

Streaming Sonnets: Efficient Generation with KV Caching and Quantization

Stanford CS224N Default Project

Michael Yang

Department of Computer Science
Stanford University
myang243@stanford.edu

Codey Sun

Department of Electrical Engineering
Stanford University
codeysun@stanford.edu

Abstract

Transformer-based language models have shown success in NLP tasks, but deploying them at scale remains a challenge due to quadratic attention costs and memory usage. This makes naive transformers impractical for long context or streaming use cases. This project aims to alleviate these issues by implementing and evaluating KV caching and quantization techniques. Building upon GPT-2, we explore two inference-time optimizations: Key-Value (KV) caching and compression with Attention Sinks, and quantization with GPTQ. Our models match or exceed TA-provided baselines across all tasks, and we observe up to a **98% speedup** with KV caching and a **16% memory reduction** using GPTQ. Further, we demonstrate that while attention sinks stabilize streaming, heavy-hitter KV compression is necessary for successful long-context retrieval. We use these techniques to develop a model capable of efficient conversational streaming and long context retrieval.

1 Key Information to include

- TA mentor: Sarah Chen
- External collaborators (if no, indicate “No”): No
- External mentor (if no, indicate “No”): No
- Sharing project (if no, indicate “No”): No

2 Introduction

Traditional dense attention has quadratic complexity with respect to sequence length, making inferring intractable for long text lengths. KV caching alleviates this problem for inference by keeping attention results in memory, sacrificing memory to achieve linear time complexity [1]; however, memory constraints limit its effectiveness. The memory constraint typically leads to a limited attention window size (e.g., 4K for LLama-2 [2]), and existing models fail to extrapolate beyond this sequence length. This tradeoff of memory and time complexity has essentially limited LLMs to finite-length generations. We explore several KV caching techniques—including naive, sliding window, attention sink, and heavy hitter compressed KV caches—and evaluate which provides the best tradeoff between speed, memory, and task performance. Importantly, we find that KV caching is also necessary for emergent behavior in streaming and long context retrieval.

The memory savings of KV caching become insufficient as model size grows. To solve this, we additionally explore memory reduction via quantization. Traditional deep learning inference stores weights in full precision (FP32), requiring significant memory that limits the deployment of LLMs on compute constrained devices. Basic quantization alleviates this by rounding weights to lower bit-widths (e.g., INT8 or INT4), reducing model weight memory usage by up to $8\times$ when using INT4. However, the accumulated rounding errors degrade model quality, worsening as context

window increases. Following GPTQ [3], we make quantization Hessian-aware: by computing the second-order sensitivity of the model’s output to each weight, we identify which weights have the largest impact on reconstruction error. Doing so achieves the full benefits of quantization while retaining performance.

In summary, our contributions are as follows:

- We **thoroughly evaluate KV caching** techniques in inference time, memory usage, and task performance. This also creates emergent behavior in **conversational streaming** and **long context retrieval**.
- We implement **model quantization** to reduce memory usage while retaining performance.
- We develop a system capable of efficiently **streaming text in constant time**.

3 Related Work

KV Caching Since the advent of the transformer architecture, all modern large language models (LLMs) are based on attention [4, 5, 6]. However, the attention operation incurs quadratic time complexity with respect to the sequence length, making inferencing applications intractable for long texts. To address this, several works introduce KV caching, in which previous key value transforms are selectively saved, either through recency [1], attention sinks [7], or attention weights [8, 9]. KV caching has shown success past language modeling into video [10] and audio [11] generation.

Quantization The field of quantization has evolved from enabling edge inference to compressing massive LLMs. Foundational work replaces floating-point operations with integer-arithmetic inference, allowing inference on mobile CPUs [12]. As model scales grew to hundreds of billions of parameters, accurate post-training quantization became essential to avoid costly retraining. GPTQ approximates second-order information to compensate for rounding errors [3]. As context windows grow, the memory usage of the KV cache has surfaced as a dominant bottleneck for inference. KVQuant [13] introduces per-channel, non-uniform key quantization to enable inference for context lengths up to 10 million.

