Effect of KV Cache Error on Inference Accuracy

Phillip Shebel

Temple University

Abstract

KV cache optimization is a popular area of interest in current AI research. Researchers have used quantization, pruning, and other methods to reduce the overall size of the cache to improve performance. In this paper we attempt to better understand the relationship between introducing error in to the cache and the effects on accuracy.

1. Introduction

Large Language Models (LLMs) are the result of many decades of research in machine learning and natural language processing. The big leap that took us into the current era of LLMs is the transformer architecture introduced by Vaswani et al. (2017) [0]. Today's state-of-the-art LLMs, with hundreds of billions or even trillions of parameters, demonstrate remarkable performance across a wide range of tasks. However, this increase in ability comes with substantial computational and memory demands. To manage these demands, modern LLMs rely on various optimizations, one of the most important being the key value (KV) cache, which improves efficiency by reusing previously computed attention information. As models grow and context lengths expand, optimizing the KV cache has become an active area of research. Techniques such as quantization, pruning, and eviction strategies aim to reduce the memory footprint of the cache but often introduce error into the model's internal representations. This paper explores the broader implications of KV cache approximations, focusing on how different forms of error affect model accuracy and offering a framework for evaluating the trade-offs between memory efficiency and performance.

2. Decoder-Only Transformer Inference

The context for this paper is decoder-only transformer inference, which represents the majority of LLM deployments that consumers interact with daily. To provide a foundation for understanding KV cache optimization, we describe the inference process step by step. Some details (such as the specifics of softmax) are omitted for clarity, and a comprehensive review can be found in Vaswani et al. (2017) [0].

2.1. Initialization

The inference process begins when a framework like `llama.cpp` or vLLM loads a pre-trained model such as GPT or Qwen. This loading process includes weight matrices for each layer of the attention mechanism, vocabulary embeddings for tokenizing input, and architectural parameters such as attention head dimensions and layer counts. Finally, memory is allocated for the KV cache at this stage.

2.2. Tokenization

When a user prompts the model with text, the input is converted into a sequence of token IDs using the model's vocabulary. Tokens typically represent words, punctuation, or individual characters, as well as special tokens like the end-of-sequence marker that signals when to terminate generation.

2.3. Embedding

Each token ID is then mapped to a representation of its semantic properties. So a token identifies a human readable word while an embedding is a numerical representation of its meaning. We also include the tokens position in the input sequence in the embedding to create a hidden state.

2.4. Attention

Each hidden state then goes through each layer of the attention mechanism. Each layer begins with normalization before doing self-attention, the defining step of the transformer architecture.

In this step, we project the hidden state into a query, key, and value space. Each layer has its own set of Q, K, and V matrices that are developed during training. You can think of the Q space as what we are looking for, the K space as what the current hidden state contains, and the V space as what it contributes to others. For each position we compute attention scores by combining a hidden states query vector with all previous key vectors. These scores are normalized into weights that determine how much each position's value vector contributes to the current position's updated representation. The weighted sum of value vectors is then projected back to the model's hidden dimension. During this process the key and value vectors computed for each input token are stored in the

KV cache for reuse in subsequent iterations, eliminating redundant computations.

3. Feed-Forward Network

Following attention, each hidden state is passed through a feed-forward network composed of linear transformations and non-linear activation functions. The non-linearity enables the model to capture more complex patterns within each hidden states representation. The output of this transformation is then combined with the original input via a residual connection, and the result is normalized to ensure stable learning and consistent scaling.

3.1. The Generation Loop

After processing the input context through all layers of the network, the model begins generating new tokens one by one. For each new token position:

1. The model takes the last token's hidden state after the final layer
2. Projects it to the vocabulary space through the output embedding matrix
3. Applies a probability distribution transformation (typically softmax)
4. Uses sampling strategies, such as top-k, to select the next token
5. The selected token is appended to the sequence
6. The process repeats with this new token as input

Critically, the model computes new query vectors for each new token, but reuses the cached key and value vectors from all previous tokens. This optimization transforms the computational complexity from quadratic to linear in sequence length, making efficient inference possible.

4. Intro to the problem

Attention helps the model generate some understanding of the relationship between words in an input, and caching helps reduce the amount of work we need to do. The current bottleneck for these systems is the KV cache. As context lengths increase into the tens of thousands of tokens, the KV cache often exceeds the memory limits of modern GPUs. When the cache must be offloaded to CPU memory, the performance impact is severe: access to CPU RAM is orders of magnitude slower than accessing high-bandwidth GPU memory, leading to significant inference slowdowns. To illustrate the scale of this issue, consider a model with:

- L number of layers in the model,
- H number of attention heads per layer,
- D dimensions of the attention head,
- N tokens in the input sequence.

