LEARNING HOW TO FORGET: FINE-TUNING FOR LONG-CONTEXT SPARSE ATTENTION

Anonymous authors
Paper under double-blind review

ABSTRACT

A lot of prior work addressed key-value (KV) cache selection and compression by sparse attention to enable long-context inference for transformer language models without excessive hardware budgets. We provide a new method for fine-tuning models with sparse attention. It works for any KV cache policy, runs on a moderate hardware budget (e.g., a single Nvidia A100 GPU with 40 GB RAM), and allows the model to co-adapt with the policy, often outperforming models trained with exact attention (sequence parallelism). We also provide an efficient implementation of H2O sparse attention (the leading policy in our experiments) with dedicated scaled dot product attention kernel support. A new open source library for long-context inference and fine-tuning will be released with the publication.

1 INTRODUCTION

Modern large language models need to process very long contexts (i.e., number of tokens) for calling many tools with sizable outputs (Schick et al., 2023; Feng et al., 2025a), running chain of thought reasoning (Wei et al., 2022), or sustaining multi-turn conversations. While naive transformer implementations scale quadratically in compute and linearly in memory with context width, a lot of progress has been made on approximations with essentially linear time and constant memory scaling. A particularly fruitful direction is *sparse attention*, where key-value (KV) information is stored in a fixed-size KV cache, slots of which are evicted once it is full.

In this paper, we address the open problem of how to *post-train* a transformer language model with sparse attention on a moderate hardware budget. Our novel method works for any KV cache policy and requires no further approximations beyond sparse attention. As we demonstrate in experiments on a range of long-context benchmarks, our training algorithm allows the model to co-adapt with the KV cache policy, often outperforming models trained with exact attention (sequence parallelism). Moreover, our fine-tuning method runs on resources comparable to sparse attention *inference*.

The heavy-hitter oracle (H2O) (Z. et al., 2023) is one of the most prominent sparse attention policies. We provide a much more efficient H2O implementation with dedicated scaled dot product attention (SDPA) kernel support. Variants of H2O outperform other KV cache policies in our experiments, and our fast implementation takes a big step towards latencies competitive with SotA inference libraries such as vLLM (Kwon et al., 2023). In summary, our contributions are:

- New method for fine-tuning transformer language models with sparse attention and arbitrary KV cache policy in place, which runs on resources comparable to sparse attention inference. It combines nested activation checkpointing and CPU offloading with exploiting a linear KV cache buffer recurrence. Our method can process sequences of arbitrary length with constant GPU resources.
- Improvements of heavy-hitter oracle (H2O) cache policy (Z. et al., 2023). In particular, we provide *Triton* code to return summed attention weights alongside a FlashInfer SDPA kernel (Ye et al., 2025) (see also Section A.3).
- A comprehensive evaluation on a range of long-context benchmarks. Our training algorithm mostly outperforms models trained with sequence parallelism (Li et al., 2023) when sparse attention inference is used.
- A novel open source library for long-context inference and fine-tuning, which will be released with the publication of this paper.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

2 Related Work

There is a large body of work on long-context inference by way of KV cache compression. Grouping heads requires fewer key and value vectors to be stored (Shazeer, 2019; Ainslie et al., 2023; Brandon et al., 2024; etal., 2024). Cache buffers can be quantized to 8 or 4 bits (Hooper et al., 2024; Liu et al., 2024b; Zhang et al., 2024a; Li et al., 2025b; Shutova et al., 2025; Lańcucki et al., 2025; Staniszewski & Lańcucki, 2026; Zandieh et al., 2026). Sparse attention comes in many variants. Big Bird (Zaheer et al., 2020) uses fixed attention sparsity patterns. The heavy-hitter oracle (H2O) (Z. et al., 2023) is discussed in Section 3.1.1, extensions include Q-Hitter (Zhang et al., 2024c), SnapKV (Li et al., 2025c), and CAKE (Qin et al., 2025). FlexGen (Sheng et al., 2023) maximizes throughput by using a cache hierarchy. FastGen (Ge et al., 2024) shows how to vote between different cache policies. Other sparse attention techniques include (Han et al., 2024; Xiao et al., 2024; Tang et al., 2024; Cai et al., 2025; Xiao et al., 2025; Tang et al., 2025; Feng et al., 2025b; Yang et al., 2025b; Wang et al., 2025; Devoto et al., 2025). KVPop (Hauzenberger et al., 2026) learns a cache policy against a future-attention target. qTTT (Bansal et al., 2026) uses a few gradient updates at test time in order to improve inference results. MInference (Jiang et al., 2024), KVPress (`https://github.com/NVIDIA/kvpress`) and ShadowKV (Sun et al., 2025) are open source libraries providing sparse attention methods. SCBench (Li et al., 2025d) provides an empirical analysis.

Optimized scaled dot product attention (SDPA) kernels are essential for fast inference and training, pioneered by FlashAttention (Dao et al., 2023; Shah et al., 2024). FlashInfer (Ye et al., 2025) is optimized for the inference case $1 \ll N_q \ll N_k$ (notation from Section 3.1). FlexAttention (Dong et al., 2025) allows users to specify mask and score modification code.

Our main contribution is on long-context *fine-tuning*. Prior work can be ordered into two groups. Proposals in the first group modify multi-head attention in ways which remedy the difficulties detailed in Section 3.2. Native sparse attention (NSA) (Yuan et al., 2025) bakes a mixture of some sparse attention kernels with different fixed policies into the model architecture. DeepSeek sparse attention (DSA) (etal., 2025) is a refined variant of this idea. Different to *selective* sparse attention, this needs to be used during pre-training already, whose cost is significantly increased. IndexCache (Bai et al., 2026) speeds up DSA somewhat by sharing the indexer between several layers. Also, by fixing the selection policy as part of the model choice, DSA offers considerable less flexibility during post-training. LSA and DSA still require storing the complete KV cache (even though each attention call uses only a part of it). LongGen (Ge et al., 2025) uses static sparsity patterns for the top and bottom third of layers and full attention for the middle third, which speeds up training a bit, but does not reduce its memory requirements. DMC (Nawrot et al., 2024) uses a form of KV cache compression where new information is either appended or accumulated with the last recent slot. They propose heuristics to train a model with this mechanism in place. This approach seems restricted to KV cache updates during generation, it is not clear what is done when a large prompt needs to be processed. Starting from Mamba (Gu & Dao, 2023), there have been several attempts to resurrect LSTMs (Hochreiter & Schmidhuber, 1997), e.g. (Dao & Gu, 2024; Beck et al., 2024). When compared against the long-context transformer SotA, none of them have been competitive enough in order to warrant costly pre-training efforts. YOCO (Sun et al., 2024) changes the transformer architecture, so that a single KV cache block serves all layers. KV buffer scaling with layers is not a major problem in practice, since we can use CPU offloading (see Section A.7.1).

In the second group, KV cache buffers are not compressed, but both storage and computation are distributed across several devices. This can be done with RingAttention (Liu et al., 2024a) or more recent variants (Liu et al., 2025), in what is called context parallelism (CP), or with sequence parallelism (SP) (Li et al., 2023). While Li et al. (2025a) observe that long sequences can be split into chunks, and that the computation graph factorizes along the chunk axis, the factor for the final chunk depends on *all* KV cache buffers of all layers, so cannot be represented on a single device[1] (attempts to sparsify this computation are heuristics, and experiments are done only on rather short sequences of up to 16k tokens). OOMB (Li et al., 2026) shares properties with our work, such as chunk-level processing, activation checkpointing, and efforts to compress KV cache buffers for *autograd*. While for exact KV caches, they run into the same issues as (Li et al., 2025a), their implementation supports NSA and DSA as well. Their way of hiding KV cache buffers from *autograd* requires a number of complex dedicated CUDA kernels, which need to be hardcoded for every sparse attention variant. We manage KV buffer size in *autograd* via delta encoding, which renders our implementation agnos-

[1] In fact, RingAttention's way to compute gradients makes use of this graph structure implicitly.

tic to the KV cache policy. Details are given in Section A.11. LongRoPE (Ding et al., 2024; Shang et al., 2025) combines SP with a search for non-uniform RoPE and several lifting stages. Other work on tuning position encoding, data mix and fine-tuning recipes, but not compressing KV cache, includes (Wu et al., 2024; Zhang et al., 2024b; Gao et al., 2023). LongStraw (Zhou et al., 2026) is a system designed for long context reinforcement learning, with a specific emphasis on sharing prompt graphs and KV caches between different roll-outs. Highly optimized implementations of CP/SP are the state of the art for long-context inference, e.g. vLLM (Kwon et al., 2023), SGLang (Zheng et al., 2024), and fine-tuning, e.g. Nvidia NeMo RL (`https://github.com/NVIDIA-NeMo/RL`), MS-SWIFT (Zhao et al., 2025). In Section 3.3, we comment on reasons why sparse attention methods are less frequently used in practice, and how this could be changed.

3 Long-Context Fine-Tuning

In this section, we introduce sparse attention and key-value caching, providing several improvements to the heavy-hitter oracle (H2O) cache policy (Z. et al., 2023), such as an efficient implementation with dedicated SDPA kernel support. We detail our main contribution: a novel method to fine-tune models with sparse attention in place, using resources comparable to sparse attention inference. While in SotA CP/SP techniques, GPUs need to be used for sharding along the context, they can be used to increase throughput (i.e., larger batch sizes) or to handle larger models in our method.

3.1 Sparse Attention. Key-Value Caching

Multi-head attention (MHA) is the most important mechanism in modern transformer architectures (Vaswani et al., 2018). At its core lies *scaled dot product attention (SDPA)*:

$$\boldsymbol{Y} = \texttt{SDPA}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}), \quad \boldsymbol{Y}, \boldsymbol{Q} \in \mathbb{R}^{(B, H_q, N_q, d_h)}, \ \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{(B, H_k, N_k, d_h)}. \tag{1}$$

Here, $\boldsymbol{Q}$ (queries), $\boldsymbol{K}$ (keys), $\boldsymbol{V}$ (values) are 4D arrays, B is the batch size, d_h the *per-head embedding dimension*, H_q, H_k are numbers of heads, and N_q, N_k are sequence lengths (i.e., the third dimension maps to token positions in the model context). The *model embedding dimension* is $d = H_q \cdot d_h$. Let us first assume that $H_q = H_k$ and drop the first two dimensions. Then:

$$\boldsymbol{Y} = \boldsymbol{M}\boldsymbol{V}, \quad \boldsymbol{M} = \texttt{softmax}\left(\texttt{mask}\left(d_h^{-1/2}\boldsymbol{Q}\boldsymbol{K}^T\right), \texttt{dim} = \texttt{1}\right). \tag{2}$$

$\boldsymbol{Y}$ are weighted combinations of values $\boldsymbol{V}$ with *attention weights* $\boldsymbol{M} \in \mathbb{R}^{(B, H_q, N_q, N_k)}$, `softmax` applies $\boldsymbol{x} \mapsto \exp(\boldsymbol{x})/(\boldsymbol{1}^T \exp(\boldsymbol{x}))$ along rows. `mask` implements causal masking: $(\texttt{mask}(\boldsymbol{X}))_{i,j} = x_{i,j} - \infty \mathrm{I}_{\{P+i<t(j)\}}$, where P and $t(j)$ are token positions (see Section A.3 for details). If arrays are indexed by (b, h, j, k), SDPA operates on (j, k) in the same way for all (b, h), computations are parallelized over batch and head positions.

Inference in transformers switches between prompt processing and token generation. Generating a token after a prompt of size N requires SDPA with $N_k = N, N_q = 1$, with keys and values of size (B, H_k, N, d_h) in memory, for each of L model layers. Exact transformer inference therefore requires $\mathcal{O}(L \cdot N \cdot BH_k d_h)$ memory for the full *key-value (KV) cache*. Even for moderate context lengths N of several hundred thousands, the KV cache far surpasses the model weights in size and cannot be stored in GPU memory as is.

A large amount of prior work confronts this problem (see also Section 2). In grouped query attention (GQA) (Ainslie et al., 2023), we set $H_q = H_k \cdot q_g$, $q_g > 1$, so that q_g heads map to the same query group, which reduces KV cache size by a factor of q_g. Most relevant to our work is *sparse attention* (or *selective KV caching*; e.g. Z. et al. (2023)), where the KV cache is represented by fixed-size buffers *independent* of the context length of the model. Once all slots are filled, new information overwrites (or *evicts*) existing ones. Formally, the cache (for one model layer) is represented by arrays $\texttt{keys}, \texttt{values} : (B, H_k, N_C, d_h)$ and $\texttt{token_pos} : (B, H_k, N_C)$. Here, N_C is the *cache length*, which is chosen as large as GPU memory permits. For up to N_C tokens, the cache is filled from left to right. After that, the *KV cache policy* $\pi_l(b, h, t) \in \{0, \ldots, N_C - 1\}$ dictates where additional key-value information is written for token position $t \geq N_C$, batch position $b \in \{0, \ldots, B-1\}$ and head (or query group) $h \in \{0, \ldots, H_k - 1\}$. Importantly, π_l can depend on b, h: a token may be in the cache for some batch positions and heads, and not for others. $t(b, h, j) = \texttt{token_pos}[b, h, j]$ lists the token position of what is stored in (b, h, j). We do not require complex

memory layouts and dedicated kernels for this setup, as for example PagedAttention (Kwon et al., 2023) needs. In fact, (2) depends on absolute token positions only via `mask`. This is as sparse as the conventional triangular one, but depends on token positions $t(\cdot) =$ `token_pos`, which through cache evictions becomes non-monotonic in general. Moreover, `torch.gather` and `torch.scatter` provide fast read and write access for these buffers (see Section A.4.1).

With sparse attention, we can run inference for any context width N. First, we process up to the first N_C tokens with a single SDPA call ($N_q = N_k = N_C$), this is known as *prefilling*. The remaining $N - N_C$ tokens are processed in *chunks* of size $S < N_C$, using SDPA calls with $N_q = S, N_k = N_C$ (see Section A.7.1 for details). Token generation uses $N_q = 1, N_k = N_C$. Memory requirements are independent of N. For each chunk, the policy $[\pi_l]$ is used to determine $B \cdot H_k \cdot S$ positions (b, h, j) which are overwritten by the new keys and values. While N_C is chosen as large as memory permits, the choice of S is more subtle. The larger S, the fewer chunks, and less sequential computation results in faster processing. The smaller S, the more fine-grained the cache policy is used, which can lead to better decisions (see also Section 3.3).

3.1.1 Variants of Heavy-Hitter Oracle

The key idea behind the heavy-hitter oracle (H2O) (Z. et al., 2023) is to make use of the attention weights $\boldsymbol{M} = [m_{i,j}]$, a by-product of SDPA (2). Dropping (b, h) for the moment, we have that $\boldsymbol{Y}_{i,:} = \sum_j m_{i,j} \boldsymbol{V}_{j,:}$ and $\sum_j m_{i,j} = 1$. $m_{i,j}$ quantifies how much values $\boldsymbol{V}_{j,:}$ are used to create $\boldsymbol{Y}_{i,:}$. The cumulative sum $\sum_i m_{i,j}$ can be used to score the usefulness of vectors $(\boldsymbol{K}_{j,:}, \boldsymbol{V}_{j,:})$ in the KV cache. Define the H2O score after having processed t tokens as

$$\phi^t_{\text{h2o}}(b, h, j) = \sum_{t(b,h,j) \le s < t} m_{b,h,s,j}, \tag{3}$$

where $t(b, h, j) =$ `token_pos`$[b, h, j]$ is the position represented at (b, h, j) right now. We sum over $s \in \{t(b, h, j), \dots, t\}$ because the slot holds KV information for token position $t(b, h, j)$, which entered the cache only then. The larger $\phi^t_{\text{h2o}}(b, h, j)$, the more valuable this information has been so far. When asked to insert new content for S tokens, for each (b, h), we overwrite these S slots j for which $\phi^t_{\text{h2o}}(b, h, j)$ is smallest.

In this paper, we modify the original H2O policy (Z. et al., 2023) (as provided by their implementation) in several ways. First, their code selects the same cache slots for each batch position b, using the score $\phi^t_{\text{h2o-orig}}(h, j) = \sum_b \phi^t_{\text{h2o}}(b, h, j)$. The rationale for this restriction is unclear, and we implement H2O without it as well. Second, the cumulative H2O score seems to favour entries (b, h, j) which have been in the cache for longer, since more terms in $[0, 1]$ are summed then. We introduce the *normalized H2O score*: $\phi^t_{\text{h2o-norm}}(b, h, j) = (t - t(b, h, j))^{-1} \phi^t_{\text{h2o}}(b, h, j)$.