4 Approach

4.1 GPT-2 Implementation

At a high level, our GPT-2 architecture comprises of dimension 768 positional and token embeddings, passed through a dropout layer. For our masked causal multi-head self attention implementation, we used query, key, and value projections split across $h = 12$ heads. Each head is of dimension $d_k = d/h = 768/12 = 64$. Also, to note, unlike the original transformer ("post-norm"), GPT-2 applies layer norm before each sub-layer. Next, the FFN expands to $4d = 3072$ dimensions internally and uses GELU activation. In accordance with the original GPT-2 design, the final logits reuse the word embedding matrix E (weight tying) to avoid needing a separate output layer. Finally, we used the Adam optimizer with decoupled weight decay (AdamW). Weight decay was applied directly to the parameters (not through the gradient), a key distinction from vanilla Adam.

Sentiment Classification For the downstream task of sentiment classification, we repurposed our GPT-2 for sentiment classification on SST (5 classes) and CFIMDB (binary). The hidden state of the last non-padding token is extracted and passed through a linear classification head:

$$\text{logits} = W_{\text{cls}} \cdot \mathbf{h}_{\text{last}} \in \mathbb{R}^C$$

Paraphrase Detection For paraphrase detection, our model was given two sentences s_1 and s_2 , which are formatted as a natural language prompt asking if s_1 is a paraphrase of s_2 . The model answers this by predicting the next token optimally as token ‘yes’ (BPE index 8505) or ‘no’ (BPE index 3919) in a cloze-style assessment.

Sonnet Generation For the task of sonnet generation, our GPT-2 model was fine-tuned on Shakespeare’s sonnets by minimizing next-token prediction loss over the full sequence:

$$\mathcal{L} = - \sum_{t=1}^T \log P(x_t \mid x_{<t})$$

At inference, sonnets are completed from a 3-line prompt using top-p (nucleus) sampling with temperature scaling:

$$P_{\text{scaled}}(x) = \text{softmax}(\text{logits}/\tau)$$

which is then truncated to the smallest set of tokens whose cumulative probability exceeds the threshold $p = 0.9$ (default $\tau = 1.2$). This avoids both the degenerate repetition of greedy decoding and the incoherence of pure random sampling.

4.2 KV Caching

Naively computing the full attention matrix recomputes key-value transforms for old tokens. This leads to significant slowdown and memory usage as the context length grows. To alleviate this, we implement a KV cache that stores the full history of transformed keys and values. During generation, rather than passing the full sequence in each loop, we need only to pass the latest token; this token attends to the saved key-value history, reducing redundant computation and reducing time complexity from quadratic to linear. Note that causal masking is no longer needed in this paradigm.

Attention Sinks The memory usage M_{KV} of a KV cache can be computed as:

$$M_{\text{KV}} = 2 \cdot B \cdot N \cdot h \cdot d_k \cdot L \cdot p$$

where B is batch size, N is context length, L is the number of layers, and p is the bytes per parameter. This becomes intractable for long contexts where N grows large.

Instead, a simple sliding window approach in which we evict old tokens can keep memory usage constant. However, this significantly degrades generation quality. We persist attention sinks—tokens at the beginning of the context—in the KV cache to stabilize generation. In other words, we divide the KV cache into two pools: a recent sliding-window pool of size r and an attention sink pool of size s_{sink} . The total cache memory usage is thus constant $O(r + s_{\text{sink}})$, where $r + s_{\text{sink}} \ll N$. We find that generation stabilizes with only $s_{\text{sink}} = 1$.

Heavy Hitter Compression While attention sinks can stabilize entropy, it fails in long context retrieval, otherwise known as the "needle in a haystack" test. For this task, we implement an alternative KV caching approach that intelligently evicts tokens based on attention weights. We replace the attention sink pool with a heavy-hitter pool of size s_{hh} . The heavy-hitter pool preserves the s_{hh} highest-scoring tokens from the non-recent portion of the cache.

