A Artifact Appendix

A.1 Abstract

We provide an artifact to demonstrate that representative `KV cache` compression methods can reduce memory consumption. Current implementations (e.g., FlashAttention, PagedAttention) do not optimize for production-level LLM serving, resulting in suboptimal throughput performance. We also provide our tools to facilitate future LLM `KV cache` compression studies.

A.2 Description and Requirements

Github Repository. The codes are available in the following Github repository https://github.com/LLMkvsys/rethink-kv-compression.

Hardware dependencies. Experiments are conducted on one NVIDIA A6000 GPU.

Software dependencies. We provide a Docker image with NVIDIA GPU support for this artifact. We use CUDA 12.1, torch 2.1.2, transformers 4.43.1, and LMDeploy v6.0.1 (modified).

A.3 Setup

To install the artifact, users should clone the repository.

```
git clone git@github.com:LLMkvsys/
    rethink-kv-compression.git
cd rethink-kv-compression/
conda env create -f mlsys_environment.
    yml
conda activate lmdeploy
conda clean -a
cd benchmark_thr/src/
pip install -e .
```

We also provide a Dockerfile to help you build a docker image.

A.4 Evaluation workflow

Given the significant time and resource costs associated with length and negative sample analysis, we do not recommend executing these scripts during artifact evaluation. Instead, we suggest focusing on throughput analysis (Figure 1). We have provided scripts to facilitate the reproduction of Figure 1 using a single GPU. However, the first row in Figure 1 involves running models with HF transformers, which can be notably slow in this context. As a result, we have not included corresponding scripts for this specific part. **We recommend users follow the README as it provides more detailed explanations.**

A.5 Major Claims

The KV cache compression methods show negative computational efficiency in certain scenarios of batch size, sequence length. Most KV cache compression methods except Gear show advantages when serving requests with a heavy KV cache.

A.6 Experiments

It might take one hour to finish the following experiments. We provide scripts to measure the prefill and decoding throughput of quantization-based approaches across various sequences with a fixed batch size. The benchmarking logs are stored in folders *0_quant_normal_logs* and *0_quant_long_logs*, respectively.

```
cd benchmark_thr/src
bash 0_quant_normal_logs/
    batch_eval_quant_normal_fixbsz.sh
bash 0_quant_long_logs/
    batch_eval_quant_long_fixbsz.sh
```

The following scripts are to measure the prefill and decoding throughput of sparsity-based approaches across various sequences with a fixed batch size. The benchmarking logs are stored in folders *0_sparse_normal_logs* and *0_sparse_long_logs*, respectively.

```
cd benchmark_thr/src
bash 0_sparse_normal_logs/
    batch_eval_sparse_normal_fixbsz.sh
bash 0_sparse_long_logs/
    batch_eval_sparse_long_fixbsz.sh
```

We provide scripts to measure the prefill and decoding throughput of quantization-based approaches across various batch sizes with a fixed sequence length. The benchmarking logs are stored in folders *0_quant_normal_logs* and *0_quant_long_logs*, respectively.

```
cd benchmark_thr/src
bash 0_quant_normal_logs/
    batch_eval_quant_normal_fixlen.sh
bash 0_quant_long_logs/
    batch_eval_quant_long_fixbsz.sh
```

We provide scripts to measure the prefill and decoding throughput of quantization-based approaches across various batch sizes with a fixed sequence length. The benchmarking logs are stored in folders *0_sparse_normal_logs* and *0_sparse_long_logs*, respectively.

```
cd benchmark_thr/src
bash 0_sparse_normal_logs/
    batch_eval_sparse_normal_fixlen.sh
bash 0_sparse_long_logs/
    batch_eval_sparse_long_fixbsz.sh
```

A.7 Experiments Results

We provide the following plotting scripts to generate Figures 1 (e)–(i), respectively. The resulting figures are saved in the folder *demo_figs/*. To account for system noise, we recommend that users verify the presence of two key findings: (1) certain KV cache compression methods fail to outperform FP16 baselines; (2) the decoding throughput advantages of KV cache compression techniques become more pronounced in scenarios with heavy KV cache usage.

```
cd benchmark_thr/src
# Figure 1 (e) & (i)
python 0_plot/plot_normal_fixbsz.py
# Figure 1 (f) & (j)
python 0_plot/plot_normal_fixseqlen.py
# Figure 1 (g) & (k)
python 0_plot/plot_long_fixbsz.py
# Figure 1 (h) & (i)
python 0_plot/plot_long_fixseqlen.py
```

A.8 Additional Experiments

We conduct length analysis and negative analysis and provide our tools for your reference. Since this process requires substantial GPU resources, we offer precomputed, cached results within our prepared environment. The cached results comprise the generated responses and associated length and evaluation metric score.