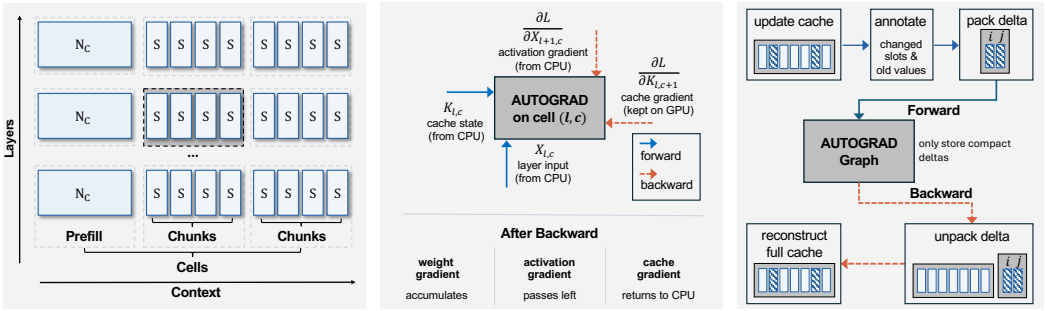
Despite convincing empirical results of H2O, both in (Z. et al., 2023) and Section 4, it is not widely used. This is mostly because current implementations of H2O are much slower than the state of the art. We need summed attention weights $\sum_i m_{b,h,i,j}$ for each (b, h, j), as by-product of SDPA (2), but none of the fast SDPA kernels derived from FlashAttention (Dao et al., 2023) provide them. Current H2O implementations use naive SDPA implementations, which are much too slow in practice. Our implementation contains *Triton* code to return summed attention weights alongside a FlashInfer SDPA kernel (Ye et al., 2025). In Section A.9, we show how FlexAttention (Dong et al., 2025) can be used to this end as well.

3.2 Fine-Tuning for Sparse Attention

How should we train a model which uses sparse attention with some KV cache policy such as H2O? As noted in Section 2, all prior long-context fine-tuning methods either restrict the MHA approximation to a particular form, or use exact MHA with KV cache buffers distributed across several devices (i.e., sequence or context parallelism). However, the choice of KV cache policy, which dictates how the model's short term memory is organized, should influence how the model is best trained. Our results in Section 4 validate this hypothesis. *Sparse attention inference for a model trained with exact MHA and sequence parallelism (which is the SotA) often performs significantly worse than for a model trained with sparse attention and the desired policy in place*.

Fine-tuning for models with sparse attention is difficult, because a huge amount of memory is required. Gradients for loss functions on sequences of length $N \gg N_C$ are computed by reverse

mode automatic differentiation (or error backpropagation, *autograd*). This creates a computation graph during the forward pass, storing arrays needed during the backward pass. With L layers and chunk size S, the training sequence is split into $1 + \lceil (N - N_C)/S \rceil$ chunks, the first (prefill) chunk of length N_C and subsequent chunks of length S. SDPA is called for each layer and chunk, creating at least one node of the size of the KV cache, so we need at least $\mathcal{O}(L \cdot S^{-1}(N - N_C) \cdot N_C \cdot \mathcal{D})$ of memory, where $\mathcal{D} = BH_k d_h$. Assuming $S = aN_C$ for some constant a, this is $\mathcal{O}(L \cdot N \cdot \mathcal{D})$: more than the *full* KV cache would need, and far beyond what is tractable.



(a) Local cells tile the lattice (b) Autograd on one cell (c) Delta encoding

Figure 1: (a) Structure of computation graph as lattice of chunks. The first (prefill) chunk in a row has full length N_C, subsequent chunks have length $S \ll N_C$. Chunks are grouped into cells (see Section A.7.1). (b) `autograd` is run separately on each cell (l, c). This needs checkpointed inputs from bottom and left, and head gradients from top (checkpointed) and right (on GPU). The direction is right to left, and head gradients for layer inputs are written back to CPU. See Section A.7.2 for details and notation. (c) Delta encoding of KV cache buffers during `autograd` on a cell. During the forward pass, deltas of the buffer sequences are extracted, matched in `pack_hook` calls, and written to the computation graph. During the backward pass, the buffer sequences are reconstructed from the deltas and supplied to `unpack_hook`. See Section A.7.3 for details.

Our goal is to bring GPU memory requirements down to levels comparable to what inference needs. First, we avoid differentiation through the KV cache policy (which is often not even possible, and in general not tractable). Along the forward pass, we store all KV cache policy decisions in a *replay log*, containing for each chunk (and each layer l) the tokens processed, and the decisions $\{\pi_l(b, h, t)\}$. Later on, we use *replay caches*, which act like normal KV caches, except that eviction decisions are replayed from the log.

Next, we use activation checkpointing (Herrmann et al., 2019). Different from autograd, there is no computation graph built during the forward pass, only the input tensors for each transformer layer are stored to CPU memory. The backward pass is split into L autograd calls, starting from the top. Head gradients are supplied from the previous layer, inputs are loaded from CPU. Computation graphs on GPU are L times smaller, while forward computations have to be run twice.

While standard activation checkpointing tackles large L, in long-context situations the context width N is the more serious problem, in that not even complete inputs or head gradients for a single layer can be kept in GPU memory (see Section A.7.1). We use activation checkpointing twice, in a nested fashion. To this end, we partition chunks into *cells*: $\{(B, S, d)\} \to (B, kS, d)$, where $k = \lfloor \alpha N_C/S \rfloor$, and $\alpha > 0$ is a hyperparameter which defaults to $\alpha = 1$. The first (prefill) chunk becomes the first cell. As a rule of thumb, we group chunks into cells which occupy about the size of KV cache buffers. The computation graph becomes a *lattice of cells*: rows are layers, columns are cells (i.e., groups of chunks) along the context (Figure 1, left). Our backward pass runs an outer loop over rows, then inner loops over cells in each row. Each inner loop starts with a (non-autograd) forward pass along the context, storing KV cache buffers going into each cell to CPU[2]. Next, autograd is run separately on each cell, starting from the right. Inputs to a cell (layer inputs from the bottom and KV cache buffers from the left) and head gradients from the top are read from CPU, while head gradients from the right stay in GPU memory (Figure 1, middle). Gradients are

[2]Our implementation writes checkpoints to disk once CPU memory is occupied.

accumulated. We reuse the same CPU and GPU buffers for all inner loops. A detailed summary of our method is given in Section A.7.2.

However, the autograd calls still need too much GPU memory. Recall that a cell consists of $k = \lfloor \alpha N_C / S \rfloor$ chunks. Autograd stores KV cache buffers for each chunk, so needs at least $\mathcal{O}(k \cdot N_C \cdot \mathcal{D})$ memory, where $\mathcal{D} = BH_k d_h$. As detailed in Section 3.1, k must be sizable to allow cache policies to make good decisions. We now show how to cut GPU memory by a factor of k, to $\mathcal{O}(N_C \cdot \mathcal{D})$ per autograd call, comparable to what is needed during inference alone. Consider KV cache buffers $(\texttt{keys}, \texttt{values})$, $(\texttt{keys}', \texttt{values}')$ for neighboring chunks. Their size is (B, H_k, N_C, d_h), but they only differ in $S \cdot \mathcal{D}$ values, because a chunk consists of S tokens only. The relationship is simple:

$$\texttt{keys}' = \texttt{scatter}(\texttt{keys}, \texttt{index}, \texttt{key_new}), \ \texttt{values}' = \texttt{scatter}(\texttt{values}, \texttt{index}, \texttt{value_new}).$$

Here, `key_new`, `value_new` are KV vectors for new tokens with sizes (B, H_q, S, d_h), `index` is based on the cache policy π_l, determining which slots are overwritten, and `scatter`, `gather` are linear `torch` operators defined in Section A.4.1. This is a linear recurrence, which is easily inverted:

$$\texttt{keys} = \texttt{scatter}(\texttt{keys}', \texttt{index}, \texttt{delta_key}), \quad \texttt{delta_key} = \texttt{gather}(\texttt{keys}, \texttt{index}), \quad (4)$$

and the same for `values`. Instead of storing `keys`, `values` for each chunk in the compute graph, it suffices to store[3] `delta_key`, `delta_value` (Figure 1, right). Since memory requirements of autograd calls are dominated by the KV cache buffers, they are reduced by a factor of k.

While the linear recurrence relation between neighboring cache buffers is simple, implementing it in the context of *PyTorch autograd* is not. We use a mechanism called *autograd saved tensors hooks*[4], originally intended to implement activation checkpointing by CPU offloading, which can be adapted to ours needs. In a nutshell, we use the *PyTorch* mechanism to store $(\texttt{delta_key}, \texttt{delta_value})$ in the autograd graph in place of $(\texttt{keys}, \texttt{values})$ (called "packing"), reconstructing the latter from the former and subsequent $(\texttt{keys}', \texttt{values}')$ during the backward pass over chunks (called "unpacking"). The key difficulty is the non-selectiveness of the mechanism: it provides a `pack_hook` function called for all arrays *PyTorch autograd* decides to place into its graph. There is no way to tag tensors in the forward code, so they can be recognized as `pack_hook` arguments. Our solution is to create *annotations* alongside the forward pass code, storing $(\texttt{index}, \texttt{delta_key})$ for `keys`, $(\texttt{index}, \texttt{delta_value})$ for `values`. In a $\texttt{pack_hook}(\boldsymbol{x})$ call, we relate $\boldsymbol{x}$ to current annotations: $\boldsymbol{x}$ matches $(\texttt{index}, \texttt{delta_key})$ (say) if $\texttt{gather}(\boldsymbol{x}, \texttt{index}) = \texttt{delta_key}$. A match leads to $\texttt{pack_hook}(\boldsymbol{x})$ returning a reference to $(\texttt{index}, \texttt{delta_key})$, which is removed from the annotation list. Note that failing to match an annotation does not lead to errors, but at most to a bit more memory being used. More details are given in Section A.7.3.

3.3 Sparse Attention and Sequence Parallelism

While with context or sequence parallelism, the context width is strictly limited by the number of GPUs available, sparse attention inference can be run for any context width on moderate GPU resources. Redundancies, which exist in multi-head attention, can be exploited by way of cache compression. Even if many GPUs are available, sparse attention allows us to increase batch size by way of distributed data parallel (DDP). A large number of sparse attention variants have been proposed (see Section 2). *Why is it then that sparse attention methods are hardly used in SotA inference libraries such as vLLM* (Kwon et al., 2023)? The short answer is that latency is significantly higher with existing sparse attention implementations. Further comments are in Section A.10.

Some of the gap is due to less low level implementation support for sparse attention. Highly optimized SDPA kernels are vital for fast inference (Dao et al., 2023; Dong et al., 2025; Ye et al., 2025). However, as noted in Section 3.1.1, existing kernels do not cater for sparse attention inference (see details in Section A.3). Other reasons for the gap are more difficult to address. A long sequence is split into chunks, the first (prefill) chunk of length N_C (cache length), subsequent chunks of length S. Larger S means fewer chunks and lower latency. But KV cache policies can make useful eviction decisions only if S is much less than N_C (in our experiments in Section 4, we use $N_C = 32768$ and $S \in \{1024, 2048\}$). For the extreme choice $S = N_C$, the whole cache is overwritten by new

[3] This is a form of *delta encoding* (`https://en.wikipedia.org/wiki/Delta_encoding`).

[4] `https://docs.pytorch.org/tutorials/intermediate/autograd_saved_tensors_hooks_tutorial.html`

content for every chunk, and the KV cache policy plays no role at all! Information cannot be kept in the cache beyond a chunk if we do not allow for substantial overlap.

For example, suppose we use 8 devices, each supporting a cache length N_C, and the sequence length is $N = 8N_C$. With RingAttention, each device holds N_C slots, and the sequence is processed in 8 sequential chunks. But for sparse attention, we need $1 + 7N_C/S$ chunks, which can be substantially larger. Despite sparse attention supporting an 8 times larger batch size via DDP, inference tends to still be slower than with RingAttention. In future work, we plan to improve sparse attention latency by appropriate kernel fusion. However, the sequential nature of decision making in sparse attention may be an inherent disadvantage over sequence or context parallelism.

Apart from large hardware requirements to even work on long sequences, RingAttention requires $\mathcal{O}(L \cdot D)$ synchronizations between all devices per gradient update, while sparse attention DDP only needs a single gradient averaging reduction. While memory transfer between devices can be run in parallel with computations, this needs double buffering[5] in RingAttention, doubling the GPU memory needed. The peer-to-peer memory transfer is more brittle than DDP used with sparse attention, and robust training code is more difficult to implement. Finally, the "waste by pre-allocation" issues which motivate the fairly complex PagedAttention Kwon et al. (2023), have a simpler solution with sparse inference: KV cache buffers are of a fixed length, there is no need to split[6] them into pages along the sequence axis, and no custom SDPA code is needed.

4 EXPERIMENTS

The key question addressed in our experiments is: *if long context inference uses sparse attention with a particular KV cache policy, how much is gained by* fine-tuning *the model with the same policy in place (using our novel method) over training it with state of the art libraries using sequence or context parallelism*? We run comparisons on the Helmet (Yen et al., 2025) benchmarks, with context widths of 64k and 128k, covering a range of cache policies:

- `lastrec` (lr): $\pi(b, h, t) = t\mathrm{I}_{\{t<N_C\}} + (\mathrm{mod}(t - N_C, N_C - \beta) + \beta)\mathrm{I}_{\{t\geq N_C\}}$, where $\beta \in [0, N_C)$. Keeps the last recent $N_C - \beta$ and first β tokens in the cache. It is important to choose $\beta > 0$ as a default "attention sink" (Xiao et al., 2024). Our default is $\beta = \min(16, \lceil N_C/8 \rceil)$.
- `smart_lastrec` (slr): Variant of `lastrec`, where the number β of initial tokens is chosen dependent on content (see Section A.5.1 for details). A simple version of this heuristic appeared in (Han et al., 2024).
- `h2o` (h2o), `h2o_norm` ($\mathrm{h2o}^{\mathrm{no}}$), `h2o_orig` ($\mathrm{h2o}^{\mathrm{or}}$): Variants of H2O (Z. et al., 2023) (see Section 3.1.1 for details). `h2o_orig` is equivalent to their code released, where $\pi(b, h, t)$ does not depend on batch position b.

We train a `Qwen3-4B-Instruct-2507`[7] model (Yang et al., 2025a), using `AdamW` (Loshchilov & Hutter, 2017) with base learning rate 0.0005 for up to 5 epochs. We train LoRA weights only (Hu et al., 2022) (rank $r = 16$, $\alpha = 16$, on all linear blocks). We run on four Nvidia A100 40 GB devices with a per-device batch size of 2, using RoPE (Su et al., 2024) and YaRN (Peng et al., 2024) for position encoding. Fine-tuning is done in different ways:

- Sequence parallelism (sp): We use MS-SWIFT (Zhao et al., 2025) for fine-tuning with DeepSpeed ZeRO-3 offload, FlashAttention, and Liger kernels enabled, running on four Nvidia A100 40GB GPUs. We use a per-device batch size of 2 and sequential gradient accumulation (effective batch size 8)[8].

[5] On each device, one set of `keys`, `values` are read for computation and transfer, another set is written to (Liu et al., 2024a).

[6] If we frequently encounter batches shorter than the cache length, we can build up buffers in chunks. But most modern agentic AI applications come with long prompts anyway.

[7] This checkpoint has been post-trained to process long contexts. We still need to adjust the RoPE position encoding, which we do with YaRN.

[8] For 64k (128k), 2 (4) devices cover the context, so we need 2 (4) sequential gradient accumulation steps.