Let C_t be the cache length at decoding step t . For layer ℓ and KV head $m \in \{1, \dots, h\}$, let the cumulative heavy-hitter score be

$$S_t^{(\ell, m)}(j) = \sum_{\tau=1}^t A_{\tau}^{(\ell, m)}(j),$$

where $A_{\tau}^{(\ell, m)}(j)$ is the attention output at step τ by KV head m in layer ℓ to token position j . Define the recent and non-recent index sets, respectively:

$$\mathcal{R}_t = \{C_t - r + 1, \dots, C_t\}, \quad \mathcal{O}_t = \{1, \dots, C_t - r\}.$$

When $C_t > r + s_{\text{hh}}$, eviction is performed independently for each (ℓ, m) by selecting heavy hitters from $\mathcal{O}_t$:

$$\mathcal{H}_t^{(\ell, m)} = \text{TopK}_{j \in \mathcal{O}_t} \left(S_t^{(\ell, m)}(j), s_{\text{hh}} \right).$$

We thus keep

$$\mathcal{K}_t^{(\ell,m)} = \mathcal{H}_t^{(\ell,m)} \cup \mathcal{R}_t,$$

where $\mathcal{K}_t^{(\ell,m)}$ denotes the position indices that are kept in the KV cache. This ensures that semantically important tokens—those that consistently receive high attention—always remain in the cache. Note that because $\mathcal{K}_t^{(\ell,m)}$ is defined per layer/head, different layers/heads may attend to different subsets of past tokens.

4.3 Quantization

While KV caching reduces the computational complexity of long-context generation, the sheer size of model weights remains a primary bottleneck for deployment. Naive 16-bit or 32-bit floating point storage often exceeds the memory capacity of consumer grade hardware. To address this, we implement 4-bit weight quantization, which reduces the weight memory footprint by approximately 8x compared to FP32 by packing two INT4 values into a single uint8 byte. The dequantization process for a weight w is defined as $w = (q - z) \cdot s$, where q is the 4-bit quantized value, z is the zero-point, and s is the row-wise scale. While naive rounding can be used to determine q , it often leads to significant accuracy degradation, especially on larger models.

GPTQ To minimize the precision loss associated with quantization, we utilize the *Generalized Post-Training Quantization (GPTQ)* algorithm [3]. GPTQ reframes quantization as an optimization problem, aiming to find a quantized weight matrix $\widehat{W}$ that minimizes the squared reconstruction error of the layer outputs:

$$\min_{\widehat{W}} \|WX - \widehat{W}X\|_2^2$$

This objective is achieved by using second-order information from the Hessian matrix $H = \frac{1}{n} X^\top X$, which represents the sensitivity of the layer’s output to changes in specific weights. To improve stability, we first permute the columns of the weight matrix based on descending Hessian diagonal values (i.e., prioritizing the most sensitive weights). We then compute the inverse Hessian. To ensure numerical stability, we apply a damping factor to the Hessian diagonal before performing a Cholesky decomposition to compute its inverse:

$$H_{damped} = LL^\top \implies H^{-1} = (L^\top)^{-1} L^{-1}$$

where L is a lower triangular matrix. We then perform quantization column-by-column, where the quantization error for a column j is compensated for by adjusting the remaining unquantized weights. For a weight column w_j , the error δ_j is computed and propagated as:

$$\delta_j = (w_j - \text{quant}(w_j)) / [H^{-1}]_{jj}$$

$$W_{remaining} = W_{remaining} - \delta_j \cdot [H^{-1}]_{j,remaining}$$

where $[H^{-1}]_{jj}$ is the j -th diagonal element of the inverse Hessian. The weights are processed in blocks to maintain computational efficiency while ensuring that the weights are accurately represented in the reduced precision format.