First, we provide cached results to reproduce Figure 3.

```
cd benchmark_len/
python -u plot_kde_shift_dist.py
```

This script yields the distribution of response length difference across various compression algorithms. The results are saved in the current directory. The users can observe that with the increase of the compression ratio, the distribution of response difference flattens, and more samples experience verbose response.

Second, we provide cached results to reproduce Figure 5.

```
cd benchmark_neg/
python -u 0_ratio_vs_no_negative.py
```

This script outputs the figures in which the number of negative samples changes with the threshold. The results are saved in the current directory. The users can find that there are many negative samples even with a threshold of 10%, indicating the fragility of compression algorithms.

B Evaluation Details

B.1 Dataset

ShareGPT. We select a subset of requests from ShareGPT (Anon, 2024) to conduct the experiments of response length difference distribution. We refer to the benchmark code in vLLM[3] to sample 1, 000 requests. Due to time and resource constraints, we set the maximum number of generation tokens as 1024 in the evaluation. We also truncate contexts for input prompts that exceed the model's maximum length allowance to ensure they fit within the model's capacity.

LongBench. LongBench (Bai et al., 2023) is a task for long context understanding that covers key long-text application scenarios, including multi-document QA, single-document QA, summarization, few-shot learning, code completion, and synthetic tasks. We keep strictly the evaluation metrics and settings in their released codebase [4] to ensure fair assessments.

B.2 Models

LLaMA Family. The LLaMA family, developed by Meta using a high-quality corpus, is widely favored by researchers working on `KV cache` compression algorithms. Many choose LLaMA models to evaluate the effectiveness of their methods. In our performance evaluation, we cover LLaMA-2-7B, LLaMA-2-13B, and LLaMA-2-70B to emphasize the advantages of `KV cache` due to their exorbitant GPU memory consumption of `KV cache`. Additionally, LLaMA-3.1-8B, known for generating high-quality responses and excelling in long-context tasks, is used in our length distribution and negative sample analysis.

Mistral Family. Similarly to the LLaMA family, many models from the Mistral family are used to demonstrate the benefits of `KV cache` algorithms. Mistral models incorporate grouped-query attention (GQA) for faster inference and are renowned for their exceptional performance. In our length difference and negative sample analysis, we utilize Mistral-7B-v0.1 to obtain relevant experimental results.

B.3 Algorithms

KIVI. KIVI (Liu et al., 2024e) is a notable quantization algorithm for `KV cache` compression, specializing in per-channel quantization for key tensors and per-token quantization for value tensors. We utilize their official implementation [5]. The critical hyperparameters in KIVI are group size G and the residual length R. G refers to the number of

[3] https://github.com/vllm-project/vllm/blob/main/benchmarks/benchmark_serving.py

[4] https://github.com/THUDM/LongBench

[5] https://github.com/jy-yuan/KIVI

channels that are grouped for quantization in the key cache, while R controls the number of most recent tokens that are kept in full precision. Following the paper's recommendations for achieving optimal performance, we have set them to $G = 32, R = 128$.

GEAR. GEAR (Kang et al., 2024) is a typical quantization error mitigation algorithm. We took their open-source code[6]. The key parameters of GEAR are sparsity ratio s and rank r. Specifically, s specifies the number of retained full-precision outlier values. r controls the richness of the low-rank approximation matrix, which recovers the model's ability from quantization errors. In line with the default settings in the official codebase, we set $s = 2\%, r = 2\%$.

StreamingLLM. StreamingLLM (Xiao et al., 2023) is an attention sparsity-based cache eviction algorithm. It retains only a limited number of initial and most recent tokens. The key parameters for controlling the sizes of the initial and recent tokens are set to 64 and 448, respectively, resulting in a total cache size of 512.

H2O. H2O (Zhang et al., 2024f) is another widely-used cache eviction algorithm that dynamically calculates and refreshes the `KV cache`. The parameters for the heavy hitter oracle token size and the recent size are configured to 64 and 448, respectively, with a total cache size of 512.

B.4 LLM Serving Engine.

Transformers Library. We directly use Torch 2.1.2 and Transformers 4.43.1 to measure the throughput performance.

FlashAttention. FlashAttention[7] can fully exploit the GPU resources to realize fast and memory-efficient attention operation. In our throughput evaluation, we measure the throughput performance of TRL+FA by enabling FlashAttention 2.5.6 in the transformers library.

LMDeploy. LMDeploy[8] allows LLM developers to compress, deploy, and serve various LLMs. It naturally supports the functionality of PagedAttention and FlashAttention. We implement various `KV cache` algorithms based on LMDeploy v6.0.1. We chose LMDeploy for three reasons.

First, LMDeploy stands out by implementing more efficient quantization kernels than vLLM. This results in superior performance in KV cache compression approaches compared to vLLM, despite the significant attention vLLM has garnered. Note that the primary focus of our paper is on KV cache compression approaches, with a particular emphasis on quantization. A prior benchmark study conducted by BentoML (BentoML) uncovers that LMDeploy obtains the best throughput performance with 4-bit quantization.

