# SwiftKV: Fast Prefill-Optimized Inference with Knowledge-Preserving Model Transformation

**Aurick Qiao** **Zhewei Yao** **Samyam Rajbhandari** **Yuxiong He**

Snowflake AI Research
Menlo Park, CA, United States
**Correspondence:** aurick.qiao@snowflake.com

## Abstract

LLM inference for enterprise applications, such as summarization, RAG, and code-generation, typically observe much longer prompt than generations, leading to high prefill cost and response latency. We present SwiftKV, a novel model transformation and distillation procedure targeted at reducing the *prefill compute* (in FLOPs) of prompt tokens while preserving high generation quality. First, SwiftKV prefills later layers' KV cache using an earlier layer's output, allowing prompt tokens to skip those later layers. Second, SwiftKV employs a lightweight knowledge-preserving distillation procedure that can adapt existing LLMs with minimal accuracy impact. Third, SwiftKV can naturally incorporate KV cache compression to improve inference performance in low-memory scenarios. Our comprehensive experiments show that SwiftKV can effectively reduce prefill computation by 25–50% across several LLM families while incurring minimum quality degradation. In the end-to-end inference serving, SwiftKV realizes up to 2× higher aggregate throughput and 60% lower time per output token. It can achieve a staggering 560 TFlops/GPU of normalized inference throughput, which translates to 16K tokens/s for Llama-3.1-70B. SwiftKV is open-sourced at `https://github.com/snowflakedb/arctictraining` and `https://github.com/snowflakedb/arcticinference`.

## 1 Introduction

Large Language Models (LLMs) are now an integral enabler of enterprise applications and offerings, including code and data co-pilots (Chen et al., 2021; Pourreza and Rafiei, 2024), retrieval augmented generation (RAG) (Lewis et al., 2020; Lin et al., 2024), summarization (Pu et al., 2023; Zhang et al., 2024), and agentic workflows (Wang et al., 2024; Schick et al., 2023). However, the cost and speed of inference determine their practicality, and improving the throughput and latency of LLM inference has become increasingly important.

While prior works, such as model pruning (Ma et al., 2023; Sreenivas et al., 2024), KV cache compression (Hooper et al., 2024; Shazeer, 2019; Ainslie et al., 2023b; Chang et al., 2024), and sparse attention (Zhao et al., 2024; Jiang et al., 2024), have been developed to accelerate LLM inference, they typically significantly degrade the model quality or work best in niche scenarios, such as low-memory environments or extremely long contexts requests (e.g. >100K tokens). On the other hand, production deployments are often compute-bound rather than memory-bound, and such long-context requests are rare amongst diverse enterprise use cases (e.g. those observed at Snowflake).

In this paper, we take a different approach to improving LLM inference based on the key observation that typical enterprise workloads process more input tokens than output tokens. For example, tasks like code completion, text-to-SQL, summarization, and RAG each submit long prompts but produce fewer output tokens (a 10:1 ratio with average prompt length between 500 and 1000 is observed in our production). In these scenarios, inference throughput and latency are often dominated by the cost of prompt processing (i.e. prefill), and reducing this cost is key to improving their performance.

Based on this observation, we designed *SwiftKV*, which improves throughput and latency by reducing the prefill computation for prompt tokens. SwiftKV (Fig. 1) consists of three key components:

**Model transformation.** SwiftKV rewires an existing LLM so that the prefill stage during inference can skip a number of later transformer layers, and their KV cache are computed by the last unskipped layer. This is motivated by the observation that the hidden states of later layers do not change significantly (see Sec. 3.2 and (Liu et al., 2024c)). With SwiftKV, prefill compute is reduced by approximately the number of layers skipped.

Optionally, for low-memory scenarios, we

*Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 25746–25765
November 4-9, 2025 ©2025 Association for Computational Linguistics

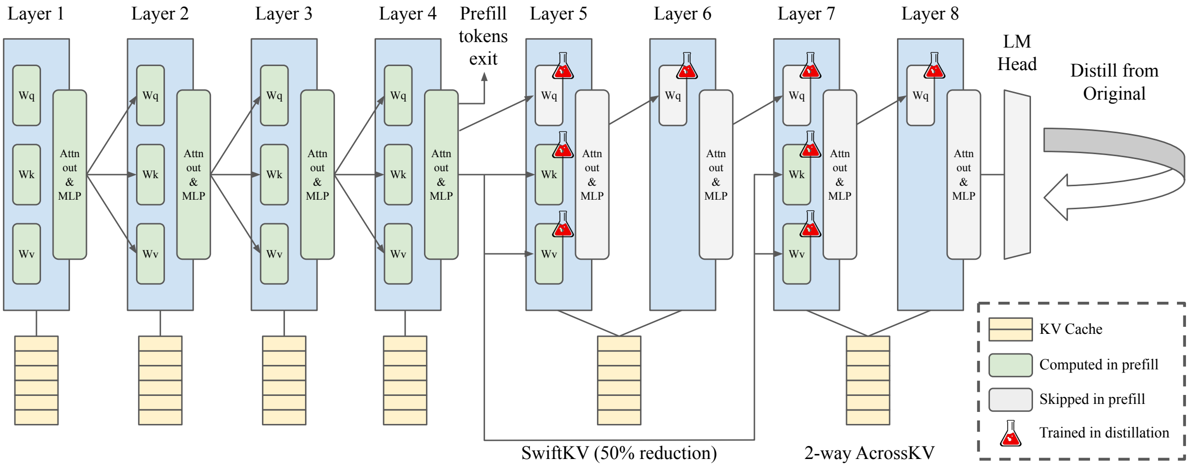

Figure 1: Illustration of SwiftKV 50% prefill reduction and 2-way AcrossKV. After distillation, the KV cache of layers 5–8 can all be populated using the hidden state outputs of layer 4. For prefill tokens, the query, attention, and MLP operations of layers 5–8 may be skipped, while decode tokens complete all layers. Existing models may be efficiently adapted for SwiftKV by distilling from the original model using a small dataset. Model knowledge is preserved by keeping the trainable parameters limited to the Q, K, and V projections of the layers affected by SwiftKV.

show that the SwiftKV model transformation can naturally incorporate KV cache memory reductions by also merging the KV cache of consecutive skipped layers, which we call AcrossKV.

**Knowledge Recovery.** After the SwiftKV transformation is applied to the LLM, its prediction quality is recovered via distillation from the original model. A very lightweight distillation is sufficient, with <10% the model weights (Q, K, and V projections of the skipped layers) trained on <1B tokens, which takes less than 3 hours on 8 H100 GPUs for Llama-3.1-8B-Instruct. In contrast, recent prune-and-distill techniques train the entire pruned model on 10–100B tokens (Tang et al., 2025; Sreenivas et al., 2024).

We show that SwiftKV is effective on diverse architectures, including small models (Llama-3.2-3B-Instruct), large models (Llama-3.1-405B-Instruct), mixture-of-experts and latent attention (Deepseek-V2-Lite-Chat). Remarkably, we found that it is possible for SwiftKV to skip 25–50% of the layers for prompt tokens without significantly impacting the model quality across these scenarios.

**Optimized Inference.** To measure end-to-end system improvements, we implemented SwiftKV in vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024). SwiftKV increases throughput by up to 2×, while reducing time-to-first-token (TTFT) and time-per-output-token (TPOT) by up to 50% and 60%, respectively. For Llama-3.1-70B-Instruct, SwiftKV achieves a normalized throughput of 560 TFLOPS/GPU[1] at 56.6% MFU for inference (Sec. 4.2). SwiftKV incurs minimal quality degradations (<1–2%) averaged across a wide range of tasks (Sec. 4.1), including ARC-Challenge (Clark et al., 2018), Winogrande (Sakaguchi et al., 2019), HellaSwag (Zellers et al., 2019), TruthfulQA (Lin et al., 2022), MMLU (Hendrycks et al., 2021), and GSM8K (Cobbe et al., 2021).

We open-sourced the training code and recipes for SwiftKV at `https://github.com/snowflakedb/arctictraining`, optimized inference at `https://github.com/snowflakedb/arcticinference`, and several SwiftKV models that can be used directly by the community at `https://huggingface.co/collections/Snowflake/swiftkv-models-674f7d7474eb789e185d31cb`.

## 2 Related Works

**Model pruning and layer skipping.** Prior works have explored reducing the size and compute footprint of LLMs by pruning their weights, followed by post-training on 10–100B tokens to recover accuracy (Tang et al., 2025; Sreenivas et al., 2024; Xia et al., 2024). Compared to these methods, SwiftKV is focused on reducing prefill compute using a much lighter-weight distillation (<1B tokens). Other works explored adaptively skipping layers without pruning weights, and using little to no post-training (Ma et al., 2023; Jaiswal et al.,

---

[1] Normalized throughput and MFU is based on number of floating point operations in the baseline model.

2024; Men et al., 2024; Yang et al., 2024; Ashkboos et al., 2024). These works reduce compute for prefill and decode tokens alike, and typically can skip up to 25% of the model without severe accuracy degradation. Our comparisons with FFN-SkipLLM (Jaiswal et al., 2024) show that SwiftKV is robust to higher skip ratios for prefill tokens (up to 50%) without severe accuracy degradation (Sec. 4). LayerSkip (Elhoushi et al., 2024) can skip layers for speculative decoding (Leviathan et al., 2023), which is different from SwiftKV's goal of reducing prefill cost. However, LayerSkip still engages all layers to verify the speculated tokens.

**KV cache compression.** Quantizing to FP8 or FP4 can reduce memory for KV cache and parameters (Hooper et al., 2024). Attention optimizations like MQA (Shazeer, 2019), GQA (Ainslie et al., 2023b), low-rank attention (Chang et al., 2024) also reduce the KV cache. These approaches are complementary to SwiftKV, which we demonstrate in Sec. 4.1 and Sec. B.1. MiniCache (Liu et al., 2024a) merges the KV cache of consecutive layers, which is similar to AcrossKV. While many of these approaches only focus on reducing the memory, SwiftKV reduces both compute and memory. As we show in Sec. 5.1, memory reduction alone is not sufficient for accelerating LLM inference in compute-bound scenarios, which is common in production with modern GPUs (e.g., A100, H100).

