

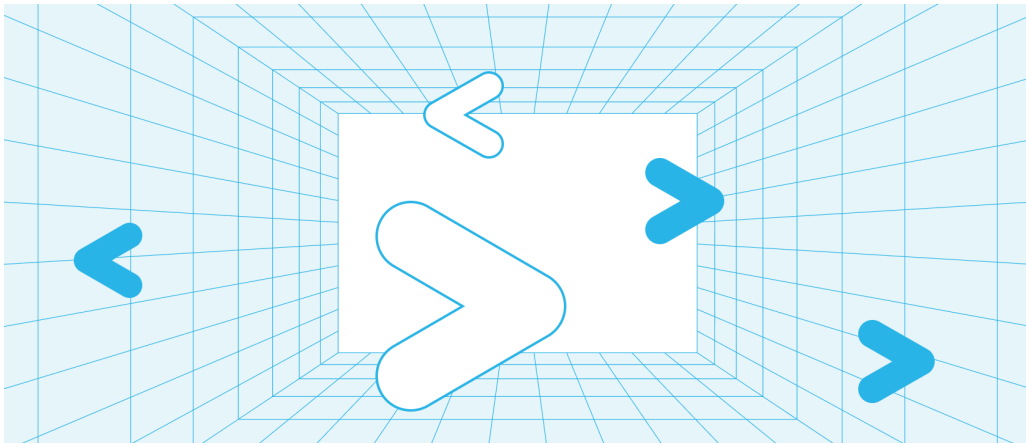
DEC 05, 2024

AUTHOR

Snowflake AI Research

SwiftKV: Accelerating Enterprise LLM Workloads with Knowledge Preserving Compute Reduction

SHARE



Large language models (LLMs) are at the heart of transformative enterprise AI solutions, powering a wide array of use cases that demand both high performance and cost efficiency. However, as enterprises adopt the technology, they are keen to optimize inference latency and costs for broader adoption.

One of the most widely explored optimizations in recent years is KV cache compression, which reduces the memory required for storing key-value (KV) pairs generated during inference. This technique has generated significant attention, with [hundreds of publications](#) highlighting its benefits, such as enabling longer input sequences and larger batch sizes on GPUs with limited memory.

However, memory savings alone are insufficient to meaningfully improve throughput and reduce the cost of inferencing enterprise workloads (see Table 1). Typical enterprise workloads process significantly more **input** tokens (prompts) than **output** tokens (generations). Snowflake's AI Research team observed that many enterprise LLM use cases exhibit a **10:1 ratio** between prompt tokens and generated tokens. As a result, a significant portion of LLM compute costs on enterprise use cases is often

associated with processing the input prompt, which is left unaffected by KV-cache-compression approaches.

Based on this observation, we introduce **SwiftKV, a groundbreaking approach that prioritizes reducing inference computation during prompt processing rather than just compressing memory**. By combining model rewiring and knowledge-preserving self-distillation, SwiftKV achieves substantial reductions in computational overhead during inference with minimal accuracy loss, leading to transformative improvements in throughput, latency and cost efficiency for enterprise LLM workloads by **up to 2x**. SwiftKV directly tackles the real bottleneck of modern inference systems, unlocking new levels of performance.

In this blog, we will delve into the core design of SwiftKV, evaluate its compute-quality tradeoffs and demonstrate its impact in production environments. For further details, check out our [arXiv paper](#). Additionally, we're excited to announce the open sourcing of SwiftKV, including [inference-ready model checkpoints](#) and knowledge-distillation pipelines (coming soon).

Why reducing computation matters more than compressing KV cache

The limitations of KV cache compression

KV cache compression has been a popular research area, as it offers memory savings that support longer sequences and larger batch sizes.