Second, LMDeploy offers a better way for faster development of KV cache compression algorithms than vLLM. The author of KVI has stated the challenges of integrating the KIVI algorithm into vLLM as early as April 2024 (kiv). As of now, there has been no significant progress on this front.

Third, our conclusions, except for Observation 2, do not pertain to any specific serving features of the inference engines. Consequently, they remain independent of the LLM inference engine used. For Observation 2, our objective is to explore the impact of KV cache compression methods like sparsity and quantization on popular serving features (e.g., Page Attention, Flash Attention) rather than focusing on any particular serving engine. As long as the selected inference engines support the efficient implementation of the necessary serving features (Page Attention, FlashAttention), it will not affect Observation 2.

B.5 Hardware Environment.

Our evaluation experiments are conducted on a GPU node with four NVIDIA A6000 GPUs interconnected via NVLink and powered by an Intel Xeon Gold 6326 CPU at 2.90 GHz.

C More results of Throughput Analysis

We evaluate throughput performance on a GPU node with four NVIDIA A6000 GPUs interconnected via NVLink and powered by an Intel Xeon Gold 6326 CPU at 2.90 GHz. We exclude the initialization overhead and average the throughput performance over three times for fair comparison. We add more experiments to demonstrate the generality of our statement in the throughput analysis as follows.

First, we measure the prefill and decoding throughput on TRL, TRL+FA, and LMD using Mistral-7B and LLaMA-13B, depicted in Figure 8 (a-b) and Figure 10 (a-b), respectively. The relative speedup of the StreamingLLM algorithm in the decoding throughput varies across LLMs and serving techniques, as shown in Figure 8 (c-d) and Figure 10 (c-d). The high speedup from TRL does not mean the significant speedup benefits.

Second, we measure the prefill and decoding throughput on LMD with various batch and prompt lengths in Figure 8 (e-h) and Figure 10 (e-l). We have observed that these Large Language Models (LLMs) show negative speedup in certain prompt lengths and batch sizes, which is consistent with the statement mentioned in Section 4. However, the prompt lengths and batch sizes that lead to this disadvantage vary among different LLMs. Worth noticing that in Figure 10, we

[6] https://github.com/opengear-project/GEAR

[7] https://github.com/Dao-AILab/flash-attention

[8] https://github.com/InternLM/lmdeploy/tree/main

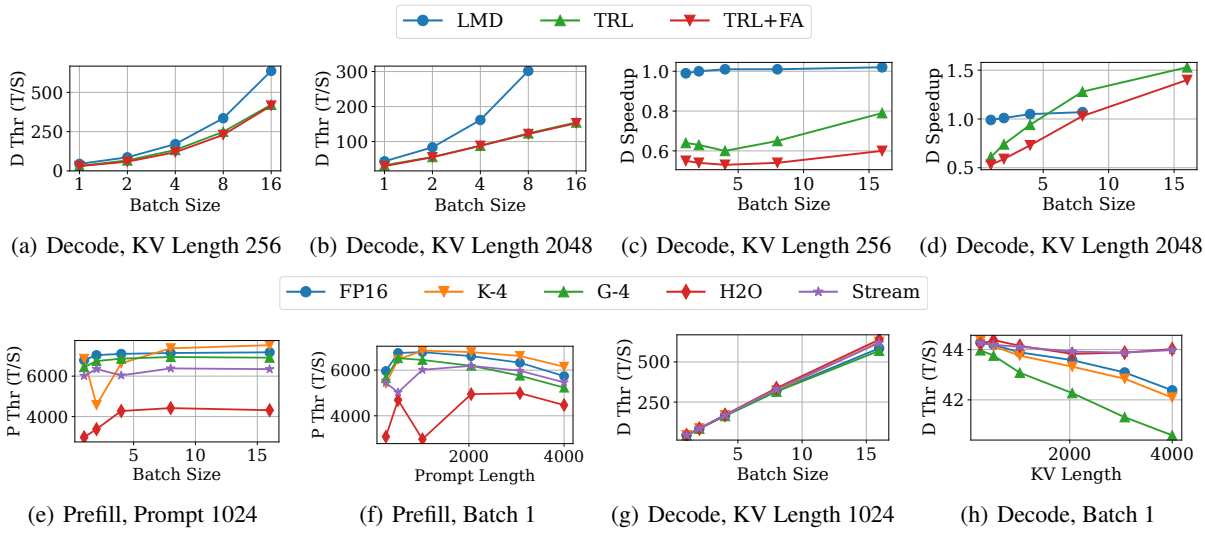


Figure 8. Throughput analysis of **Mistral-7B** (a-b) The FP16 decoding throughput on TRL (with and without FlashAttention) and LMDeploy (LMD). (c-d) The speedup of the KIVI-4bit algorithm on TRL and LMD. (e-h) The prefill and decoding throughput for inputs of moderate size.