The total memory required for the KV cache can be estimated as:

$$\text{Memory} \approx 2 \times L \times H \times D \times N \times b \text{ bytes}$$

is the number of bytes per parameter (typically 2 for half-precision FP16). As an example, a model with 26 layers, 32 attention heads, 128 dimensions per head, and 8K tokens in context would require approximately

$$2 \times 26 \times 32 \times 128 \times 8000 \times 2 = \text{ 3.4GB}$$

A commercial GPU likely has between 8 and 16GB of RAM, so if you are running multiple inferences or are trying to run a large model you can run out of memory quickly.

This memory bottleneck has spurred substantial research into KV cache optimization techniques:

- **Quantization**: Reducing numerical precision from 16-bit floating point to 8-bit integers or lower. See Liu et al. (2024) [0]
- **Pruning**: Eliminating less important keys and values based on attention scores or other importance metrics. See Xu et al. (2024) [0]
- **Structured Sparsity**: Introducing patterns of sparsity that enable hardware-accelerated processing. See Zhao et al. (2024) [0]
- **Eviction Strategies**: Selectively removing cached information for tokens deemed less relevant to future predictions. See Chen et al. (2024) [0]

Each of these approaches introduces some form of error, potentially affecting model accuracy and output quality. Understanding the relationship between KV cache modifications and model performance is thus crucial for developing optimal inference strategies.

5. Experiment

5.1. Baseline Control

The control configuration ran the model with no modifications to the KV cache, using full FP32 precision throughout the inference process. This established our performance baseline against which all error introduction methods were compared.

5.2. Quantization

Quantization represents the most common approach to KV cache optimization in production systems. This method reduces numerical precision by truncating the least significant bits in each value's binary representation, converting 32-bit floating-point values to lower precision formats (16-bit, 8-bit, or 4-bit integers). While quantization substantially reduces memory requirements, it introduces deterministic rounding errors that can be conceptualized as a form of uniform random error within bounded ranges. For our experiments, we utilized `llama.cpp`'s built-in quantization parameters, testing multiple precision levels to establish a performance-accuracy trade-off.

5.3. Uniform Random Error

To isolate the effects of uniform error distribution independent of quantization's implementation details, we implemented a function to inject uniform random error.

$$\text{error } = \text{uniform\_error(a,b)}$$

$$\text{kv } = \text{kv } + \text{kv} \times \textit{error}$$

This function adds scaled uniform random noise to each key and value vector component as it enters the KV cache. By controlling parameters a and b, we could precisely adjust the error bounds as a percentage of each value's magnitude, allowing us to test different error intensities while maintaining a uniform distribution.

5.4. Normal Random Error

To contrast with uniform error distribution, we also implemented a function to inject normal error.

$$\text{error} = \text{normal\_error(mean, std)}$$

$$\text{kv} = \text{kv} + \text{kv} \times \textit{error}$$

This approach introduces normally distributed noise centered around a specified mean with controlled standard deviation. For our experiments, we maintained a mean of 0 while varying the standard deviation to adjust error intensity. Normal distributions provide a different error profile than uniform distributions, with more values clustered near the mean and fewer at the extremes.

5.5. Benchmarking

We used two benchmarks to test the accuracy of the system. Instruction following evaluation is a dataset of questions that include specific instructions [0]. The model's response is then parsed to determine if the response correctly followed the instructions. For example, "write an email of at least 400 words" or "write a letter without using a comma". Error introduced to caching will affect the models ability to understand the context of words, and will ultimately reduce the accuracy. The second benchmark is perplexity, which is the ability for a model to correctly guess the next token in a sequence. Similarly, we expect a decrease in the ability of the model to accurately predict the next token with error introduced.

Each benchmark was run 5 times and then averaged. When too much error is introduced, the model will tend to repeat itself, so we limited the number of tokens generated to 100 (the kind of error introduced with this testing produces a pattern that often is not captured by the repeat penalty). To facilitate this choice, we removed all instructions that require a certain number of words in the response from the instruction following benchmark. The final instruction following dataset had 128 questions. For the perplexity benchmark we used sequences of 10 strings from 1000 random wikipedia articles. The 10th string is omitted from the query and used to test against the models response.

Parameters for the uniform and normal error were selected through some trial and error. We tested many values and looked for ones that we could compare to quantization. An interesting finding was that an error that changed the sign of a value was catastrophic compared to an error with the same magnitude but didn't change the sign. Additionally, error functions that add skew also produced catastrophic results. A uniform error between -1,1 and 0,2 has the same standard deviation, but 0,2 has a mean error of 1 compared to zero. For these reasons we chose values for the experiment that centered on zero and wouldn't introduce too large a negative scaling factor.

The model we used for this test was Metas Llama 3.2 1B Instruct. The inference framework we used was `llama.cpp` and the only parameters besides limiting the tokens generated and the kv quantization was flash attention. The raw scores for Each test were also included in the following tables along with a comparison to the control. For instruction following evaluation the result is an accuracy score, while for perplexity it is the number of correctly guessed next tokens.