	64k datasets								128k datasets							
	nq		tri_qa		hot_qa		pop_qa		nq		tri_qa		hot_qa		pop_qa	
	us	sp	us	sp	us	sp	us	sp	us	sp	us	sp	us	sp	us	sp
exact	-	48.8	-	78.7	-	53.3	-	59.3	-	52.5	-	55.2	-	47.7	-	45.7
lr_{2k}	32.7	14.7	56.8	48.5	43.3	5.7	44.8	28.5	21.2	9.0	55.2	23.5	27.7	5.0	31.0	37.3
slr_{2k}	32.3	15.7	58.2	43.7	43.3	6.3	41.7	24.7	19.5	9.0	50.7	20.2	23.3	7.3	30.8	37.3
$\mathrm{h2o}_{2k}$	41.0	18.2	68.2	54.2	35.7	8.7	47.3	41.2	16.3	7.2	39.5	26.5	17.0	6.0	35.8	35.0
$\mathrm{h2o}^{\mathrm{no}}_{2k}$	45.2	16.2	72.8	56.7	46.0	7.7	48.0	41.3	38.5	14.5	67.2	29.0	30.7	10.0	45.0	41.2
$\mathrm{h2o}^{\mathrm{or}}_{2k}$	40.8	18.5	70.7	54.7	42.7	9.0	49.2	42.0	37.0	14.7	70.2	39.0	22.0	6.3	48.0	46.7
lr_{1k}	32.3	15.5	64.5	45.8	41.0	6.0	43.5	26.2	20.8	8.5	56.7	20.0	26.7	5.7	32.5	38.0
slr_{1k}	30.5	14.0	66.3	49.5	45.0	6.0	39.5	25.2	22.0	9.7	45.3	19.8	22.0	5.0	27.8	36.8
$\mathrm{h2o}_{1k}$	43.5	18.0	67.2	53.5	47.3	6.3	43.2	39.7	14.8	7.2	41.2	29.7	13.7	7.7	31.7	39.0
$\mathrm{h2o}^{\mathrm{no}}_{1k}$	42.8	16.2	63.2	53.5	47.3	9.3	48.8	37.8	35.5	12.3	56.2	30.2	30.6	11.3	42.5	39.7
$\mathrm{h2o}^{\mathrm{or}}_{1k}$	43.2	17.3	67.7	56.7	41.3	6.7	47.2	41.2	38.0	16.5	61.5	40.8	24.7	5.7	47.2	44.0

	128k datasets															
	trec_c		trec_f		nlu		bank77		clc150		inf_qa		inf_mc		json_kv	
	us	sp	us	sp	us	sp	us	sp	us	sp	us	sp	us	sp	us	sp
exact	-	94.6	-	85.4	-	90.6	-	91.4	-	69.2	-	44.2	-	64.0	-	100
slr_{1k}	95.8	40.0	84.6	19.0	89.6	20.4	91.6	17.8	97.4	12.0	25.3	2.2	41.0	23.0	48.0	13.0
$\mathrm{h2o}^{\mathrm{no}}_{1k}$	96.0	33.4	90.6	13.2	86.6	18.0	91.0	19.8	96.6	19.8	32.1	2.6	53.0	26.0	52.0	1.0
$\mathrm{h2o}^{\mathrm{or}}_{1k}$	96.2	51.6	77.2	76.8	80.6	11.4	90.2	65.2	95.2	59.8	34.7	4.9	50.0	33.0	1.0	2.0

Table 1: Results for long-context inference on Helmet datasets. In the upper part, we use 4 datasets of sequence lengths 64k, 128k each (cols), run 5 KV cache policies and 2 chunk sizes $2048 = 2k, 1024 = 1k$ (rows). The *MatchFirstWordOrPhrase* metric is used with these QA datasets. In the lower part, we use 8 additional datasets and run 3 KV cache policies (chunk size 1024). Different metrics are used, depending on the dataset (see Table 2). The *exact* rows are for exact inference (sequence parallelism). Columns *us* are for models trained using our novel method with the same cache policy in place, columns *sp* are for models trained with sequence parallelism.

- Our method (us): We use a cache length $N_c = 32768$, chunk size $S \in \{1024, 2048\}$, and chunks per cell multiplier $\alpha = 1$ for $S = 2048$, $\alpha = 0.75$ for $S = 1024$ (see also Section A.7.1). KV cache buffers are quantized to 8 bits using `torchao`. We run distributed data parallel optimization on 4 devices to obtain an effective batch size of 8. We evaluate the model on a heldout validation set every 10 gradient steps (5 for `pop_qa`), and choose the checkpoint with the lowest validation loss for testing.

In Table 1, we provide results on 4 Helmet datasets (`nq, trivia_qa, hotpot_qa, pop_qa`; length 64k and 128k) (Yen et al., 2025) and 10 different cache setups (5 policies, 2 chunk sizes) in the upper part, and results for further 8 Helmet datasets (`trec_coarse, trec_fine, nlu, banking77, clinc150, inf_qa, inf_mc, json_kv`; length 128k) and 3 cache policies in the lower part. Respective results for the base checkpoint (no fine-tuning) are given in Table 3 in the Appendix. Details about Helmet datasets and metrics are given in Section A.6.1.

These results consistently confirm our hypothesis: *us* outperforms *sp* across the board, with the possible exception of `pop_qa`. If inference is to be done with sparse attention, it is a poor idea to fine-tune a model with SotA sequence parallelism, even if ample GPU resources are available. A closer look at generated samples (which can be up to 128 tokens) reveals a major failure mode of *sp* (see Section A.8.3): *its outputs are far too long and contain mostly random nonsense*. For several Helmet datasets in the lower part, targets are single numerical values, and the *Accuracy* metric (see Section A.6.1) compares this to the most frequently occurring number in the output. Poor results are due to the output for *sp* often containing many numbers. In contrast, *us* learns how

to stop properly and usually outputs a single number. The same failure mode dominates outcomes in the upper part. While targets in these QA datasets sometimes consist of multiple words, the *MatchFirstWordOrPhrase* metric scores an exact match against the first word or sentence, which often fails for *sp*: not only is random nonsense appended, but it often comes up first as well. Finally, *sp* fails for `json_kv` as well, despite this using the more lenient *SubEM* metric. Targets are UUIDs of length 32 tokens. While variants of *us* identify them about half the time, they are hardly ever contained in the outputs of *sp*. In Section A.8.3, we quantify the failure mode in Table 6. While outputs for *us* are close in length to true targets, they are longer by large factors for *sp*, which often (but not always) span the full 128 tokens (despite true targets being much shorter). We also provide randomly chosen examples for outputs there.

Target conciseness is a clear signal in the training data, expressed also by the *sp* checkpoints if exact inference is used with them (see Table 7). We should not be surprised by *sp* failing. When training with sequence parallelism (SP), each token can attend to any earlier one. This property is cut during inference, when most KV information is evicted at some point according to a logic which SP was never aware of. *Models should be trained under the same conditions and restrictions which govern inference later on*. Our new method allows practitioners to do that even on a low budget, no matter what KV cache policy they like to use during inference.

In terms of differences between cache policies, H2O variants outperform `lastrec` and `smart_lastrec`, while `h2o` tends to do worse than the variants `h2o_norm`, `h2o_orig`. Counter-intuitively, chunk size $S = 2048$ works as well as $S = 1024$, while running significantly faster (see Table 5). Despite its independence from batch position, `h2o_orig` does as well as `h2o_norm`, except it fails in a manner similar to *sp* on `json_kv` (see also Table 6).

5 Conclusions

We showed how transformer language models with sparse attention can be fine-tuned on a moderate hardware budget (e.g., a single Nvidia A100 GPU with 40 GB RAM). Our method works for any KV cache selection or compression policy and allows the model to co-adapt with the policy, often outperforming models trained with exact attention (sequence parallelism). We also provide a much more efficient implementation of H2O sparse attention (the leading policy in our experiments) with dedicated scaled dot product attention (SDPA) kernel support. By simplifying KV cache structure and clarifying the requirements on SDPA, we hope to draw the fast inference community's attention to sparse attention (see Section A.3), which despite its major potential for post-training specialization via cache selection or compression policy design does not currently play a significant role in long-context inference or fine-tuning practice.

In future work, we will combine context parallelism with sparse attention. We are also considering kernel fusion ideas in order to narrow the latency gap further. An important direction will be multi-stream asynchronous implementations which allow for on-the-fly CPU offloading (Yuan et al., 2026). We believe that once the host memory of a system can be used without much synchronization overhead and less double buffering, many current difficulties with KV caching will be much diminished. Finally, we hope that the open source library to be released with this publication will make it easier to use, compare and extend sparse attention policies, work on which so far is somewhat cluttered when it comes to implementations.

References

J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, Lebrón, and F. Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 4895–4901, 2023.

Y. Bai, Q. Dong, T. Jiang, X. Lv, Z. Du, A. Zeng, J. Tang, and J. Li. IndexCache: Accelerating sparse attention via cross-layer index reuse. Technical Report arXiv:2603.12201 [cs.CL], 2026.

R. Bansal, A. Zhang, R. Tiwari, L. Madaan, S. Duvvuri, F. Devvrit, D. Brandfonbrener, D. Alvarez-Melis, P. Bhargava, M. Kale, and S.e Jelassi. Let's (not) just put things in context: Test-time training for long-context LLMs. In *Int. Conf. Learning Representations*, 2026.

M. Beck, K. Pöppel, M. Spanring, A. Auer, O. Prudnikova, M. Kopp, G. Klambauer, J. Brandstetter, and S. Hochreiter. xLSTM: Extended long short-term memory. In Globerson et al. (2024).

D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.). *Advances in Neural Information Processing Systems 38*, 2025. Curran Associates.

W. Brandon, M. Mishra, A. Nrusimha, R. Panda, and J. Ragan-Kelley. Reducing transformer key-value cache size with cross-layer attention. In Globerson et al. (2024).

Z. Cai, Y. Zhang, B. Gao, Y. Liu, Y. Li, T. Liu, K. Lu, W. Xiong, Y. Dong, J. Hu, and W. Xiao. PyramidKV: Dynamic KV cache compression based on pyramidal information funneling. In *Conference on Language Modeling*, 2025.

T. Dao and A. Gu. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. In Salakhutdinov et al. (2024), pp. 10041–10071.

T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In Oh et al. (2023).

A. Devoto, M. Jeblick, and S. Jégou. Expected attention: KV cache compression by estimating attention from future queries distribution. Technical Report arXiv:2510.00636 [cs.AI], 2025.

Y. Ding, L. Zhang, C. Zhang, Y. Xu, N. Shang, J. Xu, F. Yang, and M. Yang. LongRoPE: Extending LLM context window beyond 2 million tokens. In Salakhutdinov et al. (2024), pp. 11091–11104.

J. Dong, B. Feng, D. Guessous, Y. Liang, and H. He. FlexAttention: a programming model for generating optimized attention kernels. In *Proceedings of the 8th MLSys Conference*, pp. 381–394, 2025.

DeepSeek-AI etal. DeepSeek-V2: A strong, economical, and efficient mixture-of-experts language model. Technical Report arXiv:2405.04434 [cs.CL], 2024.

DeepSeek-AI etal. DeepSeek-V3.2: Pushing the frontier of open large language models. Technical Report arXiv:2512.02556 [cs.CL], 2025.

J. Feng, S. Huang, X. Qu, G. Zhang, Y. Qin, B. Zhong, C. Jiang, J. Chi, and W. Zhong. ReTool: Reinforcement learning for strategic tool use in LLMs. Technical Report arXiv:2504.11536 [cs.CL], 2025a.

Y. Feng, J. Lv, Y. Cao, X. Xie, and S. Zhou. Ada-KV: Optimizing KV cache eviction by adaptive budget allocation for efficient LLM inference. In Belgrave et al. (2025).

T. Gao, A. Wettig, H. Yen, and D. Chen. How to train long-context language models (effectively). In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 2391–2404, 2023.

S. Ge, Y. Zhang, L. Liu, M. Zhang, J. Han, and J. Gao. Model tells you what to discard: Adaptive KV cache compression for LLMs. In *Int. Conf. Learning Representations*, 2024.

S. Ge, X. Lin, Y. Zhang, J. Han, and H. Peng. A little goes a long way: Efficient long context training and inference with partial contexts. In *Int. Conf. Learning Representations*, 2025.

A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.). *Advances in Neural Information Processing Systems 37*, 2024. Curran Associates.

A. Gu and T. Dao. Mamba: Linear-time sequence modeling with selective state spaces. Technical Report arXiv:2312.00752 [cs.LG], 2023.

C. Han, Q. Wang, H. Peng, W. Xiong, Y. Chen, H. Ji1, and S. Wang. LM-Infinite: Zero-shot extreme length generalization for large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pp. 3991–4008, 2024.

L. Hauzenberger, N. Schmidinger, A. Hartl, D. Stap, T. Schmied, Böck, S. Klambauer, and S. Hochreiter. KVpop – key-value cache compression with predictive online pruning. Technical Report arXiv:2607.05061 [cs.LG], 2026.

J. Herrmann, O. Beaumont, L. Eyraud-Dubois, J. Hermann, A. Joly, and A. Shilova. Optimal checkpointing for heterogeneous chains: How to train deep neural networks with limited memory. Technical Report arXiv:1911.13214 [cs.LG], 2019.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

C. Hooper, S. Kim, H. Mohammadzadeh, M. Mahoney, S. Shao, K. Keutzer, and A. Gholami. KVQuant: Towards 10 million context length LLM inference with KV cache quantization. In Globerson et al. (2024).

E. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. LoRA: Low-rank adaptation of large language models. In *Int. Conf. Learning Representations*, 2022.

H. Jiang, Y. Li, C. Zhang, Q. Wu, X. Luo, S. Ahn, Z. Han, A. Abdi, D. Li, C. Lin, Y. Yang, and L. Qiu. MInference 1.0: Accelerating pre-filling for long-context LLMs via dynamic sparse attention. In Globerson et al. (2024).

W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. Yu, J. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

A. Lańcucki, K. Staniszewski, P. Nawrot, and E. Ponti. Inference-time hyper-scaling with kv cache compression. In Belgrave et al. (2025).

S. Li, F. Xue, C. Baranwal, Y. Li, and Y. You. Sequence parallelism: Long sequence training from system perspective. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 7376–7399, 2023.

W. Li, Y. Zhang, G. Luo, D. Yu, and R. Ji. Training long-context LLMs efficiently via chunk-wise optimization. In *Findings of the Association for Computational Linguistics (ACL)*, pp. 2691–2700, 2025a.

W. Li, D. Yu, G. Luo, Y. Zhang, Y. Wu, J. Liu, Z. Gong, Z. Liao, F. Chao, and R. Ji. Out of the memory barrier: A highly memory-efficient training system for LLMs with million-token contexts. In *Int. Conf. Learning Representations*, 2026.

X. Li, Z. Xing, M. Li, L. Qu, H. Zhen, Y. Yao, W. Liu, S. Pan, and M. Yuan. KVTuner: Sensitivity-aware layer-wise mixed-precision KV cache quantization for efficient and nearly lossless LLM inference. In Singh et al. (2025).

Y. Li, Y. Huang, B. Yang, B. Venkitesh, A. Locatelli, H. Ye, T. Cai, P. Lewis, and D. Chen. SnapKV: LLM knows what you are looking for before generation. In Belgrave et al. (2025), pp. 529–536.

Y. Li, H. Jiang, Q. Wu, X. Luo, S. Ahn, C. Zhang, A. Abdi, D. Li, J. Gao, Y. Yang, and L. Qiu. SCBench: A KV cache-centric analysis of long-context methods. In *Int. Conf. Learning Representations*, 2025d.

H. Liu, M. Zaharia, and P. Abbeel. RingAttention with blockwise transformers for near-infinite context. In *Int. Conf. Learning Representations*, 2024a.

Z. Liu, J. Yuan, H. Jin, S. Zhong, Z. Xu, V. Braverman, D. Chen, and X. Hu. KIVI: A tuning-free asymmetric 2bit quantization for KV cache. In Salakhutdinov et al. (2024), pp. 32332–32344.

Z. Liu, S. Wang, S. Cheng, Z. Zhao, K. Wang, X. Zhao, J. Demmel, and Y. You. StarTrail: Concentric ring sequence parallelism for efficient near-infinite-context transformer model training. In Belgrave et al. (2025).

I. Loshchilov and F. Hutter. Decoupled weight decay regularization. Technical Report arXiv:1711.05101 [cs.LG], 2017. URL `https://arxiv.org/abs/1711.05101`.

P. Nawrot, A. Lancucki, M. Chochowski, D. Tarjan, and E. Ponti. Dynamic memory compression: Retrofitting LLMs for accelerated inference. In Salakhutdinov et al. (2024), pp. 37396–37412.