5 Experiments

5.1 Data

For sentiment classification, the SST (Stanford Sentiment Treebank) [14] and CFIMDB movie review classification datasets [15] were used. For paraphrase detection, the Quora Questions Pairs binary classification dataset [16] was used. For sonnet generation, the 154 sonnet Shakespeare’s Sonnets dataset [17] was used.

Table 1: Results split by task. We match/exceed the TA-provided baselines on all tasks.

Sentiment			Paraphrase			Sonnet		
Dataset	Model	Acc.	Split	Model	Acc.	Split	Model	ChrF
SST	TA baseline	0.513	Dev	TA baseline	0.882	Dev	TA baseline	41.414
	Ours	0.513		Ours	0.898		Ours	42.124
CFIMDB	TA baseline	0.976	Test	TA baseline	0.856	Test	TA baseline	41.208
	Ours	0.980		Ours	0.857		Ours	41.242

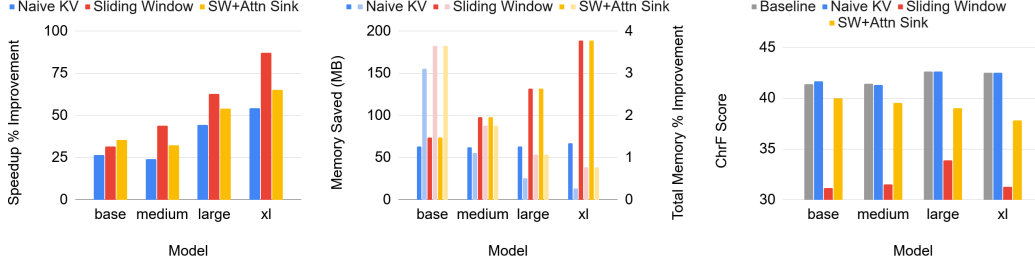


Figure 1: Inference time speedup (left), memory usage (middle), and ChrF (right) across KV caching techniques and model sizes (sliding window 64, sink size 4). Left: speedup scales with model size. Middle: memory savings (dark bars) also scale, but are overshadowed by overall parameter size—total memory % improvement (light bars) thus decreases. Right: attention sinks show decreasing performance with model size.

5.2 Evaluation method

For sentiment classification and paraphrase detection, we measured accuracy, defined as the fraction of correctly predicted labels. For sonnet generation, the Character n-gram F-score (ChrF) was used to measure the textual similarity of the generated sonnets with the actual.

To analyze inference-time optimization methods, we build off the sonnet generation task as it is the only generation task. For each optimization, we evaluate inference time and memory usage in addition to ChrF score. As a secondary measure of quality (on top of ChrF), we also evaluate the average generation lengths and compare them with the true average sonnet length. For long context and extrapolation settings, we concatenate random sonnets until the desired context length is achieved. A similar approach is used for the "needle in a haystack" retrieval test, in which we generate random facts with Gemini 2.5 and concatenate them to the desired context length.

5.3 Experimental details

All tasks used the AdamW optimizer with a fixed seed (11711) for reproducibility. We used the AdamW hyperparameters of: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-6}$. Training is run for 10 epochs on the GPT-2 base model for all tasks, with the medium, large, and XL variants trained for scaling analysis [5]. GPT-2’s 1024 context length is small by modern standards; for long context analysis, we fine-tune the Qwen2.5 models (128k context length) in the same setting [6]. All experiments are performed on a single H100 or RTX4090 GPU.

For sentiment classification, the GPT-2 base (768d, 12 layers) was fully fine-tuned with a learning rate of $1e-3$, batch sizes of 64 (SST) and 8 (CFIMDB), and 0.3 dropout. Paraphrase detection used a lower learning rate of $1e-5$ with batch size 8 and no weight decay. Sonnet generation used a learning rate of $1e-5$ with batch size 8 using standard next-token prediction cross entropy loss. At inference time, sonnets were completed from a 3-line prompt using nucleus sampling with temperature 1.2 and top-p 0.9, generating up to 1024 new tokens. All inference-time optimizations use the same sonnet generation checkpoint.

