generalization for large language models. *arXiv preprint arXiv:2308.16137*, 2023.

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

[14] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

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

[16] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*, 2024.

[17] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

[18] Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. Atom: Low-bit quantization for efficient and accurate llm serving. *Proceedings of Machine Learning and Systems*, 6:196–209, 2024.

[19] Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. Qserve: W4a8kv4 quantization and system co-design for efficient llm serving. *arXiv preprint arXiv:2405.04532*, 2024.

[20] Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W Mahoney, and Kurt Keutzer. Squeezellm: Dense-and-sparse quantization. *arXiv preprint arXiv:2306.07629*, 2023.

[21] June Yong Yang, Byeongwook Kim, Jeongin Bae, Beomseok Kwon, Gunho Park, Eunho Yang, Se Jung Kwon, and Dongsoo Lee. No token left behind: Reliable kv cache compression via importance-aware mixed precision quantization. *arXiv preprint arXiv:2402.18096*, 2024.

[22] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *arXiv preprint arXiv:2401.18079*, 2024.

[23] Shichen Dong, Wen Cheng, Jiayu Qin, and Wei Wang. Qaq: Quality adaptive quantization for llm kv cache. *arXiv preprint arXiv:2403.04643*, 2024.

[24] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[25] 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.

[26] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

[27] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems*, 35:30318–30332, 2022.

[28] 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. *arXiv preprint arXiv:2310.01801*, 2023.

[29] Tim Dettmers, Ruslan Svirschevski, Vage Egiazarian, Denis Kuznedelev, Elias Frantar, Saleh Ashkboos, Alexander Borzunov, Torsten Hoefler, and Dan Alistarh. Spqr: A sparse-quantized representation for near-lossless llm weight compression. *arXiv preprint arXiv:2306.03078*, 2023.

[30] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems*, 6:87–100, 2024.

[31] 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.

[32] Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. Unsupervised dense information retrieval with contrastive learning. *arXiv preprint arXiv:2112.09118*, 2021.

[33] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[34] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[35] Dacheng Li, Rulin Shao, Anze Xie, Ying Sheng, Lianmin Zheng, Joseph Gonzalez, Ion Stoica, Xuezhe Ma, and Hao Zhang. How long can context length of open-source llms truly promise? In *NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following*, 2023.

[36] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*, 2023.

[37] OpenAI. New and improved embedding model. https://openai.com/index/new-and-improved-embedding-model/. December 15, 2022.

[38] Jimmy Lin, Xueguang Ma, Sheng-Chieh Lin, Jheng-Hong Yang, Ronak Pradeep, and Rodrigo Nogueira. Pyserini: An easy-to-use python toolkit to support replicable ir research with sparse and dense representations. *arXiv preprint arXiv:2102.10073*, 2021.

[39] Yuhuai Wu, Markus N Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers. *arXiv preprint arXiv:2203.08913*, 2022.