A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.). *Advances in Neural Information Processing Systems 36*, 2023. Curran Associates.

B. Peng, J. Quesnelle, H. Fan, and E. Shippole. YaRN: Efficient context window extension of large language models. In *Int. Conf. Learning Representations*, 2024.

Z. Qin, Y. Cao, M. Lin, W. Hu, S. Fan, K. Cheng, W. Lin, and J. Li. CAKE: Cascading and adaptive KV cache eviction with layer preferences. In *Int. Conf. Learning Representations*, 2025.

R. Salakhutdinov, Z. Kolter, K. Heller, A. Weller, N. Oliver, J. Scarlett, and F. Berkenkamp (eds.). *International Conference on Machine Learning 41*, 2024. Proceedings of Machine Learning Research.

T. Schick, J. Dwivedi-Yu, R. Dessi, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language models can teach themselves to use tools. In Oh et al. (2023).

J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao. FlashAttention-3: Fast and accurate attention with asynchrony and low-precision. In Globerson et al. (2024), pp. 68658–68685.

N. Shang, L. Zhang, S. Wang, G. Zhang, G. Lopez, F. Yang, W. Chen, and M. Yang. LongRoPE2: Near-lossless LLM context window scaling. In Singh et al. (2025).

N. Shazeer. Fast transformer decoding: One write-head is all you need. Technical Report arXiv:1911.02150 [cs.NE], 2019.

Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, D. Fu, Z. Xie, B. Chen, C. Barrett, J. Gonzalez, P. Liang, C. Re, I. Stoica, and C. Zhang. FlexGen: High-throughput generative inference of large language models with a single GPU. In A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett (eds.), *International Conference on Machine Learning 40*, volume 202. Proceedings of Machine Learning Research, 2023.

A. Shutova, V. Malinovskii, V. Egiazarian, D. Kuznedelev, D. Mazur, S. Nikita, I. Ermakov, and D. Alistarh. Cache me if you must: Adaptive key-value quantization for large language models. In Singh et al. (2025).

A. Singh, M. Fazel, D. Hsu, S. Lacoste-Julien, F. Berkenkamp, T. Maharaj, K. Wagstaff, and J. Zhu (eds.). *International Conference on Machine Learning 42*, 2025. Proceedings of Machine Learning Research.

K. Staniszewski and A. Lańcucki. KV cache transform coding for compact storage in LLM inference. In *Int. Conf. Learning Representations*, 2026.

J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568(C), 2024.

H. Sun, L. Chang, W. Bao, S. Zheng, N. Zheng, X. Liu, H. Dong, Y. Chi, and B. Chen. ShadowKV: KV cache in shadows for high-throughput long-context LLM inference. In Singh et al. (2025), pp. 57355–57373.

Y. Sun, L. Dong, Y. Zhu, S. Huang, W. Wang, S. Ma, Q. Zhang, Z. Wang, and F. Wei. You only cache once: Decoder-decoder architectures for language models. In Globerson et al. (2024).

H. Tang, Y. Lin, J. Lin, Q. Han, D. Ke, S. Hong, Y. Yao, and G. Wang. RazorAttention: Efficient KV cache compression through retrieval heads. In *Int. Conf. Learning Representations*, 2025.

J. Tang, Y. Zhao, K. Zhu, G. Xiao, B. Kasikci, and S. Han. Quest: Query-aware sparsity for efficient long-context llm inference. In Salakhutdinov et al. (2024), pp. 47901–47911.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L Kaiser, and I. Polosukhin. Attention is all you need. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 31*, pp. 6000–6010. Curran Associates, 2018.

G. Wang, S. Upasani, C. Wu, D. Gandhi, J. Li, C. Hu, B. Li, and U. Thakker. LLMs know what to drop: Self-attention guided KV cache eviction for efficient long-context inference. In *Int. Conf. Learning Representations*, 2025.

J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, W. Chi, Q. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems 35*. Curran Associates, 2022.

T. Wu, Y. Zhao, and Z. Zheng. An efficient recipe for long context extension via middle-focused positional encoding. In Globerson et al. (2024).

G. Xiao, Y. Tian, B. Chen, S. Han, and M. Lewis. Efficient streaming language models with attention sinks. In *Int. Conf. Learning Representations*, 2024.

G. Xiao, J. Tang, J. Zuo, J. Guo, S. Yang, H. Tang, Y. Fu, and S. Han. DuoAttention: Efficient long-context LLM inference with retrieval and streaming heads. In *Int. Conf. Learning Representations*, 2025.

A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv, C. Zheng, D. Liu, F. Zhou, F. Huang, F. Hu, H. Ge, H. Wei, H. Lin, J. Tang, J. Yang, J. Tu, J. Zhang, J. Yang, J. Yang, J. Zhou, J. Zhou, J. Lin, K. Dang, K. Bao, K. Yang, L. Yu, L. Deng, M. Li, M. Xue, M. Li, P. Zhang, P. Wang, Q. Zhu, R. Men, R. Gao, S. Liu, S. Luo, T. Li, T. Tang, W. Yin, X. Ren, X. Wang, X. Zhang, X. Ren, Y. Fan, Y. Su, Y. Zhang, Y. Zhang, Y. Wan, Y. Liu, Z. Wang, Z. Cui, Z. Zhang, Z. Zhou, and Z. Qiu. Qwen3 technical report. Technical Report arXiv:2505.09388 [cs.CL], 2025a.

S. Yang, J. Guo, H. Tang, Q. Hu, G. Xiao, J. Tang, Y. Lin, Z. Liu, Y. Lu, and S. Han. LServe: Efficient long-sequence LLM serving with unified sparse attention. In *Proceedings of the 8th MLSys Conference*, 2025b.

Z. Ye, L. Chen, R. Lai, W. Lin, Y. Zhang, S. Wang, T. Chen, B. Kasikci, V. Grover, A. Krishnamurthy, and L. Ceze. FlashInfer: Efficient and customizable attention engine for LLM inference serving. Technical Report arXiv:2501.01005 [cs.DC], 2025.

H. Yen, T. Gao, M. Hou, K. Ding, D. Fleischer, P. Izsak, M. Wasserblat, and D. Chen. Helmet: How to evaluate long-context models effectively and thoroughly. In *Int. Conf. Learning Representations*, 2025.

J. Yuan, H. Gao, D. Dai, J. Luo, L. Zhao, Z. Zhang, Z. Xie1, Y. Wei, L. Wang, Z. Xiao, Y. Wang, C. Ruan, M. Zhang, W. Liang, and W. Zeng. Native sparse attention: Hardware-aligned and natively trainable sparse attention. Technical Report arXiv:2502.11089 [cs.CL], 2025.

Z. Yuan, H. Sun, L. Sun, and Y. Ye. MegaTrain: Full precision training of 100b+ parameter large language models on a single GPU. Technical Report arXiv:2604.05091 [cs.CL], 2026.

Zhang Z., Y. Sheng, T. Zhou., T. Chen, L. Zheng, R. Cai, Z. Song, Y. Tian, C. Re, C. Barrett, Z. Wang, and B. Chen. H2O: Heavy-hitter oracle for efficient generative inference of large language models. In Oh et al. (2023), pp. 34661–34710.

M. Zaheer, G. Guruganesh, A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang, and A. Ahmed. Big Bird: Transformers for longer sequences. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems 33*, pp. 17283–17297. Curran Associates, 2020.

A. Zandieh, M. Daliri, M. Hadian, and V. Mirrokni. TurboQuant: Online vector quantization with near-optimal distortion rate. In *Int. Conf. Learning Representations*, 2026.

T. Zhang, J. Yi, Z. Xu, and A. Shrivastava. KV cache is 1 bit per channel: Efficient large language model inference with coupled quantization. In Globerson et al. (2024), pp. 3304–3331.

Z. Zhang, R. Chen, S. Liu, Z. Yao, O. Ruwase, B. Chen, X. Wu, and Z. Wang. Found in the middle: How language models use long contexts better via plug-and-play positional encoding. In Globerson et al. (2024), pp. 60755–60775.

Z. Zhang, S.i Liu, R. Chen, B. Kailkhura, B. Chen, and A. Wang. Q-Hitter: A better token oracle for efficient LLM inference via sparse-quantized KV cache. In P. Gibbons, G. Pekhimenko, and C. De Sa (eds.), *Proceedings of Machine Learning and Systems (MLSys)*, volume 6, pp. 381–394, 2024c.

Y. Zhao, J. Huang, J. Hu, X. Wang, Y. Mao, D. Zhang, Z. Jiang, Z. Wu, B. Ai, A. Wang, W. Zhou, and Y. Chen. SWIFT: A scalable lightweight infrastructure for fine-tuning. In *Proceedings of the 39th Conference on Artificial Intelligence (AAAI)*, pp. 29733–29735, 2025.

L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. Gonzalez, C. Barrett, and Y. Sheng. SGLang: Efficient execution of structured language model programs. In Globerson et al. (2024), pp. 62557–62583.

C. Zhou, K. Liu, Y. Zhou, Q. Qiao, J. Gao, H. Zhang, I. Lu, N. Ho, L. Li, A. Lei, C. Cheng, S. Chiang, Y. Zeng, D. Zhang, R. Yang, K. Chen, A. Chen, P. Ma, W. Zhang, and C. Jin. LongStraw: Long-context RL beyond 2M tokens under a fixed GPU budget. Technical Report arXiv:2607.14952 [cs.LG], 2026.

A Appendix

A.1 AI Use Statement

In this work, we used generative AI tools for writing boilerplate code for experimentation (data preprocessing and loading, scripts to collect results and create result tables), debugging and profiling code (with mixed success), cleaning and reformatting datasets, supporting qualitative and thematic data analysis. We have not used generative AI tools for proposing or refining hypotheses, designing or providing feedback on research methodology or experiments, assisting with translation, interpreting results, suggesting experimental parameters, creation of artifacts, drafting parts of a research paper, transcribing recordings of research material, discovering research topics or identifying gaps, brainstorming, editing a research paper to improve readability, formatting references, suggesting a structure for a research paper, or proposing a title or keywords for a research paper. Moreover, generating synthetic data sets, helping to develop theoretical models or conceptual frameworks, formulating mathematical claims, providing critical ingredients for proving mathematical claims, or assisting in the writing of proofs are not applicable to this work (and we would not use AI for them anyway, except the first).

Additionally, we used generative AI tools for creating a figure and identifying relevant literature. We have reviewed all AI-assisted work (mostly code). We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

A.2 Reproducibility Statement

Experiments done for this paper are in principle fully reproducible. The code we used is either already open source, or will be open sourced along with this publication (it is held back for reasons of author anonymity only). An anonymous snapshot of our code can be accessed here: `https://anonymous.4open.science/r/review-artifact-2027-5850/README.md`.

The reader interested in reproduction will need at least one AWS P4 instance (or equivalent setup, containing 8 Nvidia A100 GPUs with 40 GB of RAM) as well as some patience, since our experiments require fine-tuning a 4B parameters checkpoint under hundreds of different conditions, on a range of different datasets with token lengths between 64k and 128k. All scripts we used for collecting results, creating tables, or processing data, are included with the code. Experimental protocols can be obtained on request. We are not authorized to publish the fine-tuned checkpoints.

A.3 SDPA KERNELS FOR SPARSE ATTENTION

Here, we list some ideas for SDPA kernel developers to better support sparse attention. First, we consider causal masking for sparse attention. In standard MHA (the "training case"), $(\texttt{mask}(\boldsymbol{X}))_{i,j} = x_{i,j} - \infty \mathrm{I}_{\{i<j\}}$. For sparse attention, KV information is stored in cache buffers in an ordering given by token positions $t(b,h,j)$, and the causal mask is given by $(b,h,i,j) \mapsto (-\infty)\mathrm{I}_{\{P+i<t(b,h,j)\}}$, where P is the number of tokens processed before the current MHA call (so the new information is for tokens $\{P,\dots,P+N_q-1\}$). The new key-value information has been written into the cache already, so that $\{P,\dots,P+N_q-1\}$ is part of $\{t(b,h,j)\}$ for each (b,h).

Unfortunately, none of the fast SDPA kernel implementations we know of support such variants of causal masking in an implicitly[9] defined way. In our implementation, we sort the token positions and reorder keys and values according to this index, separate for each (b,h), after which we can use standard causal masking, where queries are right-aligned with keys and values. This needs extra computation and memory which could be saved with better SDPA kernel support. Based on our experience, the following simple extensions of fast SDPA kernel libraries could make a major difference for sparse attention:

- Return summed attention weights $\sum_i m_{b,h,i,j}$ (see Section 3.1.1), an array of size (B, H_q, N_k), based on attention weights which are computed anyway. This allows for H2O and related scores to be computed, driving advanced KV cache policies. It should be a straightforward add-on, see Section A.9.
- Allow for implicitly defined causal masks of the form $(b,h,i,j) \mapsto (-\infty)\mathrm{I}_{\{P+i<t(b,h,j)\}}$, where $t(\cdot)$ is an (B, H_k, N_k) integer array as additional input to the operator. While FlexAttention (Dong et al., 2025) supports custom mask patterns, they need to be written in terms of scalar index variables, so cannot have a 3D array as input to the compute graph.

A.4 NOTATION. DEFINITIONS

Here, we add some details missing in the main text.

A.4.1 BUFFER READ/WRITE ACCESS BY `torch.scatter`, `torch.gather`

These linear operations are defined in `https://docs.pytorch.org/docs/2.12/generated/torch.Tensor.scatter_.html`. `scatter_` assigns values to entries in certain positions, `gather` extracts values of entries at certain positions. For reference, $\texttt{arr.scatter_}(\texttt{dim}, \texttt{index}, \texttt{src})$ requires $\texttt{arr.ndim} == \texttt{index.ndim} == \texttt{src.ndim}$ and $\texttt{index.shape} == \texttt{src.shape}$, whereas `arr.shape` can differ on position `dim`. Say that $\texttt{arr.ndim} == 3$. Then:

$$\begin{aligned}
\texttt{arr}[\texttt{index}[\texttt{i},\texttt{j},\texttt{k}],\texttt{j},\texttt{k}] &= \texttt{src}[\texttt{i},\texttt{j},\texttt{k}] \quad | \, dim == 0 \\
\texttt{arr}[\texttt{i},\texttt{index}[\texttt{i},\texttt{j},\texttt{k}],\texttt{k}] &= \texttt{src}[\texttt{i},\texttt{j},\texttt{k}] \quad | \, dim == 1 \\
\texttt{arr}[\texttt{i},\texttt{j},\texttt{index}[\texttt{i},\texttt{j},\texttt{k}]] &= \texttt{src}[\texttt{i},\texttt{j},\texttt{k}] \quad | \, dim == 2
\end{aligned}$$

Also, if $\texttt{res} = \texttt{arr.gather}(\texttt{dim}, \texttt{index})$, then:

$$\begin{aligned}
\texttt{res}[\texttt{i},\texttt{j},\texttt{k}] &= \texttt{arr}[\texttt{index}[\texttt{i},\texttt{j},\texttt{k}],\texttt{j},\texttt{k}] \quad | \, dim == 0 \\
\texttt{res}[\texttt{i},\texttt{j},\texttt{k}] &= \texttt{arr}[\texttt{i},\texttt{index}[\texttt{i},\texttt{j},\texttt{k}],\texttt{k}] \quad | \, dim == 1 \\
\texttt{res}[\texttt{i},\texttt{j},\texttt{k}] &= \texttt{arr}[\texttt{i},\texttt{j},\texttt{index}[\texttt{i},\texttt{j},\texttt{k}]] \quad | \, dim == 2
\end{aligned}$$

In our use case, we apply these operations to 4D arrays with $\texttt{dim} = 2$, so that $\texttt{arr}, \texttt{src}$ are 4D, but `index` is 3D. This is done by broadcasting `index` along the final axis: $\texttt{index.unsqueeze}(-1).\texttt{extend}(-1,-1,-1,\texttt{d}_\texttt{h})$.

A.5 KEY-VALUE CACHE POLICIES

In this section, we present additional details about KV cache policies used in our experiments.

[9] We cannot pass an explicit mask matrix, which would be huge and defy the purpose of fast SDPA.

