DeltaAttention

Contents

1 Introduction

Before the Transformer era, language models were built on recurrent neural networks (RNNs) and their variants such as LSTMs and GRUs. These models were used in small-scale conversational assistants like Siri and Alexa, but their usefulness was limited. RNNs encode context into a fixed-size hidden state, which means they forget information as sequences grow longer and the hidden state get diluted. More importantly, they process text sequentially token by token, making it impossible to parallelize training across the sequence. This sequential bottleneck kept these models small.

The groundbreaking paper *Attention Is All You Need* [17] changed the field. Transformers process all tokens in a sequence simultaneously, making it possible to train on massive datasets using GPU parallelism. Scaled to billions of parameters on web data, GPT-3 [2] could do in-context learning, translation, coding, and reasoning without any task-specific training. The scale of the data and model allowed a certain amount of generalization that earlier architectures could not achieve.

The mechanism behind this is the attention module. Each token attends to every other token in the sequence simultaneously, allowing the model to capture long-range dependencies. However, computing attention for a sequence of length n requires $O(n^2)$ time and memory, since every token pair must be considered. This quadratic scaling kept context windows short. GPT-3, for instance, was limited to 2048 tokens.

FlashAttention [5] addressed the memory side of this by computing attention in tiles in on-chip SRAM rather than materializing the full attention matrix in HBM, bringing memory use down to linear. On handling token position, the original Transformer used absolute positional encodings: positions were essentially hardcoded, so models simply broke on sequences longer than they were trained on. RoPE [16] encodes position relative to other tokens rather than absolutely, which generalizes much better to longer sequences, and extensions such as YaRN [13] and LongRoPE [7] pushed this further. Together, these advances enabled the jump to context windows of 100k tokens and beyond.

Long context allowed a world of practical applications for LLMs. Coding agents can now take in entire codebases and reason about them. Long documents such as legal contracts, research papers, and financial reports can be analyzed in a single pass. Retrieval-Augmented Generation (RAG) feeds retrieved documents directly into the model context, allowing LLMs to answer questions grounded in external knowledge that was not part of their training data. These applications share a common bottleneck. LLM inference consists of two stages: the prefill stage, which processes the entire input prompt in a single forward pass, and the decoding stage, which generates output tokens one by one. As context length grows, the quadratic time complexity of attention makes prefill increasingly expensive. Memory was the original wall; FlashAttention removed it but did not address the time complexity: computing attention on long contexts is slow.

A large body of work aims to accelarate attention for long context tasks by exploiting the empirical sparsity of the attention matrix: most tokens only attend strongly to a small local window and a few distant positions, so many token pair interactions do not need to be computed at all. Early work used fixed sparse patterns [3]; more recent methods determine the pattern dynamically based on the input [9, 11, 21].

This thesis exploits the insight that adjacent tokens have similar key vectors, with high cosine similarity between neighboring keys. This redundancy in K is already documented in prior work [18, 14]. We show that the same holds for query vectors: adjacent queries are also highly similar. Unlike the sparsity of the attention matrix, this adjacency redundancy lives in the Q and K matrices before attention is computed, so it can be exploited independently of existing sparse attention methods

or layered on top of them.

We apply this single insight to three prefill acceleration applications:

- **Delta-KV compression.** We compress the key and value matrices along the sequence dimension before the QK multiplication, reducing attention FLOPs proportionally to the compression ratio. We provide custom Triton kernels that turn this into wall-clock speedup, not just a theoretical reduction in FLOPs (Sections 4.1.1–4.1.4).
- **DeltaXAttention.** We apply the same compression on top of a block-sparse method, XAttention, compressing the keys inside the tiles it already selects to accelerate it further (Section 4.2).
- **Delta-based tile scoring (DPXA).** We use the adjacency structure as a tile importance estimator for block-sparse attention, choosing which tiles to compute while sampling far less of each tile than existing estimators (Section 4.3).

2 Preliminaries

2.1 Transformer Overview

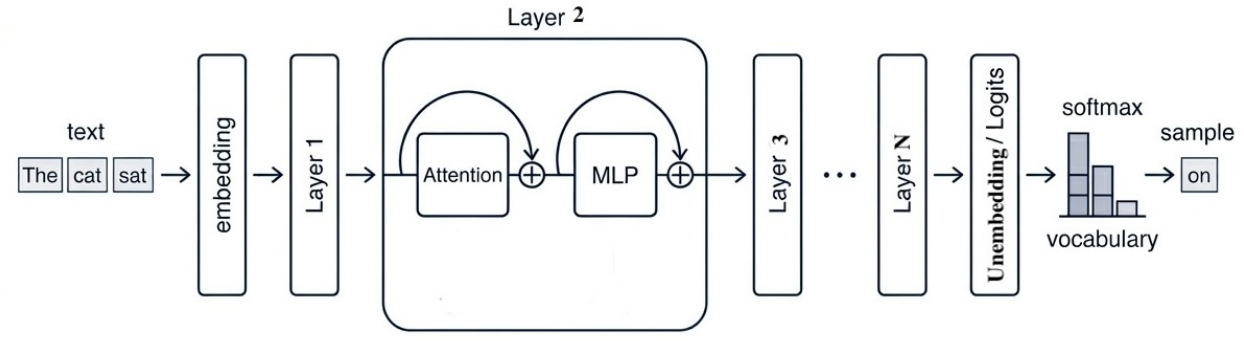

Figure 1: Overview of a Transformer language model. Text is tokenized, each token is looked up in the embedding table, and the resulting vectors pass through N layers, each consisting of an attention module and an MLP. The final embedding of the last token is projected to logits over the vocabulary, a token is sampled, and the process repeats.

A Transformer language model processes text as follows (Figure 1). Text is first split into tokens, and each token is mapped to a vector by looking it up in a learned embedding table. This gives a sequence of vectors of shape $n \times d$, where n is the sequence length and d is the embedding dimension.

These vectors are then passed through a chain of N identical layers. Each layer consists of two modules: an attention module followed by an MLP, each with a residual connection. Through these layers, the embedding of each token is updated to incorporate information from the rest of the sequence.

After the final layer, the embedding of the last token is projected back to the vocabulary size by an unembedding matrix, producing a logit for each vocabulary entry. A softmax turns these logits into a probability distribution, and the next token is sampled from it. This token is appended to

the sequence, and the whole forward pass is run again to generate the token after that. Generation continues until an end-of-sequence token is produced.

2.2 Attention Block

This section explains the inner workings of the attention block shown in Figure 1.

2.2.1 Self-Attention

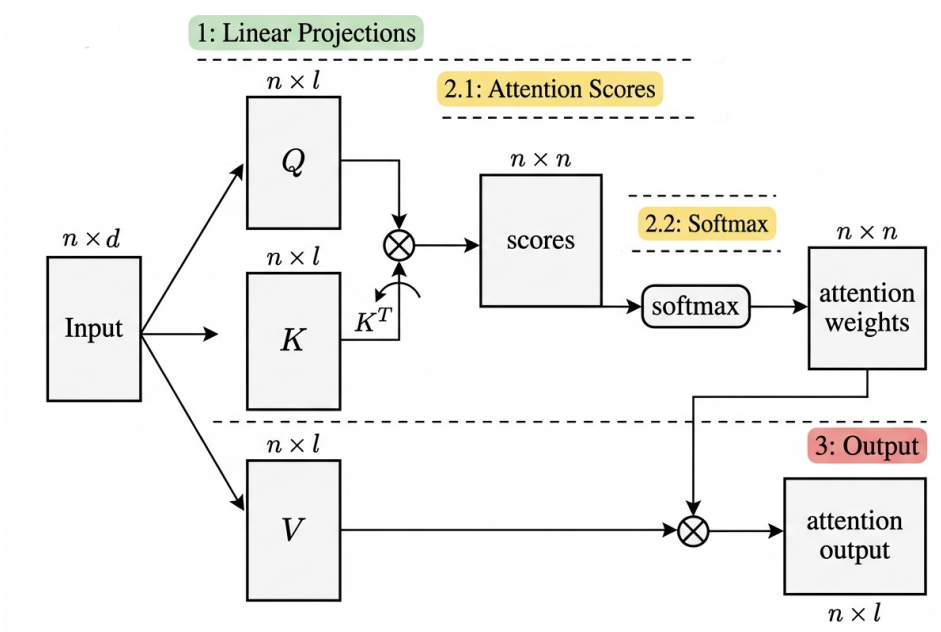

Figure 2: Self-attention mechanism. The input is projected to queries Q, keys K, and values V. The dot product $QK^\top$ gives attention scores, which are normalized by softmax to produce attention weights. The output is a weighted sum of the value vectors.

Self-attention is the operation inside each head (Figure 2). The input hidden state matrix of shape $n \times d$ is projected to three matrices, queries Q, keys K, and values V, each of shape $n \times l$, using learned projection matrices.

The attention scores are computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{l}}\right)V \tag{1}$$

The $QK^\top$ product gives an $n \times n$ matrix where entry (a, b) is the dot product of token a's query vector with token b's key vector. Intuitively, the query of a token encodes what it is looking for, and the key of a token encodes what it has to offer. Their dot product measures how relevant token b is to token a. Dividing by $\sqrt{l}$ keeps the dot products from growing too large before the softmax.

Applying softmax row-wise turns each row into a probability distribution, the attention weights, which say how much each token attends to every other token.

The attention weights are then multiplied with the value matrix V to produce the attention output: each token's output is a weighted sum of all value vectors, where the weights come from its row of the attention matrix. Intuitively, the value vector of a token encodes the information it

contributes to other tokens, and the attention weight determines how much of that contribution is picked up.

2.2.2 Multi-Head Attention

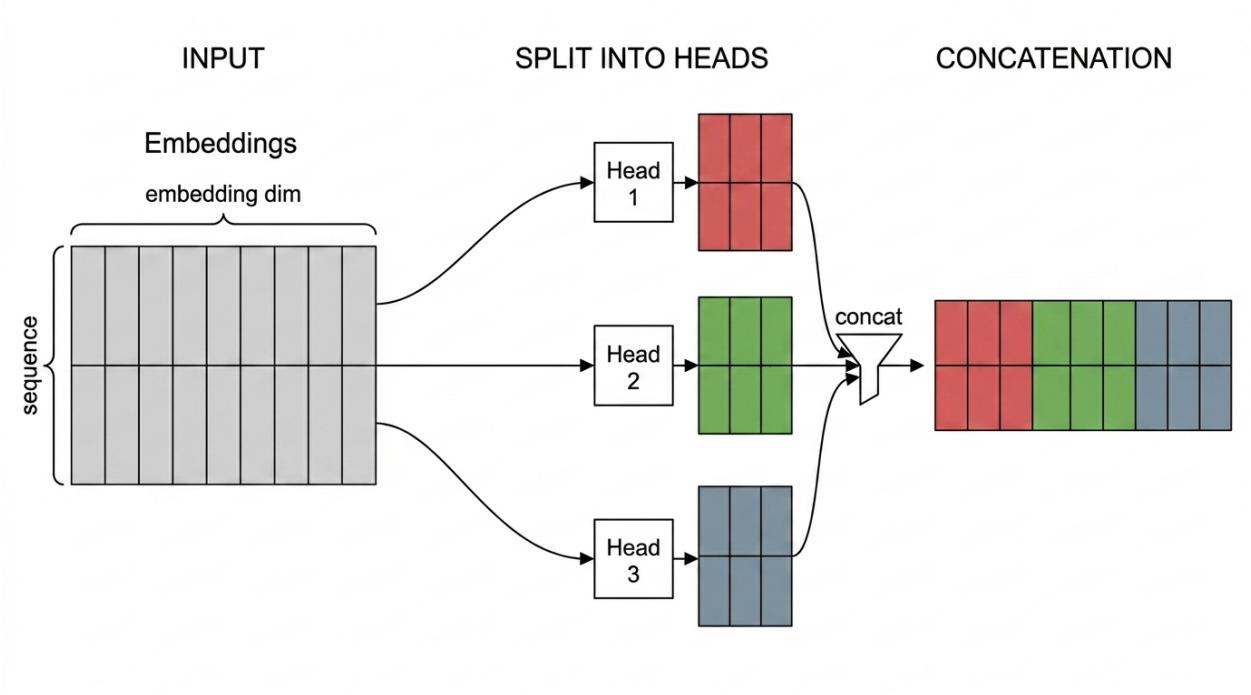

Figure 3: Multi-head attention. The input matrix of shape $n \times d$ is projected by h independent heads, each mapping the full input to a smaller subspace of width $l = d/h$ to produce per-head queries, keys, and values of shape $n \times l$. Each head runs self-attention independently. The outputs are concatenated along the embedding dimension to recover the original $n \times d$ shape.

