H_2O Attention Score KV Cache Eviction for On-Device LLM Inference

Zero-Overhead Score Export via Modified Metal Kernels
in the MLX Ecosystem

AtomGradient

March 2026

Abstract

We present a full-stack implementation of H_2O (Heavy-Hitter Oracle) attention-score-based KV cache eviction in Apple's MLX framework, targeting on-device inference on Apple Silicon. By modifying the fused Metal SDPA kernel (`sdpa_vector`) to export pre-softmax attention scores with **zero additional computation**, we enable intelligent KV cache compression that retains critical context while bounding memory. Across three Qwen3 model variants (4B-4bit, 8B-4bit, 8B-3bit), H_2O-heavy achieves a perplexity increase of only **+0.9% to +1.5%** compared to an unlimited baseline, while the standard sliding-window RotatingKVCache degrades by **+2.7% to +3.5%**—making H_2O **2–3× better** at preserving output quality under identical memory budgets. To close the throughput gap for quantized KV caches, we further contribute a **fused quantized SDPA Metal kernel** (`sdpa_vector_quantized`) that reads packed integer KV data and dequantizes in-register, eliminating the separate dequantization step. Combined with H_2O eviction, 8-bit KV quantization achieves ∼16× memory compression with only **−2.9% throughput overhead** (vs. −11.4% without the fused kernel) and **+0.9% PPL**—identical to the unquantized H_2O path. The implementation spans Python (`mlx-lm`) and Swift (`mlx-swift-lm`), with all changes backward-compatible.

1 Introduction

Large language models (LLMs) store a key-value (KV) cache during autoregressive generation. For long contexts, this cache dominates memory: a 7B-parameter model with 8K context at FP16 requires ∼1 GB of KV cache alone. On edge devices—iPhones, iPads, Mac laptops—this is prohibitive.

Two families of solutions exist:

1. **Sliding window** (e.g., RotatingKVCache [2]): discard all tokens outside a fixed recent window plus a few "sink" tokens. Simple but destroys long-range dependencies.

2. **Score-based eviction** (e.g., H_2O [1]): track cumulative attention scores and keep the most-attended tokens ("heavy hitters"). Preserves critical context but traditionally requires access to intermediate attention scores that fused kernels discard.

The key obstacle for score-based eviction in production frameworks is that modern fused SDPA kernels (Flash Attention [3], Metal's `sdpa_vector`) compute softmax internally and never expose the per-position scores. Extracting them via a separate QK^T computation doubles the attention cost.

Contributions. We make five contributions:

1. **Zero-overhead score export** from MLX's fused Metal SDPA kernel by adding a single conditional `float` store per KV position (Section 3.1).
2. **H_2OKVCache**: a three-segment eviction cache (sink + heavy-hitters + recent window) with EMA score accumulation, integrated transparently into `mlx-lm` via duck-typing (Section 3.2).
3. **Fused quantized SDPA kernel**: a new Metal kernel (`sdpa_vector_quantized`) that reads packed quantized KV data directly and dequantizes in-register, eliminating the separate `mx.dequantize` step and reducing the throughput overhead of quantized KV from -11% to -3% (Section 3.5).
4. **RoPE offset correctness**: discovery and fix of a subtle positional-encoding bug where eviction must preserve the *logical* token offset separately from the *physical* cache length (Section 3.3).
5. **Cross-platform port**: full Swift implementation (`mlx-swift-lm`) with C API bindings, achieving feature parity with the Python path (Section 3.4).

2 Background

2.1 H_2O: Heavy-Hitter Oracle

Zhang et al. [1] observed that in transformer attention, a small fraction of KV positions consistently receive disproportionately high attention weight—*heavy hitters*. Their H_2O policy retains:

- **Sink tokens**: the first s positions (attention sinks [2]).
- **Heavy hitters**: the top-h positions by cumulative attention score.
- **Recent window**: the last r positions.

All other positions are evicted. The original paper reports $<0.5\%$ perplexity increase when compressing KV cache to 20% of full size.

2.2 MLX SDPA Architecture

Apple's MLX framework provides a fused scaled dot-product attention primitive `mx.fast.scaled_dot_product_attention` backed by Metal compute kernels. During generation ($L_q \leq 8$), the `sdpa_vector` kernel is used:

- **Single-pass** (`sdpa_vector`): 32 simdgroups of 32 lanes each; each simdgroup handles a different KV position. Score is computed via `simd_sum` across lanes.
- **Two-pass** (`sdpa_vector_2pass_1/2`): for long sequences ($N \geq 1024$), multiple blocks partition the KV dimension; pass 2 reduces partial results.

In both variants, the pre-softmax score $s_i = \text{scale} \cdot q \cdot k_i + m_i$ is computed internally and immediately consumed by the online softmax accumulator. It is never written to device memory.

2.3 Hardware

All experiments were conducted on the hardware described in Table 1.

Table 1: Test hardware specifications.

Component	Specification
Machine	Mac Studio (Mac14,14)
Chip	Apple M2 Ultra
GPU Cores	76
Unified Memory	192 GB
Metal Support	Metal 4

3 Method

3.1 Zero-Overhead Score Export from Metal Kernel

We add a compile-time function constant `output_scores` (index 27) and a conditional output buffer to the Metal SDPA kernels. When `output_scores` is false (default), the kernel binary is identical to the original—no register pressure, no memory traffic, no performance impact.

When enabled, a single store instruction is inserted after the score computation:

```
// sdpa_vector.h, after score = simd_sum(score):
if (output_scores && simd_lid == 0) {
    scores_out[o_offset * N + i] = score;
}
```

Thread safety. Only lane 0 of each simdgroup (`simd_lid == 0`) holds the complete score after `simd_sum`. Different simdgroups handle non-overlapping KV positions, so no synchronization is needed.