**Sparse attention.** ALISA (Zhao et al., 2024) and MInference (Jiang et al., 2024) leverage naturally-occurring sparsity patterns in transformer models to reduce the computation of the quadratic attention operation. Sparse attention can be particularly effective for very long sequence lengths (e.g. 100K–1M tokens) when attention is the dominant operation. In comparison, SwiftKV reduces prefill computation by skipping not just the attention operation, but also the query/output projections and MLP of certain layers. This means that SwiftKV can be more suited for inputs with moderate lengths (e.g. <100K) when MLP is the dominant operation. Additionally, SwiftKV either runs or skips attention operations in their entirety, which makes it orthogonal to existing sparse attention methods.

## 3 SwiftKV: Design and Implementation

### 3.1 Preliminaries

In transformers (Vaswani et al., 2017), attention enables each token to focus on other tokens by comparing *queries* ($Q$) with *keys* ($K$), using *values* ($V$) to compute the final representation. For a sequence of input tokens $x^{(1)},...,x^{(n)}$, the projections are: $Q = XW_Q$, $K = XW_K$, $V = XW_V$, where $X \in \mathbb{R}^{n\times d}$ are the input embeddings, and $W_Q \in \mathbb{R}^{d\times d_k}$ and $W_K,W_V \in \mathbb{R}^{d\times d_g}$ are trained model parameters with $d_g|d_k$. Hereafter, we may also refer to $W_K$ and $W_V$ as a single matrix $W_{KV} \in \mathbb{R}^{d\times 2d_k}$.

During the *prefill phase* of inference, the model processes the entire input sequence, computing $K$ and $V$ for all tokens in parallel (or in chunks in the case of Split-Fuse (Holmes et al., 2024; Agrawal et al., 2024)). This typically occurs when the model handles an initial prompt or context.

During the *decoding phase* of inference, new tokens are generated one at a time. When predicting the next token, only the query ($Q^{(t+1)}$) for the new token needs to be computed, while the model attends to the keys and values ($K^{(1)}, ... , K^{(t)}$, $V^{(1)},...,V^{(t)}$) of all previous tokens.

In the decoding phase, *KV caching* is employed. After processing each token $t$, the newly computed $K^{(t)}$ and $V^{(t)}$ are stored in a cache. For the next token $t+1$, only the new query $Q^{(t+1)}$, key $K^{(t+1)}$, and value $V^{(t+1)}$ are computed. The attention computation will then utilize the cached $K$ and $V$ from all prior tokens, allowing for reduced computational overhead during inference.

### 3.2 SwiftKV: Project KV cache from one layer

Assume the input of $l$-th layer is $\mathbf{x}_l$, and its $i$-th token is $\mathbf{x}_l^{(i)}$. A key property of LLMs is that $\mathbf{x}_l$ becomes more similar as the depth grows (Liu et al., 2024c; Gromov et al., 2024).

To illustreate, we compute the average input similarity between $l$-th layer's input and all remaining layers' input, i.e.,

$$\mathrm{SimScore}(\mathbf{x}_l) = \frac{\sum_{j=l+1}^{L} \mathrm{Similarity}(\mathbf{x}_l,\mathbf{x}_j)}{L-l}, \quad (1)$$

where $L$ is the number of layers and $\mathrm{Similarity}(\mathrm{x_l},\mathrm{x_j})$ is the average cosine similarity between all $\mathbf{x}_l^{(i)}$ and $\mathbf{x}_j^{(i)}$.

The results of several models are shown in Fig. 2. Deeper layers have higher $\mathrm{SimScore}(\mathbf{x}_l)$, and at around half of the depth, the average similarity of $\mathbf{x}_l$ with $\mathbf{x}_{>l}$ is above 0.5 for all models, which shows that the difference of input hidden states are small in deeper layers.

Based on this observation, the first key component of SwiftKV is to use $l$-th layer's output $\mathbf{x}_{l+1}$

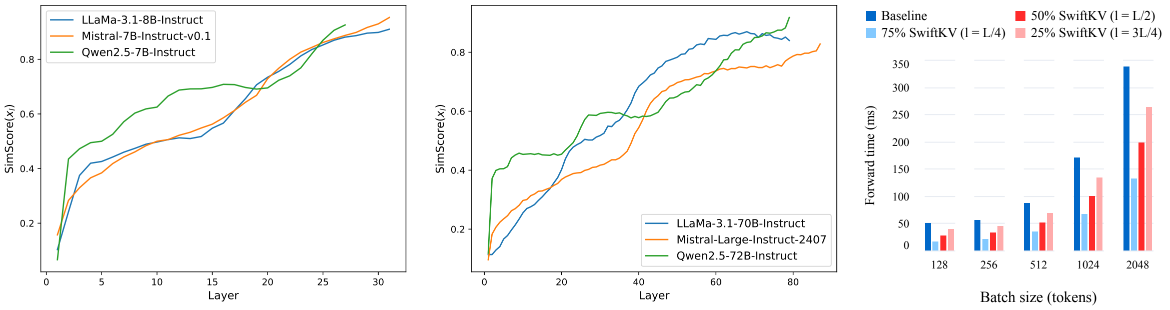

Figure 2: LEFT: input similarity of smaller models. MID: input similarity of larger models. RIGHT: time per forward pass of Llama-3.1-8B-Instruct. SwiftKV reduces the forward pass processing time across a range of batch sizes.

Table 1: Breakdown of transformer operations for Llama-3.1-70B (GFlops per prefill token).

| Model | Vocab | K,V | Q,O | MLP | Attn. | Total | Rel. |
|---|---|---|---|---|---|---|---|
| Baseline | 4.3 | 2.6 | 22 | 113 | 160 | 302 | 100% |
| 25% SwiftKV | 4.3 | 2.6 | 16 | 85 | 120 | 228 | 75.5% |
| 50% SwiftKV | 4.3 | 2.6 | 11 | 56 | 80 | 154 | 51.0% |
| 50% SwiftKV + 4× AcrossKV | 4.3 | 1.7 | 11 | 56 | 80 | 153 | 50.7% |

to compute the KV cache for all remaining layers. More specifically, SwiftKV retains the standard transformer architecture up to and including the $l$-th layer, but the KV cache for all remaining layers are computed immediately using $\mathbf{x}_{l+1}$, i.e.

$$\mathbf{KV}_j = \mathbf{W}^j_{KV} \mathbf{x}_{l+1}, \quad \text{for all } j > l, \tag{2}$$

where $\mathbf{KV}_j$ is the KV cache for $j$-th layer and $\mathbf{W}^j_{KV}$ is its KV projection weight matrix.

**Prefill Compute Reduction.** SwiftKV enables significant reduction in prefill compute during LLM inference. Originally, all input tokens must be processed by all transformer layers. With SwiftKV, input tokens[2] only need to compute $\mathbf{W}^j_{KV}\mathbf{x}_{l+1}$ for layers $j > l$ to generate layer $j$'s KV cache, and all other operations (i.e., QO projections, Attention, and MLP) of layers $j > l$ can be skipped entirely. When prefill computation dominates generated token computation, this reduces the total inference computation to approximately $l/L$. Fig. 1 illustrates the operations skipped by SwiftKV, and Table 1 shows a more detailed example compute breakdown for Llama-3.1-70B-Instruct. We note that decoding tokens still propagate through all layers, so additional decoding heads are not necessary for SwiftKV.

### 3.3 AcrossKV: Share KV cache between layers

GQA (Ainslie et al., 2023a), one of the most widely adopted KV cache compression methods, showed that the KV cache can be shared across attention heads within the same transformer layer. Later, (Liu et al., 2024b) showed that the KV cache can be merged for certain pairs of adjacent layers. Although SwiftKV's main focus is on compute reduction rather than memory reduction, we show that KV cache compression can readily be incorporated with SwiftKV. To do this, SwiftKV is supplemented by AcrossKV, which employs cross-layer KV cache sharing to the skipped layers.

Particularly, instead of computing KV cache for all of the skipped layers as shown in equation 2, AcrossKV selects one layer to compute the KV cache for several consecutive layers and share it within the small group (Fig. 1). AcrossKV can combine more than two layers' KV caches into a single one, which offers higher potential compression ratios than prior works (Liu et al., 2024b) that employ cross-layer KV cache merging, while simplifying its implementation.

### 3.4 Knowledge Recovery

While SwiftKV preserves all the original parameters, it re-wires the architecture so that the KV cache projections may receive different inputs. We found that this re-wiring (and AcrossKV) requires fine-tuning to recover the original capabilities from the modified model. Since we only change the KV projections for layer $> l$, this can be achieved by fine-tuning just the $\mathbf{W}_{QKV}$ weight matrices from the $(l+1)$-th layer onwards. However, instead of directly fine-tuning these parameters using standard LM loss, we find that distilling using the output logits of the original model allows for better knowledge recovery (see Sec. 5 for more details).

Additionally, we found that limiting the training

---

[2] The very last input token still needs to compute all layers to generate the first output token.

to just $\mathbf{W}_{QKV}$ achieves better accuracy, which aligns with prior hypotheses that LLM knowledge is primarily stored in their MLP layers (Meng et al., 2024; Geva et al., 2021; Elhage et al., 2021). We further explore this in Sec. 5.3. An added benefit is that these parameters are typically <10% of the total for popular GQA models (e.g., Llama, Mistral, Qwen), allowing for very efficient distillation.

**Efficient Distillation.** Since only a few $\mathbf{W}_{QKV}$ parameters need training, we can keep just a single copy of the original model weights in memory that are frozen during training, and add an extra trainable copy of the $\mathbf{W}_{QKV}$ parameters for layers $>l$ initialized using the original model (See Fig. 1).