Table 2: Extrapolation performance across different KV cache settings. Reposition IDs refers to renumbering the position IDs to be relative to the cache rather than the entire context. A sliding window of 128 and attention sink size of 4 are used. The true average sonnet length is 160.

Model	Pos. Embed.	Sliding window	Attention sink	Reposition IDs	ChrF	Avg seq. len.
GPT-2 Base	Absolute	✓	✓	✗	25.2	3
Qwen2.5-0.5B	RoPE	✗	✗	✗	27.5	27
Qwen2.5-0.5B	RoPE	✓	✗	✗	18.24	408
Qwen2.5-0.5B	RoPE	✓	✓	✗	35.05	204
Qwen2.5-0.5B	RoPE	✓	✓	✓	32.05	45

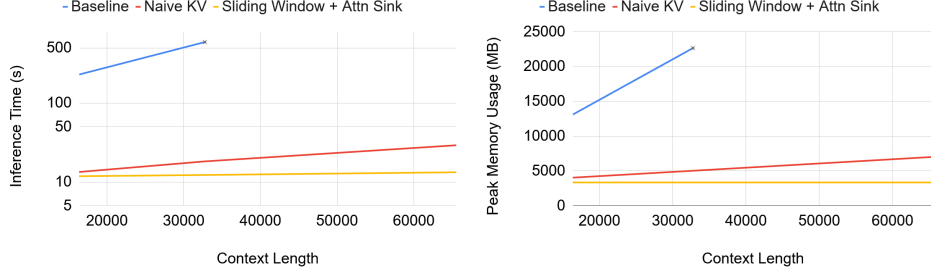


Figure 2: Inference time (left) and memory usage (right) across increasing context lengths. Sliding window KV caches are constant w.r.t. context length, making them ideal for long context scenarios. Note that the baseline method OOMs.

5.4 Results

Task Results Table 1 compares our results to the TA-provided baselines on all tasks. As shown, we match/exceed the baselines on all tasks. As of 3/11/2026, we placed 62 and 58 on the paraphrase and sonnet leaderboards, respectively. Note that our extensions improve inference speed and memory usage, and thus do not affect the final score.

5.4.1 KV Caching Results

For this section, the baseline method refers to the GPT-2 model without inference-time optimizations, shown as “Ours” in Table 1.

Speed vs. Memory Tradeoff Figure 1 shows speedup, memory savings, and ChrF across all GPT-2 sizes. Speed gains improve with model size. Memory saving also increase, but are overshadowed by model parameters. Surprisingly, attention sinks show decreasing performance with model size. We believe sliding windows on large models may be sacrificing more semantically rich information.

Figure 2 shows inference time and memory usage across increasing context lengths when generating 1024 tokens with Qwen2.5-1.5B. With the attention sink KV cache, both inference time and memory usage are constant with respect to context length, whereas the baseline and naive KV cache scale (super)linearly.

Attention Sink In Figure 3, we evaluate the ChrF score, average generation length, and memory footprint of varying sliding window+attention sink sizes. Without any attention sink tokens, generation breaks down as soon as the first token is evicted from the KV cache, leading to a low ChrF score and a long generation length. In this case, it is crucial to allocate a large enough window such that the entire sonnet can fit within the window. However, generation stabilizes with just one sink token, with the score matching the baseline method and the average generation length matching the true average sonnet length. With an attention sink, changing the window size does not noticeably improve the generation quality. As expected, the cache’s memory footprint increases linearly with the window size. More sink sizes and qualitative results can be found in the Appendix A.1, in which we visualize the output degeneration without sink tokens.