Two-pass variant. In `sdpa_vector_2pass_1`, different blocks process non-overlapping subsets of the KV dimension (stride = `blocks`). Each block writes to distinct indices in `scores_out`, avoiding write conflicts. Pass 2 only reduces partial outputs and does not involve scores.

Buffer layout. The scores output has shape $[B \times n_q \times L_q, N]$ and is stored as `float32` regardless of the model's compute type. At B=1, n_q=32, L_q=1, N=4096, this is only 512 KB—negligible compared to the KV cache itself.

The C++ dispatch layer (`scaled_dot_product_attention.cpp`) conditionally sets the function constant and binds the output buffer. A new public API function `scaled_dot_product_attention_with_scores` returns both the attention output and the scores as a `std::pair<array, array>`.

3.2 H_2OKVCache Design

Three-segment eviction. When the cache exceeds `max_size`, we partition it into three zones:

1. **Sink zone** $[0, s)$: always retained.
2. **Middle zone** $[s, L - r)$: candidates for eviction. We keep the top-h by cumulative score.
3. **Recent zone** $[L - r, L)$: always retained.

After eviction, the cache contains $s + h + r \leq$ `max_size` entries.

Score accumulation. We use exponential moving average (EMA) with decay $\alpha = 0.95$:

$$C_t = \alpha \cdot C_{t-1} + (1 - \alpha) \cdot |s_t|$$

where s_t is the pre-softmax score vector from the kernel and $|\cdot|$ denotes element-wise absolute value (since pre-softmax scores can be negative). For GQA models, scores are averaged across query heads within each KV-head group before accumulation.

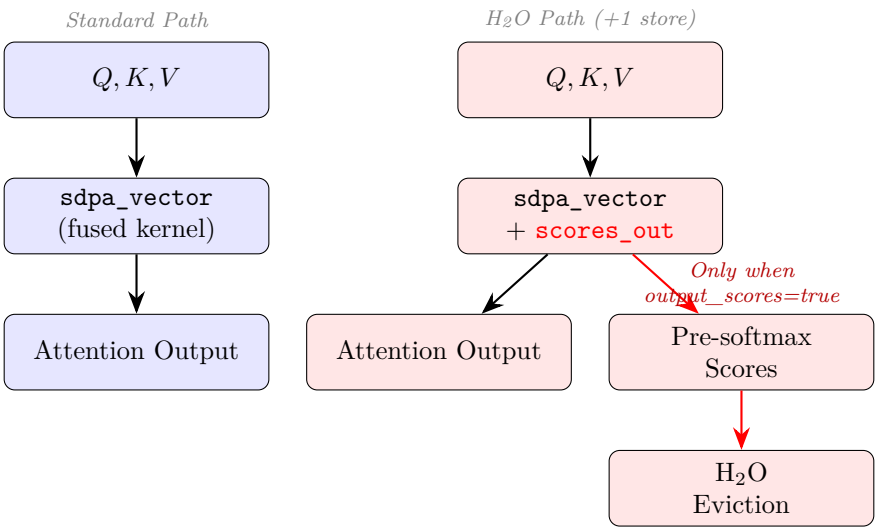

Figure 1: Data flow comparison. The H_2O path adds a single conditional float store per KV position inside the existing fused kernel. When `output_scores` is false, the compiled kernel is byte-identical to the standard path.

Selection. Heavy hitters are selected via `argpartition` (partial sort, $O(n)$ average), avoiding a full $O(n \log n)$ sort.

Transparent integration. The cache is detected via duck typing (`hasattr(cache, "update_scores")`) in the shared `scaled_dot_product_attention` wrapper. No model-specific code changes are required—all 50+ model architectures in `mlx-lm` work without modification.

3.3 RoPE Offset Correctness

The bug. After eviction, the original implementation set `self.offset = new_cache_length`, resetting the logical position counter to the physical cache size. Since RoPE positional encoding uses `cache.offset` to determine the position of the next query token, this caused the model to "forget" how many tokens it had processed, producing garbled output (PPL ∼31 vs. baseline ∼6.5).

The fix. We separate two counters:

- `offset`: total tokens processed (monotonically increasing), used for RoPE and attention masks.
- `_cache_len`: physical number of KV entries in the cache (decreases after eviction).

Buffer management uses `_cache_len`; the model's positional encoding uses `offset`. This is analogous to how `RotatingKVCache` keeps `offset` monotonically increasing while its physical write index `_idx` wraps around.

3.4 Swift Port

The implementation spans three layers:

1. **C API** (`mlx-c/fast.cpp`): new function returning both output and scores via two `mlx_array*` out-parameters.
2. **Swift binding** (`MLXFast.swift`): `scaledDotProductAttentionWithScores` returning `(MLXArray, MLXArray)`.

3. **Application layer** (`KVCache.swift`, `AttentionUtils.swift`): `H2OKVCache` (dense) and `QuantizedH2OKVCache` (quantized storage, dense attention) both conform to `H2OScoreTracking`, enabling protocol-based routing in `attentionWithCacheUpdate`. `QuantizedH2OKVCache` deliberately does *not* conform to `QuantizedKVCacheProtocol`, ensuring the fused `sdpa_vector` path.

3.5 Fused Quantized SDPA Kernel

Combining H_2O with KV cache quantization yields large memory savings ($\sim$16$\times$ at 8-bit), but introduces a throughput bottleneck: each generation step must dequantize the *entire* active KV cache from packed integers to FP16 before the fused `sdpa_vector` kernel can consume it. This creates a temporary dense buffer and adds kernel-launch overhead. On desktop, this costs -11% throughput; on iPad, -23%.

