Custom Project: Don’t Drop It, Compress It: Selective KV Quantization

Anushka Singh asing155@jh.edu
Anushri Suresh asures13@jh.edu
Mrishika Nair mnair12@jh.edu
Zongbo Bao bzongbo1@jh.edu

Abstract

As LLMs continue to support longer contexts, KV cache memory is expected to become a major bottleneck, often rivaling the model’s own size. Existing solutions either discard tokens or apply uniform compression, overlooking token relevance. In this work, we propose a selective KV quantization method that preserves high-precision representations for key tokens (i.e., sink and recent tokens) while aggressively compressing less relevant tokens. We adapt compression strategies based on content, aiming to improve memory efficiency and reduce inference latency. This hybrid approach delivers up to 2× memory savings with minimal loss in perplexity and ROUGE, offering an effective trade-off between efficiency and accuracy. We evaluate on language modeling (WikiText-2) and summarization (CNN/DailyMail), using perplexity and ROUGE-L as metrics.

1 Motivation

Large Language Models (LLMs) such as GPT-4 (OpenAI et al. [2024]) and LLaMA-2 (Touvron et al. [2023]) have demonstrated impressive capabilities across various natural language tasks. However, as these models scale in size and support longer context windows, the computational cost of inference has grown substantially. While acceleration techniques like optimized attention mechanisms (Shazeer [2019]), model pruning (Michel et al. [2019]), and speculative decoding (Leviathan et al. [2023]) have been explored, one major inefficiency remains the Key-Value (KV) cache (Vaswani et al. [2023]), whose memory footprints scale linearly with sequence length.

Recent findings (Shi et al. [2024],Xiao et al. [2024]) reveal that generating 28k tokens with an LLaMA-2 7B model may consume over 14 GB of KV cache alone, comparable to the model’s own parameter size. To address KV cache inefficiencies, existing methods either discard older tokens (e.g., sliding window (Beltagy et al. [2020]), attention sinks (Xiao et al. [2024]) or selectively retain important ones using attention-guided strategies (e.g., PyramidKV (Cai et al. [2024]) and ZipCache (He et al. [2024]). While quantization techniques (Zirui Liu et al. [2023], Hooper et al. [2024]) reduce memory by compressing all tokens, they typically apply compression uniformly, without regard for individual importance.

2 Hypothesis

We hypothesize that importance-aware selective KV cache quantization, which preserves key tokens in high precision and aggressively compresses less relevant ones, can improve memory efficiency without significantly degrading generation quality. Inspired by StreamingLLM (Xiao et al. [2024]) and ZipCache (He et al. [2024]), our method dynamically adapts compression based on token relevance to better scale LLMs for long-context generation.

3 Related Work

Recent work (Xiao et al. [2024]) has identified KV caching as a major bottleneck for LLMs, with memory usage exceeding model size at 20–30k tokens and cache access latency dominating inference time at longer sequence lengths. Early methods addressed this by limiting cache size through eviction strategies. Sliding Window Attention (Beltagy et al. [2020]) maintains a fixed-size window, discarding older tokens but risking performance degradation when long-range dependencies are needed. StreamingLLM (Xiao et al. [2024]) improved this by retaining a small set of initial "attention sink" tokens alongside recent ones, enabling infinite-length streaming without retraining. More dynamic strategies like PyramidKV (Cai et al. [2024]) and H2O (Zhang et al. [2023]) selectively retain important tokens based on attention dynamics. SnapKV (Li et al. [2024]) clusters and retains only essential token features per head, while LazyLLM (Fu et al. [2024])progressively prunes tokens layer-by-layer, reducing KV cache size to make later computations cheaper.