AUTHOR

[Snowflake AI Research](#)

SHARE



These benefits are especially valuable for optimizing workloads on GPUs with limited memory. However, enterprise-scale AI workloads often run on compute nodes equipped with high-end GPUs (such as NVIDIA H100s or A100s), where the aggregate GPU memory is typically sufficient to accommodate long input sequences (up to hundreds of thousands of tokens) or batch sizes large enough to saturate the compute resources. Consequently, the memory savings from KV cache compression have limited impact on overall throughput and cost-efficiency for many enterprise-scale use cases.

The real bottleneck: Compute operations

Model	Input length	Throughput (toks/sec) (Baseline)	Throughput (toks/sec) (Merge-all-Layers)	Throughput (toks/sec) (50% SwiftKV)
Llama 3.1 8B(1x H100)	4K	18.3k	19.8k	25.2k
Llama 3.1 8B(1x H100)	16K	16.2k	17.0k	23.5k
Llama 3.1 70B (4x H100)	4K	9.19k	10.1k	13.2k
Llama 3.1 70B (4x H100)	16K	8.21k	9.12	12.8k

Table 1. This table shows combined throughput for Llama 3.1 8B and 70B BF16 models, running enterprise workloads on a production environment using: a) baseline vLLM, b) significant KV cache compression of greater than 30x with Merge-all-Layers, and c) ~50% prefill compute reduction with SwiftKV. Table shows that for these compute-bound enterprise workloads, KV cache compression has limited impact on throughput, while compute reduction improves throughput significantly.

As shown in Table 1, even an extreme form of KV cache compression achieved by sharing KV cache across all layers (*Merge-all-Layers*) yields more modest throughput improvements, while **compute reduction techniques, such as SwiftKV, drive significant throughput gains**. For example, SwiftKV achieves up to **~50% prefill compute reduction**, delivering throughput improvements of 40% to 50%, as compared to compression-only approaches.

Therefore, in compute-bound scenarios, reducing the computation is key to unlocking faster, more cost-effective inference. This is where SwiftKV excels, going beyond traditional compression techniques to tackle the computational bottlenecks directly.

How does SwiftKV work?

AUTHOR

Snowflake AI Research

Figure 1. SwiftKV design

SHARE

Observation: What computation to reduce?

SwiftKV is built on an important observation: Several enterprise workloads process significantly more input tokens (prompts) than output tokens (generations). For example, code completion, text-to-SQL, summarization and retrieval-augmented generation (RAG) commonly submit long prompts but generate only a small number of output tokens. Many enterprise LLM use cases exhibit a **10:1 ratio** between prompt tokens and generated tokens (Figure 2).

Figure 2. Average input vs. output tokens across all Snowflake Cortex workloads. The figure shows that on average, the length of input tokens is about 10x longer than generated tokens.

As you can see, the bulk of computational cost lies in prefilling the KV cache for input tokens rather than in decoding output tokens. Based on this observation, SwiftKV was designed to reduce the computation during the prefill stage of inference computation.

Core design: How to Reduce the Computation?

During the prefill phase, KV cache corresponding to the input prompts are produced at each transformer layer to be used for attention computation during the generation phase. This KV cache is produced using a lightweight projection matrix. While projection itself is lightweight, the input to this projection matrix at a given layer depends on the output from the previous transformer layer. As such, producing KV cache across all layers requires computing all output from each of the transformer layers in the model including self-attention and multi-layer perceptron (MLP), which constitutes the majority of the computation.

With SwiftKV we introduce **SingleInputKV**, a technique that reduces the KV cache computation by leveraging a well-known observation that output of transformer layers do not change much as we get deeper into the layers. With this observation, SingleInputKV **optimizes KV cache computation by reusing the output from an earlier layer (L_{swift}) to generate the KV cache for multiple subsequent layers ($j > L_{\text{swift}}$), using just the lightweight projection (see Figure 1).**

By skipping compute-heavy operations in the later layers – such as attention and MLP projections – SingleInputKV achieves substantial efficiency improvements. It reduces prefill compute costs by up to ~50%, making prompt processing significantly faster and more cost-effective for enterprise workloads.