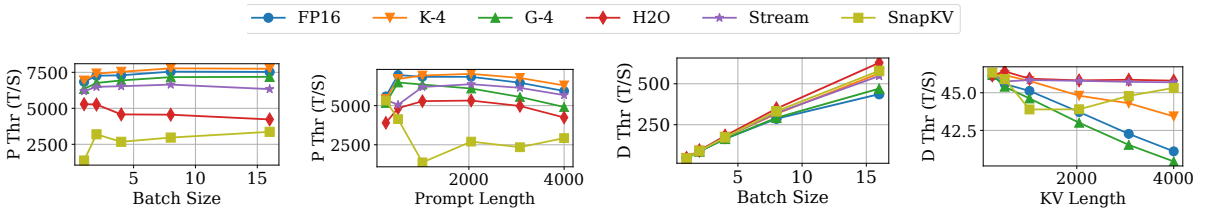


Figure 9. Throughput analysis of **LLaMA-7B**, with `KV cache` compression algorithm SnapKV (Li et al., 2024b) integrated.

omit the throughput information for the KIVI-4 algorithm due to the out-of-memory issue when evaluating on LLaMA-13B with a single A6000 GPU.

Third, we present additional findings on tensor parallelism in Figures 11, 12, 13, and 14. Performance improvements with larger tensor parallelism (TP) are notably evident during the prefill stage for various compression methods. However, larger TP does not confer significant benefits in the decoding stage when the batch size is small. Our observations indicate that the throughput advantages derived from `KV cache` compression typically become more pronounced under heavy evaluation settings (e.g., batch size, KV length, and model size).

D More Results of Length Analysis

First, we outline the configurations of the compression algorithms in Section 4.3. For text generation, we fix the temperature as 1 for the FP16 baseline and compression methods. We also vary T to assess the impact of length differences from the hyperparameter T in Table 5. For quantization-based methods, we only vary the quantization bits for KIVI and GEAR. For sparsity-based methods, we only vary the `KV cache` length for StreamingLLM and H2O. All other compression-related configurations remain consistent with those detailed in Appendix B.3.

Second, we supply more experimental results on Mistral-7B to demonstrate the generality of our statement in **Observations 3 and 4**, respectively. Particularly, we perform a similar experimental analysis as Table 5 on Mistral-7B and show the ratio (%) of samples experiencing response length variations induced by temperature and `KV cache` compression in Table 9. Similar to the LLaMA model, `KV cache` compression shows a clear tendency to produce verbose responses in Mistral-7B. We also repeat the experiments in Figure 4 and show the results of Mistral-7B in Figure 15. The impact of the compression ratio on the response length remains consistent between the LLaMA and Mistral. We also measure the end-to-end latency for various compression algorithms on Mistral-7B, as shown in Figure 16. Our observation is that the latency benefits of `KV cache` compression are not prominent, and the verbose response length should be accounted for performance measurement of various `KV cache` compression.