Parallel to eviction, quantization strategies have been applied to KV caches to reduce memory and bandwidth demands. Techniques such as KIVI (Zirui Liu et al. [2023]) and KVQuant (Hooper et al. [2024]) show that reducing KV precision to 2–3 bits can cut memory usage significantly while maintaining low perplexity, and Coupled Quantization (Zhang et al. [2024])) exploits inter-channel redundancy to reach 1-bit per channel. More recently, ZipCache (He et al. [2024]) explored importance-aware quantization by allocating higher bit-widths to salient tokens based on attention scores, though its saliency estimation introduces additional computational overhead.

Despite these advances, most work either discards old context or uniformly compresses the cache while few explore dynamically adapting compression based on token relevance. This motivates our proposed approach: Selective KV Cache Quantization, where less important tokens are compressed more aggressively while preserving crucial ones at higher fidelity.

4 Methods

4.1 Model Selection

We use the LLaMA 3.1–8B model for all experiments. This model offers a strong trade-off between performance and memory usage, and is widely available in open-weight form, enabling reproducibility and easy modification of the attention and KV caching logic.

4.2 Attention Sink KV Compression

Our primary method is Attention Sink KV Compression, based on the observation that autoregressive transformers disproportionately attend to initial prompt tokens over the course of generation, as shown in figure 1. We leverage this by:

- **Sliding Window Mechanism:** A fixed-size window of the most recent L tokens is retained in GPU memory to capture relevant local context.
- **Attention Sink Preservation:** A fixed set of initial prompt tokens—referred to as the "sink"—is always retained and never evicted, ensuring persistent access to influential context.

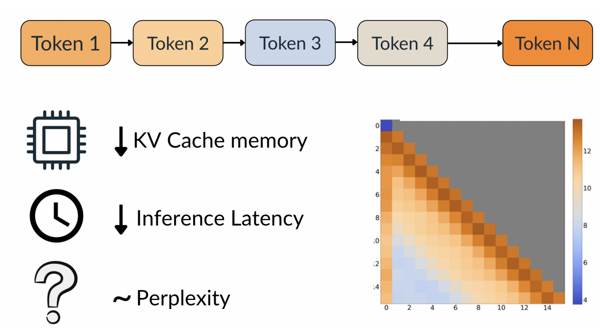

Figure 1: Attention sink compression reduces KV cache memory and inference latency with minimal impact on perplexity. Heatmap shows pruned attention, with blue areas indicating discarded context.

This serves as our *KV Compression Baseline*, replacing the full KV cache with a more compact structure consisting of just the sink and window.

4.3 Our Proposed Method: Partial KV Quantization

While prior approaches discard the KV entries outside the sink and window, we propose a more memory-efficient strategy: *Partial KV Quantization*. Instead of discarding these entries, we quantize them to lower precision, thereby preserving their utility with minimal memory overhead. The key intuition is that older tokens may be less critical but still beneficial to retain in a compressed form.

4.4 Implementation Strategy

4.4.1 Baseline: Full KV Cache

As a baseline, we use the default GPT-Fast framework with standard full KV caching. This provides an upper bound for memory consumption, where all key/value pairs are stored in full precision (fp16) throughout decoding.

4.4.2 Baseline 2.0: Compressed KV Cache

We integrate Attention Sink KV Compression into GPT-Fast using the following mechanism:

- We maintain a fixed-size key/value cache consisting of a sink region (static) and a sliding window (dynamic) for recent tokens.
- During prefill, if the number of input tokens exceeds the sink + window size, we truncate and store only the most relevant tokens.
- During decoding, the cache is updated in-place by evicting the oldest token after the sink and remapping positions to maintain correct attention behavior.

4.4.3 Proposed: Partial KV Quantization

To further reduce memory, we implement partial KV quantization as follows:

- The KV cache is split into two buffers: a full-precision (fp16) buffer for the sink + window, and a quantized (int8) buffer for older tokens.
- As decoding progresses, tokens outside the sink + window are moved from the fp16 buffer into the int8 buffer.
- At attention time, necessary quantized keys/values are dequantized on-the-fly for computation and then discarded. This allows us to retain long-range context with approximately 2× memory savings, since fp16 uses 2 bytes while int8 uses 1 byte.