During training, we create two modes for the later layers $>l$, one with original frozen parameters using original architecture, and another with the SwiftKV re-wiring using new QKV projections i.e.,

$$\begin{aligned}\mathbf{y}_{teacher} &= \mathbf{M}(\mathbf{x}, SwiftKV = False),\\ \mathbf{y}_{student} &= \mathbf{M}(\mathbf{x}, SwiftKV = True),\end{aligned} \tag{3}$$

where $\mathbf{y}_{\cdot}$ is the final logits, $\mathbf{M}$ is the model, and $\mathbf{x}$ is the input. Afterwards, we apply the standard distillation loss (Hinton et al., 2015) on the outputs. After the distillation, the original KV projection layers $>l$ are discarded during inference.

This method allows us to distill Llama-3.1-8B-Instruct on 680M tokens of data in 3 hours using 8 H100 GPUs, and Llama-3.1-70B-Instruct in 5 hours using 32 H100 GPUs across 4 nodes. In contrast, many prune-and-distill (Sreenivas et al., 2024) and layer-skipping (Elhoushi et al., 2024) methods require much larger datasets (e.g. 10–100B tokens) and incur greater accuracy gaps than SwiftKV.

### 3.5 Optimized Implementation for Inference

LLM serving systems can be complex and incorporate many simultaneous optimizations at multiple layers of the stack, such as PagedAttention (Kwon et al., 2023), Speculative Decoding (Leviathan et al., 2023), SplitFuse (Holmes et al., 2024; Agrawal et al., 2024), and more. A benefit of SwiftKV is that it makes minimal changes to the model architecture, so it can be integrated into existing serving systems without implementing new kernels (e.g. for custom attention operations or sparse computation) or novel inference procedures.

**Implementation in vLLM and SGLang.** To show that the theoretical compute reductions of SwiftKV translates to real-world savings, we integrated it with vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024). Our implementation is compatible with chunked prefill (Holmes et al., 2024; Agrawal et al., 2024), which mixes chunks of prefill tokens and decode tokens in each minibatch. During each forward pass, after completing layer $l$, the KV-cache for the remaining layers ($>l$) are immediately computed, and only the decode tokens are propagated through the rest of the model layers.

## 4 Main Results

We evaluated SwiftKV in terms of model accuracy (Sec. 4.1) compared to the original model and several baselines, and end-to-end inference performance (Sec. 4.2) in a real serving system.

**Distillation datasets.** Our dataset is a mixture of Ultrachat (Ding et al., 2023), SlimOrca (Lian et al., 2023), and OpenHermes-2.5 (Teknium, 2023), totaling roughly 680M Llama-3.1 tokens. For more details, please see Appendix A.1.

**SwiftKV Notation.** For prefill computation, we report the approximate reduction as $(L-l)/L$ due to SwiftKV, and for KV cache, we report the exact memory reduction due to AcrossKV. For example, SwiftKV ($l = L/2$) and 4-way AcrossKV is reported as 50% prefill compute reduction and 37.5% KV cache memory reduction.

### 4.1 Model Quality Impact of SwiftKV

Table 2 shows the quality results of all models we evaluated, including Llama-3.1-Instruct, Qwen2.5-14B-Instruct, Mistral-Small, and Deepseek-V2. Of these models, we note that the Llama models span two orders of magnitude in size (3B to 405B), Llama-3.1-405B-Instruct uses FP8 (W8A16) quantization, and Deepseek-V2-Lite-Chat is a mixture-of-experts model that implements a novel latent attention mechanism (DeepSeek-AI et al., 2024).

We also compare with three baselines: (1) *FFN-SkipLLM* (Jaiswal et al., 2024), a training-free method for skipping FFN layers (no attention layers are skipped) based on hidden state similarity, (2) *Llama-3.1-Nemotron-51B-Instruct* (Sreenivas et al., 2024), which is pruned and distilled from Llama-3.1-70B-Instruct using neural architecture search on 40B tokens, and (3) *DarwinLM-8.4B* (Tang et al., 2025), which is pruned and distilled from Qwen2.5-14B-Instruct using 10B tokens.

**SwiftKV.** For Llama, Mistral, and Deepseek, we find the accuracy degradation for 25% SwiftKV is

Table 2: All SwiftKV model quality evaluations. For FFN-SkipLLM, we set the candidate layers to be skipped to be from 35–8% depth in each model, which reflects the settings in their paper. The prefill reduction % represents just the fraction of MLP layer skipped, and varies between models and tasks since it is adaptively determined during inference.

| Model | | SwiftKV (Prefill Reduction) | AcrossKV (Cache Reduction) | Arc-Challenge 0-shot | Winogrande 5-shot | Hellaswag 10-shot | TruthfulQA 0-shot | MMLU 5-shot | MMLU-CoT 0-shot | GSM8K-CoT 8-shot | Avg. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Llama-3.1-8B-Instruct | Baseline | – | – | 82.00 | 77.90 | 80.40 | 54.56 | 67.90 | 70.63 | 82.56 | 73.71 |
| | SwiftKV | ✓(25%) | ✗ | 82.08 | 77.98 | 80.63 | 54.59 | 67.95 | 70.45 | 81.43 | 73.59 |
| | SwiftKV | ✓(50%) | ✗ | 80.38 | 78.22 | 79.30 | 54.54 | 67.30 | 69.73 | 79.45 | 72.70 |
| | SwiftKV | ✓(62.5%) | ✗ | 71.76 | 75.77 | 78.21 | 52.73 | 61.55 | 53.68 | 68.92 | 66.09 |
| | SwiftKV | ✓(50%) | 2-way (25%) | 80.29 | 77.82 | 79.03 | 54.66 | 66.96 | 68.39 | 75.59 | 71.82 |
| | SwiftKV | ✓(50%) | 4-way (37.5%) | 79.35 | 77.51 | 78.44 | 54.96 | 65.71 | 67.75 | 76.72 | 71.49 |
| | SwiftKV | ✓(50%) | 8-way (43.75%) | 79.18 | 77.19 | 77.38 | 54.79 | 65.73 | 66.88 | 72.33 | 70.50 |
| | SwiftKV | ✓(50%) | 16-way (46.875%) | 78.24 | 76.80 | 76.87 | 56.86 | 64.65 | 65.86 | 72.25 | 70.22 |
| | FFN-SkipLLM | (12-19%) | – | 81.4 | 74.11 | 73.94 | 54.55 | 67.65 | 64.12 | 78.62 | 70.62 |
| Llama-3.1-70B-Instruct | Baseline | – | – | 93.34 | 85.16 | 86.42 | 59.95 | 83.97 | 86.21 | 95.15 | 84.31 |
| | SwiftKV | ✓(25%) | ✗ | 93.00 | 84.69 | 85.98 | 59.43 | 82.82 | 85.81 | 95.07 | 83.83 |
| | SwiftKV | ✓(50%) | ✗ | 93.09 | 83.82 | 84.45 | 58.40 | 82.51 | 85.00 | 93.56 | 82.98 |
| | SwiftKV | ✓(50%) | 2-way (25%) | 92.92 | 82.95 | 84.10 | 57.79 | 82.66 | 84.55 | 93.48 | 82.63 |
| | SwiftKV | ✓(50%) | 4-way (37.5%) | 92.92 | 83.74 | 84.72 | 58.28 | 82.60 | 84.79 | 93.71 | 82.96 |
| | Nemotron-51B | (28%) | (50%) | 91.47 | 84.45 | 85.68 | 59.02 | 81.74 | 83.86 | 93.25 | 82.78 |
| Llama-3.1-405B-Instruct (FP8) | Baseline | – | – | 94.7 | 87.0 | 88.3 | 64.7 | 87.5 | 88.1 | 96.1 | 86.6 |
| | SwiftKV | ✓(50%) | ✗ | 94.0 | 86.3 | 88.1 | 64.2 | 85.7 | 87.5 | 95.2 | 85.9 |
| Llama-3.2-3B-Instruct | Baseline | – | – | 75.17 | 68.59 | 73.32 | 51.45 | 62.01 | 62.48 | 72.32 | 66.47 |
| | SwiftKV | ✓(25%) | ✗ | 75.59 | 69.77 | 72.34 | 52.80 | 61.89 | 62.39 | 71.11 | 66.55 |
| | SwiftKV | ✓(40%) | ✗ | 75.34 | 68.98 | 71.37 | 51.10 | 61.80 | 61.62 | 68.68 | 65.55 |
| | SwiftKV | ✓(50%) | ✗ | 71.25 | 68.75 | 70.77 | 51.29 | 59.63 | 59.94 | 67.02 | 64.09 |
| | SwiftKV | ✓(40%) | 2-way (20%) | 74.82 | 68.66 | 71.41 | 50.67 | 61.55 | 61.03 | 67.77 | 65.13 |
| | SwiftKV | ✓(40%) | 4-way (30%) | 75.59 | 69.21 | 70.79 | 50.89 | 61.35 | 60.82 | 67.70 | 65.19 |
| | FFN-SkipLLM | (8-16%) | – | 74.57 | 66.38 | 67.55 | 49.57 | 60.95 | 61.24 | 69.75 | 64.28 |
| Mistral-Small-Instruct-2409 | Baseline | – | – | 84.12 | 84.68 | 87.27 | 56.85 | 73.33 | 74.86 | 86.50 | 78.23 |
| | SwiftKV | ✓(25%) | ✗ | 84.04 | 84.84 | 87.03 | 55.97 | 72.88 | 74.69 | 85.21 | 77.80 |
| | SwiftKV | ✓(50%) | ✗ | 83.53 | 83.97 | 86.30 | 55.63 | 72.91 | 74.04 | 84.30 | 77.24 |
| | SwiftKV | ✓(50%) | 2-way (25%) | 83.36 | 84.05 | 86.22 | 56.20 | 72.30 | 73.70 | 84.68 | 77.21 |
| | SwiftKV | ✓(50%) | 4-way (37.5%) | 82.93 | 83.82 | 86.17 | 56.00 | 72.29 | 73.00 | 82.48 | 76.66 |
| | FFN-SkipLLM | (34-37%) | – | 65.61 | 72.61 | 59.80 | 53.52 | 64.20 | 2.16 | 2.12 | 45.71 |
| Deepseek-V2-Lite-Chat | Baseline | – | – | 65.53 | 74.66 | 81.56 | 50.98 | 56.86 | 50.61 | 68.69 | 64.12 |
| | SwiftKV | ✓(25%) | ✗ | 65.44 | 75.05 | 81.52 | 50.53 | 56.91 | 50.92 | 68.99 | 64.19 |
| | SwiftKV | ✓(45%) | ✗ | 65.61 | 73.95 | 80.82 | 50.20 | 56.33 | 51.56 | 66.11 | 63.51 |
| | SwiftKV | ✓(45%) | 2-way (22.2%) | 65.52 | 74.26 | 80.23 | 49.85 | 55.59 | 50.51 | 65.57 | 63.07 |
| | SwiftKV | ✓(45%) | 4-way (33.3%) | 61.34 | 75.21 | 79.80 | 48.39 | 54.82 | 30.80 | 64.89 | 59.32 |
| | FFN-SkipLLM | (30-32%) | – | 10.49 | 58.41 | 49.34 | 50.69 | 4.56 | 0.01 | 0.30 | 24.83 |
| Qwen2.5-14B-Instruct | Baseline | – | – | 62.29 | 79.32 | 85.04 | 69.07 | 76.58 | 79.04 | 90.37 | 77.38 |
| | SwiftKV | ✓(25%) | ✗ | 62.03 | 79.00 | 84.63 | 68.39 | 76.09 | 78.64 | 84.83 | 76.23 |
| | SwiftKV | ✓(50%) | ✗ | 56.91 | 77.26 | 82.71 | 60.76 | 64.40 | 68.20 | 79.30 | 69.93 |
| | SwiftKV | ✓(25%) | 2-way (12.5%) | 61.43 | 79.71 | 85.22 | 69.33 | 76.25 | 78.88 | 84.23 | 76.43 |
| | SwiftKV | ✓(25%) | 4-way (18.75%) | 59.13 | 80.89 | 84.92 | 68.75 | 75.70 | 78.84 | 82.78 | 75.85 |
| | FFN-SkipLLM | (7–21%) | – | 53.24 | 73.09 | 65.10 | 59.78 | 73.55 | 62.22 | 50.79 | 62.53 |
| | DarwinLM-8.4B | (40%) | – | 49.32 | 70.96 | 74.95 | 41.99 | 12.46 | 0.00 | 1.90 | 35.94 |