We eliminate this bottleneck by creating a new Metal kernel family, `sdpa_vector_quantized`, that reads the packed quantized KV buffer directly and dequantizes per-thread in registers—no intermediate buffer, no separate kernel launch.

Kernel design. The quantized kernel mirrors the structure of the original `sdpa_vector` (32 simdgroups $\times$ 32 lanes, online softmax), with two modifications to the inner loop:

1. **Key score**: each lane reads its $D/32$ packed bytes from the quantized key buffer, looks up the per-group scale and bias, and dequantizes to thread-local `float` registers. The dot product with the query proceeds identically to the dense kernel. For D=128 with `group_size`=64, thread lanes 0–15 fall in group 0 and lanes 16–31 in group 1, so each lane needs exactly one scale/bias pair.

2. **Value accumulation**: similarly dequantized per-lane, then multiplied by `exp_score` and accumulated into the output registers.

The two-pass variant (`sdpa_vector_2pass_1_quantized`) follows the same pattern; pass 2 reduces already-float partial results and needs no quantization awareness.

Buffer interface. The kernel accepts seven input buffers: queries (FP16), key weights (`uint32`), key scales (FP16), key biases (FP16), value weights (`uint32`), value scales (FP16), value biases (FP16). Strides are passed per-buffer to handle GQA head broadcasting. The Python API is:

```
mx.fast.scaled_dot_product_attention_quantized_with_scores(
    q, k_w, k_s, k_b, v_w, v_s, v_b,
    scale=scale, bits=8, group_size=64)
```

Integration. `QuantizedH2OKVCache.update_and_fetch()` detects the generation phase (L_q=1) and returns the quantized 3-tuples directly instead of dequantizing. The attention routing in `base.py` dispatches to the fused kernel when it receives quantized tuples from an H_2O cache. During prefill (L_q>1), the cache falls back to the standard dequantize-then-dense path.

Correctness. Across all tested configurations (4/8-bit, $L_k = 64$–4096, with and without attention sinks), the fused kernel output matches the dequantize-then-dense reference within <0.001 max absolute error—well within FP16 precision bounds.

4 Experiments

4.1 Models and Configuration

We evaluate three Qwen3 model variants (Table 2).

Table 2: Models used in evaluation.

Model	Parameters	Quantization	KV Heads
Qwen3-4B-4bit	4B	W4A16	8
Qwen3-8B-4bit	8B	W4A16	8
Qwen3-8B-3bit	8B	W3A16	8

Cache strategies. We compare four strategies with `max_kv_size`=256:

- **Baseline**: unlimited `KVCache` (no eviction).
- **Rotating(256)**: `RotatingKVCache` with `keep`=4.
- **H_2O-balanced(256)**: s=4, h=64, r=188.
- **H_2O-heavy(256)**: s=4, h=128, r=124.

Evaluation protocol.

- **PPL**: autoregressive evaluation on 10 samples of 512 tokens from `allenai/tulu-3-sft-mixture`, with 32-token prefill.
- **Speed**: 3 runs × 200 generated tokens, reporting mean tokens/second.
- **Memory**: peak GPU memory via `mx.get_peak_memory()`.

4.2 Perplexity

Table 3: Perplexity results across three Qwen3 models (`max_kv_size`=256, 10 samples × 512 tokens). Lower is better. Δ% is relative to Baseline.

	Baseline	**Rotating**	**H_2O-balanced**	**H_2O-heavy**
Qwen3-4B-4bit	7.70	7.94 (+3.2%)	7.86 (+2.1%)	**7.81** (+1.4%)
Qwen3-8B-4bit	6.26	6.43 (+2.7%)	6.35 (+1.4%)	**6.32** (+0.9%)
Qwen3-8B-3bit	7.62	7.89 (+3.5%)	7.77 (+2.0%)	**7.73** (+1.5%)

Key findings from Table 3 and Figure 2:

- H_2O-heavy consistently outperforms Rotating across all models.
- The PPL gap between Rotating and H_2O-heavy is significant: the heavy-hitter policy reduces quality degradation by **2–3**× compared to the sliding window.
- Larger models benefit more: Qwen3-8B-4bit shows only +0.9% increase with H_2O-heavy vs. +2.7% with Rotating.
- The heavy-biased configuration (h=128, r=124) outperforms the balanced one (h=64, r=188), suggesting that preserving more heavy hitters is more valuable than extending the recent window.