A.5.1 POLICY `smart_lastrec` (SLR)

Recall the simple `lastrec` (lr) policy from Section 4, which keeps the last recent $N_C - \beta$ and first β tokens in the cache. One drawback of this policy is that β is fixed, while prompts often start with initial important control information of *variable* length. Our `smart_lastrec` policy is defined in terms of a regular expression for the end of this task control prefix, as well as some maximum prefix length $M_{\text{prefix}} < N_C$. When processing the first (prefill) chunk, we search for the first match (separately for each batch position b). If this results in a prefix length $M(b) \leq M_{\text{prefix}}$, this is used, otherwise $M(b) = M_{\text{prefix}}$. For subsequent chunks and token positions $t \geq N_C$, the policy is $\pi(b, h, t) = M(b) + \text{mod}(t - N_C, N_C - M(b))$.

We also implemented a generalization where a *range* $[M_0(b), M_1(b))$ is protected from eviction.[10] Here, $M_0(b)$ is chosen as the position of the first non-padding token, and $M_1(b)$ is chosen as above. The idea is that initial padding tokens do not carry information and should not be attended to, so they can be evicted as soon as the cache is full. In this variant, if all tokens in the prefill chunk are padding for some b, the search for $[M_0(b), M_1(b))$ is shifted to subsequent chunks. The prefix case above is obtained with $M_0(b) = 0$. Surprisingly, in our experiments, the general variant did not improve over the prefix variant,[11] so that `smart_lastrec` in Section 4 is the prefix variant throughout.

A.6 LONG-CONTEXT BENCHMARKS

A.6.1 HELMET

Our training and evaluation suite is derived from Helmet (Yen et al., 2025), a benchmark designed for inference-time evaluation of long-context language models. Helmet covers five capability categories across five context-length scales (8k to 128k tokens). Helmet provides only a small number of instances per task. For supervised fine-tuning, we adapted it as follows.

Instance Construction and Task Scope. We reconstruct each task from its upstream source data, following the original Helmet logic for forming contexts and controlling sequence length. We focus on the 64k and 128k context-length settings and include 12 tasks spanning five capability categories. Table 2 provides a summary of each task.

Split Separation. For each task we produce two non-overlapping partitions. The instances used in the original Helmet evaluation are reserved as a held-out *evaluation (test) set*. All remaining instances are collected into a *development set* used for training. For RAG tasks, we sample a single depth (difficulty) variant per query in the development set to prevent the model from memorising the same question paired with multiple distractor configurations. For InfiniteBench QA/MC, we remove in-context demonstrations from development instances because the demonstrations are drawn from the same small pool and would otherwise create data leakage during training.

Evaluation Metrics. Our evaluation metrics are mostly taken from the code coming with Helmet (`https://github.com/princeton-nlp/HELMET/blob/main/utils.py`). Here, the output is a string generated by the model, the target is a string or a list of strings:

- MatchFirstWordOrPhrase: The `target` is a list of strings. Strings (both `output` and `target` entries) are normalized by stripping off whitespace and converting letters to lower-case. We build the set `extracts` by appending the first word in `output` if at least one `target` entry is a single word, and appending the first sentence in `output` if at least one `target` entry consists of more than one word. The metric is 1 if $\texttt{t} = \texttt{e}$ for some $\texttt{t} \in \texttt{target}$, $\texttt{e} \in \texttt{extracts}$, and 0 otherwise.
- Accuracy: The target is a numerical value. We extract all numerical values from the output and find the value which occurs most often (with ties, the value is chosen which appears first). We then use exact match between this value and the target.

[10] We allow for $M_1(b) < M_0(b)$, in which case the protected area is $[0, M_1(b)) \cup [M_0(b), N_C)$.

[11] This needs further analysis. Maybe some initial padding tokens are used as so-called "attention sinks" (Xiao et al., 2024).

Category	ID	Source	Metric	Dev	Eval
RAG	nq	Natural Questions	MatchFirstWordOrPhrase	893	600
	trivia_qa	TriviaQA	MatchFirstWordOrPhrase	876	600
	pop_qa	PopQA	MatchFirstWordOrPhrase	192	600
	hotpot_qa	HotpotQA	MatchFirstWordOrPhrase	787	300
Many-shot ICL	trec_coarse	TREC	Accuracy	1000	500
	trec_fine	TREC	Accuracy	1000	500
	nlu	SNIPS NLU	Accuracy	2094	500
	banking77	BANKING77	Accuracy	2580	500
	clinc150	CLINC150	Accuracy	2600	500
Long-doc QA	inf_qa	InfiniteBench QA	ROUGE-F1	251	100
	inf_mc	InfiniteBench MC	Accuracy	129	100
Synthetic Recall	json_kv	JSON-KV	SubEM	500	100

Table 2: Overview of the 10 Helmet tasks. *Dev* and *Eval* denote the number of instances in the training and evaluation partitions, respectively, at a single context-length setting.

- SubEM: Depending on the dataset, the target can be a list of strings. We normalize the output (strip whitespace, quotes, and common phrases such as "Answer:"), map output and target(s) to lower-case. The value is 1 if at least one of the targets is a substring in the output, 0 otherwise.
- ROUGE-F1: The target is a string. We compute ROUGE-N precision/recall/F1 between output (after normalization as in SubEM) and target.

Here, *MatchFirstWordOrPhrase* is our choice and replaces *SubEM* as Helmet choice for `nq`, `trivia_qa`, `pop_qa`, `hotpot_qa`. This replacement is motivated in Section A.8.3 and Section A.8.4. In essence, *SubEM* is not actionable, in that it does not prescribe a way to extract the desired substring from the output. It is also easy to game by simply generating many tokens.

In what follows, we provide details for the different tasks.

Retrieval-augmented Generation (RAG). Each instance consists of a natural-language question, one or more gold passages, and a pool of hard-negative distractors. The context is formed by inserting gold passages at a random depth among the distractors, and the whole context is truncated to the target length. *Natural Questions* (NQ) uses real Google search queries paired with Wikipedia passages. *TriviaQA* uses trivia questions authored with independently collected evidence. *PopQA* focuses on long-tail, entity-centric questions generated from Wikidata triples; we filter the evaluation set to queries whose subject entities fall below a popularity threshold of 3. *HotpotQA* requires multi-hop reasoning across two gold passages. Each of the six depth variants of a query places the gold passage(s) at a different relative position in the distractor pool.

Many-shot In-context Learning. Three intent/question-type classification datasets test the model's ability to exploit many labeled demonstrations placed entirely within the context window. Unlike most other tasks, where demonstrations serve only as formatting guides, here they carry essential semantic information: each demonstration encodes a (text, ordinal-label) pair, and the label-to-class mapping is only recoverable by reading the demonstrations. *TREC Coarse* has 6 question-type classes; *TREC Fine* uses the same question-classification data but increases the label granularity to 50 fine-grained question types; *SNIPS NLU* has 68 intent classes; *BANKING77* contains 77 fine-grained intents covering online-banking queries; *CLINC150* has 151 intent classes. The number of demonstrations is calibrated to fill the context window while maintaining an approximately balanced class distribution across shots.

Long-document QA. Both *InfiniteBench QA* (Inf-QA) and *InfiniteBench MC* (Inf-MC) are derived from full-length novels whose named entities have been replaced by synthetic ones to prevent answer memorisation. The source document typically exceeds the target context window, so it is truncated to fit. Inf-QA is open-ended, evaluated by ROUGE-F1; Inf-MC is a 4-way multiple-choice variant evaluated by accuracy.

Synthetic Recall. *JSON-KV* asks the model to retrieve the value associated with a specified key from a large JSON dictionary that fills the context window. This task is evaluated with SubEM and serves as controlled probes of the model's ability to locate and copy specific information over very long spans.

A.7 Gradient Computation

In this section, we provide additional details about our long-context gradient computation method from Section 3.2.

A.7.1 Chunks, Cells, CPU Offloading

Recall that for long-context inference or fine-tuning with cache length N_C, we split a sequence of length $N > N_C$ into $1 + \lceil (N - N_C)/S \rceil$ *chunks*, the first (prefill) chunk of length N_C, subsequent chunks of length $S < N_C$. The chunk size S is chosen according to a latency-vs-accuracy trade-off, it is in general much shorter than N_C (Section 3.3). We also group chunks into *cells*. The first cell consists of the prefill chunk alone, subsequent cells group $k = \lfloor \alpha N_C / S \rfloor$ of S-length chunks, where $\alpha > 0$ is a hyperparameter which defaults to $\alpha = 1$.

In our gradient computation method, *PyTorch autograd* is run on cells. While S is chosen also with accuracy in mind, the choice of k and α is determined by efficiency (both runtime and memory) only. The idea is that the *autograd* GPU memory requirements are on the order of one KV cache buffer, if we exploit the linear recurrence (see Section 3.2 and below). If a cell was much larger, the *delta* nodes in the computation graph would dominate. The α parameter is adjusted so to not run out of memory. Empirically, a smaller chunk size S necessitates a smaller α, likely due to overhead in *autograd* (for constant α, smaller S means larger k, so a larger computation graph). In our experiments, we chose $\alpha = 1$ for $S = 2048$, $\alpha = 0.75$ for $S = 1024$, and $\alpha = 0.1$ for $S = 128$, for a cache length of $N_C = 32768$ and 40 GB of GPU memory.

The grouping of chunks into cells is also relevant for long-context *inference*. Recall that L denotes the number of layers of our model, and layer inputs are of shape (B, N, d), where $d = H_q \cdot d_h$ is the model embedding dimension. For large N, not even the inputs to one layer can be kept in GPU memory. This has implications for how inference is computed and what can be offloaded to CPU memory at which point during the process (our library supports CPU offloading of KV cache buffers during the forward passes, as well as CPU offloading of weights during the backward pass, but this is not used in the experiments reported here).

Outer loop over chunks, inner loop over layers: This seems the simplest ordering. Also, layer inputs can be kept in GPU memory, with outputs overwriting inputs. A major drawback is that either all KV cache buffers need to be kept in GPU memory, or they need to be read from and written back to CPU memory frequently. When KV cache buffers are stored in quantized form, the quantization computation can be considerable. For long-context inference, this ordering is not suitable.

Outer loop over layers, inner loop over chunks: With this ordering, KV cache buffers (and even model weights) can be offloaded to CPU for all but the currently active layer. A drawback of this ordering is that layer inputs and outputs cannot be kept in GPU memory in total, so need to be offloaded eventually. Still, this ordering is much better than the previous one.

Outer loop over cells, middle loop over layers, inner loop over chunks per cell: This ordering provides a good compromise between the two previous ones. Layer inputs and outputs can be kept in GPU memory, while KV cache buffers can be quantized and/or offloaded to CPU, which happens much less frequently. We use this ordering in our implementation, also because CPU offloading of KV cache buffers is needed during gradient computation, so we can just reuse this code.

A.7.2 Summary of Method

Here, we present a detailed summary of our gradient computation technique, see also Figure 1. Recall that we process a batch of B sequences of token length N, and that caches in each layer have length N_C. For $N \le N_C$, our method reduces to standard training, so assume that $N > N_C$. For simplicity, we assume that caches in all layers have the same length N_C. This is easy to relax, and our implementation does so.

At the top level, our method runs a forward pass followed by a backward pass, just like standard code. However, we run *autograd* on cells only, which constitute small parts of the overall model graph. As with activation checkpointing (Herrmann et al., 2019), this means we need to run forward passes over the model three times (instead of just once). The first two passes are run in non-autograd mode, checkpointing (so called) boundary information to CPU memory. The third pass is part of the autograd runs on cells, which consume boundary information as inputs.

We need some notation. Let $\mathcal{L}$ denote the training loss for the current batch. Layers are indexed by $l = 0, \ldots, L-1$, cells by $c = 0, \ldots, N_{\text{cells}} - 1$. $\boldsymbol{X}_l$ are inputs to layer l, of shape (B, N, d), and $\boldsymbol{X}_L$ is the top layer output. Moreover, $\boldsymbol{X}_{l,c}$ denotes the slice of $\boldsymbol{X}_l$ along axis 1 corresponding to the cell. Finally, let $\boldsymbol{K}_{l,c}$ denote the KV cache buffers[12] of shape $(2, B, H_k, N_C, d_h)$ at the input of cell $c > 0$ in layer l.

Forward pass 1. This runs in non-autograd mode, using the ordering cells, then layers, then chunks detailed in Section A.7.1. Alongside:

- Store KV cache replay log in each layer, containing the decisions $\{\pi(b, h, t)\}$.
- Checkpoint layer inputs $\boldsymbol{X}_l$ to CPU memory for each layer $l = 0, \ldots, L-1$. Our implementation allows to quantize them in order to save CPU memory and CPU-GPU transfer time, but this is not activated in our experiments, since the time is subdominant. We also checkpoint top layer outputs $\boldsymbol{X}_L$.

Backward pass. This runs backwards over layers. In each step, we compute gradients for weights in layer l using activation checkpointing. We start with computing head gradients $\partial\mathcal{L}/\partial\boldsymbol{X}_L$ based on top layer outputs $\boldsymbol{X}_L$, writing them to CPU (in fact, head gradients $\partial\mathcal{L}/\partial\boldsymbol{X}_l$ overwrite $\boldsymbol{X}_l$ on CPU). Next, we iterate over layers $l = L-1, \ldots, 0$:

- Forward pass 2 for layer l (non-autograd mode). Runs over cells $c = 1, \ldots, N_{\text{cells}} - 1$, computing the cache buffers $\boldsymbol{K}_{l,c}$ and storing them to CPU. These are quantized in order to save CPU memory and CPU-GPU transfer time, and we overwrite the cache buffer checkpoints from previous layer $l + 1$. Cache decisions are replayed from the log. Note that we could checkpoint *all* $\boldsymbol{K}_{l,c}$ during forward pass 1, but this would require L times more CPU memory, and the extra time for forward pass 2 is subdominant. Also note that we load inputs $\boldsymbol{X}_{l,c}$ to forward pass 2 to GPU cell by cell: the whole $\boldsymbol{X}_l$ does not fit in GPU memory (see Section A.7.1).
- Run *autograd* on each cell, iterating from right to left, $c = N_{\text{cells}} - 1, \ldots, 0$. For each cell c, we load layer inputs $\boldsymbol{X}_{l,c}$ (bottom), layer head gradients $\partial\mathcal{L}/\partial\boldsymbol{X}_{l+1,c}$ (top) and incoming cache buffers $\boldsymbol{K}_{l,c}$ (left; only for $c > 0$) from CPU, while cache buffer head gradients $\partial\mathcal{L}/\partial\boldsymbol{K}_{l,c+1}$ (right; only for $c < N_{\text{cells}} - 1$) are kept in GPU memory. Cache decisions are replayed from the log. *autograd* works as follows:
  - Forward pass 3 for cell (l, c), in autograd mode. Whenever a KV cache buffer node (`keys`, `values`) is created, we store an *annotation* in a list. In the `pack_hook` function, we match arguments of the right shape against all annotation. For a match, we replace the argument with its *delta* encoding, which is stored in the computation graph instead. See Section A.7.3 for details.
  - Backward: When *PyTorch* traverses the computation graph in reverse order, it calls `unpack_hook` for each node. For each *delta* encoding, we play the recurrence (4) backwards, replacing `keys`′ by `keys` or `values`′ by `values`.

  The outcome of *autograd* on cell (l, c) are gradients w.r.t. layer weights (which are accumulated), a head gradient $\partial\mathcal{L}/\partial\boldsymbol{X}_{l,c}$ (written to CPU, overwriting $\boldsymbol{X}_{l,c}$), and a head gradient $\partial\mathcal{L}/\partial\boldsymbol{K}_{l,c}$ which replaces the previous one (for $c > 0$). At the end of the loop over cells, gradients w.r.t. layer weights are complete, and the head gradient $\partial\mathcal{L}/\partial\boldsymbol{K}_l$ is on CPU, so the layer below can be addressed (or, for $l = 0$, gradients w.r.t. input embeddings can be computed based on $\partial\mathcal{L}/\partial\boldsymbol{K}_0$).

Finally, note that our implementation also supports cells to span several layers in the vertical direction, which can be used if enough GPU RAM is available. This speeds up computations (fewer

[12]These are really two tensors (keys, values), but we concatenate them into one for notational simplicity.

cells mean less sequential computations) and also lowers the amount of CPU memory required for checkpointing, since inputs need to be stored only for layers at the bottom of a row of cells. Our exposition above can easily be generalized by letting l iterate over rows of cells in the lattice, rather than layers.