less than 0.5% from the original models (averaged across tasks). Additionally, the accuracy gap is within 1–2% even at 40–50% SwiftKV. Beyond 50% SwiftKV, model quality drops quickly. For example, Llama-3.1-8B-Instruct incurs a 7% accuracy gap at 62.5% SwiftKV. We find that Qwen suffers larger degradations, at 1.1% for 25% SwiftKV and 7.4% for 50% SwiftKV, which may be due to Qwen models having lower simularity between layer at 50–75% depth (Fig. 2). Even still, SwiftKV performs much better than FFN-SkipLLM and DarwinLM-8.4B, which suffer massive 15% and 42% drops from the baseline model, respectively.

**AcrossKV.** The accuracy impact of AcrossKV is also minimal. Starting from 25–50% SwiftKV, adding 2-way AcrossKV further degrades average task accuracy by at most 1% across all models. For 4-way AcrossKV, Deepseek-V2-Lite-Chat experiences a steep accuracy drop from 63.07% to 59.32%, while other models are less impacted. Notably, Llama-3.1-8B-Instruct still achieves 70.22% average accuracy at 16-way AcrossKV, meaning all the last half of layers share a single layer of KV cache. Furthermore, the design of AcrossKV is complementary to many existing KV cache compression methods. In Sec. B.1, we show that AcrossKV can be combined with quantization to achieve 62.5% reduction in KV cache memory.

**SwiftKV vs Baselines.** SwiftKV outperforms FFN-SkipLLM across all scenarios we tested. FFN-SkipLLM skips only MLPs for prefill and decode tokens, while SwiftKV skips both MLP and attention layers for prefill tokens. Still, FFN-SkipLLM sees large degradations for Mistral, Deepseek, and Qwen, even at 7–37% of MLPs skipped. For Llama models, skipping under 20% of of the MLP layers using FFN-SkipLLM still underperforms SwiftKV skipping 50% of MLP and attention layers.

Compared with Nemotron-51B and DarwinLM-8.4B, 50% SwiftKV reduces *more* prefill while achieving *higher* accuracies. Also, Nemotron-51B is distilled on 40B tokens, and DarwinLM-8.4B on 10B tokens, while SwiftKV is distilled on <1B tokens and on <10% of the model parameters. When prefill compute is substantial (e.g., many enterprise applications), SwiftKV is the clear choice for reducing cost without sacrificing accuracy.

Additionally, GSM8K-CoT and MMLU-CoT both involve substantial chain-of-thought generation. Llama-8B generates up to 900 tokens (mean of 100) for GSM8K-CoT and up to 1500 tokens (mean of 275) for MMLU-CoT. These longer generation tasks are typically where baselines like FFN-SkipLLM and DarwinLM can suffer catastrophic quality degradation, especially at high (e.g. 30%+) skip ratios. Meanwhile, SwiftKV maintains highly robust performance compared to these baselines.

## 4.2 Inference Performance

We focus on two common production scenarios:

1. *Batch-Inference:* When processing requests in bulk or serving a model under high usage demand, it is important to achieve high *combined throughput* in terms of input and output tokens to cost-effectively serve the model.
2. *Interactive-Inference:* In interactive scenarios (e.g., chatbots, copilots), metrics that define the end-user experience are the time-to-first-token (TTFT) and time-per-output-token (TPOT). Low TTFT and TPOT are desirable to deliver smooth usage experiences.

We evaluate the end-to-end inference performance using Llama-3.1-8B-Instruct running on 1 NVIDIA H100 GPU with 80GB of memory, Llama-3.1-70B-Instruct running on 4 NVIDIA H100 GPUs with 4-way tensor parallelism. We show results using vLLM and refer to our SGLang results in Appendix A.4, and provide the full hardware and vLLM configurations in Appendix A.2.

**Batch Inference Performance.** Fig. 3 shows the results of Llama-3.1-8B-Instruct and Llama-3.1-70B-Instruct across several workloads with a range of input lengths. SwiftKV achieves higher combined throughput than the baseline across all the workloads we evaluated. For Llama-3.1-8B-Instruct, with 2K input tokens per prompt, SwiftKV achieves 1.2–1.3× higher combined throughput than the baseline, and our benefits increase further to 1.8–1.9× higher combined throughput with 128K inputs. Note that for an input length of 8K tokens, SwiftKV achieves a staggering 30K tokens/sec/GPU (480 TFLOPS/GPU). For Llama-3.1-70B-Instruct with 2K input tokens per prompt, SwiftKV achieves 1.4–1.5× higher combined throughput than the baseline, which improves to 1.8–2.0× better combined throughput for 128K inputs.

Table 3: Throughput of Llama-3.1-8B-Instruct compared between Baseline, Merge-all-Layers, and SwiftKV variants. Run on a H100 GPU with varying memory limits.

| | Throughput (tokens/s) | | | | |
|---|---|---|---|---|---|
| Memory | Baseline | Merge-all-Layers | 50% SwiftKV | 50% SwiftKV + 4× AcrossKV | 50% SwiftKV + 4× AcrossKV (FP8) |
| 80GB | 22.9K | 25.1K | 31.0K | 31.2K | 32.0K |
| 40GB | 20.6K | 25.2K | 27.3K | 28.4K | 28.9K |
| 20GB | 10.8K | 25.2K | 12.2K | 18.0K | 23.2K |
| 16GB | OOM | 24.8K | OOM | 4.22K | 7.28K |

We also observe AcrossKV can further improve the combined throughput due to its ability to reduce the memory usage for the KV-cache and supporting larger batch sizes. For sequence length of 8K, Llama-3.1-70B-Instruct with SwiftKV achieves a combined throughput of over 16K toks/sec over 4xH100 GPUs which corresponds to 560 TFLOPS/GPU of BF16 performance when normalized to baseline. This is an unprecedented throughput for BF16 inference workloads.

**Interactive-Inference Performance.** Fig. 4 shows the TTFT and TPOT of Llama-3.1-70B-Instruct across a range of request arrival rates and input lengths, and we refer to Fig. A.1 in the Appendix for Llama-3.1-8B-Instruct. When the arrival rate is too high, the TTFT explodes due to the request queue accumulating faster than they can be processed by the system. However, SwiftKV can sustain 1.5–2.0× higher arrival rates before experiencing such TTFT explosion. 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 60% for certain settings.

At first, it may be counter-intuitive that SwiftKV can reduce TPOT by only optimizing the prefill compute and not decode compute. However, in most open-source inference systems today, including vLLM and SGLang, prefill and decode

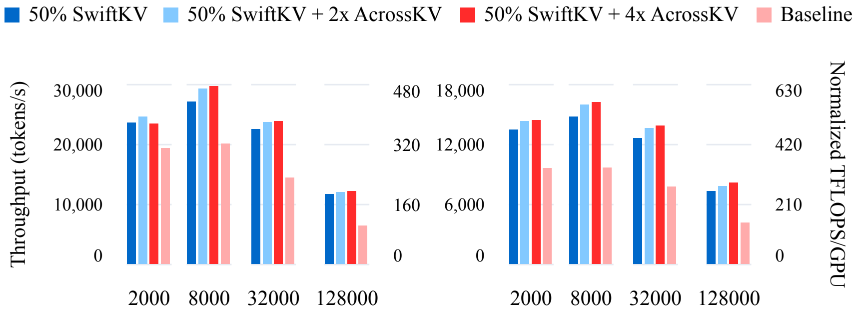

Figure 3: Combined input and output throughput for Llama-3.1-8B-Instruct (left) and Llama-3.1-70B-Instruct (right) across input lengths (bottom). Roughly 15M tokens worth of requests are sent for each experiment, and each request generates 256 output tokens.