Table 2 demonstrates the computational savings achieved, with reductions of up to ~50% during prefill.

Per-token prefill compute (GFlops), Layers=80, Hidden Size 8K, Vocab 128K, SeqLen 128K									
	Vocab	Q	K	V	O	MLP	ATTN	Total per Prefill Token	Prefill Compute
Baseline	4.29	10.7	1.34	1.34	10.7	113	160	301	100%
25% SingleInputKV	4.29	8.05	1.34	1.34	8.05	84.6	120	228	75.5%

50% SingleInputKV	4.29	5.37	1.34	1.34	5.37	56.4	80	154	51.1%
-------------------	------	------	------	------	------	------	----	-----	-------

Table 2. SingleInputKV at 25% and 50% reduces the total prefill compute by 24.5% and 48.9%, respectively. Firstly, SingleInputKV allows most of the compute-heavy operations (Q, O, MLP, ATTN) to be skipped entirely for the later layers. Second, SingleInputKV skips the attention operation of the later layers to retain its relative performance improvements for long and short sequences alike.

Knowledge recovery: How to Preserve Accuracy?

Due to model rewiring, SingleInputKV preserves all the original model parameters, and therefore the original knowledge is preserved. However, due to the rewiring, this knowledge is scrambled and requires a lightweight fine-tuning¹ to retrain the SingleInputKV model to extract this knowledge from its parameters.

Key aspects of the fine-tuning process (see Figure 1):

- Selective fine-tuning:** As we only change the computation of the attention part for later layers ($j > L_{\text{swift}}$), adjusting only the WQ, WK and WV weight matrices for layers $j > L_{\text{swift}}$ minimizes disruption to the model's parameters.
- Self-distillation:** Using the original model's output logits as supervision during fine-tuning helps the rewired model retain knowledge more effectively than standard LM loss.

This approach limits accuracy loss to an average of **~1 point**, as shown in Tables 3, 4 and 5 , while reducing prefill computation by up to ~50%.

	Arc Challenge	Wino grand e	Hella Swag	Truth fulQ A	MML U	MML U cot	GSM 8K	Avg
Basel ine	94.7	87.0	88.3	64.7	87.5	88.1	96.1	86.6
50% Singl elnpu tKV	94.0	86.3	88.1	64.2	85.7	87.5	95.2	85.9

Table 3. Llama 3.1 405B Instruct (FP8)

	Arc Challenge	Winogrande	HellaSwag	TruthfulQA	MMLU	MMLU cot	GSM8K	Avg
Baseline	84.13	84.69	87.27	56.86	72.70	74.86	86.5	78.23
50% SingleInputKV	83.5	83.98	86.22	55.67	72.6	74.04	84.3	77.24

Table 4. Mistral Small Instruct 2409 22B (BF16)

	Arc Challenge	Winogrande	HellaSwag	TruthfulQA	MMLU	MMLU cot	GSM8K	Avg
Baseline	82.00	77.90	80.40	54.56	67.90	70.63	82.56	73.71
50% SingleInputKV	80.38	78.22	79.30	54.54	67.30	69.73	79.45	72.70

Table 5. Llama 3.1 8B Instruct (BF16)

In fact, with this knowledge preservation, SwiftKV can achieve significantly better quality than those of a model half its size, while incurring similar computation costs (see Table 6).

Benchmark	Llama 2 7B	Llama 2 13B	Llama 2 13B + 50% SingleInputKV
Arc-Challenge	0.4352	0.5009	0.4676
Winogrande	0.7316	0.7569	0.7545
Hellaswag	0.7867	0.8250	0.8139
TruthfulQA	0.4558	0.4412	0.4385
MMLU	0.4384	0.5339	0.5346
MMLU-CoT	0.2310	0.3372	0.3180
GSM8K-CoT	0.2547	0.3783	0.3745
Average	0.4762	0.5390	0.5288

Table 6. Comparing model quality between Llama 2 7B and Llama 2 13B + 50% SingleInputKV. The results show that the latter achieves nearly 5% higher accuracy on average than the former while requiring

approximately the same amount of compute (50% of 13B is roughly 7B) for enterprise workloads.