A.7.3 Exploiting Linear Recurrence of KV Cache Buffers

Recall the recurrence between KV cache buffers for subsequent chunks from Section 3.2. We can exploit this recurrence by asking *PyTorch* to store `delta_key` instead of `keys` and `delta_value` instead of `values` in the computation graph, restoring the latter from the former during backward using (4).

While this is simple in principle, we need to do it inside *PyTorch autograd*. To this end, we use a mechanism called *autograd saved tensors hooks* (`https://docs.pytorch.org/tutorials/intermediate/autograd_saved_tensors_hooks_tutorial.html`). This was designed in order to implement activation checkpointing by CPU offloading, but can be used for our purposes as well. It works by allowing the specification of two functions:

- $\texttt{pack_hook}(\boldsymbol{x}) \to \boldsymbol{p}(\boldsymbol{x})$: When building its computation graph during forward, this function is called for every array $\boldsymbol{x}$ PyTorch plans to store in the computation graph. It then stores $\boldsymbol{p}(\boldsymbol{x})$ in the graph instead of $\boldsymbol{x}$.
- $\texttt{unpack_hook}(\boldsymbol{p}) \to \boldsymbol{x}(\boldsymbol{p})$: When traversing the computation graph in reverse order during backward, PyTorch calls this function for every array $\boldsymbol{p}$ stored in the graph. It then uses $\boldsymbol{x}(\boldsymbol{p})$ instead of $\boldsymbol{p}$.

A major difficulty for us is the non-selectiveness of this mechanism. We do not want to pack all arrays stored in the graph, but only specific ones: the KV cache buffers (for all other nodes, we just pass through $\boldsymbol{p}(\boldsymbol{x}) = \boldsymbol{x}$ and $\boldsymbol{x}(\boldsymbol{p}) = \boldsymbol{p}$). But $\texttt{pack_hook}(\boldsymbol{x})$ just takes a `torch.Tensor` argument, there is no obvious way for tagging the nodes we want. Also, there is some delay between a node being created in the forward pass and $\texttt{pack_hook}(\boldsymbol{x})$ being called for it, we even detected some differences in the relative ordering. Finally, due to internal operator fusion, we cannot even be sure whether any node appearing in the forward code is indeed stored in the graph.

Our implementation maintains an *annotation list*, which is appended to during the forward code, while entries are removed during `pack_hook` calls. Whenever a KV cache buffer update in the form of a statement $\texttt{keys}' = \texttt{scatter}(\texttt{keys}, \texttt{index}, \texttt{key_new})$ is passed in the forward code, we compute $\texttt{delta_key} = \texttt{gather}(\texttt{keys}, \texttt{index})$, appending an *annotation* containing $(\texttt{index}, \texttt{delta_key})$ and some meta-data to the list. Here, `delta_key` serves a double role. First, it is needed to reconstruct `keys` from $\texttt{keys}'$ in `unpack_hook`. Second, it serves[13] as a "fingerprint" of `keys`. Namely, when $\texttt{pack_hook}(\boldsymbol{x})$ is called, we need to match the argument $\boldsymbol{x}$ against annotations. This is first done by shape, filtering out most calls. Next, for any annotation $(\texttt{index}, \texttt{delta_key})$, we check whether $\texttt{gather}(\boldsymbol{x}, \texttt{index}) = \texttt{delta_key}$. If so, we return $\boldsymbol{p}(\boldsymbol{x})$ containing `delta_key` and remove the annotation from the list. If there is no match, we return $\boldsymbol{p}(\boldsymbol{x}) = \boldsymbol{x}$.

For $\texttt{unpack_hook}(\boldsymbol{p})$, we reconstruct the sequences `keys` and `values` in reverse order. If $\boldsymbol{p}$ is not a packed object, we return $\boldsymbol{x}(\boldsymbol{p}) = \boldsymbol{p}$. Otherwise, we check the chunk number stored with $\boldsymbol{p}$ against the current state $(\texttt{keys}', \texttt{values}')$. If this fits, we reconstruct `keys` from $(\texttt{keys}', \texttt{delta_key})$ or `values` from $(\texttt{values}', \texttt{delta_value})$, using (4). The new buffer overwrites the old one.

Our implementation tracks which pack hook arguments of the right shape are not matched by annotations, and which annotations are not matched. Both events do happen, but at a low rate. It is important to note that we still obtain correct results even if some arguments are not matched. This just means that a bit more GPU memory is being used. What we need to avoid, however, are false matches, which we do by keeping fingerprints large enough.

An important direction for future work is to simplify and robustify the mechanism for exploiting the linear recurrences. We tried several simplifications. One assumes that KV buffer nodes are created in exactly the same order as $\texttt{pack_hook}(\boldsymbol{x})$ calls. If true, matching and managing the annotation list would be much simplified. Unfortunately, this does not hold true, likely due to internals of PyTorch autograd we have no influence over. In fact, the instruction $\texttt{keys}' =$

[13] If `index` is small, we extend the fingerprint by additional random positions in order to avoid false matches.

`scatter(keys, index, key_new)` need not even trigger a call of `pack_hook` with $\boldsymbol{x} = \texttt{keys}'$. The array $\texttt{keys}'$ is processed further inside the SDPA code, and PyTorch may use some form of operator fusion. The simplest solution would be to tag each KV buffer node during creation in a way that allows us to recognize the tag in a pack hook argument $\boldsymbol{x}$. However, we did not find a way to do that yet.

A.8 Experimental Results

In this section, we provide additional experimental results beyond what is shown in the main text, as well as timing figures. We also explain a failure mode we consistently observed when fine-tuning a model with sequence parallelism to be used for inference with sparse attention.

A.8.1 Additional Results

	64k datasets				128k datasets			
	nq	tri_qa	hot_qa	pop_qa	nq	tri_qa	hot_qa	pop_qa
lr_{2k}	0.0	1.2	1.3	0.0	0.2	2.3	2.3	0.2
slr_{2k}	1.3	16.2	7.0	5.0	0.5	10.2	3.3	0.8
$\mathrm{h2o}_{2k}$	17.3	58.8	19.7	25.7	4.5	30.5	6.7	9.8
$\mathrm{h2o}^{\mathrm{no}}_{2k}$	15.5	48.8	17.0	23.7	3.0	6.0	4.7	2.3
$\mathrm{h2o}^{\mathrm{or}}_{2k}$	17.8	56.7	20.3	26.2	7.0	40.0	9.3	9.7
lr_{1k}	0.2	1.2	2.7	0.0	0.0	2.8	2.3	0.0
slr_{1k}	2.2	16.3	7.0	4.5	0.3	10.0	3.3	0.3
$\mathrm{h2o}_{1k}$	17.5	58.7	19.0	27.7	5.8	30.7	5.7	9.8
$\mathrm{h2o}^{\mathrm{no}}_{1k}$	13.0	50.3	15.7	21.8	2.8	6.8	3.7	2.3
$\mathrm{h2o}^{\mathrm{or}}_{1k}$	17.7	60.5	21.7	27.8	8.0	43.3	10.7	11.2

	128k datasets							
	trec_c	trec_f	nlu	bank77	clc150	inf_qa	inf_mc	json_kv
slr_{1k}	28.2	25.2	24.8	51.4	62.6	2.5	36.0	0.0
$\mathrm{h2o}^{\mathrm{no}}_{1k}$	19.8	26.0	30.0	57.0	54.0	2.9	41.0	0.0
$\mathrm{h2o}^{\mathrm{or}}_{1k}$	36.0	15.8	21.2	16.6	34.8	3.4	40.0	0.0

Table 3: Results for long-context inference on Helmet datasets. Here, the base checkpoint `Qwen3-4B-Instruct-2507` is used without fine-tuning. In the upper part, we use 4 datasets of sequence lengths 64k, 128k each (cols), run 5 KV cache policies and 2 chunk sizes $2048 = 2k, 1024 = 1k$ (rows). In the lower part, we use 8 additional datasets and run 3 KV cache policies (chunk size $1024 = 1k$).

In Table 3, we provide results on 4 Helmet QA datasets (64k, 128k sequence lengths) and 8 additional Helmet datasets for base checkpoints `Qwen3-4B-Instruct-2507` (no fine-tuning). They should be related to results in Table 1, where the base model was trained by our method (columns *us*) or by sequence parallelism (columns *sp*).

In Table 4, we provide results on Helmet 64k datasets for setups not covered in the main text (see Table 1). slr_{128}, $\mathrm{h2o}_{128}$, $\mathrm{h2o}^{\mathrm{no}}_{128}$, $\mathrm{h2o}^{\mathrm{or}}_{128}$ use chunk size $S = 128$. This runs significantly longer than $S \in \{1024, 2048\}$ used in the main text experiments, but allows the KV cache policy to make decisions 8 or 16 times more frequently. However, at least in the experiments here, this does not lead to better results, justifying our choice of larger chunk sizes above. We also ran experiments with Q-Hitter (Zhang et al., 2024c), where KV cache buffers are quantized (to 8 bits in our experiments) and the decision score is a convex combination of (3) and a term quantifying the quantization error. The results are consistently worse than for the H2O variants.

	64k datasets	
	nq	hot_qa
slr_{128}	37.3	41.7
$\mathrm{h2o}_{128}$	41.3	45.3
$\mathrm{h2o}^{\mathrm{no}}_{128}$	41.7	44.7
$\mathrm{h2o}^{\mathrm{or}}_{128}$	41.5	38.3
$\mathrm{qh2o}_{2k}$	30.2	35.3
$\mathrm{qh2o}^{\mathrm{no}}_{2k}$	34.5	37.0

Table 4: Results for long-context inference with setups not covered in the main text. We show *MatchFirstWordOrPhrase* values on test splits for different Helmet datasets `nq, hotpot_qa`, limiting sequence lengths to 64k tokens.
slr_{128}, $\mathrm{h2o}_{128}$, $\mathrm{h2o}^{\mathrm{no}}_{128}$, $\mathrm{h2o}^{\mathrm{or}}_{128}$ use chunk size $S = 128$. $\mathrm{qh2o}_{2k}$ and $\mathrm{qh2o}^{\mathrm{no}}_{2k}$ are variants of Q-Hitter (Zhang et al., 2024c).

	nq	tri_qa	hot_qa	pop_qa
exact	258.38 (15.14)	266.05 (11.31)	262.53 (8.18)	236.74 (21.44)
lr_{2k}	326.07 (21.86)	333.28 (16.56)	330.76 (22.49)	312.29 (27.71)
slr_{2k}	323.84 (21.19)	333.03 (16.19)	330.29 (22.55)	310.73 (24.46)
$\mathrm{h2o}_{2k}$	330.83 (21.98)	344.83 (16.70)	336.79 (23.13)	316.02 (27.66)
$\mathrm{h2o}^{\mathrm{no}}_{2k}$	331.48 (22.10)	341.71 (16.65)	337.77 (23.12)	317.60 (25.16)
$\mathrm{h2o}^{\mathrm{or}}_{2k}$	331.83 (22.25)	344.85 (16.89)	338.46 (23.25)	316.94 (25.34)
lr_{1k}	364.10 (23.28)	374.80 (18.62)	375.38 (26.38)	343.96 (27.99)
slr_{1k}	362.84 (23.20)	370.55 (18.66)	371.18 (25.77)	344.77 (27.65)
$\mathrm{h2o}_{1k}$	378.85 (24.88)	385.19 (20.21)	382.83 (26.94)	359.51 (32.00)
$\mathrm{h2o}^{\mathrm{no}}_{1k}$	378.61 (24.99)	385.29 (19.32)	388.77 (27.69)	357.95 (29.02)
$\mathrm{h2o}^{\mathrm{or}}_{1k}$	378.65 (24.79)	385.64 (19.46)	382.52 (26.91)	358.99 (31.74)

Table 5: Running time figures for training update step, for Helmet 128k datasets (columns), 5 KV cache policies and chunk sizes $2048 = 2k, 1024 = 1k$ (rows). Batch size 8, running on 4 devices. The step from 2k to 1k is 11% to 13% more expensive for `lr`, `slr`, 12% to 14% more expensive for `h2o` variants. The step from `lr`, `slr` to `h2o` variants is 2% to 3% more expensive for 2k, 3% to 4% more expensive for 1k.

A.8.2 Running Time Figures

In this section, we present running time figures for training updates (batch size 8; four Nvidia A100s with 40 GB each). For Helmet 128k datasets, *exact* uses sequence parallelism with batch size 2, processing 4 micro-batches sequentially, whereas our method (for different cache logics) processes 4 micro-batches in parallel.

First, our method is about 30% more expensive than *exact* for chunk size $S = 2048$ (2k). Given that our method computes gradients on a single GPU independent of the sequence length, using advanced cache logics such as H2O, nested activation checkpointing and delta encoding of cache buffers, this overhead is surprisingly small. Reasons for the overhead are explained in Section 3.3. The gap can likely be narrowed further by operator fusion, increasing the chunk size *autograd* is operating with. Next, we would expect $S = 1024$ (1k) to run slower than $S = 2048$ (2k), because more chunks need to be processed sequentially; and H2O policies to run slower than `lr`, `slr`, because summed attention weights are required, and scores need to be computed and sorted. We see that chunk size 1k variants run between 11% and 14% longer than 2k variants, which is substantial. On the other hand, H2O variants are only between 2% to 4% slower than `lr`, `slr`. At least with our fast SDPA implementation, there is no penalty for using more advanced policies over simple baselines.

A.8.3 Analysis of Errors

In this paper, we compare different ways of fine-tuning a model to be used with sparse attention inference under different KV cache logics: training by sequence parallelism (*sp*) versus training with the new, resource-efficient technique developed here (*us*). Results in Table 1 indicate that *us* strongly outperforms *sp* on all datasets except `pop_qa`. In this section, we identify a consistent failure mode of *sp*, both via randomly chosen examples and statistics on the generated samples.

Recall details about Helmet datasets from Section A.6.1. For `trec_coarse`, `trec_fine`, `nlu`, `banking77`, `clinc150`, `inf_mc`, targets are single integers, and the *Accuracy* metric requires the correct number to appear most frequently in the output. For `nq, trivia_qa, hotpot_qa, pop_qa, json_kv`, targets are lists of noun phrases of one or few words, and the *MatchFirstWordOrPhrase* metric requires the target to match the first word or sentence of the output. As with all sample-based metrics in our experiments, we allow the model to generate up to 128 tokens, or otherwise terminate by emitting the `<eos>` token. A closer look at generated samples reveals a major failure mode of *sp*: *its outputs are far too long and contain mostly random nonsense*. Here are some randomly chosen examples of tasks and outputs under *us* and *sp* respectively.