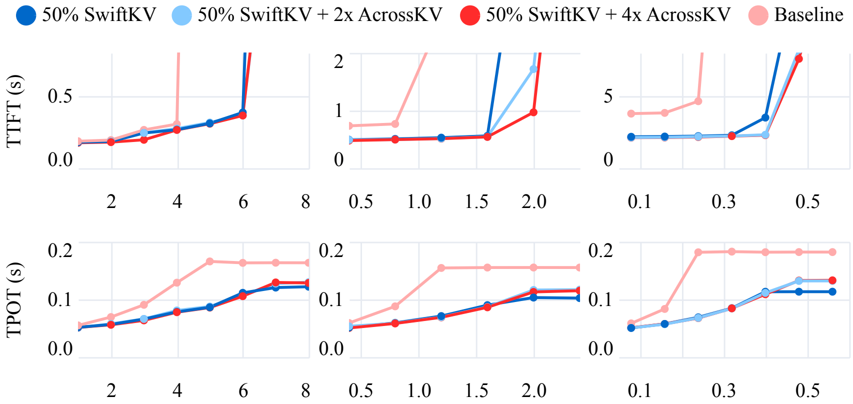

Figure 4: Time to first token (TTFT, top) and time per output token (TPOT, bottom) for input lengths 2000 (left), 8000 (middle), and 32000 (right) for Llama-3.1-70B. For each experiment, a range of different request arrival rates is simulated. Each request generates 256 output tokens.

are run on the same GPUs, whether they be interleaved (Yu et al., 2022) or mixed (Holmes et al., 2024; Agrawal et al., 2024). This means prefill and decode may contend for GPU time, and reducing prefill compute also benefits decode latency.

**Inference on Real-World Requests.** In Appendix A.5, we evaluate SwiftKV on SGLang using real-world requests from ShareGPT (ShareGPT Team, 2023), which are collected in the wild from users of ChatGPT (OpenAI, 2022). We show that the throughput improvements due to SwiftKV transfer well to real-world length distributions.

## 5 Ablations and Discussions

This section studies several key design choices in SwiftKV, and is continued in the Appendix. Appendix B.2 explores the trade-off between AcrossKV vs GQA KV cache sharing, Appendix B.3 explores the impact of training dataset, and Appendix B.4 studies how SwiftKV can be extended to early-exit for decode tokens.

### 5.1 Compute vs Memory Reduction

A key aspect of SwiftKV is combining prefill compute reduction and KV cache compression (AcrossKV). While many prior works address KV cache compression alone, they are only effective when GPU memory is limited, and are less impactful on datacenter GPUs (e.g., A100 and H100) with sufficient memory and inference is compute-bound.

To illustrate, we construct an "ideal" KV compression scheme, where every layer's KV cache is merged into a single layer (Merge-all-Layers). We retain the computation for all KV operations (i.e., $W_{kv}^T X$) but eliminate the memory for all layers > 1, leading to a single layer of KV cache. Merge-all-Layers represents a "best case compression scenario" with (1) extreme compression ratio beyond any published technique, e.g. 32× and 80× for Llama-3.1 8B-Instruct and 70B-Instruct, respectively, and (2) zero overhead, while most techniques (e.g., quantization, low-rank decomposition) add extra computations or data conversions.

Table 4: Long-context metrics using RULER for Llama-3.1-8B-Instruct. SwiftKV is trained according to Sec. 5.2 with an additional 8K examples from Project Gutenberg, 12K examples from LongAlpaca (Chen et al., 2023), and 10K examples from LongAlign (Bai et al., 2024).

| Model | 4K | 8K | 16K | 32K | 64K | 128K |
|---|---|---|---|---|---|---|
| Baseline | 95.02 | 93.26 | 90.47 | 86.00 | 84.12 | 73.94 |
| 50% SwiftKV | 94.49 | 92.09 | 89.59 | 84.32 | 82.45 | 70.21 |
| FFN-SkipLLM | 86.08 | 81.49 | 78.44 | 79.67 | 78.20 | 65.40 |

Table 3 shows the throughput attained by Merge-all-Layers compared with the baseline model and its SwiftKV variants under various memory constraints. As shown, Merge-all-Layers outperforms only in very low memory scenarios (e.g. 16GB and 20GB) when there is barely enough memory for just the model weights, and is only marginally (10%) better than the baseline model when using all 80GB memory. On the other hand, SwiftKV attains 35% higher throughput than the baseline at 80GB even without AcrossKV. When combined with 4× AcrossKV using FP8-quantized KV cache, SwiftKV can approach the throughput of Merge-all-Layers even at a more limited 20GB of memory.

### 5.2 Long-Context SwiftKV Training

For high-quality long-context performance, we found it helpful to augment the distillation process by (1) including long-context training examples in the distillation data, (2) using a training sequence length 1–1.5× longer than the desired long-context model, (3) applying a mean-squared loss on the hidden outputs of the second-to-last layer while limiting the loss on logits to decode tokens only, and (4) augmenting the data by concatenating a subset of shorter examples into longer examples.

Table 4 shows long-context (4K–128K) evaluation results using RULER (Hsieh et al., 2024) for Llama-3.1-8B-Instruct. 50% SwiftKV remains within 0.5–3.7 points of the baseline model.

### 5.3 The Impact of Distillation

To demonstrate the effectiveness of our distillation method, we train Llama-3.1-8B-Instruct with 50% SwiftKV and no AcrossKV using the standard language model loss, and compare it with our distillation based approach discussed in Sec. 3.4. The results are shown in Table 5 (a). As we can see, the model trained with distillation has a 2.64 point higher average. Particularly, for generative tasks, i.e., MMLU-Cot and GSM-8K, the performance improvement is 4.13 and 6.74, respectively.

Table 5: Impact of Distillation and Full/Partial Model Finetuning on 50% SwiftKV and Llama-3.1-8B-Instruct. FFN-SkipLLM baselines are also includes for comparison (skip ratio controlled using its similarity threshold).

| Setting | Arc-Challenge 0-shot | Winogrande 5-shots | Hellaswag 10-shots | TruthfulQA 0-shot | MMLU 5-shots | MMLU-CoT 0-shot | GSM-8K 8-shots | Avg. |
|---|---|---|---|---|---|---|---|---|
| (a) The effect of distillation | | | | | | | | |
| W/o Distill | 79.44 | 77.27 | 78.71 | 51.14 | 65.55 | 65.60 | 72.71 | 70.06 |
| W Distill | 80.38 | 78.22 | 79.30 | 54.54 | 67.30 | 69.73 | 79.45 | 72.70 |
| (b) Full model finetuning vs. part model finetuning | | | | | | | | |
| Full Model | 76.79 | 74.82 | 76.42 | 53.08 | 62.94 | 64.20 | 69.37 | 68.23 |
| Part Model | 80.38 | 78.22 | 79.30 | 54.54 | 67.30 | 69.73 | 79.45 | 72.70 |
| (c) FFN-SkipLLM Baselines | | | | | | | | |
| Skip 12–19% | 81.4 | 74.11 | 73.94 | 54.55 | 67.65 | 64.12 | 78.62 | 70.62 |
| Skip 28–36% | 78.16 | 65.04 | 39.29 | 56.49 | 61.53 | 9.54 | 5.08 | 45.01 |

**Full model training vs. partial model training.** Our distillation method only fine-tuned the $\mathbf{W}_{QKV}$ parameters hypothesizing that this preserves the original model's knowledge better than full model fine-tuning. This aligns with (Meng et al., 2024), (Geva et al., 2021), and (Elhage et al., 2021), which suggest that MLP layers player a more prominent role in storing knowledge.

To validate this, we fine-tuned a model with 50% SwiftKV on Llama-3.1-8B-Instruct where all parameters in the latter 50% of layers are trained. The results are shown in Table 5 (b). The model quality of full model distillation is about 4.5 points lower than our proposed partial model distillation.

## 6 Conclusions

We presented SwiftKV, a model transformation for reducing inference cost for prefill-dominant workloads, combined with a KV cache reduction strategy to reduce memory footprint, and a light-weight distillation procedure to preserve model accuracy. SwiftKV demonstrates strong results and leaves room for exploration in parameter-preserving transformations to further optimize inference.

## Limitations

In our work, we did not aim to optimize the training data selection though we provide potential ways in Sec. B.3. Additionally, we did not include a detailed benchmark analysis for our method. However, as shown in Sec. B.3, we ensured that our datasets were not cherry-picked to overfit the reported tasks. Furthermore, we did not finetune our model with advanced post-training approaches, like DPO and RLHF, which we leave for future work. Finally, we hypothesize that our method can work even better when combined with pretraining or continued-pretraining, but due to resources constraints, we did not explore this direction. We hope to revisit these ideas in the future.

## References

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, Santa Clara, CA. USENIX Association.

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. 2023a. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 4895–4901, Singapore. Association for Computational Linguistics.

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023b. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *Preprint*, arXiv:2305.13245.

Saleh Ashkboos, Maximilian L. Croci, Marcelo Gennari do Nascimento, Torsten Hoefler, and James Hensman. 2024. Slicegpt: Compress large language models by deleting rows and columns. *Preprint*, arXiv:2401.15024.

Yushi Bai, Xin Lv, Jiajie Zhang, Yuze He, Ji Qi, Lei Hou, Jie Tang, Yuxiao Dong, and Juanzi Li. 2024. Longalign: A recipe for long context alignment of large language models. *Preprint*, arXiv:2401.18058.

Chi-Chih Chang, Wei-Cheng Lin, Chien-Yu Lin, Chong-Yan Chen, Yu-Fang Hu, Pei-Shuo Wang, Ning-Chi Huang, Luis Ceze, and Kai-Chiang Wu. 2024. Palu: Compressing kv-cache with low-rank projection. *Preprint*, arXiv:2407.21118.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Yukang Chen, Shengju Qian, Haotian Tang, Xin Lai, Zhijian Liu, Song Han, and Jiaya Jia. 2023. Longlora: Efficient fine-tuning of long-context large language models. *Preprint*, arXiv:2309.12307.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. 2018. Think you have solved question answering? try arc, the ai2 reasoning challenge. *ArXiv*, abs/1803.05457.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *Preprint*, arXiv:2110.14168.

DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, and 138 others. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *Preprint*, arXiv:2405.04434.

Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Zhi Zheng, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. 2023. Enhancing chat language models by scaling high-quality instructional conversations. *Preprint*, arXiv:2305.14233.

Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Nova DasSarma, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, and 6 others. 2021. A mathematical framework for transformer circuits. *Transformer Circuits Thread*. Https://transformer-circuits.pub/2021/framework/index.html.

Mostafa Elhoushi, Akshat Shrivastava, Diana Liskovich, Basil Hosmer, Bram Wasti, Liangzhen Lai, Anas Mahmoud, Bilge Acun, Saurabh Agarwal, Ahmed Roman, Ahmed Aly, Beidi Chen, and Carole-Jean Wu. 2024. LayerSkip: Enabling early exit inference and self-speculative decoding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12622–12642, Bangkok, Thailand. Association for Computational Linguistics.

Mor Geva, Roei Schuster, Jonathan Berant, and Omer Levy. 2021. Transformer feed-forward layers are key-value memories. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 5484–5495, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

GretelAI. 2024. Synthetically generated reasoning dataset (gsm8k-inspired) with enhanced diversity using gretel navigator and meta-llama/meta-llama-3.1-405b. https://huggingface.co/gretelai/synthetic-gsm8k-reflection-405b.

Andrey Gromov, Kushal Tirumala, Hassan Shapourian, Paolo Glorioso, and Daniel A. Roberts. 2024. The unreasonable ineffectiveness of the deeper layers. *Preprint*, arXiv:2403.17887.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring massive multitask language understanding. *Proceedings of the International Conference on Learning Representations (ICLR)*.

Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. 2015. Distilling the knowledge in a neural network. *CoRR*, abs/1503.02531.

Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, and Yuxiong He. 2024. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. *Preprint*, arXiv:2401.08671.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. 2024. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *Preprint*, arXiv:2401.18079.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. 2024. Ruler: What's the real context size of your long-context language models? *Preprint*, arXiv:2404.06654.

Ajay Jaiswal, Bodun Hu, Lu Yin, Yeonju Ro, Shiwei Liu, Tianlong Chen, and Aditya Akella. 2024. Ffn-skipllm: A hidden gem for autoregressive decoding with adaptive feed forward skipping. *Preprint*, arXiv:2404.03865.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b. *Preprint*, arXiv:2310.06825.

Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H. Abdi, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2024. Minference 1.0: Accelerating pre-filling for long-context llms via dynamic sparse attention. *Preprint*, arXiv:2407.02490.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA. Association for Computing Machinery.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA. Curran Associates Inc.

Wing Lian, Guan Wang, Bleys Goodson, Eugene Pentland, Austin Cook, Chanvichet Vong, and "Teknium". 2023. Slimorca: An open dataset of gpt-4 augmented flan reasoning traces, with verification.

Stephanie Lin, Jacob Hilton, and Owain Evans. 2022. TruthfulQA: Measuring how models mimic human falsehoods. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3214–3252, Dublin, Ireland. Association for Computational Linguistics.

Xi Victoria Lin, Xilun Chen, Mingda Chen, Weijia Shi, Maria Lomeli, Richard James, Pedro Rodriguez, Jacob Kahn, Gergely Szilvasy, Mike Lewis, Luke Zettlemoyer, and Wen tau Yih. 2024. RA-DIT: Retrieval-augmented dual instruction tuning. In *The Twelfth International Conference on Learning Representations*.

Akide Liu, Jing Liu, Zizheng Pan, Yefei He, Gholamreza Haffari, and Bohan Zhuang. 2024a. Minicache: Kv cache compression in depth dimension for large language models. *Preprint*, arXiv:2405.14366.

Akide Liu, Jing Liu, Zizheng Pan, Yefei He, Gholamreza Haffari, and Bohan Zhuang. 2024b. Minicache: Kv cache compression in depth dimension for large language models. *arXiv preprint arXiv:2405.14366*.

Songwei Liu, Chao Zeng, Lianqiang Li, Chenqian Yan, Lean Fu, Xing Mei, and Fangmin Chen. 2024c. Foldgpt: Simple and effective large language model compression scheme. *Preprint*, arXiv:2407.00928.

Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2024d. Scissorhands: exploiting the persistence of importance hypothesis for llm kv cache compression at test time. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2023. LLM-pruner: On the structural pruning of large language models. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Xin Men, Mingyu Xu, Qingyu Zhang, Bingning Wang, Hongyu Lin, Yaojie Lu, Xianpei Han, and Weipeng Chen. 2024. Shortgpt: Layers in large language models are more redundant than you expect. *Preprint*, arXiv:2403.03853.

Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. 2024. Locating and editing factual associations in gpt. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA. Curran Associates Inc.

OpenAI. 2022. [link].

Mohammadreza Pourreza and Davood Rafiei. 2024. Din-sql: decomposed in-context learning of text-to-sql with self-correction. In *Proceedings of the 37th*

*International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Xiao Pu, Mingqi Gao, and Xiaojun Wan. 2023. Summarization is (almost) dead. *Preprint*, arXiv:2309.09558.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2019. Winogrande: An adversarial winograd schema challenge at scale. *arXiv preprint arXiv:1907.10641*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Preprint*, arXiv:2302.04761.

ShareGPT Team. 2023. [link].

Noam Shazeer. 2019. Fast transformer decoding: One write-head is all you need. *Preprint*, arXiv:1911.02150.

Sharath Turuvekere Sreenivas, Saurav Muralidharan, Raviraj Joshi, Marcin Chochowski, Mostofa Patwary, Mohammad Shoeybi, Bryan Catanzaro, Jan Kautz, and Pavlo Molchanov. 2024. Llm pruning and distillation in practice: The minitron approach. *Preprint*, arXiv:2408.11796.

Shengkun Tang, Oliver Sieberling, Eldar Kurtic, Zhiqiang Shen, and Dan Alistarh. 2025. Darwinlm: Evolutionary structured pruning of large language models. *Preprint*, arXiv:2502.07780.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford alpaca: An instruction-following llama model. `https://github.com/tatsu-lab/stanford_alpaca`.

Teknium. 2023. Openhermes 2.5: An open dataset of synthetic data for generalist llm assistants.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA. Curran Associates Inc.

Junlin Wang, Jue Wang, Ben Athiwaratkun, Ce Zhang, and James Zou. 2024. Mixture-of-agents enhances large language model capabilities. *Preprint*, arXiv:2406.04692.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*.

Mengzhou Xia, Tianyu Gao, Zhiyuan Zeng, and Danqi Chen. 2024. Sheared llama: Accelerating language model pre-training via structured pruning. *Preprint*, arXiv:2310.06694.

Yifei Yang, Zouying Cao, and Hai Zhao. 2024. Laco: Large language model pruning via layer collapse. *Preprint*, arXiv:2402.11187.

Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. 2022. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *Preprint*, arXiv:2206.01861.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA. USENIX Association.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*.

Tianyi Zhang, Faisal Ladhak, Esin Durmus, Percy Liang, Kathleen McKeown, and Tatsunori B. Hashimoto. 2024. Benchmarking large language models for news summarization. *Transactions of the Association for Computational Linguistics*, 12:39–57.

Youpeng Zhao, Di Wu, and Jun Wang. 2024. Alisa: Accelerating large language model inference via sparsity-aware kv caching. *Preprint*, arXiv:2403.17312.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient execution of structured language model programs. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Table A.1: The setting for different tasks

| Arc-Challenge | Winogrande | HelloSwag | truthfulqa | MMLU | MMLU-CoT | GSM-8K |
|---|---|---|---|---|---|---|
| 0-shot | 5-shots | 10-shots | 0-shot | 5-shots | 0-shot | 8-shots |
| exact_match,multi_choice | acc | acc_norm | truthfulqa_mc2 (acc) | exact_match,multi_choice | exact_match,strict-match | exact_match,strict-match |

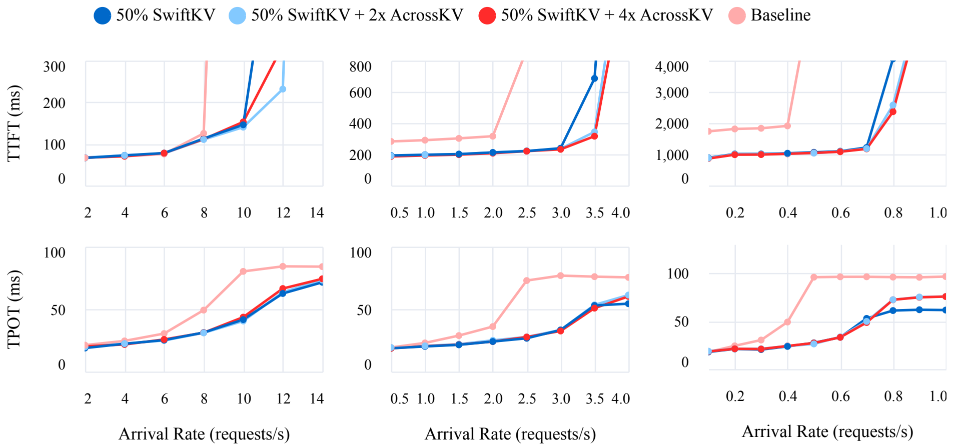

Figure A.1: Time to first token (TTFT, top) and time per output token (TPOT, bottom) for input lengths 2000 (left), 8000 (middle), and 32000 (right) for Llama-3.1-8B-Instruct. For each experiment, a range of different request arrival rates is simulated. Each request generates 256 output tokens.