KV cache compression: How to reduce memory together with computation?

While SingleInputKV focuses on reducing computation, it does not inherently reduce the memory footprint of the KV cache. To address this, SwiftKV incorporates **AcrossKV**, a technique that compresses the KV cache by sharing it across multiple consecutive layers (see Figure 1). This approach reduces memory requirements while maintaining compatibility with the computation-reduction benefits of SingleInputKV.

AUTHOR

Snowflake AI Research

SHARE



Table 7. Llama 3.1 8B KV cache quantization results

SwiftKV is also compatible with existing KV cache compression techniques, such as quantization. When AcrossKV is combined with KV cache quantization, SwiftKV achieves up to **62.5% KV cache compression** with minimal accuracy impact, averaging around **a single-point drop in accuracy** (see Table 5).

Table 8. Impact of AcrossKV on throughput of Llama 3.1 70B BF16 model when running with limited GPU memory.

As shown in Table 8. AcrossKV can significantly help improve throughput when running on memory constrained systems.

System- and model-level extensions: How to further reduce cost and latency?

SwiftKV is compatible with a wide range of system-level and model-level optimizations, including:

- Precision reduction: FP8 computation for faster processing and reduced memory usage.

Parallelism techniques: Tensor parallelism to distribute workloads across GPUs.

Attention optimization: Paged attention for efficient memory access patterns.

AUTHOR

[Snowflake AI Research](#)

SplitFuse scheduling: Combines prefill and decoding stages to achieve larger average batch size for better compute efficiency.

Speculative decoding: Advanced decoding strategies, such as Medusa and MLP-Speculator.

SHARE

By combining these system- and model-level extensions with SwiftKV's core innovations, users can achieve cost and latency reductions that surpass the current state-of-the-art.

vLLM implementation: How to bring these benefits to users?

SwiftKV is integrated into vLLM to make its computation and memory optimizations readily accessible for real-world enterprise workloads. Key features of the implementation include:

1. Pre-fill compute reduction: After processing up to layer L_{swift} , SwiftKV computes the KV cache for all subsequent layers ($j > L_{\text{swift}}$) in one step, fully leveraging its compute savings.
2. SplitFuse/chunked prefill: SwiftKV utilizes vLLM's SplitFuse mechanism, processing input tokens efficiently while interleaving prefill and decode stages in the same minibatch.
3. CUDA graph: SwiftKV combines vLLM's existing CUDA graph optimization with its own inner CUDA graph that optimizes the remaining decode tokens that propagate through the second half of the model, leading to performance improvements for both small and large minibatches.
4. Seamless integration: SwiftKV works out of the box with vLLM's existing features, including tensor parallelism, memory paging and speculative decoding, offering immediate performance improvements without disrupting user workflows.

We demonstrate the significant throughput, latency and memory improvements enabled by this integration in the next section.

End-to-end performance

In our inference evaluation, we focus on two common scenarios: batch inference for cost-sensitive scenarios and interactive inference for latency-sensitive scenarios².

We also evaluate key performance metrics relevant to both scenarios, such as combined throughput, time-to-first-token (TTFT) and time-per-output-token (TPOT).

Key Metrics: What defines inference efficiency?

To evaluate SwiftKV's performance, we focus on the following key

f .ne in ,: ✉

Combined throughput: The total number of input and output tokens processed per second. This determines:

For batch processing, the time required to complete jobs.

For interactive use, the volume of concurrent requests a system can handle.

TTFT: The latency between a user request and receiving the first token in the response.

TPOT: The latency between subsequent tokens after the first token.

These metrics are critical for optimizing cost, latency and user experience in enterprise workloads.