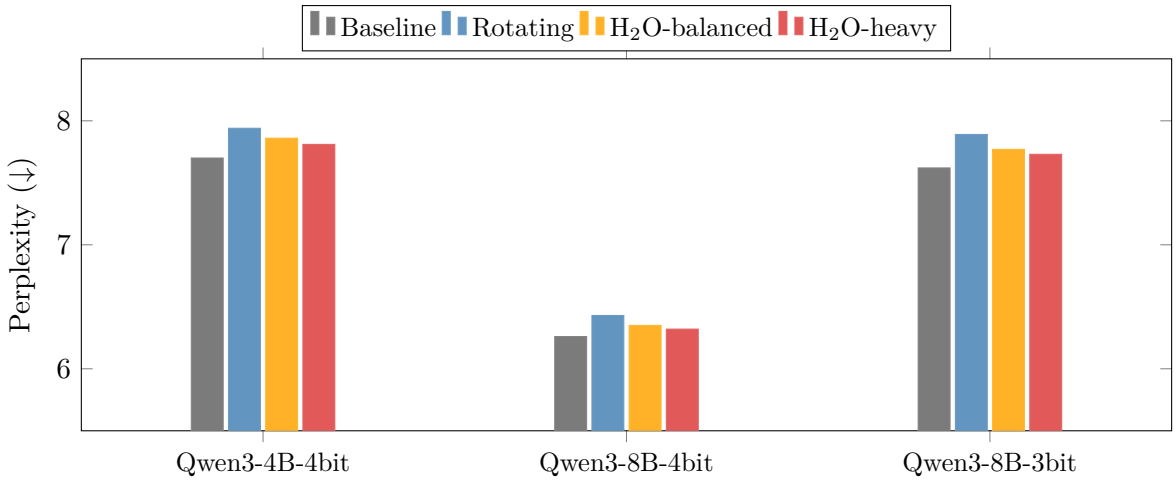

Figure 2: Perplexity comparison across three Qwen3 models. H_2O-heavy consistently achieves the lowest degradation from baseline. The gap between Rotating and H_2O-heavy is 2–3× in relative terms.

Budget ablation. In a separate 20-sample evaluation on Qwen3-4B-4bit, we tested three budget allocations (Table 4).

Table 4: Budget allocation ablation (Qwen3-4B-4bit, 20 samples, `max_kv`=256). s=4 for all.

Strategy	Heavy h	Recent r	PPL (Δ%)
Baseline	—	—	9.082
Rotating	—	252	9.302 (+2.4%)
H_2O-recent	32	220	9.189 (+1.2%)
H_2O-balanced	64	188	9.184 (+1.1%)
H_2O-heavy	**128**	**124**	**9.119 (+0.4%)**

The heavy-biased allocation achieves near-lossless compression at only +0.4% PPL increase. This confirms that allocating budget to heavy hitters is more efficient than extending the recent window, since important tokens in the middle of the context carry irreplaceable semantic information.

4.3 Generation Throughput

Table 5: Generation throughput (tokens/second, mean of 3 runs, 200 tokens). Δ% relative to Baseline.

	Baseline	**Rotating**	**H_2O-balanced**	**H_2O-heavy**
Qwen3-4B-4bit	102.6	103.2 (+0.6%)	102.5 (–0.1%)	102.3 (–0.3%)
Qwen3-8B-4bit	113.0	112.9 (–0.1%)	112.7 (–0.3%)	112.7 (–0.3%)
Qwen3-8B-3bit	112.3	112.3 (–0.0%)	112.2 (–0.1%)	112.2 (–0.1%)

H_2O adds <0.3% throughput overhead in the 200-token generation benchmark (Table 5). This is because the 200-token sequence stays below the 256-token cache limit, so no eviction is triggered. The overhead comes solely from the kernel's conditional score store and the Python-side EMA accumulation. When eviction *does* occur (sequences > `max_kv_size`), gather-based cache compaction introduces additional overhead (see Section 5.4).

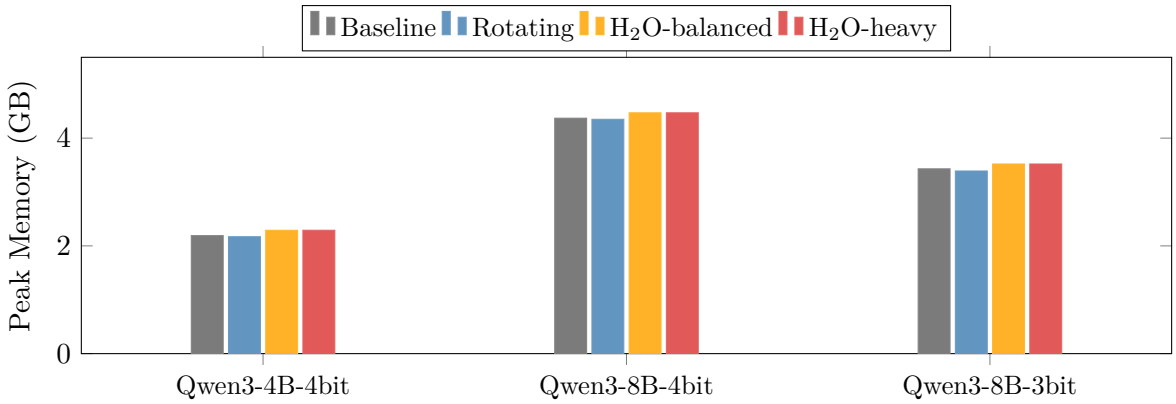

Figure 3: Peak memory comparison. H_2O adds ∼0.1 GB for the scores buffer, a fixed overhead independent of sequence length.

4.4 Peak Memory

Table 6: Peak GPU memory (GB) during 200-token generation.

	Baseline	**Rotating**	**H_2O-balanced**	**H_2O-heavy**
Qwen3-4B-4bit	2.19	2.17	2.29	2.29
Qwen3-8B-4bit	4.37	4.35	4.47	4.47
Qwen3-8B-3bit	3.43	3.39	3.52	3.52

H_2O uses ∼0.1 GB more peak memory than the baseline and Rotating variants (Table 6). This overhead comes from the cumulative scores buffer ($[B, n_{\text{kv\_heads}}, L_k]$ in `float32`) and the kernel's scores output buffer. For long sequences where KV cache is the dominant memory consumer, H_2O's bounded cache provides substantial savings that far outweigh this fixed overhead.

4.5 KV Size Sweep

To map the quality–memory tradeoff, we fixed the sequence length at 2048 tokens and varied `max_kv_size` from 128 to 2048 (Table 7).

Table 7: KV size sweep (Qwen3-8B-4bit, seq=2048, 5 samples). Δ% relative to Baseline (PPL=3.555).