The attention module inside each Transformer layer is a multi-head attention (MHA) block (Figure 3). Rather than running a single attention computation over the full embedding dimension, MHA runs h attention computations (heads) in parallel, each operating in a smaller subspace of width $l = d/h$. The input to the block is the hidden state matrix of shape $n \times d$, where n is the sequence length and d is the embedding dimension. Each head has its own learned projection matrices that map the full d-dimensional input down to l dimensions, producing per-head queries, keys, and values of shape $n \times l$. Each head sees the full sequence and the full embedding, but through its own projection, different heads facilitate different attention patterns and capture different types of token interactions. The output of each head has the same shape $n \times l$, and the h outputs are concatenated along the embedding dimension to give a final output of shape $n \times d$, matching the input.

2.3 FlashAttention

Materializing the full $n \times n$ attention scores matrix has two critical problems.

The first is memory capacity. The matrix grows quadratically with sequence length: at 10k tokens it is roughly 200 MB per head, at 100k tokens it is around 20 GB per head, and at 1 million

tokens it exceeds 2 TB per head. MHA runs h heads in parallel, so these numbers scale by h, a model like Llama 3 8B uses 32 heads. An H100 has 80 GB of HBM, so at long sequences the scores matrices alone exhaust GPU memory by a large factor.

The second problem is memory bandwidth. Even when the matrix fits, standard attention is bottlenecked by repeated data movement between HBM and the small on-chip SRAM. Computing attention requires multiple passes over the scores: write scores to HBM, read them back for softmax, write the result, read it again for the matmul with V. Each of these round trips is slow, and they dominate the runtime at long sequences.

FlashAttention [5] solves both problems by computing attention in tiles that fit in SRAM. Within each tile, the score computation, softmax, and matmul with V are fused into a single pass, so intermediate results are newer written to HBM. This requires an incremental softmax update to accumulate the correct result across tiles without materializing the full scores matrix. The peak memory footprint drops from $O(n^2)$ to $O(n)$, and the intermediate HBM read/writes are eliminated.

2.3.1 FlashAttention Algorithm

Q, K, and V are divided into fixed-size blocks along the sequence dimension (typically 64 or 128 tokens per block). The outer loop iterates over Q blocks; for each Q block, an inner loop iterates over all K and V blocks in sequence.

For each (Q block, K/V block) pair, three steps happen in SRAM:

- The Q block and K block are multiplied to produce an attention scores tile of shape $B_q \times B_k$, where B_q is the query block size and B_k is the key block size.
- Online softmax is applied to the scores tile. Standard softmax requires the full row to normalize, but online softmax maintains a running maximum and a running sum that are corrected as each new tile arrives, producing numerically identical results without having the full row in memory.
- The attention weights tile is multiplied with the V block to get a partial attention output of shape $B_q \times d$. This is added to a running output accumulator for the current Q block, which collects the contributions from all K/V blocks seen so far.

Once all K/V blocks have been processed, the running accumulator holds the sum of all partial contributions. It is rescaled by the final softmax normalizer from the online softmax and written back to HBM as the attention output for this Q block.

FlashAttention 1 parallelizes this computation over the batch and head dimensions. FlashAttention 2 [4] additionally parallelizes over Q blocks: each kernel instance handles one Q block independently, iterating over all K and V blocks on its own. Since there are no dependencies between Q blocks, this is feasible and enables significantly more parallelism.

2.3.2 FlashAttention Demonstration

Figure 4 shows a FlashAttention 2 program computing the second query block (Q_2), with block size 2, sequence length 8, and head dimension 1. Other programs handle the remaining Q blocks in parallel. The gray upper-triangular blocks of S mark KV blocks skipped entirely: every token in those blocks lies in the future of the corresponding query block.

In iteration 1 (left), Q_2 and K_1 are loaded into SRAM and multiplied to produce scores tile S_{21}. After online softmax, S_{21} is multiplied with V_1 and accumulated into O_2.

In iteration 2 (right), the K and V blocks advance to block 2. Q_2 and K_2 produce tile S_{22}, which is multiplied with V_2 and again accumulated into O_2. The program stops here: blocks K_3 and beyond lie entirely in the future relative to Q_2, so they are skipped due to causality.

After all K/V blocks have been processed, O_2 holds the correct attention output for the tokens in Q_2, rescaled by the final softmax normalizer.

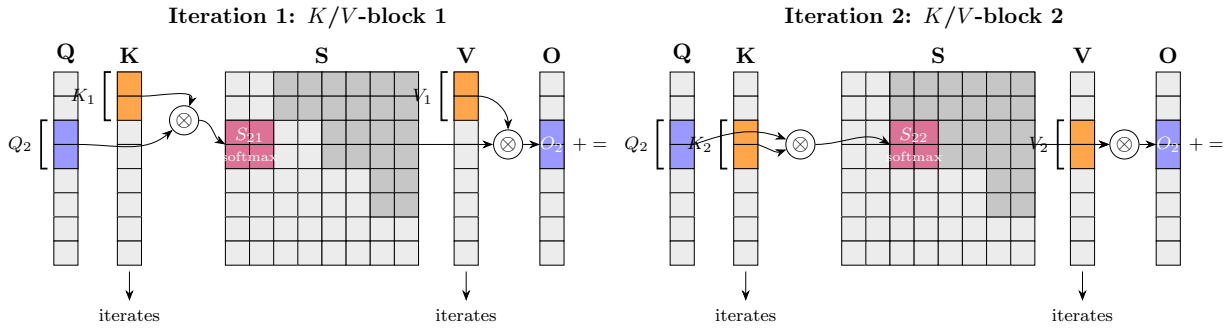

Figure 4: Two consecutive inner-loop iterations for Q-block 2. Left: K_1 and V_1 produce tile S_{21}, accumulated into O_2. Right: K_2 and V_2 produce tile S_{22}, accumulated into O_2.

2.4 Local Attention Window

Attention in LLMs is empirically sparse: tokens strongly attend to their neighbors and a few distant tokens. Xiao et al. [20] identified two regions that receive strong attention. The first is a narrow band along the diagonal, the **critical diagonal**, meaning tokens attend mostly to their neighbors. The second is the first few tokens of the sequence, the attention sink.

Figure 5 shows this for two layers of LLaMA 3.2. Both layers show the same structure: high attention weights on the diagonal and on the initial token.

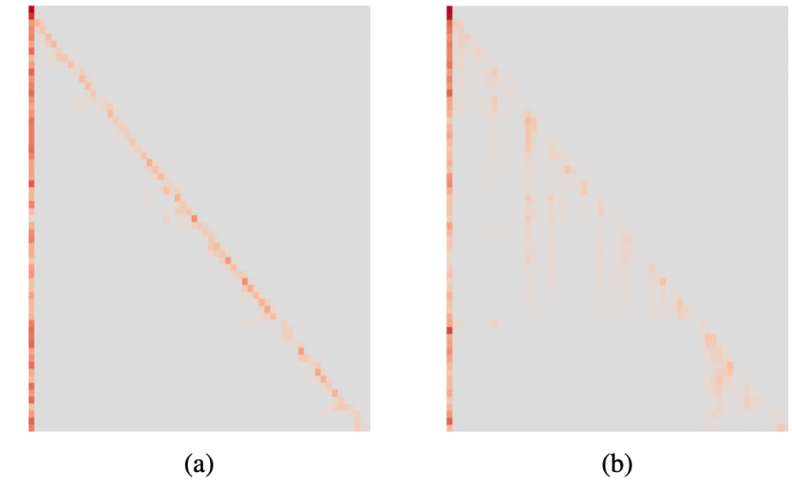

Figure 5: Heatmap of attention scores in two layers of LLaMA 3.2 during prefill. Figure reproduced from Qi et al. [14].

This observation motivates computing the attention exactly for tokens in the critical diagonal in prefill acceleration methods. Jiang et al. [9] have their A-shape attention pattern. Qi et al.

[14] maintain an exact window for the most recent tokens while compressing the rest via delta key vectors. DeltaAttention follows the same principle: the critical diagonal is processed with the full uncompressed keys and values.

2.5 xAttention

XAttention [21] builds on the FlashAttention tiling structure. The key observation is that many tiles contribute little to the final attention output and can be skipped. XAttention predicts which tiles matter before any full attention is computed, producing a binary block mask that is passed to a block-sparse FlashAttention kernel, which skips the zero entries entirely.

Tile importance is estimated using the antidiagonal of the attention score tile. For a query block Q_i and key block K_j, XAttention samples elements along the antidiagonals of the $B \times B$ score tile at a stride S, without computing the full tile. In implementation, the B tokens of each block are partitioned into B/S groups of S consecutive tokens. Within each group, the S per-head token vectors of dimension l are concatenated along the head dimension, producing a single packed vector of dimension Sl (with block size $B = 128$, stride $S = 8$, and head dimension $l = 128$, the packed dimension is 1024). This yields the compressed query representation $\tilde{Q}_i$ of shape $(B/S) \times Sl$, shown in Figure 6. For K_j, tokens within each group are concatenated in reverse order, giving $\tilde{K}_j$ of the same shape.

Their product $\tilde{Q}_i \tilde{K}_j^\top$ has shape $(B/S) \times (B/S)$, where each entry equals the sum of the S scores along the antidiagonal of the corresponding $S \times S$ sub-block of the full score tile (Figure 6). The exponentials of these approximate scores are summed to produce a single tile importance score, estimating the total attention mass the tile would contribute if computed fully. The total cost is $O(B^2/S)$, rather than $O(B^2/S^2)$ for naive subsampling (taking every S-th token without concatenation along the head dimension) or $O(B^2)$ for the full tile.

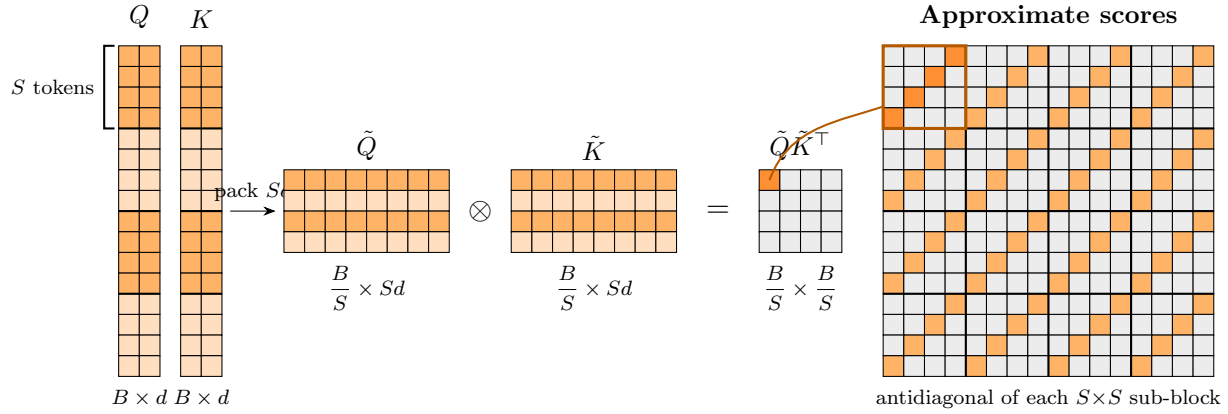

Figure 6: XAttention antidiagonal scoring. Groups of S consecutive tokens are packed along the head dimension ($d \to Sd$), with K groups reversed. Each entry of the $(B/S) \times (B/S)$ product $\tilde{Q}\tilde{K}^\top$ sums the S antidiagonal scores of one $S \times S$ sub-block (arrow).