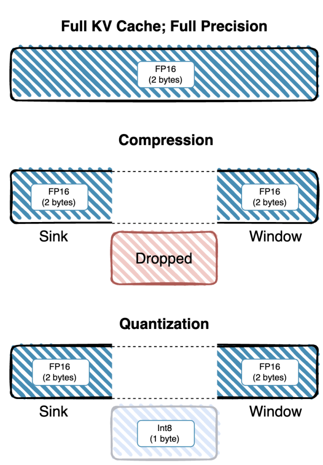

Figure 2: Different Caching Strategies

Algorithm 1 Prefill KV Cache with Compression or Quantization

KV slices for attention
if compress mode **then**
  if $T \leq$ total **then**
    Load all tokens as full precision
  else
    Retain *sink* tokens and last *window* tokens
  end if
  return compressed FP slices
else if quantize mode **then**
  if $T \leq$ total_fp **then**
    Load all tokens into FP cache
    return FP slice
  else
    Store *sink* and *window* in FP16
    Quantize middle tokens $\rightarrow$ int8 + scale
    Store into quant buffers
    return FP + quant buffers via index map
  end if
else
  return (k_{new}, v_{new}) {No compression}
end if

Algorithm 2 KVCache.update: Append token with Compression or Quantization

$k, v \in \mathbb{R}^{B\times H\times 1\times D}$, $pos \in \mathbb{N}$
Updated k_{cache}, v_{cache}
if no compression or quantization **then**
  Insert k, v into cache at position pos
else if compression enabled **then**
  Insert k, v into sliding window at index $(sink + ptr)$
  Update $kv\_positions$ and increment ptr
else if quantization enabled **then**
  if step < mid **then**
    Quantize k, v and store in quant buffers
    Update kv_map with quant slot
  else
    Store k, v in FP window and update kv_map
  end if
  Increment ptr and $cache\_len$
end if
return k_{cache}, v_{cache}

Quantization Mathematical Formulation To quantize a floating-point tensor $x \in \mathbb{R}^{B\times H\times T\times D}$ into 8-bit integers, we use per-token symmetric quantization over the last dimension. Specifically, we define:

$$scale_{b,h,t} = \frac{\max(|x_{b,h,t,:}|)}{127} + \epsilon$$

$$\hat{x}_{b,h,t,d} = clip\left(round\left(\frac{x_{b,h,t,d}}{scale_{b,h,t}}\right), -127, 127\right)$$

where:

- $x_{b,h,t,:}$ is the vector of values for a given token position (b, h, t),
- $scale_{b,h,t}$ is the quantization scale for that token,
- $\hat{x}_{b,h,t,d} \in \mathbb{Z}$ is the quantized int8 output,

4.5 Dataset

We evaluate model performance on two core tasks: language modeling and summarization.

For language modeling, we compute perplexity on the test split of the WikiText-2-raw-v1Merity et al. [2016] dataset, which contains 4,358 Wikipedia-derived text samples. We chose WikiText due to its wide adoption in the language modeling community, high-quality and well-structured text, and relatively small size that enables fast and reproducible evaluation.

For summarization, we use cnn_dailymail Nallapati et al. [2016] dataset.The CNN/DailyMail dataset consists of news articles paired with multi-sentence summaries, originally constructed from the CNN and Daily Mail websites. Each data point includes a full article as input and a human-written summary as the reference. In our summarization experiments, we evaluate the quality of model-generated summaries by computing ROUGE-L scores against the ground truth summaries. We choose this dataset because it is one of the most established benchmarks for text summarization, featuring real-world content with clear structure and concise summaries.