Table 3: Long context retrieval results by KV caching method. The total context length is 900 tokens. Compressing heavy hitter tokens consistently retrieves the correct password, while attention sinks only retrieve what is in the sliding window. More results in Appendix A.2

Password depth	Method	Result
50% (450 tokens)	Full window	✓
	4 attn sink + 256 sliding window	✗
	4 attn sink + 512 sliding window	✓
	128 heavy hitters + 128 sliding window	✓

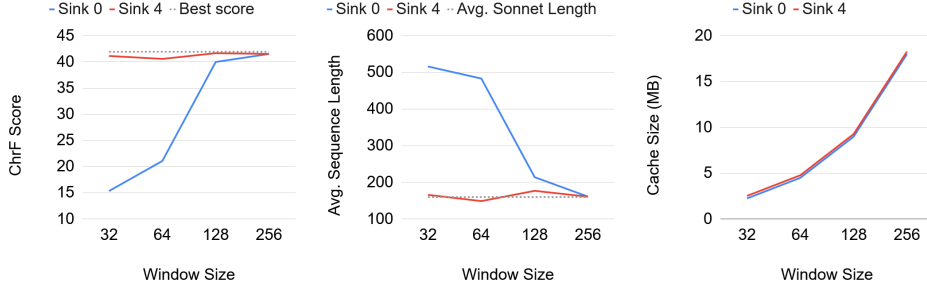


Figure 3: Sliding window size vs. generation quality/length and memory usage. The dotted lines indicate the best score from Table 1 (left) and the true average sonnet length (middle). With no attention sink, we get degenerate outputs until the entire context length fits within the window. With an attention sink, increasing window size does not noticeably improve quality. Memory usage increases linearly with window size, with a constant offset added for the attention sink.

Streaming Extrapolation We evaluate extrapolation capability by inserting a random complete sonnet in the context before prompting for a new sonnet generation. This mimics a conversational setting and evaluates the model beyond its trained context length. As shown in Table 2, GPT-2 fails to extrapolate due to its use of absolute position embeddings—it immediately outputs <EOS> due to the context length. By using Qwen2.5-0.5B, which uses RoPE [18] embeddings instead, we can get reasonable sonnet generations beyond the trained context length. Note that StreamingLLM’s ID repositioning method [7] fails. We attribute this to 1) Qwen2.5 already using YARN [19] for context extrapolation and 2) repositioned IDs being out of distribution for the pretrained model.

Long Context Retrieval We conduct a "needle in a haystack" test using Qwen2.5-7B-Instruct with different KV caching methods. A password is inserted into 900 tokens of randomly generated facts at varying depths; we then prompt the model to repeat the password. As shown in Table 3, by preserving heavy hitters in the KV cache we are able to retrieve the password, whereas the attention sink fails. However, due to the nature of the pretraining data, uncommon tokens will never receive high attention scores and may be discarded—this makes retrieving uncommon tokens still fail. This behavior and more results are further discussed in Appendix A.2.

5.5 Inference Quantization Results

Both naive and GPTQ quantization reduce memory usage by 13–16% compared to the FP32 baseline, with GPTQ achieving slightly lower memory (Table 4). The key difference is in generation quality: naive quantization degrades ChrF by 1.18 points, while GPTQ preserves quality much better with only a 0.35 drop in score. This confirms that Hessian-aware error compensation effectively minimizes quantization error on the most sensitive weights. Both methods increase inference time due to the on-the-fly INT4 dequantization overhead in our implementation, which lacks native INT4 kernel support unlike specialized libraries such as AutoGPTQ.

Adding KV caching with quantization yields subtle improvements in both speed and memory. Unfortunately, the dominant cost—dequantizing INT4 weights to FP32 on every forward pass—remains unchanged. KV caching would provide more substantial speedups when combined with

Table 4: Inference efficiency and quality under different quantization methods.

Method	Inference time ↓	Memory usage (MB) ↓	ChrF ↑
Baseline	10.1292	1980	41.94
Naive quantization	18.86	1715	40.76
GPTQ	20.71	1654	41.59
Naive quantization + KV cache	18.71	1696	40.65
GPTQ + KV cache	19.39	1657	41.57

optimized quantization kernels (e.g., AutoGPTQ, bitsandbytes) that eliminate the dequantization bottleneck. These optimized kernels make the attention computation savings the dominant factor, particularly in long contexts.