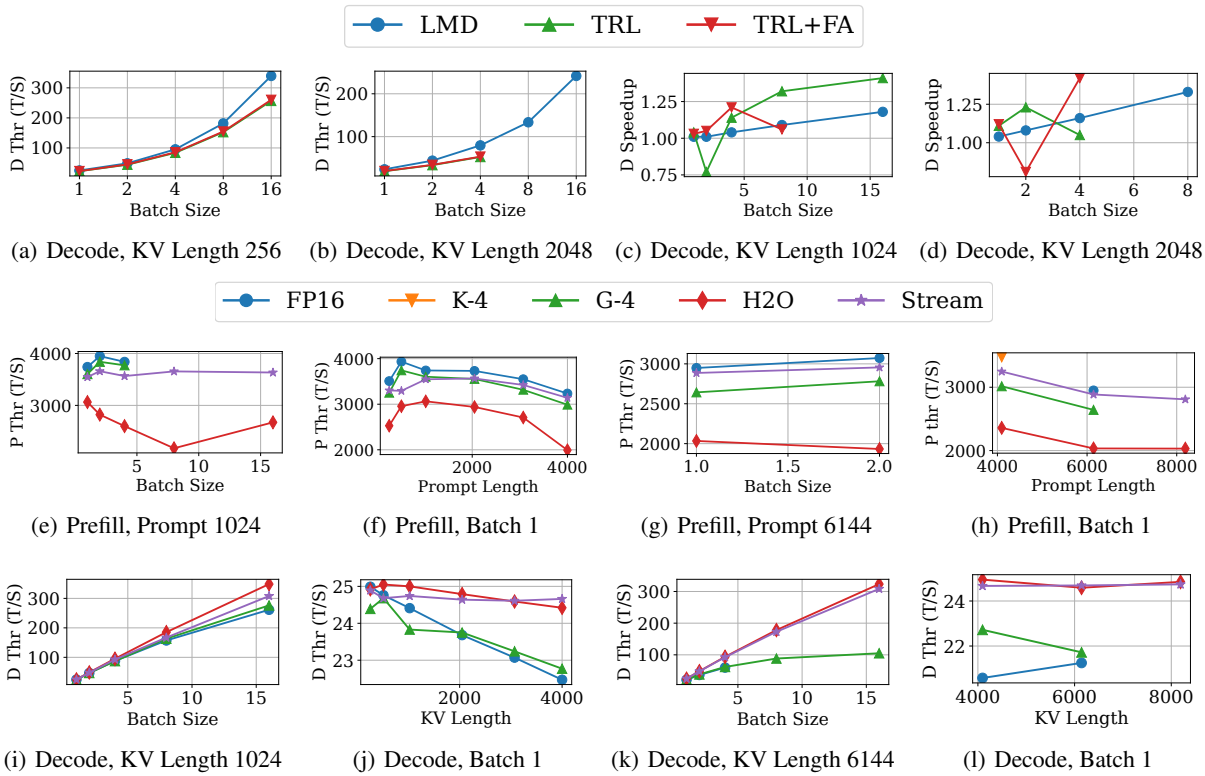


Figure 10. Throughput analysis of **LLaMA-13B**: (a-b) The FP16 decoding throughput on TRL (with and without FlashAttention) and LMDeploy (LMD). (c-d) The speedup of the StreamingLLM algorithm on TRL and LMD. (e-h) The prefill throughput for various sizes of inputs. (i-l) The decoding throughput for various sizes of inputs.

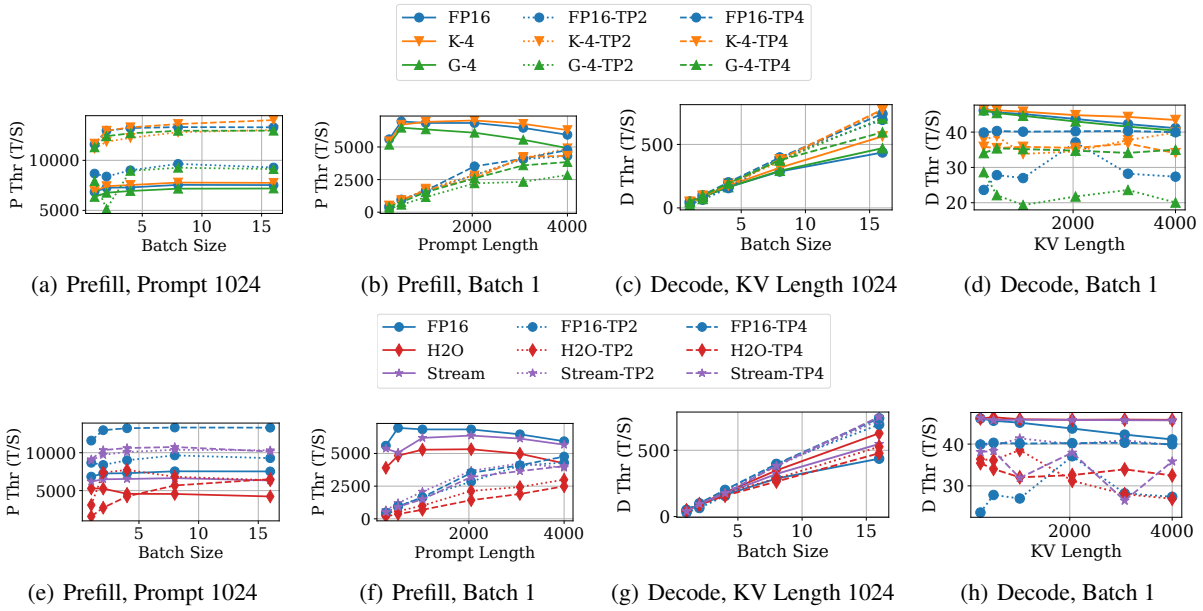
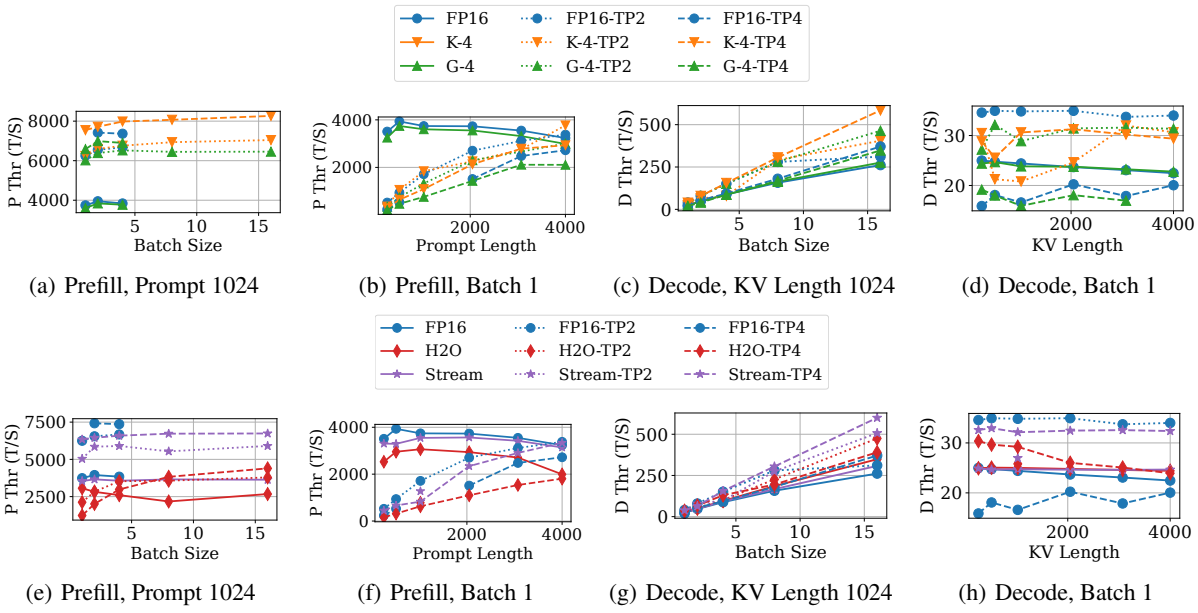