4.6 Evaluation Metrics

We benchmark both memory usage and inference efficiency using the following metrics:

- Peak Memory Usage (GB): Maximum GPU memory consumed during inference, including model weights and KV cache.
- Token Per Second: Number of tokens generated per second; higher means faster generation.
- Perplexity: Measuring next token prediction ability, lower is better.
- ROUGE L: Measuring the similarity between generated summary and reference summary, higher is better.
- Torch Profiler Metrics: Collected detailed runtime statistics including CUDA memory allocation, FLOPs, and kernel-level execution timelines to analyze model efficiency and bottlenecks.

These metrics allow us to quantify trade-offs, aiming for reduced memory usage with minimal or positive impact on inference speed, and model capability.

4.7 Experiments

4.7.1 Setup and Configurations

All experiments were performed on a high-memory GPU server equipped with a single NVIDIA A100 80GB GPU. We used PyTorch and Hugging Face Transformers, with custom modifications to implement the attention sink compression mechanism. Model weights and activations were kept in FP16 precision. The sizes of the attention sink and the sliding context window were fixed during each experiment and treated as task-specific hyperparameters, chosen based on empirical performance and available memory constraints.

Our experiments were divided into two parts:
Memory Evaluation Subset :

- 10 diverse prompts, LLM-generated to reduce prompt-specific bias
- Task: Narrative generation
- Prompt lengths: 10–200 tokens

Perplexity Evaluation Subset :

- Language modeling on the WikiText-2 dataset

- Summarization on the CNN/DailyMail dataset
- Evaluation metric: ROUGE-L score (higher is better)

Each prompt was tested under three conditions:

- **Full cache (Baseline)** : Full KV cache retention without compression.
- **Compressed** : Attention sink with a sliding window applied; older KV pairs evicted.
- **Quantized** : Sink tokens and recent tokens maitained in FP16; older KV pairs quantized to INT8

We used a batch size of 1 (one prompt at a time) to isolate per-sequence behavior. Greedy decoding was used with fixed maximum output lengths (ranging from 40 to 200 tokens) to ensure consistent comparison across runs.

4.7.2 Procedure

For each run, we recorded:

- Peak GPU memory usage via PyTorch profile monitoring. (model + KV cache)
- Tokens-per-second throughput.
- Estimated FLOPs/sec based on model operation counts and runtime.
- Perplexity to measure output accuracy; lower is better

Each prompt was individually evaluated for different caching conditions, and the environment was reset between runs to avoid memory contamination between the baseline and compressed cases.

5 Results and Discussion

We begin by examining the memory breakdown during inference , which clearly shows that KV cache accounts for over 90% of total GPU memory consumption, as shown in the figure 3. This makes it the single most significant bottleneck when scaling LLMs to longer contexts, highlighting why optimizing KV cache storage is essential for practical deployment.

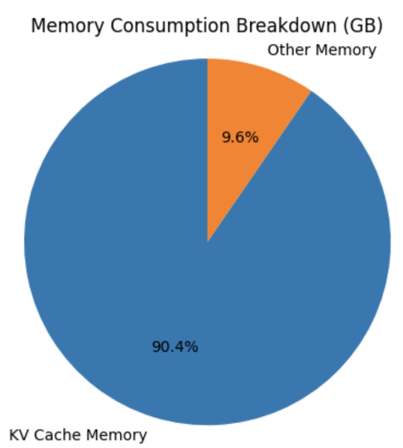

Figure 3: Memory consumption breakdown in GB

To address this, we evaluated three strategies using LLaMA-3.1 8B: Full Cache, Compressed Cache (sink + sliding window), and Selective Quantization. While Compressed Cache achieves $\sim 4\times$ memory savings, it does so at the cost of severe accuracy degradation: perplexity spikes to 132.08 and ROUGE-L drops to 0.1030. In contrast, our quantization approach achieves a strong balance, delivering $\sim 2\times$ memory savings while maintaining nearly identical perplexity (5.56) and only a minor ROUGE-L drop (from 0.2073 to 0.1709).