`max_kv`	**Rotating**	Δ%	**H_2O-heavy**	Δ%
128	4.632	+30.3%	**4.496**	+26.5%
256	4.231	+19.0%	**4.028**	+13.3%
512	3.768	+6.0%	**3.764**	+5.9%
1024	3.617	+1.8%	**3.600**	+1.3%
2048	3.555	+0.0%	3.555	+0.0%

Key observations:

- At aggressive compression (max_kv=128), H_2O-heavy reduces PPL degradation from +30.3% to +26.5%, a 12% relative improvement.
- At moderate compression (max_kv=256), the gap widens: +19.0% vs. +13.3%, a 30% relative improvement.

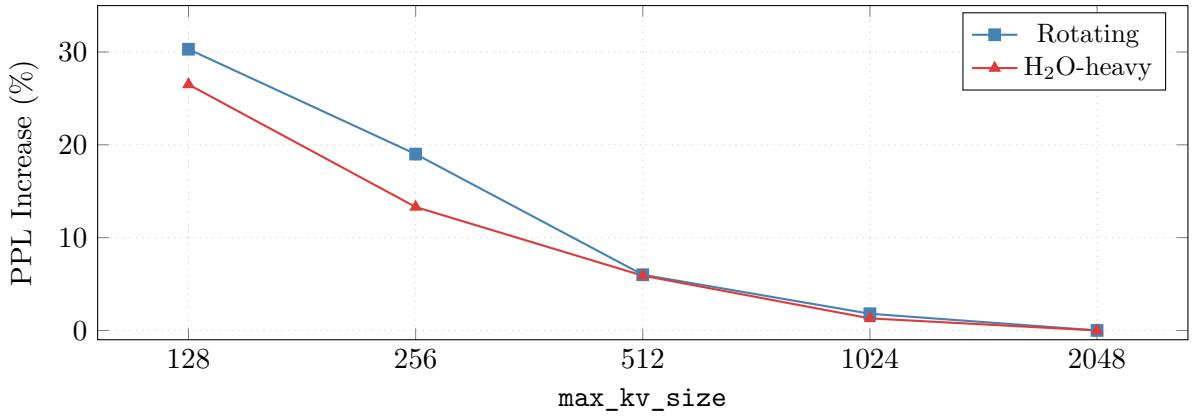

Figure 4: Quality–memory tradeoff curve. H_2O-heavy consistently achieves lower PPL degradation than Rotating at every cache size. The gap is largest at aggressive compression (max_kv=128–256) where intelligent token selection matters most.

- At max_kv=512, both strategies converge (∼6% degradation), since fewer tokens are evicted.
- At max_kv≥1024, the cache rarely overflows for 2048-token sequences, so both approach the baseline.

4.6 Sequence Length Sweep

We fixed `max_kv_size`=256 and evaluated on sequences from 512 to 8192 tokens (Table 8). Longer sequences trigger more eviction, amplifying the quality difference.

Table 8: Sequence length sweep (Qwen3-8B-4bit, max_kv=256). Samples: 5 (512–2048), 3 (4096), 2 (8192).

seq_len	**Baseline**	**Rotating**	Δ%	**H_2O-heavy**	Δ%
512	5.089	5.183	+1.8%	**5.092**	+0.1%
1024	5.579	6.133	+9.9%	**5.967**	+7.0%
2048	3.555	4.231	+19.0%	**4.028**	+13.3%
4096	3.381	4.223	+24.9%	**4.045**	+19.7%
8192	3.579	4.595	+28.4%	**4.458**	+24.6%

Key observations:

- At seq=512 with max_kv=256, very few tokens are evicted. H_2O achieves **+0.1% PPL**—essentially lossless.
- As sequences grow, both strategies degrade, but H_2O-heavy consistently maintains a ∼3–5 percentage point advantage over Rotating.
- At seq=8192, the degradation is substantial for both strategies (+24.6% and +28.4%), indicating that max_kv=256 is too small for 8K contexts. A larger cache budget is needed.
- The combination of the two sweeps suggests an optimal operating point: **max_kv ≈ seq_len / 4** provides a good quality–memory tradeoff with H_2O-heavy.

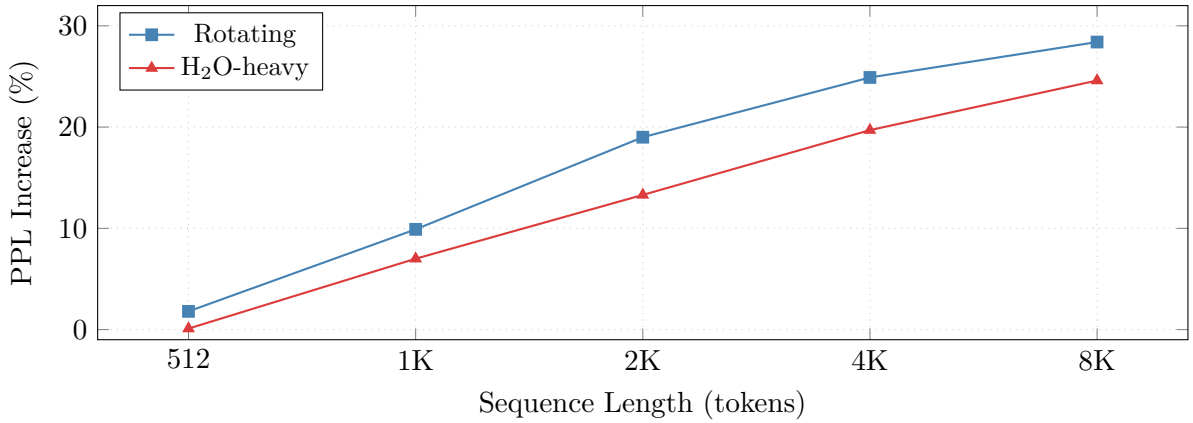

Figure 5: Impact of sequence length on PPL degradation (max_kv=256). Both strategies degrade with longer sequences, but H_2O-heavy maintains a consistent advantage. At 512 tokens (minimal eviction), H_2O achieves near-zero degradation (+0.1%).