Figure 11. Throughput analysis of **LLaMA-7B**, with different tensor parallelism configurations. (a-d) The throughput of quantization-based methods. (e-h) The throughput of sparsity-based methods.

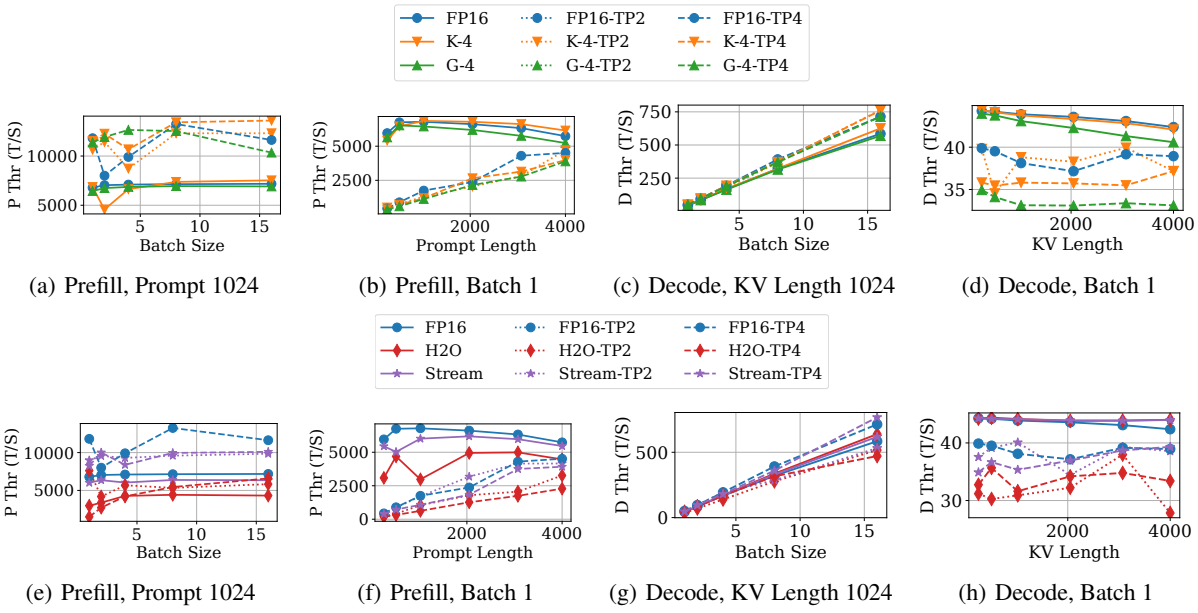
Third,



(a) Prefill, Prompt 1024 (b) Prefill, Batch 1 (c) Decode, KV Length 1024 (d) Decode, Batch 1

(e) Prefill, Prompt 1024 (f) Prefill, Batch 1 (g) Decode, KV Length 1024 (h) Decode, Batch 1

Figure 12. Tensor parallelism analysis of **LLaMA-13B**. (a-d) The throughput of quantization-based methods. (e-h) The throughput of sparsity-based methods.



(a) Prefill, Prompt 1024 (b) Prefill, Batch 1 (c) Decode, KV Length 1024 (d) Decode, Batch 1

(e) Prefill, Prompt 1024 (f) Prefill, Batch 1 (g) Decode, KV Length 1024 (h) Decode, Batch 1

Figure 13. Tensor parallelism analysis of **Mistral-7B**. (a-d) The throughput of quantization-based methods. (e-h) The throughput of sparsity-based methods.