The tile scores are normalized with softmax across all key blocks for each query block, producing tile weights. Tiles are then selected greedily in decreasing order of weight until the cumulative weight exceeds a threshold τ, yielding a binary mask over the $\lceil N/B \rceil \times \lceil N/B \rceil$ grid of tiles. The threshold τ is tuned per attention head.

Since tile scores are a proxy for true attention mass, τ cannot be set analytically; the same τ value would retain different fractions of true mass in different heads. XAttention calibrates τ

independently per head: a small set of sample sequences is run through the model, and for each head τ is chosen so that the selected tiles cover a target fraction r of the true attention mass. XAttention reports results at stride $S = 8$ and $S = 16$, each with a 32×32 calibrated threshold table covering all layers and heads.

XAttention outperforms both MInference [9] and FlexPrefill [11] on RULER and LongBench while achieving up to 13.5× speedup in attention computation during prefill, with accuracy comparable to full attention.

3 Related Work

Attention complexity and FlashAttention. Self-attention scales quadratically in both time and memory with sequence length, which made long-context inference impractical for early Transformer models. FlashAttention [5] addressed the memory side by computing attention in tiles in on-chip SRAM, avoiding materializing the full attention matrix in HBM. This brought memory use down to linear and cut HBM traffic substantially, making attention practical for sequences up to tens of thousands of tokens. However, FlashAttention is exact (every attention score is still computed), so the quadratic time complexity remains. As sequences grow attention computation becomes the dominant runtime cost and approximate methods become necessary.

Sparse attention for prefill. A natural way to reduce the time cost is to exploit the fact that attention matrices are empirically sparse: most tokens attend strongly to only a small fraction of others. Child et al. [3] were among the first to demonstrate this, observing that trained Transformer attention matrices concentrate on local windows and a few distant positions. They proposed fixed sparse patterns (local windows combined with strided or fixed summary positions) to approximate full attention with fewer FLOPs. The limitation is that a fixed pattern cannot adapt to the content of a given input.

This motivated a shift to dynamic sparse attention, where the sparsity pattern is determined on the fly. Jiang et al. [9] found that LLM attention heads exhibit consistent structural patterns across different inputs (A-shape, Vertical-Slash, and Block-Sparse) and assigned each head its dominant pattern offline, building sparse indices at inference time accordingly. SpargeAttn [22] is a parallel work that takes a more general approach designed to work across diverse model types beyond language models. It selects a representative per block and uses representative-level attention scores to predict which block pairs are near-zero and can be skipped entirely, without assuming any fixed pattern. FlexPrefill [11] and XAttention [21] are more recent works that push the state of the art further. FlexPrefill adapts both the sparsity pattern and compute budget dynamically per head using Jensen-Shannon divergence to classify the pattern type at runtime. XAttention uses the sum of antidiagonal values within each attention block as a lightweight proxy for block importance, achieving aggressive pruning at minimal overhead.

As a parallel line, Qi et al. [14], a concurrent work from our group at LIACS, targets a different kind of sparsity. While the methods above exploit the empirical sparsity of the attention map, they observe that adjacent key vectors are similar, so encoding each key as a delta from the previous one yields a sparse delta key matrix. They then propose a method to compute attention directly using this delta key matrix, reducing the overall computation. However, the delta is applied element-wise, giving unstructured sparsity scattered throughout the delta matrix, a pattern that GPU hardware cannot exploit efficiently, requiring custom hardware to accelerate.

KV cache compression for decoding. A separate line of work targets the memory bottleneck of autoregressive decoding. Decoding is memory-bound: at every generation step the full KV cache must be loaded from HBM, so the KV cache dominates both memory capacity and memory bandwidth. As context length grows, the cache can exceed GPU memory entirely, but even when it fits, a smaller cache directly translates to faster decoding by reducing the amount of data loaded per step. Zhang et al. [24] made the foundational observation that a small set of "heavy hitter" tokens receives the majority of attention mass, and proposed dynamically retaining only those tokens while evicting the rest. Li et al. [12] refined this with a query-aware eviction policy: each attention head observes which tokens are attended to within an observation window at the end of the prompt, and retains only those positions for subsequent decoding. Zhang et al. [23] pointed out that evicting tokens causes irreversible information loss, and proposed merging evicted tokens into their neighbors rather than discarding them. Wang et al. [18] arrived at merging from a different angle: they observed that adjacent token embeddings in the key matrix exhibit high cosine similarity within a sequence, and that this pattern is consistent across datasets and layers. This motivates merging consecutive similar key tokens into a single representative, compressing the cache without throwing away information.

InfLLM [19] spans both prefill and decoding. It offloads the KV cache to CPU memory and selectively loads blocks to the GPU based on predicted attention relevance, using a block-representative scheme to decide which blocks a given query is likely to attend to. Since CPU RAM is far more abundant than GPU VRAM, this effectively extends the usable context window without requiring the full cache on device at once.

Delta-based prefill acceleration. The three methods in this thesis are all training-free prefill acceleration methods built on a single observation: adjacent tokens have highly similar key and query vectors. Each method applies this insight in a different setting, and each has a different closest relative among the methods above.

Delta-KV compression is a direct prefill optimization and is closest to KVMerger [18]. Both compress the KV by merging adjacent keys that are nearly identical. KVMerger does this to shrink the KV cache for decoding, while we compress K at prefill to cut the cost of the QK multiplication, turning the redundancy into structured sparsity along the sequence dimension. To our knowledge this is the first work to use this redundancy as a prefill optimization with structured, GPU-friendly sparsity.

DeltaXAttention asks whether the delta principle can be layered on top of an existing block-sparse method rather than used on its own. We take XAttention [21] and apply delta compression to the keys inside the tiles it selects, compressing within each tile on top of the tiles XAttention already skips.

Delta-based tile scoring (DPXA) is closest to XAttention [21] and SpargeAttn [22]. All three compute a cheap importance score for each tile to decide which tiles to compute exactly. XAttention sums the antidiagonal values of each block, SpargeAttn mean-pools each block into a single representative, and DPXA compresses Q and K to form a proxy for the tile's attention mass.

4 Methodology

This section describes the three methods. We first introduce Delta-KV compression, the core mechanism that compresses the key and value matrices before attention (Section 4.1). We then layer this mechanism on top of a block-sparse method to give DeltaXAttention (Section 4.2), and finally repurpose it as a tile importance estimator for delta-based tile scoring (Section 4.3).

4.1 Delta-KV Compression

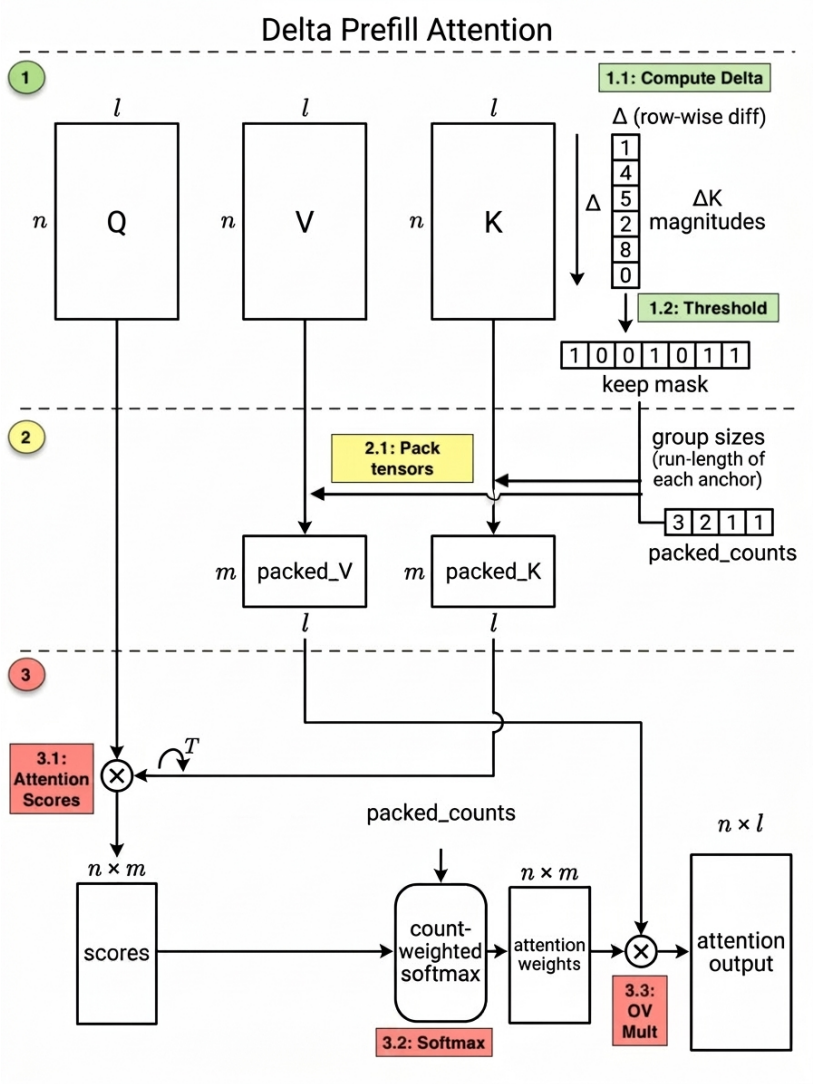

Figure 7: The three stages of Delta-KV compression. Stage 1 (pre-processing) computes row-wise deltas of K and thresholds them into a keep mask. Stage 2 (packing) drops the masked rows to form $\hat{K}$ and $\hat{V}$ of length $m < n$, along with packed_counts, the run length of each anchor. Stage 3 (self-attention) multiplies Q against $\hat{K}$, applies count-weighted softmax, and multiplies by $\hat{V}$ to give the $n \times l$ output.

Delta-KV compression runs in three stages: pre-processing analyzes the key matrix to decide which rows to keep, packing drops the rest to produce the compressed matrices, and self-attention computes the output against the compressed matrices. We then adapt FlashAttention 2 to realize the method on long context.

4.1.1 Stage 1: Pre-processing

The pre-processing stage takes the key matrix K as input and outputs a binary keep mask of length n. It analyzes K row by row to decide which rows to keep and which to drop (steps 1.1 and 1.2 in

Figure 7).

We initialize a keep mask of length n to all ones, meaning all rows are initially kept. The anchor is set to the first row K_0. We then iterate over the remaining rows:

1. Compute the Euclidean distance from row K_i to the current anchor:

$$\delta_i = \|K_i - \text{anchor}\|_2 \tag{2}$$

2. If $\delta_i > \tau$ for some threshold τ: the row is sufficiently different from the anchor, so we keep it and move the anchor to K_i.

3. If $\delta_i \leq \tau$: the row is similar enough that the anchor can represent it, so we drop it by setting keepmask$[i] = 0$ and leave the anchor unchanged.

The output keep mask has a 1 at each anchor position and a 0 at each dropped position.

Crucially, the sequential comparison is performed over chunks of 512 tokens. The sequence is divided into non-overlapping chunks and each chunk is processed in parallel. The first token of each chunk is treated as an anchor, which breaks the sequential dependency between chunks and allows them to be processed in parallel.