6 Analysis

By adding KV caching, we observe upwards of **98% speedup** over the baseline model. Without it, any long context inference is intractable and will quickly OOM. To stabilize generation, just one attention sink token is needed; we find 4 tokens gives the best results. With attention sinks, the sliding window size no longer matters. In addition to speedup, KV caching also enables streaming behavior, but only if the positional embedding allows it. Without relative positional embeddings like RoPE, extrapolation beyond the trained context length will still fail. Thus, positional embedding strategy is more impactful to extrapolation than KV caching in our experiments. Alternative KV caching strategies such as heavy-hitter KV compression can benefit long context retrieval tasks. However, without any fine-tuning, the attention score accumulation is still subject to the pretraining data—tokens rarely seen in the training data will continue to attract low attention scores and cannot be retrieved. Further, the sparse distribution of attention corrupts reasoning traces, leading to nonsensical reasoning even when the answer is correct.

While the primary benefit of KV caching is inference speedup, it does not address the growing memory cost of model weights as parameter counts scale. For GPT-2 small (124M parameters), weight memory is a modest portion of the total footprint, but for larger models (billion-scale parameters) weights dominate, making quantization essential. KV caching and quantization are thus complementary: KV caching eliminates redundant computation during autoregressive generation, while quantization compresses the static weight memory that scales with model size. Combining both methods with optimized INT4 kernels would yield the full benefit of reduced memory from quantization and reduced computation from caching, enabling efficient deployment of LLMs under tight compute constraints.

7 Conclusion

This project successfully implements and fine-tunes a GPT-2 architecture that matches or exceeds TA-provided baselines across sentiment analysis, paraphrase detection, and sonnet generation. Our evaluation of inference-time optimizations demonstrates that KV caching provides a significant 98% speedup, enabling long-context generation that would otherwise result in out-of-memory (OOM) errors. While attention sinks effectively stabilize streaming generation with constant memory complexity, we found that heavy-hitter compression is essential for maintaining retrieval accuracy in "needle in a haystack" scenarios. Furthermore, GPTQ quantization reduces the model's memory footprint by 16% with a negligible 0.83% loss in performance, proving to be a highly effective complement to caching. Together, these techniques provide a robust framework for deploying high-performance LLMs under the tight compute constraints of consumer-grade hardware.

Team contributions (Required for multi-person team)

Codey: GPT-2, KV caching, KV compression, Quantization/GPTQ + KV caching
Michael: Quantization, GPTQ, Quantization/GPTQ + KV caching

References

- [1] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference, 2022.
- [2] 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.
- [3] 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.
- [4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- [5] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 2019.
- [6] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025.
- [7] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations (ICLR)*, 2024.
- [8] 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*, 36:34661–34710, 2023.
- [9] 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.
- [10] Jung Yi, Wooseok Jang, Paul Hyunbin Cho, Jisu Nam, Heeji Yoon, and Seungryong Kim. Deep forcing: Training-free long video generation with deep sink and participative compression. *arXiv preprint arXiv:2512.05081*, 2025.
- [11] Kai Qiu, Xiang Li, Hao Chen, Jie Sun, Jinglu Wang, Zhe Lin, Marios Savvides, and Bhiksha Raj. Efficient autoregressive audio modeling via next-scale prediction, 2024.
- [12] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *arXiv preprint arXiv:1712.05877*, 2017.