Batch Inference: How does SwiftKV reduce cost?

Batch-inference scenarios include bulk-text processing or serving models for cost-sensitive workloads. Achieving high **combined throughput** is critical, as it determines the efficiency of the system in processing large volumes of data or requests.

Figure 3. Combined input and output throughput for Llama 3.1 8B (left) and Llama 3.1 70B (right) across a range of input lengths (bottom).

BF16 Results: Figure 3 shows the results of Llama 3.1 8B and Llama 3.1 70B across several workloads with a range of input lengths running on 8xH100 GPUs. SwiftKV achieves higher combined throughput than the baseline model across all the workloads we evaluated. At 2K input tokens per prompt, SwiftKV achieves 1.2x to 1.5x higher throughput than the baseline model, and the benefits increase further to 1.8x to 2x for 128K inputs.

AUTHOR

Snowflake AI Research

Note that for an input length of 8K tokens, SwiftKV achieves a staggering 30K tokens/sec/GPU (480 TFLOPS/GPU) and 240K tokens/sec aggregate for Llama 3 8B, and 32K toks/sec over 8xH100 GPUs for Llama 3.1 70B model, which corresponds to 560 TFLOPS/GPU of BF16

SHARE


on Facebook, LinkedIn, and Email when normalized to baseline. Considering this is over 56% of the normalized hardware peak, it is an unusually high throughput for BF16 inference workloads, which is difficult to achieve even in training scenarios.

Figure 4. Combined input and output throughput for Llama 3.1 70B (left) and Llama 3.1 405B (right) across a range of input lengths (bottom).

FP8 Results: Figure 4 shows SwiftKV can be combined with FP8 to further boost throughput for Llama 3.1 70B and Llama 3.1 405B. SwiftKV (FP8) consistently improves the baseline (FP8) performance by 1.4x-1.5x, achieving normalized inference performance of up to 770 TFlops/GPU, while achieving up to 2.3x the throughput of baseline (BF16).

Interactive inference: How does SwiftKV improve latency?

SHARE

 **Figure 5.** TTFT (top) and TPOT (bottom) for input lengths 2000 (left), 8000 (middle), and 32000 (right) for Llama 3.1 70B BF16 model. For each request, a range of different request arrival rates is simulated. Each request generates 256 output tokens.

Interactive scenarios, such as chatbots and AI copilots, prioritize low-latency metrics like **TTFT** and **TPOT**, which directly impact user experience.

Figure 5 shows the TTFT and TPOT of Llama 3.1 70B Instruct across a range of request arrival rates and input lengths. When the arrival rate is too high, the TTFT increases significantly due to the request queue accumulating faster than they can be processed by the system. However, SwiftKV can sustain 1.5x to 2.0x higher arrival rates before experiencing such TTFT increase. When the arrival rate is low, SwiftKV can reduce the TTFT by up to 50% for workloads with longer input lengths. In terms of TPOT, SwiftKV achieves significant reductions for all but the lowest arrival rates, up to 61% for certain settings.

Figure 6. TTFT (top) and TPOT (bottom) for input lengths 2000 (left), 8000 (middle), and 32000 (right) for Llama 3.1 405B fp8 model. For each experiment, a range of different request arrival rates is simulated. Each request generates 256 output tokens.

Figure 6 shows very similar results using Llama 3.1 405B Instruct with FP8, with SwiftKV sustaining 1.5x to 2.0x higher arrival rates before experiencing TTFT increase and reducing TPOT for all but the lowest arrival rates, up to 68% for certain settings.

AUTHOR

Snowflake AI Research

SHARE



Figure 7. TPOT reduction by combining SwiftKV with Speculative Decoding using MLP-Speculator for Llama 3.1 70B on FP8.

At the lowest arrival rates, where inference is memory bandwidth bound, SwiftKV by itself does not reduce TPOT, however, it is compatible with existing techniques like Speculative Decoding that allows SwiftKV to reduce TPOT not only at high arrival rates but also at low arrival rates (see Figure 7).