4.1.2 Stage 2: Packing

The packing stage takes the keep mask from stage (1) as input and outputs compressed key and value matrices $\hat{K}$ and $\hat{V}$, and a vector packed_counts (step 2.1 in Figure 7).

We drop all rows of K and V where the keep mask is 0, keeping only the anchor rows. The resulting $\hat{K}$ and $\hat{V}$ have shape $m \times l$, where $m < n$ is the number of anchors.

We also compute packed_counts of length m, where packed_counts$[j]$ is the run length of anchor j, the number of rows it represents including itself.

4.1.3 Stage 3: Self-Attention

The final stage takes the original query matrix Q and the packed matrices $\hat{K}$, $\hat{V}$, and packed_counts from stage (2) as input, and outputs the attention output of shape $n \times l$. It follows the same steps as standard self-attention (Section 2.2.1), with two differences: Q is multiplied against the compressed $\hat{K}$ and $\hat{V}$ instead of the full matrices, and the softmax is weighted by packed_counts.

1. **Attention scores** (step 3.1): Multiply Q by $\hat{K}^\top$ to get the score matrix:

$$S = \frac{Q\hat{K}^\top}{\sqrt{l}} \tag{3}$$

   S has shape $n \times m$, where entry (i, j) encodes how much token i attends to anchor j.

2. **Count-weighted softmax** (step 3.2): An anchor representing many tokens should have a proportionally larger effect than one representing a single token. To account for this, we scale each column of S by the corresponding entry in packed_counts before applying the softmax:

$$W_{i,j} = \frac{\text{packed\_counts}[j] \cdot \exp(S_{i,j})}{\sum_{k=1}^{m} \text{packed\_counts}[k] \cdot \exp(S_{i,k})} \tag{4}$$

For example, if token i attends to anchor j with score 5 and anchor j represents 10 tokens, the score is updated to 50 before the softmax. This is equivalent to running standard self-attention where the anchor key vector is copied to all token positions it represents.

3. **Attention output** (step 3.3): Multiply the attention weights by the packed value matrix:

$$\text{output} = W\hat{V} \tag{5}$$

The compressed sequence dimension m disappears after this multiplication, giving an output of shape $n \times l$,identical in shape to the output of standard self-attention.

4.1.4 FlashAttention Adaptation

To test Delta-KV compression on long context tasks we modified Flash Attention 2 to operate on the compressed $\hat{K}$ and $\hat{V}$ matrices. The kernel follows the same structure as FlashAttention 2 (Section 2.3.1): an outer loop over query blocks, with each kernel program handling one query block independently. The inner loop is modified to operate in two phases.

Alongside $\hat{K}$, $\hat{V}$, and packed_counts, we store a `packed_timestamps` vector of length m. Entry j holds the index in the original sequence of the last token that anchor j represents. For example, if anchor j represents tokens 5, 6, and 7 from the original sequence, then `packed_timestamps`$[j] = 7$. This vector can be computed as a cumulative sum of packed_counts, minus one. It lets the kernel track how far into the original sequence each anchor's coverage extends.

Phase 1: Compressed history. The first phase iterates over blocks of $\hat{K}$ and $\hat{V}$ instead of blocks of K. For each packed block, the kernel reads the `packed_timestamps` of the anchors in that block and takes their maximum. This is the reach of the block: the furthest position in the original sequence that any anchor in the block represents.

We define the **critical diagonal** of a query block as the span of positions the block itself covers, [query_block_start, query_block_end]. Any anchor whose reach falls within this span represents tokens that overlap with the current query block and must be processed exactly. This follows the local attention window principle described in Section 2.4: tokens attend most strongly to their immediate neighbors, so those neighbors must be kept exact.

If the reach falls before the critical diagonal (reach $<$ query_block_start), the block is safe to process in compressed form. The kernel computes the attention scores against the packed key vectors and applies count-weighted softmax exactly as in Equation 4, updating the online softmax running state (m_i, l_i, acc) as in standard FlashAttention.

If the reach falls inside or past the critical diagonal, the first phase stops. Those anchors cover tokens that belong to the current query block, so they need to be processed exactly.

Phase 2: Critical diagonal. The second phase switches to the original uncompressed K and V. Starting from just after the last safe anchor's reach and going up to the end of the current query block, it runs standard FlashAttention over the exact token vectors.

4.2 DeltaXAttention

Block-sparse attention methods such as XAttention [21] and SpargeAttention [22] are orthogonal to Delta-KV compression: one reduces the number of tiles computed, the other reduces the cost of each tile. The two can be combined to decrease both computation and data movement. We implement Delta-KV compression on top of XAttention, a strong recent block-sparse prefill method, and call the result DeltaXAttention.

XAttention produces a binary tile mask before the attention computation (Section 2.5). For tiles the mask selects, the standard approach loads the full uncompressed K and V blocks. In our

adaptation, we instead load the compressed $\hat{K}$ and $\hat{V}$ blocks and compute the attention output using count-weighted softmax as described in Section 4.1.3.

To maintain a one-to-one correspondence between tile indices and compressed key indices, the pre-processing and packing stages (Sections 4.1.1 and 4.1.2) are applied per block rather than over the full key matrix. Each tile then has its own independent $\hat{K}$, $\hat{V}$, and packed_counts.

4.2.1 Preserving accuracy

Using packed KV for all non-skipped tiles hurts accuracy. The source of this degradation is the diagonal tiles. As established in Section 2.4, diagonal tiles correspond to local attention between neighboring tokens and carry the most attention mass. Substituting compressed keys for these tiles introduces approximation error where the model is most sensitive.

To address this, we never apply delta compression to diagonal tiles. They are always computed using the full uncompressed K and V.

For off-diagonal tiles, we additionally keep the top 10% by XAttention tile importance score exact, computing them with full K and V as well. The remaining off-diagonal tiles use the packed $\hat{K}$ and $\hat{V}$. The two mechanisms operate together: diagonal tiles and the most important off-diagonal tiles are computed exactly, and everything else uses the compressed representation.

We measure the effect of this approach using a KV reduction metric computed per attention head. KV reduction is the decrease in total KV loads relative to standard XAttention. If a fraction f of tiles are computed with packed KV and the packing reduces those KV matrices to a fraction p of their original size, then the KV reduction is $f(1-p)$. For example, if half the tiles use packed KV and delta packing halves the KV size, the KV reduction is $0.5 \times 0.5 = 25\%$.

4.2.2 Ablation

XAttention can reduce KV loads without delta compression by simply skipping more tiles. Concretely, for a target KV reduction of $\alpha\%$, we skip the $\alpha\%$ least important off-diagonal tiles that XAttention would otherwise compute. This gives the same KV reduction as a DeltaXAttention configuration tuned to $\alpha\%$, but through tile skipping alone rather than key compression.

We compare the two approaches at matched KV reduction. If XAttention tile skipping preserves more accuracy than DeltaXAttention at the same reduction, then applying delta compression on top of XAttention adds no value. This serves as a direct check on whether the combination is meaningful.

4.3 Delta-Based Tile Scoring (DPXA)

XAttention estimates tile importance by compressing Q and K into $\tilde{Q}$ and $\tilde{K}$ of shape $(B/S) \times Sl$ using a fixed stride S (Section 2.5, Figure 6). Groups of S consecutive tokens are concatenated along the head dimension, so the scoring cost is $O(B^2/S)$, a factor of $1/S$ of the full tile. For stride $S = 8$ this is 12.5% of full tile cost.

Delta compression offers a content-adaptive alternative. Rather than sampling every S-th token by fixed stride, we apply delta pre-processing (Section 4.1.1) independently to Q and K to select anchors, giving Q^δ and K^δ of shapes $(\alpha B) \times l$ and $(\beta B) \times l$, where α and β are the keep ratios of Q and K respectively (Figure 8). Crucially, no dimension expansion is performed: the head dimension stays at l. Their product $Q^\delta {K^\delta}^\top$ has shape $(\alpha B) \times (\beta B)$ and costs a fraction $\alpha\beta$ of the full tile. We call this the **scoring ratio**. It has a second, equivalent reading: since each entry of

$Q^\delta {K^\delta}^\top$ is a single cell of the full $B \times B$ attention tile, $\alpha\beta$ is also the fraction of the tile's cells that are sampled to estimate its importance.

This differs from the XAttention approach in two ways. First, Q^δ and K^δ can have different numbers of anchors, since Q and K are compressed independently (Figure 8). Second, each entry of $Q^\delta {K^\delta}^\top$ corresponds to a single dot product between one Q anchor and one K anchor, a single position in the full $B \times B$ score tile, rather than a sum over an $S \times S$ sub-block as in XAttention.

Each entry of the resulting score matrix is exponentiated and summed to produce a tile importance score, which feeds into the same greedy threshold selection as standard XAttention to produce the binary block mask. The attention itself is then computed using the same block-sparse kernel.

Running delta compression on both Q and K with cosine similarity threshold 0.75 gives keep ratios of $\alpha \approx 0.20$ and $\beta \approx 0.20$, yielding a scoring ratio of $\approx 4\%$. This is below XAttention at both stride $S = 8$ (12.5%) and stride $S = 16$ (6.25%), while selecting anchors adaptively based on content rather than at a fixed stride.

As shown in Figure 8 and Figure 6, delta-based scoring uses a different sampling pattern from XAttention. It looks more like grid sampling on the attention score matrix. We want to isolate the effect of content-adaptive sampling from this grid sampling pattern. To do this we add two ablation configurations: naive subsampling at stride 3 and stride 4. By naive we mean no expansion along the head dimension, just sampling every S-th token of Q and K and discarding the rest. This produces a regular grid on the score matrix, similar in shape to delta-based scoring but with a fixed period instead of content-adaptive anchors.

Naive stride 3 gives a scoring ratio of $1/9 \approx 11.1\%$. This is close to XAttention at stride $S = 8$ (12.5%) and delta with cos 0.78 (8.8%), so we use these three together as one comparison group. Naive stride 4 gives $1/16 = 6.25\%$, the same as XAttention at stride $S = 16$ and close to delta with cos 0.75 (5.7%), even though the delta scoring ratio is a bit lower.

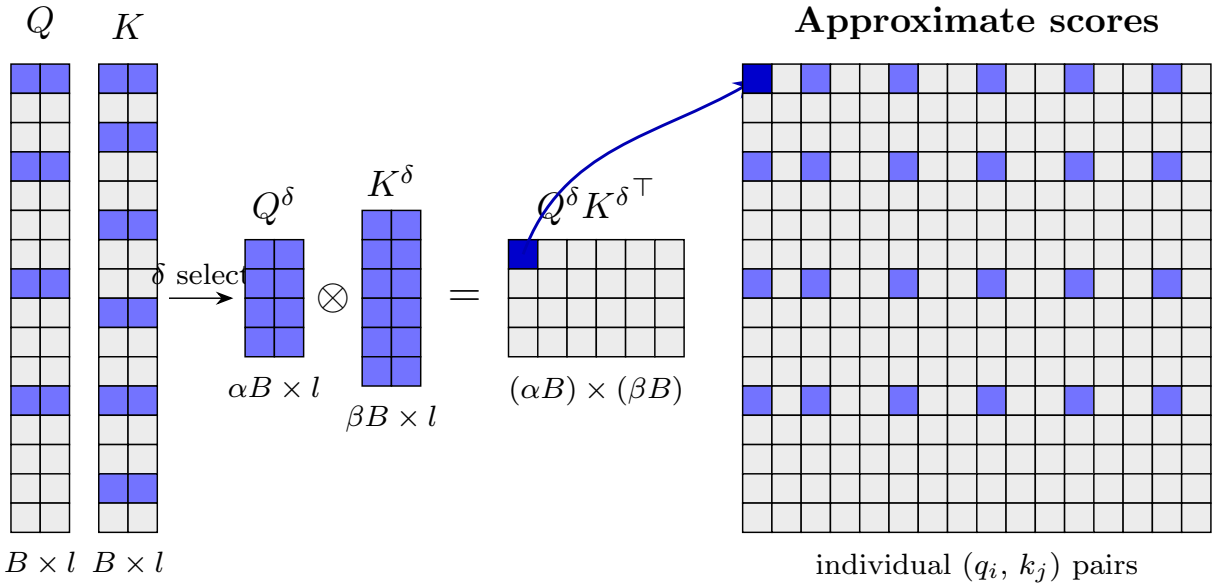

Figure 8: Delta-based tile scoring. Delta pre-processing selects content-adaptive anchors (blue) from Q and K independently, producing Q^δ and K^δ of potentially different sizes — no dimension expansion. Their product gives an $(\alpha B) \times (\beta B)$ score matrix; each entry (highlighted cell, arrow) corresponds to a single dot product at one position in the full $B \times B$ score tile.