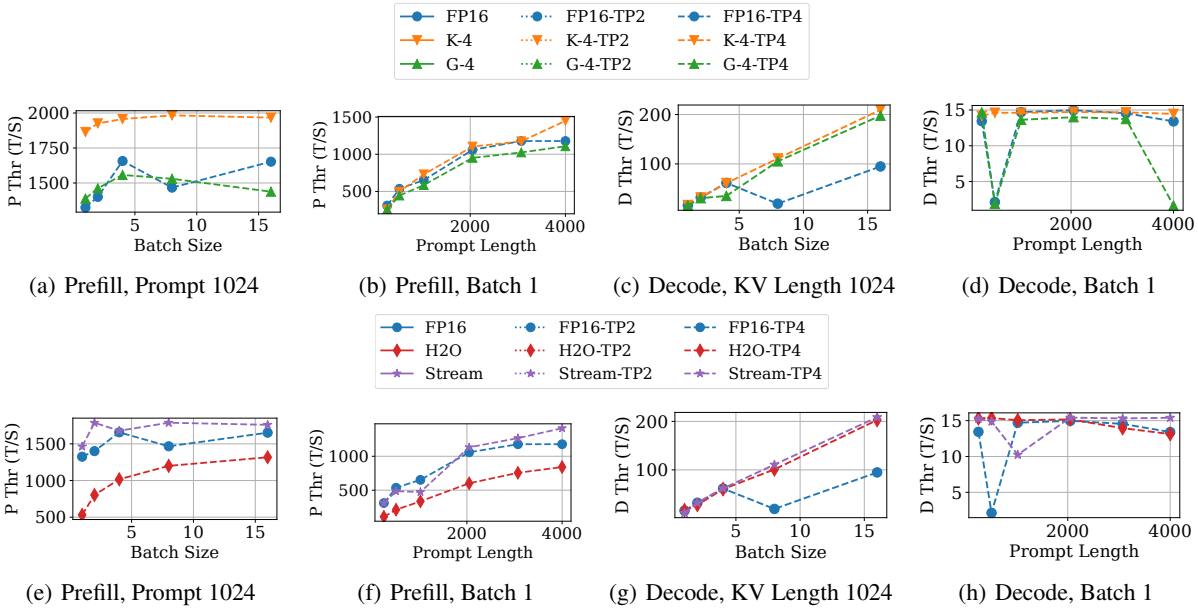
Table 9. The results of length analysis similar to Table 5, but measured on Mistral-7B.

Metric	T=0.9	T=1.1	KIVI	GEAR	H2O	Stream
% of samples D of which $\geq 50\%$	45.1%	45.9 &	2.8%	0.8%	11.0%	17.3%
% of samples D of which $\leq -50\%$	17.7%	20.0 %	44.9%	49.4%	14.3%	16.3%

E More Results of Negative Sample Analysis

First, the detailed task description of the LongBench used in the negative sample analysis can be found in https://huggingface.co/datasets/THUDM/LongBench#task-description. It contains the detailed task description of the LongBench dataset, including the task name, task type, evaluation metric, and average length. We use the corresponding task type to collect the number of negative samples.

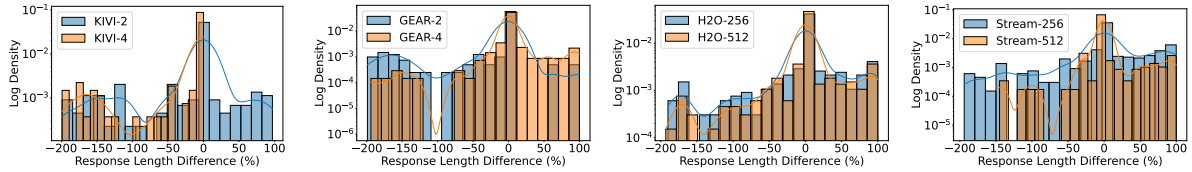
Section 3.2 suggests that compression algorithms excel in proceeding short prompt lengths with no accuracy loss. Thus, we use LongBench and Llama-3.1-8B-instruct to con-



(a) Prefill, Prompt 1024 (b) Prefill, Batch 1 (c) Decode, KV Length 1024 (d) Decode, Batch 1

(e) Prefill, Prompt 1024 (f) Prefill, Batch 1 (g) Decode, KV Length 1024 (h) Decode, Batch 1

Figure 14. Tensor parallelism analysis of **LLaMA-70B**. (a-d) The throughput of quantization-based methods. (e-h) The throughput of sparsity-based methods.



(a) KIVI (b) GEAR (c) H2O (d) StreamingLLM

Figure 15. The Mistral-7B's distribution of response length difference across different compression algorithms.