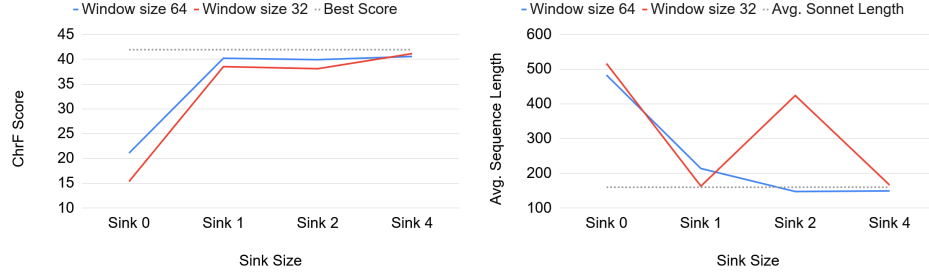


Figure 4: Attention sink size vs. generation quality/length. The dotted lines indicate the best score from Table 1 (left) and the true average sonnet length (right). As sink size increases, ChrF score increases, and the average sequence length gets closer to the true average sonnet length of 160 tokens.

- [13] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Amir Gholami, Michael W. Mahoney, Kurt Keutzer, and Zhen Dong. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *arXiv preprint arXiv:2401.18079*, 2024.
- [14] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2013.
- [15] Antonio Davi Macedo Coelho de Castro. minbert and downstream tasks. <https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1234/final-reports/final-report-169724933.pdf>, 2023.
- [16] Shankar Iyer, Nikhil Dandekar, and Kornel Csernai. First quora dataset release: Question pairs. <https://quoradata.quora.com/First-Quora-Dataset-Release-Question-Pairs>, 2017.
- [17] William Shakespeare. Shakespeare’s sonnets. <https://www.folger.edu/explore/shakespeares-works/shakespeares-sonnets/>. Folger Shakespeare Library edition, ed. Barbara Mowat, Paul Werstine, Michael Poston, and Rebecca Niles.
- [18] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023.
- [19] Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. Yarn: Efficient context window extension of large language models, 2026.

A Appendix

A.1 Attention Sink Results

In Figure 4, we evaluate the ChrF score and average generation length of varying attention sink sizes. With just one sink token, generation stabilizes, with the score approaching the baseline method and the average generation length approaching the true average sonnet length. Table 5 shows sonnet generations with varying attention sink sizes (constant window size of 64). Without any attention sink tokens, generation breaks down as soon as the first token is evicted from the KV cache.

A.2 Needle in a Haystack Results

Tables 6 and 7 show further results on the "needle in a haystack" test with Qwen2.5-7B-Instruct. The heavy hitter approach successfully retrieves the correct context, but only for common tokens.

Table 5: Qualitative generations for different attention sink sizes (window size 64). Just one attention sink token drastically improves generation quality. Otherwise, generation breaks down as soon as the first token is evicted from the KV cache sliding window.

[illegible]

Table 6: Long context retrieval results by password depth and KV caching method. The total context length is 900 tokens. Compressing heavy hitter tokens consistently retrieves the correct password, while attention sinks only retrieve what is in the sliding window.

Password depth	Method	Result
10%	Full window	✓
	4 attn sink + 256 sliding window	✗
	4 attn sink + 512 sliding window	✗
	128 heavy hitters + 128 sliding window	✓
50%	Full window	✓
	4 attn sink + 256 sliding window	✗
	4 attn sink + 512 sliding window	✓
	128 heavy hitters + 128 sliding window	✓
100%	Full window	✓
	4 attn sink + 256 sliding window	✓
	4 attn sink + 512 sliding window	✓
	128 heavy hitters + 128 sliding window	✓

Table 7: Long context retrieval when the secret code has special characters. The heavy hitter approach correctly retrieves the context, but is missing special tokens from the password. These special tokens have lower attention weights and are evicted. Further, reasoning traces are corrupted with this approach.

Password depth	Method	Result	Example output (optional)
10%	Full window	✓	<i>The special secret code is: PINEAPPLE-7429</i>
	128 heavy hitters + 128 sliding window	✗	<i>The secret code provided in the text is "PINEAPPLE7". This can be deduced by looking at the first letter of each each sentence that contains a number or a specific measurement.</i>