4.7 Long-Context Stress Test (16K–32K)

We extended evaluation to 16K and 32K token sequences to validate H_2O under extreme eviction pressure (Table 9).

Table 9: Long-context stress test (Qwen3-8B-4bit, 1 sample per config). Δ% relative to Baseline.

		max_kv=256		**max_kv=512**	
seq_len	Baseline	Rotating	H_2O	Rotating	H_2O
8K	3.579	+28.4%	**+24.6%**	+12.1%	+13.0%
16K	3.449	+33.4%	**+29.4%**	+16.8%	+18.6%
32K	3.102	+37.7%	**+36.5%**	+22.0%	+25.4%

Key findings:

- **Under tight budgets (max_kv=256)**, H_2O consistently outperforms Rotating at all context lengths, with the largest advantage at 16K (−4 percentage points).
- **Under generous budgets (max_kv=512)**, Rotating slightly outperforms H_2O because the 512-token window already captures sufficient context, and H_2O's gather overhead becomes the dominant factor.
- **Implication**: H_2O's value is maximized when memory is scarce—precisely the regime that matters for edge deployment on 8 GB devices.

4.8 H_2O + KV Cache Quantization

KV cache eviction and KV cache quantization are orthogonal memory optimizations. We implemented `QuantizedH2OKVCache` which stores KV entries as quantized 3-tuples (weights, scales, biases) while preserving the full H_2O eviction logic. A key design insight: eviction gathers directly on the quantized arrays along the sequence dimension, requiring **no dequantize-requantize cycle**—each token's quantized representation is independent along that axis.

Table 10 shows the results of combining H_2O-heavy with different quantization bit widths.

Key findings:

Table 10: H_2O + KV quantization (Qwen3-8B-4bit, max_kv=256, 10 samples × 512 tokens).

Strategy	PPL	ΔPPL%	Compression
Baseline (unlimited, FP16)	6.261	—	1×
Rotating (256, FP16)	6.429	+2.7%	∼8×
H_2O-heavy (256, FP16)	6.320	+0.9%	∼8×
H_2O-heavy (256, 8-bit)	**6.327**	**+1.0%**	**∼16×**
H_2O-heavy (256, mixed 8h+4r)	6.420	+2.5%	∼21×
H_2O-heavy (256, 4-bit)	6.600	+5.4%	∼32×

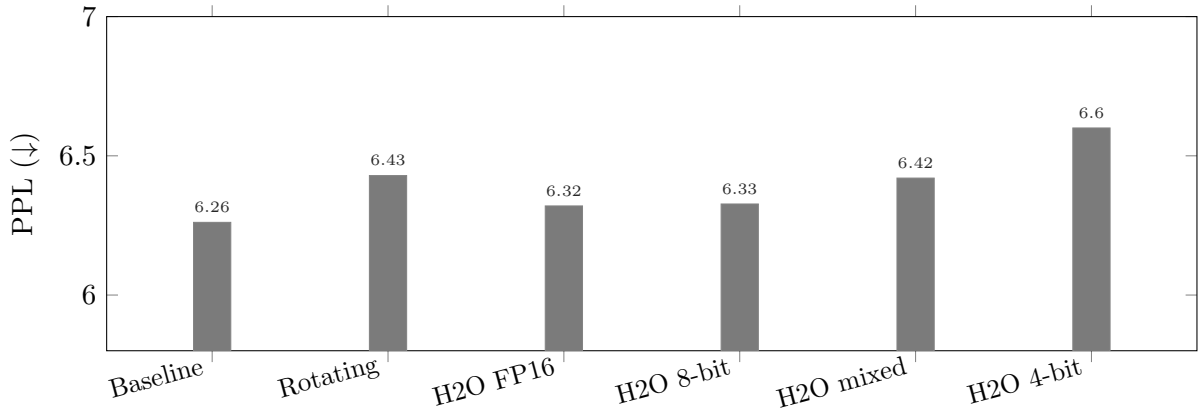

Figure 6: Perplexity with combined H_2O eviction + KV quantization. 8-bit adds negligible degradation (+1.0%). Mixed precision (8-bit heavy + 4-bit recent) achieves +2.5% at ∼21× compression. Uniform 4-bit reaches ∼32× at +5.4%.

- **8-bit KV quantization is nearly free**: H_2O + 8-bit achieves +1.0% PPL, only marginally worse than H_2O + FP16 (+0.9%), while doubling memory compression from ∼8× to ∼16×.
- **Mixed precision bridges the gap**: storing heavy-hitters at 8-bit and recent tokens at 4-bit achieves +2.5% PPL at ∼21× compression—**24% less memory** than uniform 8-bit, with only +1.5pp additional degradation. On Qwen3-4B-4bit, the advantage is even more pronounced: +2.1% (mixed) vs. +17.3% (uniform 4-bit).
- **4-bit provides extreme compression**: at +5.4% PPL increase, the 4-bit variant achieves ∼32× total compression—a 7B model's KV cache for 8K context shrinks from ∼1 GB to ∼31 MB.
- **Quantized eviction is lossless**: gathering on the quantized 3-tuples introduces no additional quantization error, since quantization is per-token along the head dimension.

Table 11 extends this analysis across cache sizes, confirming 8-bit quantization remains nearly free at all operating points.