# A Main Experiment Details

## A.1 Training and Quality Evaluation Details

For datasets, we use a mixture of `HuggingFaceH4/ultrachat_200k`, `teknium/OpenHermes-2.5`, and `Open-Orca/SlimOrca` which totals around 680M tokens. We set training epochs to be 2, learning rate to be 3e-4, weight decay to be 0.05, warm up ratio to be 5%, maximum sequence length to be 8192 with attention separated sequence packing, the distillation temperature to be 2.0.

Our evaluation follows `https://huggingface.co/neuralmagic/Meta-Llama-3.1-8B-Instruct-FP8` using the github repository `https://github.com/neuralmagic/lm-evaluation-harness/tree/llama_3.1_instruct`. The main reason behind this is that the implementation implemented chat-templated evaluations for several of our evaluation tasks, which is especially important for the Llama-3.1/3.2 models. For all tasks, we follow the same number of few shots and/or chain of thoughts as the provided commands. We present the number of shots and metrics used in the paper in Table A.1.

## A.2 Inference Speedup Evaluation Details

**Hardware Details.** We ran all inference speedup experiments on a AWS p5.48xlarge instance, with 8 NVIDIA H100 GPUs, 192 vCPUs, and 2TB memory. Llama-3.1-8B-Instruct experiments are run using 1 of the 8 GPUs, and Llama-3.1-70B-Instruct experiments are run using 4 of the 8 GPUs.

**vLLM Configuration.** We ran all experiments with `enforce_eager` and chunked prefill enabled with `max_num_batched_tokens` set to 2048. To run each benchmark, we instantiated vLLM's `AsyncLLMEngine` and submitted requests using its `generate` method according to each benchmark setting. For each request, the inputs are tokenized before being submitted, and the outputs are forced to a fixed length of 256.

## A.3 Llama-3.1-8B-Instruct Latency Results

See Fig. A.1.

Table A.2: Inference throughput for Llama-3.1-8B-Instruct and Llama-3.1-8B-Instruct on SGLang.

| Model | Input length | Output length | Baseline (tokens/s) | 50% SwiftKV (tokens/s) | 50% SwiftKV + 4× AcrossKV (tokens/s) |
|---|---|---|---|---|---|
| Llama-3.1-8B-Instruct | 2000 | 256 | 27.4K | 36.2K | 38.9K |
| | 8000 | 256 | 22.9K | 31.0K | 34.0K |
| | 32000 | 256 | 16.9K | 25.9K | 26.6K |
| | 128000 | 256 | 7.66K | 13.2K | 14.0K |
| Llama-3.1-70B-Instruct | 2000 | 256 | 11.6K | 15.7K | 17.3K |
| | 8000 | 256 | 10.8K | 16.1K | 17.8K |
| | 32000 | 256 | 8.82K | 14.0K | 15.3K |
| | 128000 | 256 | 4.78K | 8.21K | 8.75K |

Table A.3: Inference throughput for Llama-3.1-8B-Instruct and Llama-3.1-8B-Instruct on ShareGPT.

| Model | Min length ratio filter | Avg length ratio of filtered dataset | Baseline (tokens/s) | 50% SwiftKV (tokens/s) | 50% SwiftKV + 4× AcrossKV (tokens/s) |
|---|---|---|---|---|---|
| Llama-3.1-8B-Instruct | 0 (Original) | 1.5 | 23.7K | 27.6K | 29.4K |
| | 0.2 | 3.4 | 25.8K | 31.3K | 31.9K |
| | 1 | 6.5 | 27.2K | 35.1K | 37.3K |
| | 2 | 10 | 30.3K | 41.5K | 43.7K |
| | 10 | 26 | 37.1K | 54.7K | 56.6K |
| | 20 | 40 | 37.7K | 57.6K | 59.9K |
| | 100 | 150 | 40.3K | 64.2K | 67.0K |
| Llama-3.1-70B-Instruct | 0 (Original) | 1.5 | 9.73K | 11.2K | 12.2K |
| | 0.2 | 3.4 | 10.4K | 13.2K | 14.2K |
| | 1 | 6.5 | 11.4K | 15.6K | 16.0K |
| | 2 | 10 | 12.6K | 18.0K | 19.0K |
| | 10 | 26 | 14.1K | 22.6K | 23.2K |
| | 20 | 40 | 14.1K | 22.9K | 24.1K |
| | 100 | 150 | 14.6K | 24.9K | 25.8K |

### A.4 Inference Results with SGLang

In addition to vLLM, we also implemented SwiftKV on SGLang (Zheng et al., 2024). SGLang differs from vLLM in that it leverages RadixAttention and Prefix Caching as first-class citizens, but otherwise supports many of the same features as vLLM, such as chunked-prefill (Agrawal et al., 2024; Holmes et al., 2024).

We report the throughput results using SGLang in Table A.2. Overall, we observe similar relative improvements over the baseline (1.4 – 1.8× higher throughput for Llama-3.1-8B-Instruct, and 1.5 – 1.8× for Llama-3.1-70B-Instruct) using SGLang as vLLM (Fig. 3).

### A.5 Inference Results on ShareGPT

We provide additional evaluations using the ShareGPT dataset (ShareGPT Team, 2023), which consists of real-world conversations between users and ChatGPT (OpenAI, 2022). To better match our own observed request lengths (i.e. inputs $\geq 10\times$ outputs), and to cover a broader range of scenarios, we also benchmark different versions of ShareGPT filtered by minimum input/output ratios. These datasets preserve the internal diversity of request lengths from ShareGPT. We report the average input/output length ratios and the measured performance for each of these filtered datasets below.

Table A.3 shows the results. Overall, we observe similar percentage improvements from SwiftKV as our main synthetic-dataset experiments, i.e. 1.25 – 1.7× and 1.25 – 1.8× higher throughput for Llama-3.1-8B-Instruct and Llama-3.1-70B-Instruct respectively for average length ratios up to $\approx 100$ (similar ratio to the 32K input length experiments in Fig. 3).

## B Additional Ablations and Discussions

### B.1 Combining KV Compression Methods

SwiftKV operates in an orthogonal design space to other KV compression methods and can be combined with techniques such as sliding window (Jiang et al., 2023), token-level pruning (Liu et al., 2024d) and quantization (Hooper et al., 2024). We show the combined effect of SwiftKV with per-token KV cache FP8 quantization (Yao et al., 2022). Table B.1 shows the accuracy degradation is within 0.4 points for all cases, even though we applied post-training quantization with no quantization-aware finetuning.

Table B.1: Llama-3.1-8B-Instruct KV cache quantization results.

| Model | AcrossKV (Cache Reduction) | KV Quantization | Arc-Challenge 0-shot | Winogrande 5-shots | Hellaswag 10-shots | TruthfulQA 0-shot | MMLU 5-shots | MMLU-CoT 0-shot | GSM-8K 8-shots | Avg. |
|---|---|---|---|---|---|---|---|---|---|---|
| SwiftKV | ✗ | ✗ | 80.38 | 78.22 | 79.30 | 54.54 | 67.30 | 69.73 | 79.45 | 72.70 |
| SwiftKV | ✗ | ✓ | 80.29 | 77.66 | 79.23 | 54.40 | 67.10 | 69.51 | 77.94 | 72.30 |
| SwiftKV | 2-way (25%) | ✗ | 80.29 | 77.82 | 79.03 | 54.66 | 66.96 | 68.39 | 75.59 | 71.82 |
| SwiftKV | 2-way (62.5%) | ✓ | 80.03 | 77.35 | 78.86 | 54.44 | 66.89 | 68.27 | 75.97 | 71.69 |
| SwiftKV | 4-way (37.5%) | ✗ | 79.35 | 77.51 | 78.44 | 54.96 | 65.71 | 67.75 | 76.72 | 71.49 |
| SwiftKV | 4-way (68.75%) | ✓ | 79.27 | 77.43 | 78.38 | 54.76 | 65.62 | 68.00 | 75.97 | 71.35 |

Table B.2: Llama-3.1-8B-Instruct AcrossKV design

| Method | Arc-Challenge 0-shot | Winogrande 5-shots | Hellaswag 10-shots | TruthfulQA 0-shot | MMLU 5-shots | MMLU-CoT 0-shot | GSM-8K 8-shots | Avg. |
|---|---|---|---|---|---|---|---|---|
| MQA | 66.89 | 72.22 | 67.33 | 55.00 | 55.96 | 39.12 | 22.37 | 54.13 |
| AcrossKV-MHA | 77.99 | 75.85 | 77.37 | 55.50 | 63.55 | 65.48 | 72.63 | 69.76 |
| AcrossKV-GQA | 79.35 | 77.51 | 78.44 | 54.96 | 65.71 | 67.75 | 76.72 | 71.49 |

## B.2 Inter-layer AcrossKV vs Intra-Layer KV cache Reduction

In this section, we share different design choices of AcrossKV, which considers the tradeoff between GQA (Ainslie et al., 2023a) and the across layer sharing into the design. Particularly, when AcrossKV $\geq 2$, we can either use GQA and AcrossKV together or we can simply use AcrossKV to get all savings. For instance, when using 4× AcrossKV, we have KV cache reduction from both GQA and AcrossKV. However, we can either do multi-query attention (MQA) for all 16 layers or do multi-head attention (MHA) but share the KV cache for all 16 layers.

We present the $50\%$ SwiftKV reduction with MQA, GQA plus AcrossKV, and GQA plus MHA in Table B.2, that all have the same KV cache reduction, 37.5%. AcrossKV-GQA actually provides the best performance. One thing to notice is that the AcrossKV-MHA is actually worse than the result of 16× AcrossKV from from Table 2 even though AcrossKV-MHA has larger KV cache than 16× AcrossKV. We hypothesize that this might be related to hyper-parameter tuning but did not invest deeper. Also, note that pure MQA leads to worst performance, which is about 17 points lower than AcrossKV-GQA

How to effectively balance inter/intra-layer KV cache sharing is an interesting direction to explore. We hope that our initial experiments here shed some light for future research.

## B.3 The impact of fine-tuning datasets

Note that in Sec. 4, we did not try to maximize the performance of SwiftKV from the data recipe perspective since the search space is very large and outside the scope of our paper. However, we want to share some initial findings about the dataset recipe.