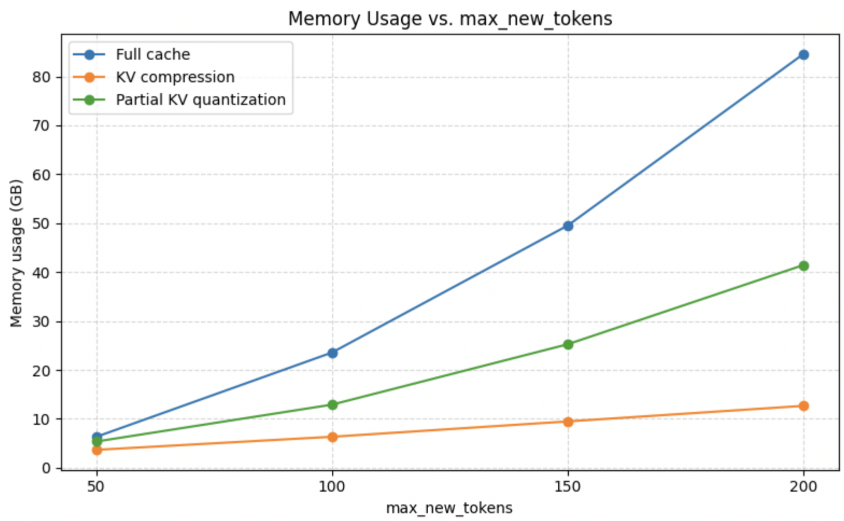

Figure 4: Memory Usage Vs max new tokens across different KV cache strategies

Prompt Length (tokens)	Full Cache (GB)	Compressed (GB)	Quantized (GB)
50	6.33	3.70	5.37
100	23.56	6.33	12.91
150	49.52	9.48	25.25
200	84.58	12.66	41.44

Table 1: CUDA memory usage across different prompt lengths and caching conditions.

Caching Strategy	Perplexity	ROUGE-L	Tokens/sec
Full Cache	5.56	0.2073	6.75
Compressed Cache	132.08	0.1030	5.53
Quantized Cache	5.56	0.1709	4.47

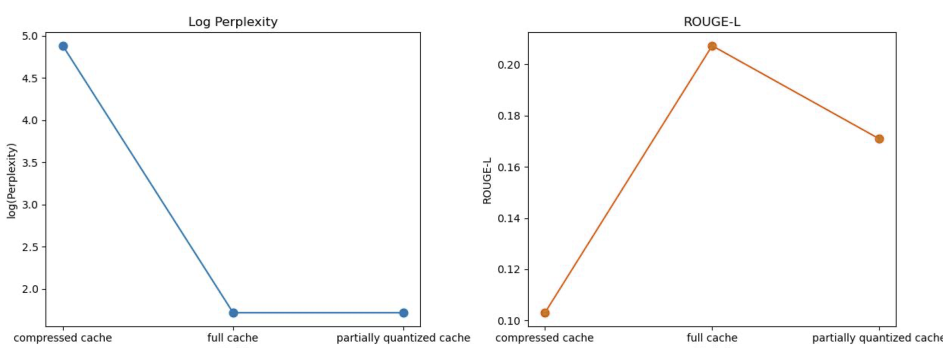

Figure 5: Log perplexity and ROUGE-L under different KV cache strategies, illustrating that selective quantization preserves full-cache performance while aggressive compression degrades it.

Importantly, unlike naive cache dropping, our method preserves recent and high-importance tokens in full precision, while aggressively quantizing the rest. This enables smarter memory use without compromising context or coherence.

Although token throughput dips slightly (4.47 tokens/s vs. 6.75 tokens/s), this is due to on-the-fly dequantization overhead. With future support for custom CUDA kernels, this runtime cost can be reduced significantly, making selective quantization a practical and scalable solution for memory-constrained environments.

6 Future Work

One promising direction is integrating dequantization directly into the attention mechanism via custom CUDA kernels. This would eliminate the overhead of on-the-fly `int8` to `fp16` conversion, boosting tokens-per-second throughput and closing the performance gap with full-precision inference. Future work could also explore mixed-bit quantization strategies—such as combining `fp16` with `int4` or dynamically assigning bitwidths per token or attention head—to push memory savings further without sacrificing accuracy.