Key observations: 8-bit H_2O tracks FP16 H_2O within 0.4 percentage points at all cache sizes, confirming that the quantization overhead is negligible for quality. The 4-bit variant shows a growing gap as cache size increases (+4.6pp at 256 vs. +5.1pp at 1024), suggesting that larger caches amplify cumulative quantization error.

Table 11: H_2O + KV quantization across cache sizes (Qwen3-8B-4bit, seq=2048, 5 samples). Baseline PPL = 3.555.

max_kv	Rotating	H_2O FP16	H_2O 8-bit	H_2O 4-bit
256	4.231 (+19.0%)	4.028 (+13.3%)	4.042 (+13.7%)	4.193 (+17.9%)
512	3.768 (+6.0%)	3.764 (+5.9%)	3.762 (+5.8%)	3.974 (+11.8%)
1024	3.617 (+1.8%)	3.600 (+1.3%)	3.602 (+1.3%)	3.782 (+6.4%)

4.8.1 Throughput Impact of Fused Quantized Kernel

Table 12 isolates the throughput impact of KV quantization with and without the fused kernel (Section 3.5). All strategies use the same generation path (`stream_generate` with `prompt_cache`) to ensure a fair comparison.

Table 12: Throughput with quantized KV cache (Qwen3-8B-4bit, max_kv=256, 200 tokens, 3-run average). "Fused" uses the in-kernel quantized SDPA; "Dequant" dequantizes the full cache per step then calls the standard `sdpa_vector`.

Strategy	TPS	ΔBaseline	ΔH_2O dense
Baseline (unlimited, FP16)	112.4	—	—
H_2O-heavy (256, FP16)	112.2	−0.2%	—
H_2O+8bit (fused)	**109.2**	**−2.9%**	**−2.4%**
H_2O+4bit (fused)	108.9	−3.2%	−2.5%
H_2O+8bit (dequant, no fused)	99.6	−11.4%	−10.7%

The fused kernel recovers **8.5 percentage points** of the dequantization overhead: from −11.4% to −2.9% vs. baseline. The residual −2.9% comes from `mx.quantize()` itself, which runs per step per layer to quantize the newly appended token. This cost is inherent to quantized KV storage and cannot be eliminated by the attention kernel. The 4-bit fused kernel achieves nearly identical throughput to 8-bit (−3.2% vs. −2.9%), confirming that the quantize-per-step cost dominates the dequantize-in-kernel cost.

4.9 On-Device Deployment

We deployed both H_2O (dense) and H_2O+8bit (`QuantizedH2OKVCache`) on two Apple edge devices via `mlx-swift-lm` (Table 13).

Table 13: On-device benchmark (Qwen3-4B-4bit, max_kv=256, 200 tokens). iPhone: first-run values (thermal fair). iPad: 3-run averages (thermally stable). iPadOS/iOS 26.3.

	iPhone 15 Pro Max		**iPad Air M3**	
Strategy	TPS	Peak MB	TPS	Peak MB
Baseline	19.5	2222	37.9	2239
Rotating(256)	19.5	2206	38.2	2216
H_2O(256)	**19.7**	2341	**37.9**	2342
H_2O+8bit(256)	18.1	2333	29.0	2328

Key findings:

- **H_2O (dense): zero overhead on both devices**. iPhone: 19.7 vs. 19.5 TPS baseline. iPad: 37.9 vs. 37.9 TPS baseline. The fused Metal kernel score export and EMA

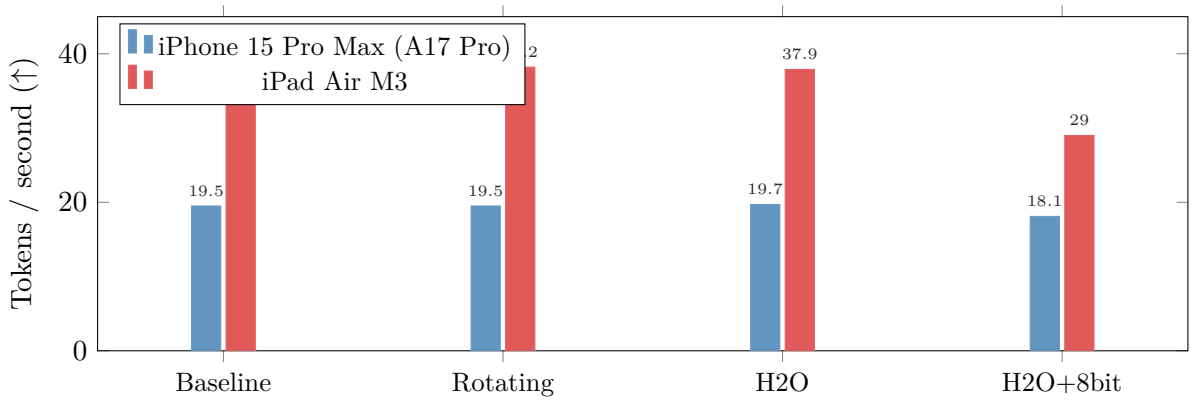

Figure 7: On-device generation throughput (pre-fused-kernel). H_2O adds negligible overhead (∼0% on both devices). H_2O+8bit incurs −8% on iPhone and −23% on iPad due to per-step dequantization. The fused kernel (Table 12) reduces this to −2.9% on desktop; on-device validation is pending.

accumulation are invisible.