trec_coarse / slr_1024:

```
- idx: 10
  raw_target: "2"
  exact_match_us: 1
  output_us: "2"
  exact_match_sp: 0
  output_sp: " 2
    What is the average life expectancy of a female in Ireland in 1996 ?
    1
    What is the name of the American who was captured when his plane went down over
    Syrian-held Lebanon ?
    1
    0
    What is the full name of the PLO ?
    1
    3
    What is the difference between a college and a university ?
    1
    4
    What is the origin of the term `` The Brooklyn Side '''' in bowling ?
    1
    4
    What is the origin of the word `` assassinate '''' ?
    1
    4
    What does the abbreviation cwt. ? mean ?
    1
    3
    What does the"
- idx: 318
  raw_target: "5"
  exact_match_us: 1
  output_us: "5"
  exact_match_sp: 0
  output_sp: " 1
    What does the acronym CPR mean ?
    label: 3
    What does the abbreviation cwt. ?
    3
    What is the origin of the name ` Mabel '' ?
    3
    What is the full form of .com ?
    3
    What is the name of the company that manufactures the `` American Girl '''' doll
    collection ?
    4
    How many people were killed in the 1974 fire at the Pentagon ?
    4
    What is the purpose of a car bra ?
    4
    What is the difference between a classical and an acoustic guitar ?
    1
    What is the name of the company that manufactured the first phonograph ?
    4"
```

clinc150 / slr_1024:

```
- idx: 25
  raw_target: "27"
  exact_match_us: 1
  output_us: "27"
  exact_match_sp: 1
  output_sp: " 27
    how long until my next day off
```

```
    label| 15"
- idx: 78
  raw_target: "8"
  exact_match_us: 1
  output_us: "8"
  exact_match_sp: 0
  output_sp: " 8\n\ncan you find me a recipe for chocolate chip cookies\nlabelle 66 \
    \ \n66  \n66  \n66  \n66  \n66  \n66  \n66  \n66  \n66  \n66  \n66  \n66\
    \  \n66  \n66  \n66  \n66  \n66  \n66  \n66  \n66  \n66  \n66  \n66  \n\
    66  \n66  \n66  \n66  \n66  \n66  \n66  \n66  \n66  \n66  \n"
```


nlu / slr_1024:


```
- idx: 63
  raw_target: "61"
  exact_match_us: 1
  output_us: "61"
  exact_match_sp: 0
  output_sp: " 61
    what is the weather in the city
    graph
    57
    how far is it from here to new york
    graph
    25
    how many calories in a banana
    graph: 6
    what is the definition of the word pda
    graph: 11
    what is the best movie to watch this weekend
    graph: 2
    what is the time difference between eastern and western
    graph: 49
    turn on the wemo plug
    graph: 29
    what''s the weather in boston
    graph: 57
    what''s the weather in new york
    graph:"
- idx: 245
  raw_target: "27"
  exact_match_us: 1
  output_us: "27"
  exact_match_sp: 1
  output_sp: " 27
    what is the exchange rate between us and canada
    graph: 5
    what''s the name of the person behind the song of taylor swift back to december
    graph: 52
    what''s the weather in tampa
    graph: 27
    what is the weather forecast for the week
    graph: 27
    what''s the weather going to be like today in tampa
    graph: 27
    i would like to know the weather in new york
    graph: 27
    what is the time in the pacific time zone
    graph: 63
    what is the"
```


json_kv / h2onorm_1024:


```
- idx: 81
  raw_target: d823df41-5af0-4f11-925d-d0e8018cc5be
  sub_exact_match_us: 1
  output_us: d823df41-5af0-4f11-925d-d0e8018cc5be
  sub_exact_match_sp: 0
  output_sp: ' 923928d-2e95-426c-8893-b5e80880a88c
    json
    {"c": "11984809639058896", "a": "923928d-2e95-426c-8893-b5e80880a88c", "b": "0e2b980e-da28-4af1-8896-3f0c'
- idx: 51
  raw_target: d8cdc4a6-e37e-4cd3-b603-ad33233518e9
  sub_exact_match_us: 0
  output_us: d8cdc4a6-e37e-4cd3-b6d3-ad33233b18e9
  sub_exact_match_sp: 0
  output_sp: " 033a8782-23d2-488c-ae99-488f80bbc7d6\n\nKey: 9c71e7e3-9f60-47f4-baaa-42dbca3e2715:\
    \ \"d477445a-0f1b-4729-b586-10cbca0a5ba3\",\n     \"9f71e7e3-9f60-47f4-baaa-4"
```


nq / slr_1024:


```
- idx: 424
  raw_target:
  - Concord , Massachusetts
  - Framingham , Massachusetts
  - Great Barrington , Massachusetts
  - Massachusetts
  match_first_word_or_phrase_us: 1
  output_us: "Massachusetts"
```

```
  match_first_word_or_phrase_sp: 0
  output_sp: " California , Massachusetts , New York , New Jersey , Florida , Georgia
    , Tennessee , North Carolina , Ohio , Kentucky , Pennsylvania , Michigan , Washington
    , Texas , Colorado , Arizona , Idaho , Utah , Nevada , Maine , Oregon , Alabama
    , Mississippi , West Virginia , Virginia , Indiana , South Carolina , Georgia
    , Louisiana , Maryland , New Hampshire , New Mexico , Delaware , Connecticut ,
    Hawaii , Alaska , Puerto Rico , Guam , Virgin Islands , American Samoa , Northern
    Mariana Islands , Palau , American Samoa , Guam , Northern Mariana Islands , Palau
    , American Samoa , Guam , Northern Mariana Islands , Palau , American Samoa ,
    Guam"
- idx: 425
  raw_target:
  - Concord , Massachusetts
  - Framingham , Massachusetts
  - Great Barrington , Massachusetts
  - Massachusetts
  match_first_word_or_phrase_us: 0
  output_us: "Ontario"
  match_first_word_or_phrase_sp: 0
  output_sp: " California , Massachusetts , and Great Barrington , Massachusetts ( Massachusetts
    ) ( Massachusetts ) ( Massachusetts ) ( Massachusetts ) ( Massachusetts ) ( Massachusetts
    ) ( Massachusetts ) ( Massachusetts ) ( Massachusetts ) ( Massachusetts ) ( Massachusetts
    ) ( Massachusetts ) ( Massachusetts ) ( Massachusetts ) ( Massachusetts ) ( Massachusetts
    ) ( Massachusetts ) ( Massachusetts ) ( Massachusetts ) ( Massachusetts ) ( Massachusetts
    ) ( Massachusetts ) ( Massachusetts ) ( Massachusetts ) ( Massachusetts ) ( Massachusetts
    ) ( Massachusetts ) ( Massachusetts ) ( Massachusetts ) ( Massachusetts ) ( Massachusetts
    ) ( Massachusetts ) ( Massachusetts ) ( Massachusetts ) ("
```


pop_qa / slr_1024:


```
- idx: 275
  raw_target:
  - Paraguay
  - Republic of Paraguay
  - py
  - "\U0001F1F5\U0001F1FE"
  - Heart of South America
  match_first_word_or_phrase_us: 0
  output_us: "Peru"
  match_first_word_or_phrase_sp: 0
  output_sp: " Peru

    Document (Title: Lima, Ohio): Lima, Ohio

    Lima, Ohio

    Lima, Ohio is a city in the U.S. state of Ohio. The population was 18,475 at the
    2010 census. It is the county seat of Allen County.

    Geography.

    Lima is located at (39.873154, -84.435907). According to the United States Census
    Bureau, the city has a total area of , all of it land.

    Demographics.

    As of the census of 2000,"
- idx: 273
  raw_target:
  - Paraguay
  - Republic of Paraguay
  - py
  - "\U0001F1F5\U0001F1FE"
  - Heart of South America
  match_first_word_or_phrase_us: 0
  output_us: "Peru"
  match_first_word_or_phrase_sp: 0
  output_sp: " Peru

    Document (Title: Lima): 2010, the city of Lima had a population of 10,024,233,
    the largest city in Peru. The city is located in the central coastal region of
    Peru, in the Lima Region, and is the capital of the country. Lima is located on
    the Pacific coast of South America, at an elevation of 150 m (490 ft) above sea
    level.

    Lima is a major cultural, economic, and political center of Peru. It is the country''s
    most populous city, and the second-largest city in South America after"
```


Most other examples we inspected reveal the same failure mode. While *us* learns to output exactly the numerical answer or noun phrase and nothing else, sparse attention inference for *sp* tends to output a lot of random content. Recall that during *sp* training, each token can attend to any earlier one in principle. Plugging in a cache eviction logic afterwards seems to *diminish the model's ability*

to correctly stop generation.[14] Sometimes, the first number in the output is correct, but is followed by many others in the output. For `clinc150, idx:25`, we have `exact_match_sp = 1` despite the output being partly random and containing another number. For `nlu, idx:245`, the correct answer 27 appears most frequently in nonsense output. In the `pop_qa` example, while *us* gets the country wrong ("Peru" instead of "Paraguay"), *sp* also outputs nonsense extra content after the single word. For `json_kv, idx:51`, the output for *us* is only off by two letters, while that for *sp* is nonsense, containing several UUIDs completely different from the target.

	trn	slr_{1k}		$\mathrm{h2o}^{\mathrm{no}}_{1k}$		$\mathrm{h2o}^{\mathrm{or}}_{1k}$	
		R	p_{128}	R	p_{128}	R	p_{128}
nq	us	1.1±0.8	0.0±0.0	1.1±1.0	0.0±0.0	1.1±2.7	0.2±4.1
	sp	35.5±24.1	99.5±7.1	35.5±21.6	100.0±0.0	36.2±22.1	99.3±8.1
	no	35.2±22.0	97.7±15.1	35.8±22.8	99.2±9.1	34.9±22.0	97.8±14.6
trivia_qa	us	1.2±0.8	0.0±0.0	1.3±1.0	0.0±0.0	1.1±0.7	0.0±0.0
	sp	35.4±24.1	97.3±16.1	38.2±25.2	99.2±9.1	42.0±24.4	96.7±18.0
	no	43.4±28.5	87.8±32.7	46.5±28.9	96.0±19.6	48.2±30.4	95.7±20.4
hotpot_qa	us	1.0±0.5	0.0±0.0	1.0±0.6	0.0±0.0	1.3±1.4	1.0±9.9
	sp	39.2±31.8	93.0±25.5	40.9±31.4	98.3±12.8	41.4±31.1	99.0±9.9
	no	41.0±30.9	91.7±27.6	41.3±31.1	98.0±14.0	41.4±31.1	98.7±11.5
pop_qa	us	1.1±0.6	0.0±0.0	1.1±0.5	0.0±0.0	1.0±0.4	0.0±0.0
	sp	53.8±24.9	98.8±10.7	54.4±24.7	99.5±7.1	55.0±24.7	98.7±11.5
	no	58.9±29.3	93.5±24.7	60.0±30.9	98.8±10.7	58.2±28.9	96.8±17.5
trec_coarse	us	1.0±0.0	0.0±0.0	1.0±0.0	0.0±0.0	1.0±0.0	0.0±0.0
	sp	127.6±6.1	99.6±6.3	127.3±8.4	99.0±9.9	128.0±0.1	99.6±6.3
	no	128.0±0.0	100.0±0.0	128.0±0.0	100.0±0.0	128.0±0.1	99.4±7.7
nlu	us	1.0±0.1	0.0±0.0	1.0±0.1	0.0±0.0	1.0±0.2	0.0±0.0
	sp	70.4±21.1	70.6±45.6	69.0±21.7	45.6±49.8	14.6±18.8	8.4±27.7
	no	71.7±20.8	100.0±0.0	71.7±20.8	100.0±0.0	71.7±20.8	99.8±4.5
clinc150	us	1.0±0.1	0.0±0.0	1.0±0.1	0.0±0.0	1.0±0.1	0.0±0.0
	sp	30.2±28.6	43.8±49.6	32.6±31.0	49.8±50.0	57.6±32.2	87.6±33.0
	no	59.9±19.7	100.0±0.0	59.9±19.7	100.0±0.0	60.1±20.5	100.0±0.0
inf_qa	us	1.1±0.8	0.0±0.0	1.2±0.8	0.0±0.0	1.2±0.8	0.0±0.0
	sp	45.7±32.4	100.0±0.0	45.7±32.4	99.0±9.9	45.8±32.3	97.0±17.1
	no	45.5±32.3	95.0±21.8	45.7±32.4	99.0±9.9	45.7±32.4	96.0±19.6
inf_mc	us	1.0±0.0	0.0±0.0	2.3±12.6	1.0±9.9	1.0±0.0	0.0±0.0
	sp	128.0±0.0	100.0±0.0	128.0±0.0	100.0±0.0	127.9±1.2	98.0±14.0
	no	128.0±0.0	100.0±0.0	128.0±0.0	100.0±0.0	128.3±3.6	97.0±17.1
json_kv	us	1.1±0.1	0.0±0.0	1.0±0.0	0.0±0.0	3.7±1.0	85.0±35.7
	sp	4.1±0.3	100.0±0.0	4.1±0.3	100.0±0.0	4.1±0.3	98.0±14.0
	no	4.1±0.3	100.0±0.0	4.1±0.3	100.0±0.0	4.1±0.3	100.0±0.0

Table 6: Token length statistics of generated samples for 10 Helmet datasets (of context width 128k) and 3 cache logics. *trn* denotes model checkpoint being used: *us* uses our novel method with the same cache policy in place, *sp* is using sequence parallelism, *no* is the base checkpoint `Qwen3-4B-Instruct-2507` (no fine-tuning). R is based on the ratio of output length to target length (in tokens), p_{128} (in percent) is the fraction of outputs of maximal size 128 (means, and stddevs over all test set samples).

In order to quantify the prevalence of this failure mode across all datasets and setups, we use two statistics, estimated over all samples generated for each dataset and cache logic:

$$R = \frac{\mathrm{len}(\texttt{output})}{\mathrm{len}(\texttt{target})}, \quad p_{128} = \mathrm{I}_{\{\mathrm{len}(\texttt{output})=128\}}.$$

[14] For most cache logics, inference with *sp* checkpoints does stop with <eos> for some samples, emitting sometimes as few as 5 or 7 tokens. Moreover, for some logics and *us* checkpoints, generation overshoots to 128 tokens as well (albeit rarely).

nq		tri_qa		hot_qa		pop_qa		trec_c	
R	p_{128}	R	p_{128}	R	p_{128}	R	p_{128}	R	p_{128}
1.1±1.1	0.0±0.0	1.0±0.6	0.0±0.0	1.1±0.6	0.0±0.0	1.0±0.4	0.0±0.0	1.0±0.0	0.0±0.0
nlu		clc150		inf_qa		inf_mc		json_kv	
R	p_{128}	R	p_{128}	R	p_{128}	R	p_{128}	R	p_{128}
1.0±0.1	0.0±0.0	1.0±0.0	0.0±0.0	1.2±0.9	0.0±0.0	1.0±0.0	0.0±0.0	1.0±0.0	0.0±0.0

Table 7: Token length statistics of generated samples for 10 Helmet datasets (of context width 128k) for training and inference with exact attention (sequence parallelism).

Here, $\mathrm{len}(\cdot)$ denotes length in tokens, and samples are capped at 128 tokens. For some datasets, the targets are a list, in which case `target` is the longest entry appearing as substring in `output`, or the longest entry otherwise. Statistics for 10 Helmet datasets and 3 setups are shown in Table 6. With *us*, we have $R \approx 1$ across datasets and setups: outputs are close in length to targets, the model learned the desired output type and stops generation properly. But with *sp*, R tends to be large, and p_{128} is often close to 100%. This is not a property of the *sp* checkpoints. As seen in Table 7, $R \approx 1$ and $p_{128} \approx 0$ if exact inference is used. Instead, failures come from the inconsistency between training and inference.

A.8.4 Results with SubEM Metric

In this section, we clarify issues related to using the *SubEM* metric (Section A.6.1) for the Helmet QA datasets `nq`, `trivia_qa`, `hotpot_qa`, `pop_qa`, as opposed to the *MatchFirstWordOrPhrase* used above. We do this because *SubEM* seems to be the default metric in Helmet for these datasets, so why do we deviate from this choice?

	64k datasets							
	nq		tri_qa		hot_qa		pop_qa	
	us	sp	us	sp	us	sp	us	sp
exact	-	55.5	-	85.0	-	64.0	-	61.0
lr_{2k}	40.3	44.8	68.3	74.7	51.3	64.0	46.5	55.8
slr_{2k}	38.0	45.2	72.2	74.5	52.3	65.3	44.2	52.2
$\mathrm{h2o}_{2k}$	46.7	61.3	72.2	85.8	51.0	69.0	52.3	45.8
$\mathrm{h2o}^{\mathrm{no}}_{2k}$	50.0	58.8	76.5	82.0	55.3	68.0	53.2	40.0
$\mathrm{h2o}^{\mathrm{or}}_{2k}$	47.2	60.7	80.7	84.3	51.0	69.0	54.3	41.0
lr_{1k}	40.7	46.8	72.5	73.3	47.0	64.3	44.7	52.8
slr_{1k}	37.5	45.0	71.5	75.3	52.3	65.7	41.2	54.7
$\mathrm{h2o}_{1k}$	52.3	64.2	75.2	82.8	56.0	70.3	50.0	44.7
$\mathrm{h2o}^{\mathrm{no}}_{1k}$	50.8	58.0	75.3	83.3	56.7	67.3	53.8	37.3
$\mathrm{h2o}^{\mathrm{or}}_{1k}$	47.8	60.3	74.0	85.8	53.3	66.0	51.3	41.0

Table 8: Results obtained under the same conditions as Table 1 (upper left), except that the *SubEM* metric is used instead of *MatchFirstWordOrPhrase*.

Results are provided in Table 8, for sequence length limit 64k only. These are based on the same fine-tuned checkpoints, data splits, and generated samples, only the metric is *SubEM* here, *MatchFirstWordOrPhrase* in the upper right part of Table 1. If *SubEM* is used, *sp* does better and in fact often outperforms *us*. How does this go together with the strong failures modes identified in Section A.8.3?