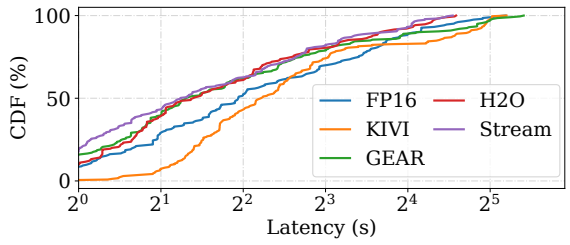


Figure 16. The Mistral-7B's CDF of the end-to-end latency (seconds) of various algorithms.

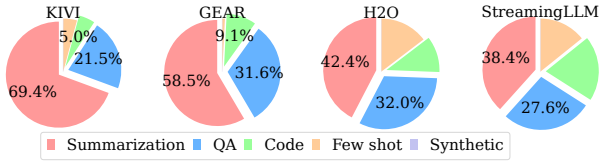
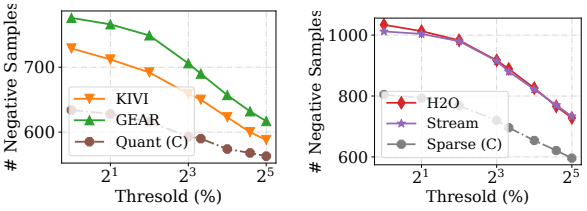


Figure 18. The pie chart details the proportion of negative samples over task types across varying compression algorithms on Mistral.



(a) Quantization (b) Sparsity

Figure 17. The threshold (x-axis) versus the number of negative samples (y-axis) for quantization-based (a) and sparsity-based (b) methods. The experimental results are measured on Mistral-7B.

duct negative sample analysis. We assess the average scores of LLaMA-3.1-8B-instruct for baseline, KIVI, GEAR, H2O, and StreamingLLM on the LongBench test dataset are 41.2, 41.3, 40.9, 39.1, and 38.9, respectively. We also cover more experimental results about negative sample analysis on Mistral-7B in Section 3.2. In Mistral-7B, the average scores for baseline, KIVI, GEAR, H2O, and StreamingLLM on the LongBench test dataset are 33.3, 33.4, 33.4, 31.8, and 30.4, respectively. First, we vary the threshold in Algorithm 1 to uncover the relationship between the threshold and the number of negative samples on Mistral-7B, as shown in Figure 15. We conclude that the minor accuracy loss from compression algorithms does not indicate that each sample

experiences a minor performance loss. It is not easy to eliminate the existence of negative samples. Second, we present the sensitivity of task types to `KV cache` compression in Figure 18 on Mistral. Similar to LLaMA, performing `KV cache` compression on Mistral-7B considerably affects the accuracy performance on summarization and QA tasks. Overall, the experimental results further reinforce our statement in Section 4.4.

Table 10. The accuracy of the length predictor for Mistral-7B.

Tools	FP16	KIVI	GEAR	H2O	Stream
Length Predictor	92.6%	92.3%	88.8%	92.8%	89.5%

F THROUGHPUT PREDICTOR

We use Vidur's released code[9] to realize the throughput predictor. The runtime time information of each operator in LLMs is profiled on A6000 NVIDIA RTX. The key difference between different compression algorithms and the FP16 baseline hinges upon the attention operation. Hence, apart from attention operators, all other operators are reused among different `KV cache` compression algorithms. We enumerate various combinations of batch sizes, sequence lengths, and stages to attain ample profiled runtime speed information for LLaMA-7B and Mistral-7B. Vidur provides the implementation code to construct and optimize the throughput predictor. We define the accuracy as $(1 - \frac{|T^{\text{pred}} - T^{\text{gt}}|}{T^{\text{gt}}}) \times 100\%$.

G LENGTH PREDICTOR

We collect the response length information from ShareGPT to synthesize the response length dataset. To account for the long-context prompt, we choose LongFormer with a maximum sequence size of 4096. We set the input of the length predictor as the input response and the target of the length predictor as the ratio between the response length and the prompt length. We define the accuracy as $(1 - \frac{|L^{\text{pred}} - L^{\text{gt}}|}{L^{\text{gt}}}) \times 100\%$. Table 6 has reported the prediction results on LLaMA3-8B. We include the prediction results on Mistral-7B in the second row of Table 10. Overall, the bert-based length predictor can deliver accurate response length prediction for LLaMA and Mistral models.

Table 11. The measured score of various algorithms evaluated on the negative sample benchmark dataset and Mistral-7B using LongBench's provided metric.

Task Type	Baseline	KIVI	GEAR	H2O	Stream
Summarization	27.2	15.2	16.6	15	11.1
Question Answering	26.8	17.6	16.4	18.0	15.4
Code	90.8	47.5	47.5	64.7	59.6

[9] https://github.com/microsoft/vidur

H PERFORMANCE ON NEGATIVE BENCHMARK

We use the Mistral-7B and report the corresponding measured score on the negative sample benchmark dataset in Table 11.