Beyond bit-level optimization, adapting the sink and window sizes dynamically could improve memory use based on input context. Lightweight controllers (e.g., MLPs or RL agents) could learn to prioritize tokens for full-precision retention. Incorporating a pointer-generator mechanism would also help preserve rare or salient content by copying directly from the input. Finally, scaling evaluations to longer sequences (10k–30k tokens), broader model families, and more diverse tasks (e.g., code generation, QA) would further validate selective quantization's generalizability, especially when combined with speculative decoding or retrieval-augmented generation.

7 Individual Contributions

All of our code can be found here.

Anushka Singh
Implemented profiling tools and conducted experiments to measure memory usage across different KV caching strategies. Contributed to writing the results, generating plots, and drafting the future work section.

Anushri Suresh
Developed and integrated the compression and selective quantization logic into the GPT-Fast framework. Authored the methods section, detailing both the KV compression and quantization approaches.

Bao Zongbo
Wrote the code for evaluating model capability using perplexity and ROUGE-L metrics. Ran experiments on WikiText-2 and CNN/DailyMail datasets, and contributed to the dataset and metrics section.

Mrishika Nair
Led the integration of the LLaMA-3.1 8B model and implemented baseline caching strategies. Conducted baseline and compressed caching experiments and authored the abstract, introduction, hypothesis and experiments. Designed the project poster.

References

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon

Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical report, 2024. URL `https://arxiv.org/abs/2303.08774`.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023. URL `https://arxiv.org/abs/2307.09288`.

Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019. URL `https://arxiv.org/abs/1911.02150`.

Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one?, 2019. URL `https://arxiv.org/abs/1905.10650`.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding, 2023. URL `https://arxiv.org/abs/2211.17192`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. URL `https://arxiv.org/abs/1706.03762`.

Luohe Shi, Hongyi Zhang, Yao Yao, Zuchao Li, and Hai Zhao. Keep the cost down: A review on methods to optimize llm' s kv-cache consumption, 2024. URL https://arxiv.org/abs/2407.18003.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks, 2024. URL https://arxiv.org/abs/2309.17453.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020. URL https://arxiv.org/abs/2004.05150.

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Baobao Chang, Junjie Hu, and Wen Xiao. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling, 2024. URL https://arxiv.org/abs/2406.02069.

Yefei He, Luoming Zhang, Weijia Wu, Jing Liu, Hong Zhou, and Bohan Zhuang. Zipcache: Accurate and efficient kv cache quantization with salient token identification, 2024. URL https://arxiv.org/abs/2405.14256.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi : Plug-and-play 2bit kv cache quantization with streaming asymmetric quantization. 2023. doi: 10.13140/RG.2.2.28167.37282. URL https://rgdoi.net/10.13140/RG.2.2.28167.37282.

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, 2024. URL https://arxiv.org/abs/2401.18079.

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. URL https://arxiv.org/abs/2306.14048.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation, 2024. URL https://arxiv.org/abs/2404.14469.

Qichen Fu, Minsik Cho, Thomas Merth, Sachin Mehta, Mohammad Rastegari, and Mahyar Najibi. Lazyllm: Dynamic token pruning for efficient long context llm inference, 2024. URL https://arxiv.org/abs/2407.14057.

Tianyi Zhang, Jonah Yi, Zhaozhuo Xu, and Anshumali Shrivastava. Kv cache is 1 bit per channel: Efficient large language model inference with coupled quantization, 2024. URL https://arxiv.org/abs/2405.03917.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.

Ramesh Nallapati, Bowen Zhou, Caglar Gulcehre, Bing Xiang, et al. Abstractive text summarization using sequence-to-sequence rnns and beyond. *arXiv preprint arXiv:1602.06023*, 2016.