6. Results

6.1. Quantization

In table 2 we can see that 4 bit quantization actually outperforms the control. This is an outlier. The `llama.cpp` parameters do not simply mask the associated bits of a float when doing quantization, and after some research others have also reported improved accuracy performance from this parameter. We also tested using the bit masking and was able to produce the expected decrease in performance.

	IFEval	Percent of Control
Control	0.46	1
Quant8	0.45	.98
Quant4	0.45	.98

Table 1: *Quantization Instruction Following Scores*

	Perplexity	Percent of Control
Control	51.3	1
Quant8	46.8	.91
Quant4	59.2	1.15

Table 2: *Quantization Perplexity Scores*

6.2. Uniform Error

Most of our testing before benchmarking was done with uniform error. Test results with `uniform_error(-2,2)` and `uniform_error(0,2)` were omitted because they produced random tokens as output and therefore were not interesting.

	IFEval	Percent of Control
Control	0.46	1
Uniform Error(-1,1)	0.38	0.83
Uniform Error(-1.5,1.5)	0.39	0.85

Table 3: *Uniform Error Instruction Following Scores*

	Perplexity	Percent of Control
Control	51.3	1
Uniform Error(-1,1)	27.4	0.83
Uniform Error(-1.5,1.5)	31.4	0.85

Table 4: *Uniform Error Perplexity Scores*

6.3. Normal Error

From the testing we see that normal error degrades quickly as standard deviation is increased. `uniform_error(-1.5,1.5)` has a larger standard deviation than either normal distribution tested but performed better. This shows that the standard deviation parameter in this context is more sensitive to change and more testing will need to be done to find precise bounds.

	IFEval	Percent of Control
Control	0.46	1
Normal Error(0,0.25)	0.45	0.98
Normal Error(0,0.5)	0.43	0.93

Table 5: *Normal Error Instruction Following Scores*

	IFEval	Percent of Control
Control	0.46	1
Normal Error(0,0.25)	41.2	0.80
Normal Error(0,0.5)	31.2	0.60

Table 6: *Normal Error Perplexity Scores*

6.4. Analysis

An interesting observation is the asymmetric impact of error distributions that cross zero versus those that preserve sign. Even when matched for overall magnitude, error distributions that frequently changed the sign of cached values led to catastrophic degradation at much lower error rates. This finding suggests that the directional information encoded in KV cache values may be more fundamental to model function than their precise magnitudes.

Similarly, we found that error distributions with non-zero means caused more severe performance drops than zero-centered distributions with equivalent variance. This observation indicates that transformer architectures may be more robust to random noise than to systematic shifts in representation space.

7. Conclusion

This paper has established an empirical foundation for understanding the relationship between KV cache error and model performance degradation in decoder-only transformers. We have demonstrated that meaningful KV cache optimization is possible within specific error bounds – specifically, uniform error distributions between -1.5 and 1.5, or normal distributions with mean 0 and standard deviation 0.5 – while maintaining 80% of baseline model accuracy. These findings provide practical guidance for developing and evaluating KV cache compression techniques. Our work highlights several key principles that should inform future KV cache optimization research:

- Error Distribution Matters: The distribution characteristics of introduced error are as important as its magnitude. Zero-centered, sign-preserving error distributions offer better performance preservation than alternatives with similar magnitude.
- Graceful Degradation: Below certain error thresholds, model performance degrades gradually rather than catastrophically, suggesting that controlled compression is viable for many applications.
- Evaluation Methodology: Establishing benchmarks across multiple tasks is essential for understanding the full impact of KV cache optimizations, as different capabilities may exhibit varying sensitivity to error.

While our findings provide valuable heuristics for acceptable error bounds, a comprehensive understanding will require further experimentation across diverse models, tasks, and hardware configurations. Future work should focus on developing compression techniques specifically designed to introduce errors within the bounds we have identified, potentially enabling significant memory savings with acceptable performance impact.

References

Yilong Chen, Guoxia Wang, Junyuan Shang, Shiyao Cui, Zhenyu Zhang, Tingwen Liu, Shuohuan Wang, Yu Sun, Dianhai Yu, and Hua Wu. Nacl: A general and effective kv cache eviction framework for llms at inference time. *arXiv preprint arXiv:2408.03675*, 2024. 2

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. 2

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. 1

Yuhui Xu, Zhanming Jie, Hanze Dong, Lei Wang, Xudong Lu, Aojun Zhou, Amrita Saha, Caiming Xiong, and Doyen Sahoo. Think: Thinner key cache by query-driven pruning. *arXiv preprint arXiv:2407.21018*, 2024. 2

Junqi Zhao, Zhijin Fang, Shu Li, Shaohui Yang, and Shichao He. Buzz: Beehive-structured sparse kv cache with segmented heavy hitters for efficient llm inference. *arXiv preprint arXiv:2410.23079*, 2024. 2

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. Instruction-following evaluation for large language models. *arXiv preprint arXiv:2311.07911*, 2023. 3