Scope: When does SwiftKV have the most benefits?

As mentioned earlier, SwiftKV is designed to optimize common enterprise workloads where prefill tokens are significantly larger than decoded tokens. For use cases where the distribution of prefill vs. generated tokens are different, and the ratio between prefill and decoding is small, SwiftKV may not be effective in reducing cost. Our primary objective with SwiftKV was to lower the cost inference based on the prefill to decoding distribution we observed for majority enterprise workloads running in Snowflake Cortex.

Getting started with SwiftKV

We are thrilled to make SwiftKV accessible to the community. Here's how you can get started:

1. **Model checkpoints:** SwiftKV model checkpoints are available via

HuggingFace:

[SwiftKV Llama 3.1 8B](#)

- 2. Inference with vLLM:** We've integrated SwiftKV optimizations into vLLM to enhance inference efficiency. To try these models with vLLM, please use our dedicated [vLLM branch](#) (currently in the process of being upstreamed). Please see our [getting started instructions](#) for more details.

- 3. Train your own SwiftKV models:** Interested in customizing SwiftKV for specific workloads? Check out the SwiftKV knowledge distillation recipe in our new post-training library, ArcticTraining (*coming soon*).

We are excited to see what you build with the SwiftKV models!

Contributors

Aurick Qiao, Zhewei Yao, Samyam Rajbhandari (tech lead), Jeff Rasley, Michael Wyatt, Ye Wang, Flex Wang, Harshal Pimpalkhute, Yuxiong He

¹ 4 hours of distillation for Llama-3.1-8B-Instruct on a H100 node with 8 GPUs over 320M training tokens.

² For all experiments on Llama 3.1 8B Instruct, we use one NVIDIA H100 GPU with 80GB of memory. For experiments on Llama 3.1 70B Instruct, we use 4 NVIDIA H100 GPUs running the model with 4-way tensor parallelism for BF16, and two NVIDIA H100 GPUs running the model with 2-way tensor parallelism for FP8. For Llama 3.1 405B Instruct (FP8), we use eight NVIDIA H100 GPUs running the model with 8-way tensor parallelism.

SHARE



RELATED CONTENT

DEC 04, 2024

Snowflake’s Arctic Embed 2.0 Goes Multilingual: Empowering Global-Scale Retrieval with Inference Efficiency and High-Quality Retrieval

Snowflake is excited to announce the release of Arctic Embed L 2.0 and Arctic Embed M 2.0, the next iteration of our frontier embedding models, which now empower multilingual search....

[Have a look](#)

NOV 19, 2024

Benchmarking LLMs on Writing Feature Engineering Code

Today, the limitations of LLMs are predominantly assessed using benchmarks focused on language understanding, world...

[Here's How](#)

SEP 12, 2024

LLM Interactive Workloads: Optimizing GPU Capacity for Interactive and Batch Workloads

At Snowflake, we offer a wide variety of LLM-powered features in Cortex AI, including Cortex...

[Here's How](#)

READ THE SNOWFLAKE AI + DATA PREDICTIONS 2025

DOWNLOAD YOUR COPY

AUTHOR

Snowflake AI Research

SHARE

f

in

Sign up for Snowflake Communications

diana.shaw@snow

United States

By submitting this form, I understand Snowflake will process my personal information in accordance with their **Privacy Notice**. Additionally, I consent to my information being shared with Event Partners in accordance with Snowflake's **Event Privacy Notice**. I understand I may withdraw my consent or update my preferences **here** at any time.

SUBSCRIBE NOW

[Privacy Notice](#) | [Site Terms](#) | [Cookie Settings](#) | [Do Not Share My Personal Information](#)

© 2024 Snowflake Inc. All Rights Reserved | If you'd rather not receive future emails from Snowflake, [unsubscribe here](#) or [customize your communication preferences](#)

X

in

f