5 Experimental Setup

5.1 Benchmarks

5.1.1 Needle in a Haystack and RULER

Benchmarks for long-context transformers are a big research area by themselves. The original Needle in a Haystack (NIAH) test inserts a distinctive sentence at a random position in a long text and asks the model to retrieve it. NIAH only tests retrieval and is a limited evaluation method on its own. Models that do well on NIAH can still fail on real-world tasks [10].

RULER [8] is the modern extension of NIAH, expanding it to 13 more complex tasks such as multi-hop variable tracking and question answering over long contexts. RULER has become a standard benchmark in the field, with MInference [9], FlexPrefill [11], and XAttention [21] all reporting results on it. RULER evaluates models at a range of context lengths, typically from 4K to 128K tokens. DeltaAttention is evaluated on RULER for a direct comparison with existing methods.

Our RULER setup. The full RULER protocol uses 100 samples per task across all 13 tasks, which is very expensive to run, especially when repeated across every method and ablation configuration. We therefore use a reduced setup for all RULER results in this thesis: 30 samples per task across 9 tasks (niah_single_1/2/3, niah_multikey_1/2/3, vt, cwe, fwe) at context lengths 4K, 8K, 16K, 32K, 64K, and 128K. We drop the question-answering tasks from the full set, as these are real-world style tasks already covered by LongBench.

5.1.2 LongBench

LongBench [1] is another standard benchmark used in the field. Unlike RULER, which uses synthetic tasks, LongBench uses real-world documents. It covers 21 datasets across six task categories: single-document QA, multi-document QA, summarization, few-shot learning, synthetic retrieval, and code completion. We evaluate on 9 of these tasks, spanning single-document QA, multi-document QA, and summarization, at 100 samples per task. We drop the synthetic retrieval tasks, since RULER already covers retrieval. This makes LongBench a good complement to RULER, testing whether a method holds up on the kinds of tasks models are actually used for.

5.2 Speedup Experiment

To measure raw attention speedup, we run an isolated prefill test independent of any downstream task, following the latency evaluation in MInference [9]. The model is fed a randomly sampled sequence from WikiText-103 at context lengths from 4K to 512K tokens. We measure the time to first token, which covers the full prefill stage. DeltaAttention is compared against PyTorch SDPA, which dispatches to a FlashAttention 2 kernel, as the dense baseline.

6 Observations

6.1 Key Similarity

High cosine similarity between adjacent key states has been observed and studied in KVMerger [18]. They derive a theoretical lower bound on the cosine similarity between key vectors at positions n and m of $\cos(n-m)$, which puts the minimum similarity of any adjacent pair at $\cos(1) \approx 0.54$. As the distance between tokens grows, the bound decreases toward zero. Since this pattern does not appear in value states, they attribute the phenomenon entirely to RoPE.

In this section we look at the pre-RoPE key states to check whether other factors also contribute. We feed LLaMA-3 8B Instruct a sequence of 512 tokens sampled from WikiText-103 and record key states at layers 0, 8, 16, and 23. At each layer we compute the average cosine similarity between all adjacent token pairs and average over heads. We repeat this for five configurations:

1. **pre-RoPE real**: key states before RoPE, original token order
2. **pre-RoPE shuffled**: key states before RoPE, token positions randomly shuffled
3. **post-RoPE real**: key states after RoPE, original token order
4. **post-RoPE content-shuffled**: input sequence shuffled before keys are computed, RoPE applied in natural positional order
5. **post-RoPE key-shuffled**: pre-RoPE key vectors shuffled, RoPE then applied

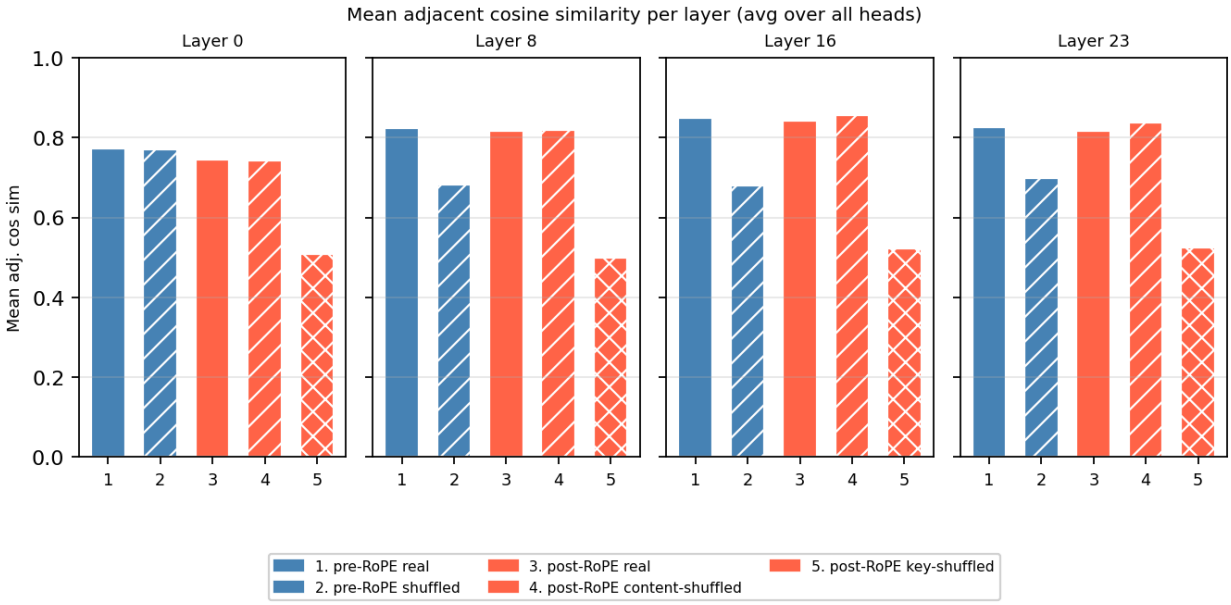

Figure 9: Average cosine similarity of adjacent key states under five configurations for LLaMA-3 8B Instruct at layers 0, 8, 16, and 23.

Configuration 2 in Figure 9 (pre-RoPE shuffled) already shows high cosine similarity. Since the token order is random here, neither positional encoding nor content adjacency can explain it. The key projection itself pushes key states into a narrow region of the key space, raising the global similarity. This is the structure that low-rank KV compression methods exploit, such as EigenAttention [15] and DeepSeek MLA [6].

In mid and deep layers, configuration 1 (pre-RoPE real) in Figure 9 shows noticeably higher similarity than configuration 2. Adjacent tokens in natural text tend to be semantically related, and this is reflected in their key states.

Comparing configurations 1 and 3 in Figure 9 (pre- and post-RoPE on the real sequence), the similarity values are nearly identical. When content is in natural order, applying RoPE does not significantly change adjacent token similarity.

Configuration 4 in Figure 9 shows a different picture. The input content is shuffled before keys are computed, so the key states lack content-driven adjacency. After RoPE is on the shuffled keys, the similarity recovers to nearly the level of configuration 3. RoPE raises the similarity of adjacent positions regardless of the underlying content order.

Key similarity at the token level is therefore a combination of three factors: the key projection creates a high global similarity floor, semantic content raises similarity further for adjacent tokens in natural text, and RoPE contributes a positional adjacency structure on top of that.

6.2 Query Similarity

The same analysis on query states shows that the query projection has the same properties as the key projection. Query vectors show high cosine similarity regardless of token order, which means the projection itself pushes queries into a narrow region of the query space. In later layers, similarity is higher for real sequences than for shuffled ones, suggesting that query vectors pick up contextual information and adjacent tokens in natural text end up closer together. RoPE recovers adjacent similarity for shuffled sequences, contributing a positional structure on top of the content signal.

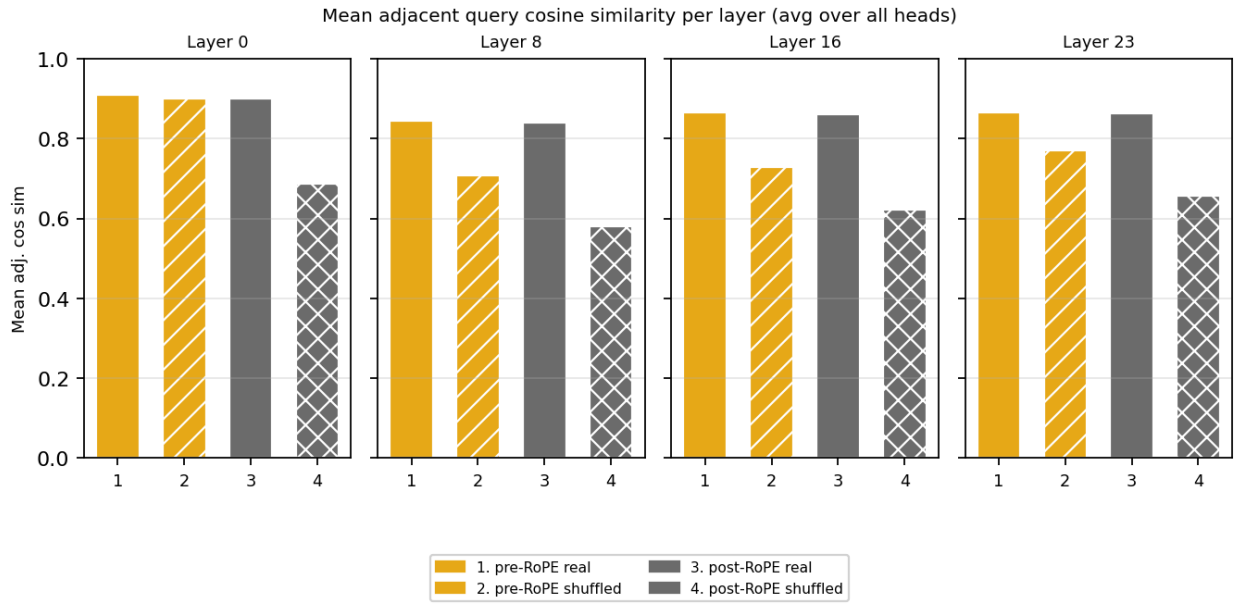

Figure 10: Mean adjacent query cosine similarity per layer (averaged over all heads) for LLaMA-3 8B Instruct at layers 0, 8, 16, and 23, under four configurations: pre-RoPE real, pre-RoPE shuffled, post-RoPE real, and post-RoPE shuffled.

7 Results

We report results for the three methods in turn: Delta-KV compression, DeltaXAttention, and delta-based tile scoring (DPXA). All evaluations use LLaMA-3.1 8B Instruct.