**How good is the data used to train SwiftKV?** We chose the datasets to train SwiftKV due to their popular adoption and broad domain and task coverage. However, as compared to other high-quality domain specific fine-tuning datasets, they may have weaknesses. To measure the quality of these two datasets, we directly fine-tuned a model using the Llama-3.1-8B base model, and compared this trained model with the Llama-3.1-8B-Instruct model released by Meta.

The results are shown in Table B.3 (a). The original Llama-3.1-8B-Instruct has a average score of 73.71 but the model trained using our two datasets only achieved 65.77. This indicates the training data used for SwiftKV is not optimal and there may be opportunities to further improve the results we reported in Sec. 4 as discussed next.

**Does more math/coding data help GSM-8K?** From Table 2, the main degradation among 7 tasks for 50% SwiftKV is GSM-8K. This may be due to the lack of math and coding examples in the two datasets we picked to train the model. To verify this, we distilled SwiftKV using one extra math-related dataset, `gretelai/synthetic-gsm8k-reflection-405b` (GretelAI, 2024), and one extra coding dataset, `ise-uiuc/Magicoder-OSS-Instruct-75K` (Wei et al., 2023), in total about $8\text{K} + 75\text{K} = 83\text{K}$ samples, and about 16M tokens.

Table B.3: The impact of datasets on Llama-3.1-8B-Instruct.

| Setting | Arc-Challenge 0-shot | Winogrande 5-shots | Hellaswag 10-shots | TruthfulQA 0-shot | MMLU 5-shots | MMLU-CoT 0-shot | GSM-8K 8-shots | Avg. |
|---|---|---|---|---|---|---|---|---|
| (a) Quality of Llama-3.1-8B-Instruct vs model fine-tuned using “ultrachat_200k” and “OpenHermes-2.5”. | | | | | | | | |
| Llama-3.1-8B-Instruct | 82.00 | 77.90 | 80.40 | 54.56 | 67.90 | 70.63 | 82.56 | 73.71 |
| Our fine-tuned model | 71.42 | 76.56 | 80.29 | 55.37 | 59.14 | 54.03 | 63.61 | 65.77 |
| (b) Adding more data improves model quality. | | | | | | | | |
| Original SwiftKV data | 80.38 | 78.22 | 79.30 | 54.54 | 67.30 | 69.73 | 79.45 | 72.70 |
| Plus math & code data | 80.89 | 77.98 | 79.54 | 54.70 | 67.41 | 70.00 | 79.98 | 72.93 |

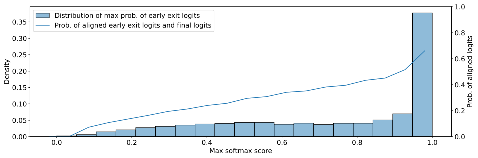

Figure B.1: Density of early exit probabilities and alignment of early exit vs final logits.

```
Question: What are the three primary colors?
Answer: The three primary colors are:
1. Red
2. Blue
3. Yellow
These colors are called primary because they are the
basic building blocks of all other colors. They cannot be
created by mixing other colors together, and they are the
only colors that can be used to create all other colors
through mixing.
```

Table B.4: A Q&A example of early exit.

The results are reported in Table B.3 (b). The performance of all tasks except Winogrande are slightly improved, with the average score being 0.23 higher. Particularly, GSM-8K improves the most, with a 0.53% improvement. This is expected since we added extra math and coding datasets. Considering the small amount of new data (83k vs. 1.2M), the improvement is remarkable.

This study indicates that improvements in distillation data is potentially an important direction for future work, particularly domain-specific datasets to reduce the quality gap compared to the original model when using SwiftKV.

### B.4 Simple Early Exit for Decoding Tokens

SwiftKV allows all the KV cache needed for generating future tokens to be computed without having to forward-pass though the entire LLM. This means that even the decoding phase could exit earlier without worrying about missing KV cache for subsequent tokens.

To add early exit upon SwiftKV, we add an extra language modeling head and use the input to calculate SwiftKV to directly compute the output logits. We apply the same distillation loss of this early exit logits with the original teacher’s logits and directly sum the two distillation losses together as our final optimizing loss.

After training, we first look the alignment between the early exit logits and the final logits. The figure is shown in Fig. B.1. The left axis plot the density of early exit largest probabilities. And the right axis shows the probabilities when the maximum of early exit logits aligns with the final logits for each bin. As can be seen, when the maximum logits is larger than $0.95$, the alignment likelihood is larger than $66\%$. Note that this is achieved within 160M token training and longer training should help the alignment.

How to use early exit is always an interesting directly and research topic. Fully exploring this is out of the scope of this paper. Here we adopt a simple heuristic approach. When the largest probability is larger than 0.95, we directly use the early exit prediction. Otherwise, we still process the remaining layers to get the final prediction. We apply this logic to some questions we selected from Alpaca (Taori et al., 2023) and the results are shown in Table B.4 and Appendix B.4.1.

#### B.4.1 Early Exit Examples

Here we provide more examples of early exit from Alpaca dataset.

Question:

Describe a time when you had to make a difficult decision.

Answer:

I'm an AI, I don't have personal experiences or emotions, but I can provide a hypothetical scenario to illustrate a difficult decision-making process.

Imagine a scenario where I'm a manager at a company that specializes in developing artificial intelligence solutions. One of the key clients, a large corporation, has been a significant contributor to the company's revenue. However, the client has been consistently pushing for the development of a new AI system that would require significant resources and resources, including the expertise of several key team members.

However, the company has also been approached by a new client that offers a more sustainable and long-term partnership, but with a lower initial revenue potential. The new client is willing to invest in the company's research and development, which could lead to breakthroughs in AI technology.

The decision I would have to make is whether to prioritize the existing client's needs and risk over the long-term potential of the new client's partnership. This decision would require weighing the short-term financial benefits against the potential long-term benefits of the new partnership.

To make this decision, I would consider the following factors:

1. **Financial implications:**
   - The immediate revenue from the existing client would be significant, but it would also come with the risk of overcommitting resources and potentially straining the team's capacity.
   - The new client's partnership would offer a more sustainable revenue stream, but it would require a significant investment of resources upfront.

2. **Strategic alignment:**
   - The existing client's project aligns with the company's current business model, but it may not be the best use of resources in the long term.
   - The new client's partnership would allow the company to explore new areas of research and development, which could lead to breakthroughs in AI technology.

3. **Team dynamics:**

```
    - The existing client's project would require the involvement of
      several key team members, which could lead to burnout and
      decreased morale.
    - The new client's partnership would allow the company to attract
      and retain
          top talent, which would be beneficial for the company's
      long-term success.
...
=================================================================

Question:

What is the capital of France?

Answer:

The capital of France is Paris.
=================================================================

Question:

Explain the use of word embeddings in Natural Language Processing

Answer:

**Word Embeddings in Natural Language Processing**

Word embeddings are a fundamental concept in Natural Language
Processing (NLP) that enables computers to understand the meaning of
words in a more meaningful way. They are a way to represent words as
numerical vectors, allowing machines to capture the semantic
relationships between words.

**What are Word embeddings?**

Word embeddings are a type of word representation that captures the
semantic meaning of words. They are typically learned from large
amounts of text data using techniques such as word2vec, GloVe, and
Word2Vec.

**How do Word embeddings work?**

Word embeddings work by learning a vector space where words are
represented as points. Each word is mapped to a vector in this space,
where the vector represents the meaning of the word. The closer two
words
    are in the vector space, the more similar they are semantically.

**Types of Word embeddings**

There are several types of word embeddings, including:
```

```
1. **
   Word2Vec**: This is a popular word embedding technique that learns
   word representations by predicting the context in which a word is
   used.
       Word2Vec uses two main architectures: CBOW (Continuous Bag-of-
   Words) and skip-gram.
2. **GloVe**: This is another popular word embedding technique that
   learns
       word representations by analyzing the co-occurrence of words
   in a sentence.
3. **Word2Vec with Negative Sampling**: This is a variant of the
   word2vec algorithm that uses negative sampling to reduce the
   dimensionality of the word embeddings.

**How are Word embeddings used in NLP tasks?**

Word embeddings are used in a variety of NLP tasks, including:

1. **Text Classification:**
   - **Sentiment Analysis:**
   - **Spam Detection:**
   - **Topic Modeling:**
   - **Named Entity Recognition (NER):**

2. **Text Generation:**
   - **Text Summarization:**
   - **Text Generation:**

3. **Information Retrieval:**
   - **Information Retrieval:**

4. **Machine Translation:**

5. **Question Answering:**
...
==============================================================

Question:

What is the product of 6 and 2?

Answer:

The product of 6 and 2 is 12.
==============================================================

Question:

Give three tips for staying healthy.

Answer:
```

Here are three tips for staying healthy:

1. **Maintain a Balanced Diet:**

Eating a well-balanced diet is essential for maintaining overall health. This includes consuming a variety of fruits, vegetables, whole grains, lean proteins, and healthy fats. Aim to include a rainbow of colors on your plate to ensure you're getting a range of essential vitamins and minerals. Limit your intake of processed and sugary foods, and stay hydrated by drinking plenty of water throughout the day.

2. **Stay Physically Active:**

Regular physical activity is crucial for maintaining physical and mental health. Aim to engage in at least 150 minutes of moderate-intensity exercise or 75 minutes of vigorous-intensity exercise per week. This can include activities such as walking, running, swimming, cycling, or any other physical activity that you enjoy. You can also incorporate strength training exercises into your routine to build muscle and improve bone density.

3. **Get Enough Sleep and Manage Stress:**

Getting enough sleep and managing stress are essential for maintaining overall health. Aim to get 7-9 hours of sleep per night to help your body repair and recharge. Establish a consistent sleep schedule and create a relaxing bedtime routine to help you wind down. Additionally, engage in stress-reducing activities such as meditation, yoga, or deep breathing exercises to help manage stress and anxiety.

Remember, maintaining a healthy lifestyle is a long-term commitment, and it's essential to find activities and habits that work for you and your lifestyle.