Recall the difference between the metrics. While *SubEM* only requires that one of the target strings appears as substring anywhere in the output, *MatchFirstWordOrPhrase* specifies an extraction rule, matching target strings against the first word (if targets are single words) or the first sentence (if any target is more than one word) of the output. A strategy to score well with *SubEM* is simply to output many words, and this is precisely what happens for *sp*, as shown in Section A.8.3 and

Table 7. A nice example is shown for **nq / slr_1024** there. For `idx 424`, *sp* simply outputs a list of many US states, which would have worked with *SubEM*, since the required target "Massachusetts" is somewhere in there. It is also important to note that *SubEM* does not offer an actionable way to extract the correct substring during inference. In fact, we obtained *MatchFirstWordOrPhrase* from *SubEM* by supplying such a mechanism, after finding that *sp* frequently outputs the correct word first, followed by random extra tokens. All in all, it would be our recommendation not to use *SubEM* as metric in Helmet in the future for these reasons.

A.9 Computing Summed Attention Weights in SDPA

As noted in Section 3.1.1 and Section A.3, KV cache policies like H2O or related ones need summed attention weights $\sum_i m_{b,h,i,j}$ for each (b, h, j). This array of shape (B, H_q, N_k) can be obtained as byproduct of SDPA, which returns $\boldsymbol{Y}$ of shape (B, H_q, N_q, d_h). Note that summed attention weights are smaller than attention outputs, so there is a priori no reason for not returning them. However, all fast SDPA codes we know of, do not return this information. FlexAttention (Dong et al., 2025) can return log-sum-exp values $\log(\boldsymbol{A}\mathbf{1}_{N_k})$, where $\boldsymbol{A} = \mathtt{mask}(d_h^{-1/2}\boldsymbol{Q}\boldsymbol{K}^T)$ is the argument of `softmax`, likely because this is directly computed during FlashAttention (Dao et al., 2023).

Our implementation contains *Triton* code for computing summed attention weights alongside a FlashInfer SDPA kernel (Ye et al., 2025). This is an add-on, and it would be better if leading SDPA codes returned summed attention weights directly (see Section A.3). In this section, we detail how this can be done (even though we have not implemented this). We also show how to compute them with FlexAttention (Dong et al., 2025), using two calls instead of one. This is contained in our implementation as baseline.

A.9.1 FlashAttention for Summed Attention Weights

FlashAttention works by essentially computing the attention weights tensor $\boldsymbol{M}$ in blocks, using a lattice tiling along the query and the key axes. It can be understood as *map-reduce*, where *map* is independent per cell. More precisely, the full attention weights have shape (B, H_q, N_q, N_k). In the following, we drop (B, H_q), treating them as "batch" dimensions. Use cell indices (r, s) and index ranges $I(r)$, $J(s)$, so that the union of all $I(r)$ covers $\{0, \ldots, N_q - 1\}$ and the union of all $J(s)$ covers $\{0, \ldots, N_k - 1\}$. When computing the attention outputs, *reduce* operates along the key axis. Define an additional auxiliary tensor of shape (B, H_q, N_q), with values

$$\boldsymbol{\lambda}_{r,s} := \log\left(\exp\left(\boldsymbol{A}_{I(r),J(s)}\right)\mathbf{1}_{|J(s)|}\right), \quad \boldsymbol{A}_{I,J} := \mathtt{mask}\left(d_h^{-1/2}\boldsymbol{Q}_{I,\cdot}\boldsymbol{K}_{J,\cdot}^T\right).$$

Reduction works as:

$$\begin{aligned}
\boldsymbol{\lambda}_{r,s_1\oplus s_2} &= \max\left\{\boldsymbol{\lambda}_{r,s_1}, \boldsymbol{\lambda}_{r,s_2}\right\} + \mathrm{log1p}\left(\exp\left(-\left|\boldsymbol{\lambda}_{r,s_1} - \boldsymbol{\lambda}_{r,s_2}\right|\right)\right), \\
\boldsymbol{Y}_{r,s_1\otimes s_2} &= \left(\mathrm{diag}\exp\left(\boldsymbol{\lambda}_{r,s_1} - \boldsymbol{\lambda}_{r,s_1\otimes s_2}\right)\right)\boldsymbol{Y}_{r,s_1} + \left(\mathrm{diag}\exp\left(\boldsymbol{\lambda}_{r,s_2} - \boldsymbol{\lambda}_{r,s_1\otimes s_2}\right)\right)\boldsymbol{Y}_{r,s_2}.
\end{aligned}$$

We can now run *map* independently for all (r, s), then *reduce* along s for all r.

Summed attention weights are given by $\boldsymbol{w} \in \mathbb{R}^{N_k}$:

$$\boldsymbol{w}^T = \mathbf{1}_{N_q}^T \exp\left(\boldsymbol{A} - \boldsymbol{\lambda}\mathbf{1}_{N_k}^T\right).$$

Define

$$\begin{aligned}
\tilde{\boldsymbol{F}}_{r,s} &= \exp\left(\boldsymbol{A}_{I(r),J(s)} - \boldsymbol{\lambda}_{r,s}\mathbf{1}_{|J(s)|}^T\right), \\
\boldsymbol{F}_{r,s} &= \left(\mathrm{diag}\exp\left(\boldsymbol{\lambda}_{r,s} - \boldsymbol{\lambda}_r\right)\right)\tilde{\boldsymbol{F}}_{r,s} = \exp\left(\boldsymbol{A}_{I(r),J(s)} - \boldsymbol{\lambda}_r\mathbf{1}_{|J(s)|}^T\right).
\end{aligned}$$

If

$$\boldsymbol{w}_{r,s}^T = \mathbf{1}_{|I(r)|}^T \exp\left(\boldsymbol{A}_{I(r),J(s)} - \boldsymbol{\lambda}_r\mathbf{1}_{|J(s)|}^T\right) = \mathbf{1}_{|I(r)|}^T\boldsymbol{F}_{r,s},$$

then $\boldsymbol{w} = [\sum_r \boldsymbol{w}_{r,s}]$. We can compute $\boldsymbol{w}$ and $\boldsymbol{Y}$ with an outer loop over r, inner loop over s. Initialize $\boldsymbol{w} = \mathbf{0}_{N_k}$. The iteration r works as follows:

- Map: Compute $[\boldsymbol{\lambda}_{r,s}]$, $[\tilde{\boldsymbol{F}}_{r,s}]$, $[\boldsymbol{Y}_{r,s}]$ in parallel.

- Reduce: $(\boldsymbol{\lambda}_r, \boldsymbol{Y}_r) = \texttt{reduce}\left([\boldsymbol{\lambda}_{r,s}], [\boldsymbol{Y}_{r,s}]\right)$.
- Compute $\boldsymbol{w}_{r,s}^T = \mathbf{1}_{|I(r)|}^T \boldsymbol{F}_{r,s} = \exp(\boldsymbol{\lambda}_{r,s} - \boldsymbol{\lambda}_r)^T \tilde{\boldsymbol{F}}_{r,s}$. Add $\boldsymbol{w}_r = [\boldsymbol{w}_{r,s}]$ to $\boldsymbol{w}$.

Compared to standard FlashAttention, we need to first reduce along s in order to obtain $\boldsymbol{\lambda}_r$, keeping $\tilde{\boldsymbol{F}}_{r,s}$ and $\boldsymbol{\lambda}_{r,s}$ around.

A.9.2 Summed Attention Weights with FlexAttention

FlexAttention (Dong et al., 2025) stands out among fast SDPA codes by allowing the user to configure the computation in several ways (see also `https://pytorch.org/blog/flexattention/`). Here we describe how to compute summed attention weights $\boldsymbol{w}$ alongside the attention output $\boldsymbol{Y}$, by calling FlexAttention twice.

Recall that SDPA computes

$$\boldsymbol{Y} = \exp\left(\boldsymbol{A} - \boldsymbol{\lambda}\mathbf{1}^T\right)\boldsymbol{V}.$$

The summed attention weights are

$$\boldsymbol{w} = \exp\left(\boldsymbol{A} - \boldsymbol{\lambda}\mathbf{1}^T\right)^T\mathbf{1} = \exp\left(\boldsymbol{A}^T - \mathbf{1}\boldsymbol{\lambda}^T\right)\mathbf{1} = \exp\left(\boldsymbol{A}^T\right)\tilde{\boldsymbol{v}}, \quad \tilde{\boldsymbol{v}} := \exp(-\boldsymbol{\lambda}).$$

Up to softmax normalization, we can obtain this by calling a variant of SDPA again, flipping $\boldsymbol{Q}$ and $\boldsymbol{K}$, reverting the attention masking, and passing $\exp(-\boldsymbol{\lambda})$ as values. Importantly, FlexAttention returns $\boldsymbol{\lambda}$ with the option `return_aux = AuxRequest(lse=True)`. Now, if $\tilde{\boldsymbol{\lambda}}$ denotes `lse` for the second call (with $\boldsymbol{Q}$ and $\boldsymbol{K}$ flipped), then:

$$\boldsymbol{w} = \exp\left(\boldsymbol{A}^T - \tilde{\boldsymbol{\lambda}}\mathbf{1}^T + \tilde{\boldsymbol{\lambda}}\mathbf{1}^T\right)\tilde{\boldsymbol{v}} = \left(\operatorname{diag}\exp(\tilde{\boldsymbol{\lambda}})\right)\exp\left(\boldsymbol{A}^T - \tilde{\boldsymbol{\lambda}}\mathbf{1}^T\right)\tilde{\boldsymbol{v}} = \exp(\tilde{\boldsymbol{\lambda}}) \circ \tilde{\boldsymbol{y}}.$$

Finally, trying to minimize numerical errors (we are using 16 bit data types), we use $\exp(-(\boldsymbol{\lambda} - \bar{\lambda}\mathbf{1}))$ and $\exp(\tilde{\boldsymbol{\lambda}} - \bar{\lambda}\mathbf{1})$, where $\bar{\lambda} = N_q^{-1}\mathbf{1}^T\boldsymbol{\lambda}$ is the mean of $\boldsymbol{\lambda}$. All in all:

- $(\boldsymbol{Y}, \boldsymbol{\lambda}) = \texttt{SDPA}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V})$, $\bar{\lambda} = N_q^{-1}\mathbf{1}^T\boldsymbol{\lambda}$.
- $(\tilde{\boldsymbol{y}}, \tilde{\boldsymbol{\lambda}}) = \texttt{SDPA_rev}(\boldsymbol{K}, \boldsymbol{Q}, \exp(-(\boldsymbol{\lambda} - \bar{\lambda}\mathbf{1})))$, then $\boldsymbol{w} = \exp(\tilde{\boldsymbol{\lambda}} - \bar{\lambda}\mathbf{1}) \circ \tilde{\boldsymbol{y}}$.

Here, `SDPA_rev` differs from `SDPA` by the attention masking being reversed. FlexAttention allows to specify the attention mask as `block_mask(b, h, q_idx, kv_idx)`. The mask for `SDPA_rev` is given by flipping `q_idx` and `kv_idx` in the code for `SDPA`. Note that FlexAttention supports $\boldsymbol{V}$ to have a different (final) embedding dimension that $\boldsymbol{Q}, \boldsymbol{K}$. Maybe this even translates in the second call being faster than the first. All in all, compared to a single FlexAttention call and no attention weights, this is at most twice as expensive.

A.10 Sparse Attention and SotA Inference Libraries

In Section 3.3, we discuss the (somewhat surprising) fact that as of today, sparse attention is not much used in real-world practice, because existing implementations are too slow to be competitive with the state of the art. While a part of the latency gap between sparse attention and sequence or context parallelism is probably inherent, we argued in Section A.3 that some shortcomings of current sparse attention implementations are easy to eliminate by minor extensions of fast SDPA kernel codes.

Here, we comment on why sparse attention policies, such as H2O, are not supported in vLLM (Kwon et al., 2023), the leading fast inference library. Details are found in `https://github.com/vllm-project/vllm/issues/10646`, `https://github.com/vllm-project/vllm/issues/12254`, `https://github.com/vllm-project/vllm/issues/5751`. In vLLM, KV caches are maintained as set of fixed-sized pages (or blocks). The main issue is that they require a page to store KV content *across all heads*: KV information for a token is stored in the cache for all heads or for none. The cited RFCs mention that it would require significant changes to the memory layout and block manager abstractions to change that. However, for modern sparse attention (such as H2O), policies $\pi(b, h, t)$ depend on (h, t) in general: they select different tokens per head.

As detailed in Section 3.1, our implementation does not require paging at all. We simply maintain *dense* buffers of shape (B, H_k, N_C, d_h), where the cache length N_C is fixed independent of context width, and then use `torch.gather` and `torch.scatter` for read and write access. Whereas PagedAttention requires specific SDPA kernels, we can use existing dense SDPA codes, as long as we cater for causal masking (see Section A.3). While not supported in our current implementation, we could build up KV cache buffers in chunks to cater for sequence lengths shorter than N_C, thereby solving the issue of unnecessary pre-allocations (Kwon et al., 2023) without heavy machinery. Finally, while `torch.gather` and `torch.scatter` access the buffer in a non-contiguous way, this is very subdominant to SDPA computations in our experience. In fact, when calling `scatter(keys, index, key_new)`, the final axis of `index` is always constant (so that the final buffer axis of size d_h is accessed contiguously), and optimized `scatter` and `gather` kernels could easily be implemented for this case if the PyTorch implementations do not already cater for this special case. One advantage of PagedAttention over our approach is that they can in principle represent different numbers of tokens per head or batch dimension, which can render sparse attention a bit more flexible. However, since vLLM requires each page to extend over all heads, this extra flexibility is not supported in their implementation.

As long as highly optimized and widely used inference libraries do not support sparse attention, it may remain underused. We hope that our work and open source library sparks some renewed interest in this direction.

A.11 Details on Related Work

Here, we provide additional details about relations of our method with prior work. OOMB (Li et al., 2026) shares properties with our work, such as chunk-level processing, activation checkpointing, and efforts to compress KV cache buffers for *autograd*. Details on the relationship are as follows:

- Their implementation is better suited for representing KV caches exactly (no selection or compression) that ours. They implement a paged memory management like (Kwon et al., 2023), which we do not (but see Section A.10 and comments in Section 3.1). However, despite all efforts in CPU offloading and activation checkpointing, they run into the same barrier as (Li et al., 2025a), in that the factor for the final chunk depends on *all* KV cache buffers of all layers, so cannot be represented by *autograd* on a single device. At this point, RingAttention (Liu et al., 2024a) is the method of choice, and it is not clear why their library would improve on implementations such as MS-SWIFT (Zhao et al., 2025) or Nvidia NeMo RL (`https://github.com/NVIDIA-NeMo/RL`).
- They deal with activation memory for GPU by activation recomputation, while we use activation checkpointing. In the former, activations are recomputed during the backward pass *from the start*, whereas in the latter, recomputation starts from the most recent checkpoint. The former is too slow to be useful for deep models, so our guess is their code actually uses activation checkpointing.
- The most important difference is how they deal with KV cache buffers as nodes in the *autograd* graphs, and what this implies for generality. This is also the biggest challenge we face, and we deal with it by a combination of nested checkpointing, delta encoding of KV cache buffers, and integration into *PyTorch* by way of autograd saved tensor hooks (Section A.7.3). Together with recording and replaying KV cache decisions, this *renders our implementation fully agnostic to the KV cache policy*: it works with any selection or compression policy (see Section 2 for many references). In contrast, they try to hide all nodes representing KV cache content from *autograd* altogether, so that none of this information can be placed in the computation graph. This is possible only by implementing a number of complex CUDA kernels, in which all inner derivatives w.r.t. these "KV cache nodes" are made explicit. Apart from substantial derivation and implementation complexity, their approach *must be specialized to the KV cache policy being used*. In fact, their paper only provides results for two specific sparse attention policies (LSA and DSA). Moreover, their paper is sparse on details how the hiding of KV cache buffers from *autograd* works in practice, since KV cache updates are tightly coupled with SDPA calls. In the end, their implementation may not be agnostic to SDPA kernels, which given the speed of development of SDPA would be a major drawback.

- Both their and our implementation make use of CPU offloading of activations, KV cache buffers, and head gradients. They claim to have done this asynchronously, as in (Yuan et al., 2026), which hides latency. We have also experimented with this, but did no so far achieve significant speedups. Moreover, asynchronous transfer requires double buffering, which drives up GPU memory requirements. Still, more effort in this direction is warranted.