7.1 Delta-KV Compression

7.1.1 RULER

Table 1 reports RULER accuracy on LLaMA-3.1 8B Instruct across context lengths from 4K to 128K tokens. All methods are run using the XAttention codebase and their RULER setup, so the MInference result is the XAttention authors' implementation. DeltaAttention is slightly below XAttention, but the gap is small. Both XAttention and DeltaAttention degrade at 64K and 128K tokens, while MInference retains accuracy at those lengths. MInference outperforms XAttention in our evaluation, which differs from the results reported in the original XAttention paper [21]; we attribute this to dataset versioning differences in the RULER benchmark. Overall, DeltaAttention achieves comparable accuracy to the state-of-the-art methods while reducing KV loads by 49.4%.

Table 1: RULER accuracy (%) on LLaMA-3.1 8B Instruct (100 samples, 13 tasks). Higher is better. KV Red is the average fraction of KV loads saved relative to full attention.

Method	4K	8K	16K	32K	64K	128K	Avg	KV Red
Full attention	96.21	93.78	93.30	90.69	86.27	74.27	89.09	0%
MInference	96.11	94.10	93.51	90.65	86.00	74.42	89.13	–
XAttention	95.59	93.98	92.95	90.44	81.75	71.01	87.62	–
DeltaAttention (cos083)	95.56	93.07	91.68	89.02	82.35	71.08	87.13	49.4%

7.1.2 Prefill speedup

Figure 11 shows the prefill speedup of DeltaAttention at thresholds 15 and 17 against two baselines. The Flash baseline is our own Triton Flash Attention 2 implementation, which is the same kernel DeltaAttention is built on, so this is a direct comparison of the effect of delta packing alone. SDPA is a standard PyTorch Flash Attention 2 implementation included as a common reference point.

At short sequences the overhead of computing the delta pattern costs more than it saves, so DeltaAttention is slower. Around 16K tokens the reduction in K iterations starts to dominate and DeltaAttention crosses above both baselines. At 262K tokens, threshold 15 reaches roughly 1.35x over SDPA and 1.5x over Flash, while threshold 17 reaches roughly 1.8x over SDPA and 2.1x over Flash.

The observed speedup is lower than what the KV reduction alone would predict. Threshold 15 achieves 50% KV reduction (Section 7.1.3), which would suggest a 2x attention speedup, but attention is only one part of the forward pass: MLP layers take a fixed share of the time regardless of sequence length. As sequence length grows, attention increasingly dominates latency (Figure 12), so the end-to-end speedup approaches the theoretical attention speedup. Our experiments on an H100 ran up to 262K tokens, and the trend suggests the speedup would continue climbing toward 2x. At threshold 17, the 65% KV reduction implies a theoretical attention speedup of roughly 3x, but the measured value is 2.1x at 262K for the same reason.

A second factor limiting the realized speedup is the overhead of delta computation and packing (Sections 4.1.1 and 4.1.2). Table 2 shows that at 512 tokens this overhead accounts for 39.1% of total prefill time, dropping to 22.6% at 4K and 8.5% at 16K. The absolute overhead grows slowly with sequence length, from 21ms at 512 tokens to 95ms at 262K, while the total prefill time grows much faster. As a result the ratio falls to 1.4% at 65K tokens and 0.24% at 262K, making it negligible at the sequence lengths where DeltaAttention provides the most speedup.

Final factor is that the DeltaAttention kernel cannot benefit from memory pipelining. Standard FlashAttention iterates over key blocks with a for loop, so the GPU can prefetch the next block's data from HBM while the current block's matrix multiply is still running, hiding most of the memory latency. DeltaAttention uses a while loop to handle the two phases dynamically, and the number of iterations is not known in advance, so the compiler cannot schedule prefetches ahead of time. Each block load must complete before the next iteration can start, leaving memory latency exposed. This is an inherent cost of the adaptive two-phase structure and one of the main reasons DeltaAttention does not fully realize its theoretical speedup.

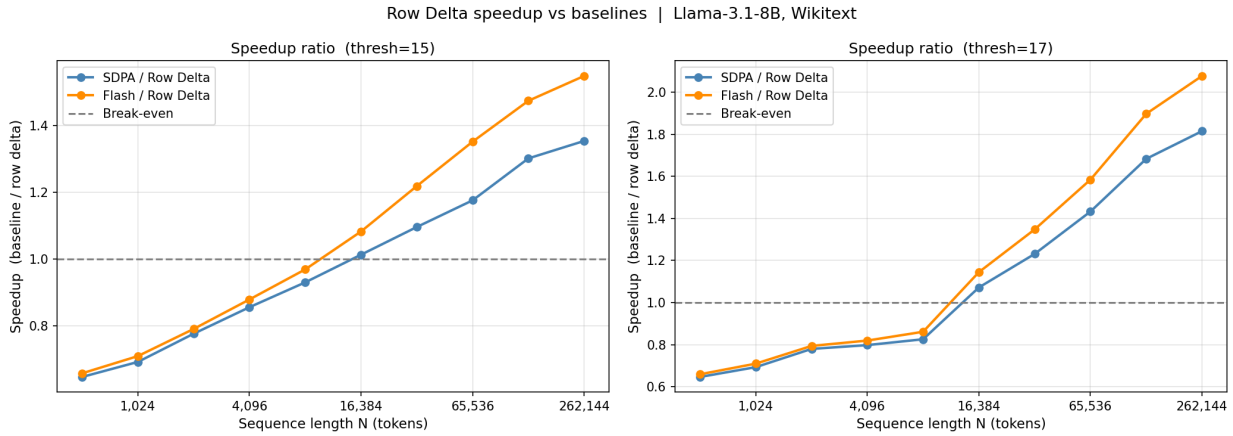

Figure 11: Prefill speedup of DeltaAttention relative to SDPA and Flash baselines at thresholds 15 and 17 on LLaMA-3.1 8B Instruct. Values above 1.0 mean DeltaAttention is faster.

Table 2: Overhead of delta computation and packing for threshold 15 on LLaMA-3.1 8B Instruct. Total prefill time is the full forward pass including all layers. Overhead ratio is the fraction of the total prefill time spent on delta computation.

Sequence length (tokens)	Overhead (ms)	Total prefill (ms)	Overhead / Total
512	21.10	53.91	39.14%
1,024	21.15	57.60	36.71%
2,048	21.41	69.65	30.74%
4,096	21.93	96.93	22.62%
8,192	26.12	164.36	15.89%
16,384	29.37	347.72	8.45%
32,768	34.84	949.27	3.67%
65,536	42.94	3,037.21	1.41%
131,072	57.76	10,358.60	0.56%
262,144	95.41	40,039.47	0.24%

7.1.3 Ablations

We ablate the threshold metric and the main design choices of Delta-KV compression. Unless noted, the design-choice ablations use Euclidean thresholds at three compression levels (t13 32%, t15 50%, t17 65% KV reduction).

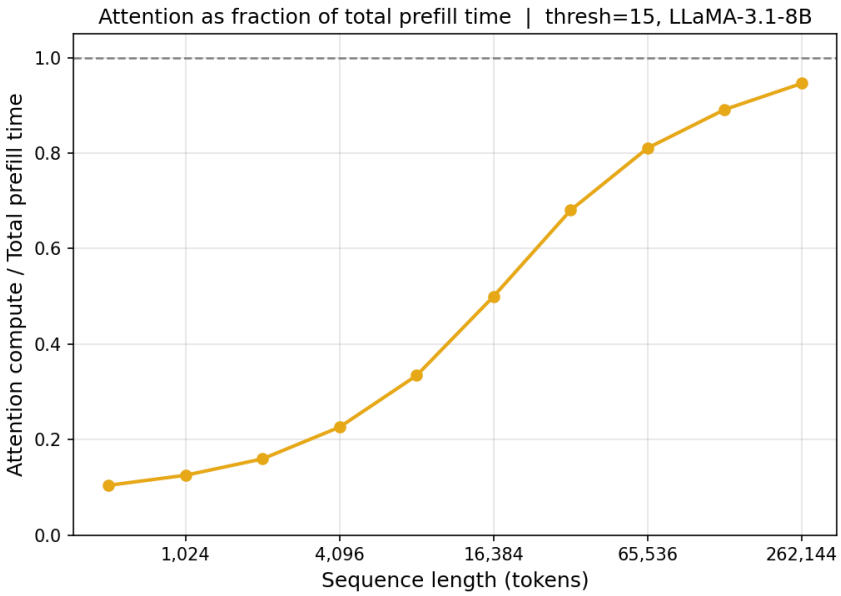

Figure 12: Attention compute as a fraction of total prefill time for DeltaAttention at threshold 15 on LLaMA-3.1-8B. At short sequences attention is a small fraction of total time; at 262K tokens it accounts for 95% of the forward pass.

Threshold metric. Table 3 compares Euclidean distance and cosine similarity as the threshold metric in the pre-processing stage. Pairs of rows are matched by KV reduction level: low (30%), medium (50%), and high (65%). Both metrics perform similarly at low compression, but cosine is consistently better at higher compression, most visibly at 128K tokens where it retains 66.57% accuracy versus 60.57% for Euclidean at 65% KV reduction. Cosine is the better metric overall. We use cos083 (50% KV reduction) as the default operating point: it roughly halves the KV loads with minimal accuracy loss, whereas the 30% setting yields too little speedup and the 65% setting loses more accuracy.

Table 3: RULER accuracy (%) comparing Euclidean and cosine thresholds (30 samples, 9 tasks). Pairs are matched by KV reduction.

Method	4K	8K	16K	32K	64K	128K	Avg	KV Red
Full attention	99.31	97.41	96.52	93.65	88.16	74.35	91.57	0%
Euclidean t13	99.11	96.63	95.63	92.31	86.96	75.19	90.97	32.0%
Cosine cos088	99.38	97.54	96.70	92.31	86.73	72.82	90.91	29.5%
Euclidean t15	97.94	96.05	94.04	91.62	83.77	69.44	88.81	49.7%
Cosine cos083	99.27	96.99	95.11	90.79	83.94	71.43	89.59	50.9%
Euclidean t17	98.27	96.27	93.36	90.45	82.16	60.57	86.85	65.2%
Cosine cos078	99.01	97.12	94.07	86.99	81.27	66.57	87.50	65.8%

Count-weighted softmax. Count-weighted softmax is theoretically necessary for correctness: without it, an anchor representing many tokens receives the same attention weight as one repre-

senting a single token, which underestimates its contribution. With it, the result is equivalent to exact attention where each anchor key is repeated for every token it covers (Section 4.1.3).

Table 4 shows that removing count-weighted softmax has almost no effect on accuracy across all three compression levels. The differences are small. A likely explanation is that the per-anchor counts are similar across anchors in practice, so the scaling barely shifts the softmax distribution. Dropping count-weighting loses no accuracy. We keep it because it maintains theoretical correctness.

Table 4: RULER accuracy (%) with and without count-weighted softmax (30 samples, 9 tasks).

Method	4K	8K	16K	32K	64K	128K	Avg	KV Red
Full attention	99.31	97.41	96.52	93.65	88.16	74.35	91.57	0%
DeltaAttention t13	99.11	96.63	95.63	92.31	86.96	75.19	90.97	32.0%
DeltaAttention t15	97.94	96.05	94.04	91.62	83.77	69.44	88.81	49.7%
DeltaAttention t17	98.27	96.27	93.36	90.45	82.16	60.57	86.85	65.2%
w/o count-weighted softmax t13	98.90	97.31	95.32	94.25	86.62	75.31	91.29	30.9%
w/o count-weighted softmax t15	98.68	95.52	95.00	91.43	81.55	69.74	88.65	48.8%
w/o count-weighted softmax t17	98.62	96.05	94.04	89.72	79.00	67.04	87.41	64.7%

Critical diagonal. The diagonal band of the attention matrix carries the majority of attention mass, as each token attends most strongly to its immediate neighbors (Section 2.4). Phase 2 of the FlashAttention adaptation switches to exact uncompressed attention for the tiles covering this band, so those neighbor interactions are computed precisely (Section 4.1.4).

Table 5 shows that removing phase 2 causes a catastrophic accuracy drop across all compression levels. At t13 the average falls from 90.97% to 57.90%, and at t17 it collapses to 15.19%. Phase 2 is a critical component of DeltaAttention.

Table 5: RULER accuracy (%) with and without phase 2 (critical diagonal) (30 samples, 9 tasks).

Method	4K	8K	16K	32K	64K	128K	Avg	KV Red
Full attention	99.31	97.41	96.52	93.65	88.16	74.35	91.57	0%
DeltaAttention t13	99.11	96.63	95.63	92.31	86.96	75.19	90.97	32.0%
DeltaAttention t15	97.94	96.05	94.04	91.62	83.77	69.44	88.81	49.7%
DeltaAttention t17	98.27	96.27	93.36	90.45	82.16	60.57	86.85	65.2%
w/o phase 2 (critical diag.) t13	74.16	64.04	61.70	58.39	50.36	38.75	57.90	35.7%
w/o phase 2 (critical diag.) t15	54.75	45.15	42.72	37.41	35.44	19.85	39.22	55.0%
w/o phase 2 (critical diag.) t17	17.19	18.44	19.82	17.07	11.85	6.79	15.19	71.3%

Phase 2 only. In this ablation all phase 1 iterations are skipped. Only the diagonal band tiles are computed exactly. Tokens attend only to their immediate neighbors and have no access to the compressed history from phase 1. Table 6 shows accuracy collapse, confirming that the compressed history carried by phase 1 is essential.

The small accuracy increase from t13 to t17 is an artifact of how the critical diagonal is determined. Phase 2 starts as soon as a packed key block's reach overlaps with the current query block's

span. At higher thresholds each anchor represents a longer range, so packed blocks reach further into the sequence and the critical diagonal is entered earlier. This means phase 2 computes more exact tiles near the diagonal at higher thresholds, giving a slight accuracy boost. This is a useful property of the algorithm: higher compression thresholds naturally expand the exact local window, partially compensating for the coarser compressed history.

Table 6: RULER accuracy (%) for phase 2 only (local diagonal, no compressed history) (30 samples, 9 tasks).

Method	4K	8K	16K	32K	64K	128K	Avg	KV Red
Full attention	99.31	97.41	96.52	93.65	88.16	74.35	91.57	0%
DeltaAttention t13	99.11	96.63	95.63	92.31	86.96	75.19	90.97	32.0%
DeltaAttention t15	97.94	96.05	94.04	91.62	83.77	69.44	88.81	49.7%
DeltaAttention t17	98.27	96.27	93.36	90.45	82.16	60.57	86.85	65.2%
phase 2 only t13	12.97	11.01	6.62	3.32	3.53	1.98	6.57	40.2%
phase 2 only t15	14.17	10.62	8.59	4.63	3.86	1.85	7.29	56.0%
phase 2 only t17	17.27	10.44	8.84	5.10	3.78	2.10	7.92	67.9%

Non-adaptive baselines. The phase 2 only ablation showed that the compressed history is essential for retrieval. Here we check whether DeltaAttention's content-adaptive key selection is what makes it effective, or whether any compression scheme at the same reduction works. Random packing randomly drops key vectors to reach a target compression. Periodic packing keeps every nth token and drops the rest, with $n = 2$ giving 50% reduction and $n = 3$ giving 65% reduction. These match the KV reductions of DeltaAttention t15 and t17 for direct comparison.

Both baselines retain some performance, but DeltaAttention is consistently better. At 50% reduction, periodic packing is surprisingly competitive at 87.46% versus 88.81% for DeltaAttention, while random packing falls to 84.22%. At 65% reduction the gap widens: DeltaAttention holds at 86.85%, periodic drops to 81.85%, and random to 75.12%. The content-adaptive selection becomes more important at higher compression.

Table 7: RULER accuracy (%) comparing DeltaAttention against non-adaptive packing baselines at matched KV reduction (30 samples, 9 tasks).

Method	4K	8K	16K	32K	64K	128K	Avg	KV Red
Full attention	99.31	97.41	96.52	93.65	88.16	74.35	91.57	0%
DeltaAttention t15	97.94	96.05	94.04	91.62	83.77	69.44	88.81	49.7%
periodic packing (kr=50%)	97.53	95.01	92.80	90.90	81.62	66.91	87.46	50.0%
random packing (kr=50%)	97.17	92.51	91.22	83.83	75.72	64.88	84.22	50.0%
DeltaAttention t17	98.27	96.27	93.36	90.45	82.16	60.57	86.85	65.2%
periodic packing (kr=35%)	92.12	91.02	86.57	83.19	75.15	63.06	81.85	66.7%
random packing (kr=35%)	89.62	86.61	80.26	73.81	68.08	52.37	75.12	65.0%

7.2 DeltaXAttention

Table 8 evaluates DeltaAttention applied on top of XAttention as described in Section 4.2. Applying delta packing to all tiles XAttention selects results in severe accuracy degradation: 51.39% at dt=13 and 12.81% at dt=17. This mirrors the critical diagonal finding in Section 4.1.4: packing the diagonal tiles approximates the neighbor interactions that carry the most attention mass. Computing diagonal tiles exactly recovers accuracy dramatically, as seen in the diag kept rows.

Adding a second accuracy guard, computing the top 10% of off-diagonal tiles exactly as well, recovers additional accuracy. At dt=15 this configuration achieves 87.06% with 45.6% KV reduction relative to XAttention, a loss of 3.0 percentage points against the XAttention baseline.

The key question is whether XAttention could have achieved the same KV reduction natively by skipping more tiles. At matched reductions, DeltaXAttn tw10 consistently outperforms XAttention tile skipping by a small margin: 88.95% vs 87.51% at 37% reduction, 87.06% vs 85.98% at 46%, and 85.14% vs 84.43% at 57%. However, skipped tiles in XAttention translate directly to fewer FlashAttention iterations, which is straightforward to realize as speedup. DeltaXAttention saves KV loads within each iteration, which is harder to exploit in practice since it does not reduce the number of tiles. We conclude that DeltaAttention can further optimize block-sparse attention methods with minimal accuracy degradation, but its practical value here is limited: XAttention can achieve similar reductions through tile skipping with similar accuracy and a more direct path to speedup.

Table 8: RULER accuracy (%) for DeltaXAttention ablation on LLaMA-3.1 8B Instruct (30 samples, 9 tasks). KV Red is relative to standard XAttention.

Method	4K	8K	16K	32K	64K	128K	Avg	KV Red
XAttention	98.14	97.37	96.62	92.84	84.01	71.49	90.08	0%
XAttn tile skip dr=40%	98.47	95.56	92.63	89.51	81.26	67.64	87.51	37.4%
XAttn tile skip dr=50%	98.72	94.20	91.05	89.43	78.67	63.81	85.98	48.4%
XAttn tile skip dr=60%	97.48	93.86	89.53	88.05	76.83	60.83	84.43	57.6%
DeltaXAttn all packed dt=13	68.00	62.83	59.12	53.72	42.00	22.69	51.39	38.8%
DeltaXAttn all packed dt=15	55.53	48.63	40.59	33.95	28.27	23.86	38.47	54.8%
DeltaXAttn all packed dt=17	19.17	17.14	15.06	12.01	10.04	3.46	12.81	70.1%
DeltaXAttn diag kept dt=13	97.64	95.68	94.01	91.75	77.75	67.61	87.41	36.2%
DeltaXAttn diag kept dt=15	96.24	93.38	92.15	88.63	75.74	61.95	84.68	50.5%
DeltaXAttn diag kept dt=17	95.06	93.46	90.65	84.47	65.72	51.53	80.15	63.5%
DeltaXAttn tw10 dt=13	97.90	96.36	95.21	91.73	82.62	69.90	88.95	32.3%
DeltaXAttn tw10 dt=15	97.99	95.04	94.22	89.57	80.27	65.28	87.06	45.6%
DeltaXAttn tw10 dt=17	97.54	95.19	92.48	87.15	75.62	62.89	85.14	57.0%

7.3 Delta-Based Tile Scoring (DPXA)

We evaluate delta-based tile scoring (DPXA) against a set of strong training-free sparse attention baselines. DPXA is the direct descendant of XAttention, so XAttention at stride 8 and stride 16, both with their r90 calibrated threshold tables, are the most direct comparison points. We also compare against MInference, FlexPrefill, and SpargeAttention, which together cover the main lines of work in training-free sparse attention. All baselines are run from their original GitHub

repositories, with no reimplementation. We report DPXA at two calibration points, cosine threshold 0.75 with r=0.75 and r=0.90 (see the preliminary section), and evaluate on RULER and LongBench together with an isolated speedup analysis.

7.3.1 RULER

Table 9 reports RULER accuracy. The most direct comparison is against XAttention at stride 8 and 16, both calibrated with r90. DPXA retains accuracy substantially better than either XAttention variant. The gap is clearest at 128K tokens, where DPXA reaches 74.79 at r=0.75 and 72.94 at r=0.90, while XAttention drops to 71.49 at stride 8 and 69.53 at stride 16. On average DPXA reaches 91.42 at r=0.90 and 91.30 at r=0.75, both above either XAttention variant (90.40 and 90.08) and close to the SDPA full-attention baseline (91.52). DPXA also stays ahead of MInference (89.55) and FlexPrefill (90.40), and well ahead of SpargeAttention (64.43).

Table 9: RULER accuracy (%) comparing delta-based tile scoring against baseline and block-sparse methods on LLaMA-3.1 8B Instruct (30 samples, 9 tasks). Best score per column in **bold**.

Method	4K	8K	16K	32K	64K	128K	Avg
SDPA	99.31	97.21	96.59	93.76	88.23	74.00	91.52
MInference	99.18	97.25	96.44	91.96	87.21	65.26	89.55
SpargeAttn (adaptive)	93.46	84.68	71.75	58.56	40.99	37.15	64.43
FlexPrefill $\gamma = 0.90$	98.72	97.09	94.63	92.15	86.22	73.61	90.40
XAttn $s = 16$	99.20	**97.57**	96.52	**94.26**	85.30	69.53	90.40
XAttn $s = 8$	98.14	97.37	96.62	92.84	84.01	71.49	90.08
DPXA $\cos 0.75$ r=0.75 (ours)	**99.43**	**97.57**	96.10	93.00	86.90	**74.79**	91.30
DPXA $\cos 0.75$ r=0.90 (ours)	99.38	97.30	**96.90**	94.06	**87.94**	72.94	**91.42**

7.3.2 LongBench

Table 10 shows LongBench scores on LLaMA-3.1 8B Instruct across nine real-world long-context tasks. FlexPrefill's γ parameter controls the same quantity as DPXA's r: the minimum fraction of cumulative attention mass that must be retained per head. At the same operating point (γ=r=0.90), DPXA achieves 39.08 average versus FlexPrefill's 36.50, a gap of 2.6 points, showing that DPXA's delta-based block scoring selects more informative blocks than FlexPrefill's runtime estimation at equal coverage budget. DPXA r=0.90 trails only MInference (39.34) overall, and its 39.08 average is above both XAttention variants (38.28 at stride 8 and 38.37 at stride 16). On the single-document QA tasks NarrativeQA, Qasper, and MultifieldQA-en, DPXA r=0.75 records the best score of any method, exceeding even the full-attention baseline. Compared to the r=0.90 variant, this lighter r=0.75 setting gives up multi-hop accuracy on 2WikiMultiHopQA and MuSiQue, which the higher threshold recovers: moving to r=0.90 improves 2WikiMultiHopQA by 6.6 points and MuSiQue by 2.8 points. Notably, DPXA $\cos 0.75$ r=0.90 scores 49.27 on 2WikiMultiHopQA, exceeding the full-attention baseline (47.97). Single- and multi-document summarization tasks (GovReport, QMSum, MultiNews) are largely unaffected across all sparse variants.

Table 10: LongBench scores on LLaMA-3.1 8B Instruct (100 samples per task). NQA = NarrativeQA, Qas = Qasper, MFQ = MultifieldQA-en, HotP = HotpotQA, 2Wiki = 2WikiMultiHopQA, Mus = MuSiQue, GovR = GovReport, QMS = QMSum, MNews = MultiNews. Best score per column in **bold**.

Method	NQA	Qas	MFQ	HotP	2Wiki	Mus	GovR	QMS	MNews	Avg
Full attention	31.91	43.06	56.25	54.87	47.97	**34.88**	34.47	25.27	**27.15**	**39.54**
MInference	31.74	42.25	56.50	54.70	**50.17**	31.80	34.51	25.27	27.13	39.34
FlexPrefill γ=0.90	26.02	39.00	55.19	**55.11**	38.53	28.73	33.44	25.53	26.93	36.50
XAttn s=8	29.84	41.08	54.43	53.23	47.59	31.01	34.49	**26.37**	26.46	38.28
XAttn s=16	31.01	37.56	56.65	53.35	48.37	32.71	34.83	24.37	26.46	38.37
DPXA $\cos 0.75$ r=0.75 (ours)	**32.32**	**43.50**	**57.13**	53.73	42.66	29.28	35.03	25.38	26.28	38.37
DPXA $\cos 0.75$ r=0.90 (ours)	31.26	42.35	56.76	53.11	49.27	32.12	**35.11**	25.27	26.51	39.08

7.3.3 Speedup and discussion

Table 11 reports an isolated speedup analysis at 131K tokens. The light variants DPXA r=0.75 and XAttention stride 8 reach the same end-to-end speedup of about 2x over SDPA (2.02x and 2.01x). The notable result is that DPXA reaches this while computing fewer attention tiles: DPXA r=0.75 computes 8.5% of the tiles against 10.4% for XAttention stride 8, and DPXA r=0.90 computes 15.7% against 17.1% for stride 16. In both pairs DPXA is the sparser method, which is reflected in its lower attention time (Attn ms). Taken together with the RULER and LongBench results, where DPXA matches or beats XAttention, this means DPXA selects more important tiles: it reaches better or equal accuracy while computing fewer tiles. This is the central argument for DPXA as a higher quality tile importance estimator.

DPXA reaches this estimate while sampling much less of each tile, only 4.1% of the tile for r=0.75 against 12.5% for XAttention stride 8 (Score%). In theory this means a better tile score from far fewer FLOPs. In practice the lower sampling cost does not translate into faster scoring in wall-clock time: the dynamic nature of the computation makes the scores harder to compute efficiently, and the scoring time (Score ms) is slightly higher than XAttention rather than lower, 3278ms against 3073ms for the light pair. So we do not realize a speedup in the scoring phase, though it is not meaningfully slower either. The gain shows up in the attention phase, where the higher sparsity reduces attention time.

DPXA does not reach the end-to-end speedups of MInference, FlexPrefill, or SpargeAttn. SpargeAttn achieves 2.32× with negligible scoring overhead: its tile estimation is extremely cheap, which lets it translate sparsity directly into wall-clock speedup. However, it operates at 47.2% sparsity and suffers significant accuracy degradation, showing that cheap estimation is not the same as accurate estimation. MInference and FlexPrefill are not block-sparse methods. Rather than selecting tiles, they identify important tokens and apply fixed sparse attention patterns at token-level granularity, which is finer than tile-level. MInference uses patterns calibrated before inference and has no per-request selection overhead, reaching 2.48×. It retains accuracy well on LongBench but degrades sharply at 128K tokens on RULER. FlexPrefill adapts patterns at runtime by estimating token importance from the tokens the last query block attends to, reaching the highest speedup at 3.03×. It holds RULER accuracy well across all sequence lengths but drops substantially on LongBench, particularly on 2WikiMultiHopQA and MuSiQue. Looking at the tokens the last query block attends to works for most tasks, but multi-hop reasoning requires attending to context that is not predictable from a local window, and this assumption breaks down there. Notably, DPXA r=0.75 and FlexPrefill γ=0.90 operate at similar sparsities (8.5% and 9.8%), but FlexPrefill selects at token level while DPXA selects at tile level. Even so, DPXA achieves higher accuracy on both

benchmarks, indicating that its content-adaptive tile scoring identifies the informative regions of the attention matrix more reliably.

Table 11: Prefill speedup at 131K tokens on LLaMA-3.1 8B Instruct. Speedup is end-to-end prefill relative to SDPA. Sparsity is the fraction of attention tiles computed. Score% is the fraction of full tile cost spent on tile importance scoring. Score ms and Attn ms are absolute latencies for the scoring and attention phases.

Method	Speedup	Sparsity	Score%	Score (ms)	Attn (ms)
SDPA	1.00×	–	–	–	14,491
MInference	2.48×	2.0%	–	–	3,789
SpargeAttn (adaptive)	2.32×	47.2%	–	–	3,565
FlexPrefill $\gamma = 0.90$	3.03×	9.8%	–	–	2,735
XAttn $s = 16$	1.91×	17.1%	6.2%	2,499	2,738
XAttn $s = 8$	2.01×	10.4%	12.5%	3,073	1,717
DPXA $\cos 0.75$ r=0.75 (ours)	2.02×	8.5%	4.1%	3,278	1,409
DPXA $\cos 0.75$ r=0.90 (ours)	1.81×	15.7%	3.8%	3,191	2,520

8 Conclusion

This thesis applied a single observation, that adjacent query and key vectors are highly similar, to three prefill acceleration methods. We summarize what each one showed.

Delta-KV compression halves the KV with cos083 thresholding while retaining accuracy at a competitive level. The Triton kernels realize an end-to-end speedup, but the speedup is not dramatic. The theoretical maximum is 2x, and we cannot fully reach even that at the sequence lengths we tested. The method demonstrates the strengths of delta packing, but in this direct form the speedup is dominated by existing work. MInference reaches far higher speedups, up to 10x at one million tokens [9, Figure 1b], while the most Delta-KV compression can reach in this setup is 2x. This makes it impractical as a standalone method in this form.

DeltaXAttention halves the KV operations of XAttention while trading some accuracy. However, XAttention can reach the same KV reductions natively, with a similar accuracy trade-off. XAttention translates these reductions directly into speedup because it skips entire blocks. DeltaXAttention does not realize speedups as easily: it does not reduce the number of tiles, which is what XAttention does natively, but instead reduces the number of KV loads and multiplications within each tile. In theory this can lower KV loading and multiplication time, but there is little motivation to build an implementation where these savings turn into real speedup.

DPXA shows the most promising results. The first two methods explore the capabilities of delta compression, and DPXA applies it in a setting where it fits naturally. At 128K tokens DPXA matches the speedup of XAttention while retaining much more accuracy on RULER and matching it on LongBench, and it does so at higher sparsity. Together with the fact that DPXA needs fewer samples from each attention tile, this shows it is a higher quality tile importance estimator. XAttention positions itself as the state of the art in training-free sparse attention, so outperforming it places DPXA in a strong position.

Compared with the other methods, DPXA's realized speedup is lower than SpargeAttn, FlexPrefill, and MInference. SpargeAttn cannot retain much accuracy, so it is not a strong competitor.

MInference reaches about 1.25x more speedup than DPXA at 128K but cannot match its RULER accuracy. In fact DPXA dominates every other method in our RULER setup. FlexPrefill reaches about 1.5x more speedup than DPXA, but it suffers multi-hop accuracy degradation on 2WikiMultiHopQA. These results place DPXA in a strong position in sparse attention research: it leads on accuracy retention, beating the other methods overall. Where it falls behind is realized end-to-end speedup against FlexPrefill and MInference, and the cause is the scoring overhead. DPXA operates at high sparsity, and the isolated attention timings show that its attention computation is dramatically reduced, but the tile importance scoring overhead remains prominent. In theory it samples very few points of each attention tile, around 4%, but the current implementation cannot realize these theoretical numbers because of the dynamic matrix shapes produced by delta packing.

8.1 Future Work

References

[1] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. LongBench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024.

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

[3] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

[4] Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations*, 2024.

[5] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, volume 35, pages 16344–16359, 2022.

[6] DeepSeek-AI. DeepSeek-V2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024.

[7] Yiran Ding, Li Lyna Zhang, Chengruidong Zhang, Yuanyuan Xu, Ning Shang, Jiahang Xu, Fan Yang, and Mao Yang. LongRoPE: Extending LLM context window beyond 2 million tokens. *arXiv preprint arXiv:2402.13753*, 2024.

[8] Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. RULER: What's the real context size of your long-context language models? In *Proceedings of the First Conference on Language Modeling*, 2024.

[9] Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H. Abdi, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. MInference 1.0: Accelerating pre-filling for long-context LLMs via dynamic sparse attention. In *Advances in Neural Information Processing Systems*, 2024.

[10] Yuri Kuratov, Aydar Bulatov, Petr Anokhin, Ivan Rodkin, Dmitry Sorokin, Artyom Sorokin, and Mikhail Burtsev. BABILong: Testing the limits of LLMs with long context reasoning-in-a-haystack. In *Advances in Neural Information Processing Systems*, 2024.

[11] Xunhao Lai, Jianqiao Lu, Yao Luo, Yiyuan Ma, and Xun Zhou. FlexPrefill: A context-aware sparse attention mechanism for efficient long-sequence inference. In *International Conference on Learning Representations*, 2025.

[12] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. SnapKV: LLM knows what you are looking for before generation. *arXiv preprint arXiv:2404.14469*, 2024.

[13] Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. YaRN: Efficient context window extension of large language models. *arXiv preprint arXiv:2309.00071*, 2023.

[14] Jiawen Qi, Chang Gao, Zhaochun Ren, and Qinyu Chen. DeltaLLM: A training-free framework exploiting temporal sparsity for efficient edge LLM inference. *TODO: add arXiv ID or venue*, 2024.

[15] Utkarsh Saxena, Gobinda Saha, Sakshi Choudhary, and Kaushik Roy. Eigen attention: Attention in low-rank space for KV cache compression. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, 2024.

[16] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

[18] Zheng Wang, Boxiao Jin, Zhongzhi Yu, and Minjia Zhang. Model tells you where to merge: Adaptive KV cache merging for LLMs on long-context tasks. *arXiv preprint arXiv:2407.08454*, 2024.

[19] Chaojun Xiao, Pengle Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, Zhengyan Zhang, Zhiyuan Liu, and Maosong Sun. InfLLM: Training-free long-context extrapolation for LLMs with an efficient context memory. *arXiv preprint arXiv:2402.04617*, 2024.

[20] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations*, 2024.

[21] Ruyi Xu, Guangxuan Xiao, Haofeng Huang, Junxian Guo, and Song Han. XAttention: Block sparse attention with antidiagonal scoring. *arXiv preprint arXiv:2503.16428*, 2025.

[22] Jintao Zhang, Chendong Xiang, Haofeng Huang, Jia Wei, Haocheng Xi, Jun Zhu, and Jianfei Chen. SpargeAttention: Accurate and training-free sparse attention accelerating any model inference. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*. PMLR, 2025.

[23] Yuxin Zhang, Yuxuan Du, Gen Luo, Yunshan Zhong, Zhenyu Zhang, Shiwei Liu, and Rongrong Ji. CaM: Cache merging for memory-efficient LLMs inference. *TODO: add arXiv ID or venue*, 2024.

[24] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H2O: Heavy-hitter oracle for efficient generative inference of large language models. In *Advances in Neural Information Processing Systems*, 2023.