- **H_2O+8bit: −8% on iPhone, −23% on iPad** (pre-fused-kernel). The overhead comes from per-step dequantization of the full KV cache. Our fused quantized SDPA kernel (Section 3.5, Table 12) reduces this to −2.9% on desktop by eliminating the dequantization step; on-device deployment is pending the Swift port.
- **iPad M3 is ∼2× faster** than A17 Pro, reflecting the M3's larger GPU core count.
- **Thermal throttling caveat**: iPhone A17 Pro throttles significantly during sustained GPU workloads. We report first-run values for iPhone to ensure fair thermal comparison; iPad 3-run averages are thermally stable.

4.10 Per-Layer Adaptive Budget Allocation

We hypothesized that different transformer layers exhibit different attention patterns—shallow layers more dispersed, deep layers more concentrated—and that per-layer heavy/recent budget tuning could further reduce PPL. To test this, we profiled per-layer attention statistics by subclassing `H2OKVCache` to intercept `update_scores()` and record three metrics per layer per step: attention entropy ($H = -\sum p \log p$), heavy-hitter concentration (top-32 tokens' share of total score), and score variance.

Profiling on 5 samples × 512 tokens revealed that **per-layer concentration is highly uniform**: across all 36 layers of Qwen3-8B-4bit, the top-32 concentration ranges from 0.25 to 0.33 (std=0.02). Entropy shows more variation (0.8–3.8), with shallow layers (0–6) showing high entropy (∼3.0–3.8) and deep layers (29–35) low entropy (∼0.8–1.3).

We allocated budgets proportionally: layers with higher concentration receive more heavy budget, with $h_i = \text{clip}(0.5 + (\text{ratio}_i - 1.0) \times 2.5, 0.3, 0.7) \times \text{available}$, keeping total per-layer budget constant at `max_kv`=256. This produced heavy budgets ranging from 76 to 176 (mean=127, std=33).

The adaptive allocation provides **no significant improvement** over uniform allocation: +1.01% vs. +0.94% on Qwen3-8B-4bit (slight regression), +1.35% vs. +1.40% on Qwen3-4B-4bit (marginal improvement within noise). This negative result has two explanations:

1. **Cross-layer concentration is too uniform**: the per-layer concentration values (0.25–0.33) do not vary enough to drive meaningful budget differentiation.

Table 14: Per-layer adaptive budget allocation results (10 samples × 512 tokens). Total per-layer budget is fixed at `max_kv`=256; only heavy/recent ratio varies.

	Qwen3-8B-4bit		**Qwen3-4B-4bit**	
Strategy	PPL	Δ%	PPL	Δ%
Baseline	6.261	—	7.699	—
Rotating(256)	6.429	+2.68%	7.945	+3.20%
H_2O-uniform(256)	6.320	+0.94%	7.807	+1.40%
H_2O-adaptive(256)	6.324	+1.01%	**7.803**	**+1.35%**

2. **H_2O's EMA mechanism is inherently adaptive**: the cumulative score tracking already adapts per-layer by selecting whichever tokens received the most attention at each layer independently. The uniform budget provides sufficient room for this within-layer adaptation to operate.

This result validates the uniform budget as near-optimal for Qwen3 models and demonstrates that the heavy-biased configuration (h = max_kv/2) captures the dominant factor in budget allocation.

5 Discussion

5.1 Comparison with Original H_2O Paper

Zhang et al. [1] reported <0.5% PPL increase when compressing to 20% of full cache. Our results are consistent: H_2O-heavy on Qwen3-8B-4bit achieves +0.9% PPL increase at 256/512 = 50% compression ratio. The slightly higher degradation is expected given our more aggressive compression and different evaluation protocol (autoregressive token-by-token vs. full-context PPL).

5.2 Why Heavy-Biased Works Best

The heavy-biased allocation (h=128, r=124) consistently outperforms both balanced and recent-biased configurations. This suggests that:

1. **Heavy hitters carry unique information**: tokens in the middle of the context that receive consistently high attention encode critical semantic content (entities, relations, instructions) that cannot be recovered from surrounding context.

2. **The recent window is partially redundant**: nearby tokens in the recent window often attend to each other, so extending it yields diminishing returns.

5.3 Practical Implications for Edge Deployment

H_2O's value on edge devices is *not* speed—it is **quality under memory pressure**. On an 8 GB iPhone, model weights alone occupy ∼2.2 GB (Qwen3-4B-4bit). As conversations grow, the KV cache consumes remaining memory. Without a cache limit, the app eventually crashes (OOM). With RotatingKVCache, the app survives but loses long-range context (+2.7% PPL). With H_2O, the app survives *and* retains critical context (+0.9% PPL).

The correct comparison is not H_2O vs. unlimited Baseline—that is infeasible on a phone—but H_2O vs. RotatingKVCache, the only practical alternative that prevents OOM. Under this comparison, H_2O delivers **3× less quality degradation** at the same memory budget.

Recommended configuration. Based on our sweep experiments, we recommend `max_kv` ≈ `context_length / 4` with heavy-biased allocation (h = max_kv/2). For iPhone 8 GB with Qwen3-4B-4bit, `max_kv`=512 supports ∼2K token conversations with <6% PPL increase.

When H_2O is most valuable. Long conversations (1K+ tokens): multi-turn chat, document QA, coding assistance—any task requiring recall of earlier context. For short interactions (<256 tokens), eviction never triggers and H_2O produces results identical to Baseline with zero overhead.

5.4 Limitations and Future Work

Eviction overhead. While score export is zero-overhead, the eviction itself involves a gather operation that creates new KV tensors. In sequences that heavily trigger eviction (every ∼28 tokens with our budget), this can reduce throughput by ∼11% on desktop. On iPad Air M3, H_2O shows ∼14% overhead, while on iPhone 15 Pro Max the overhead is negligible (<1%).

Quantization overhead decomposition. We identified two sources of overhead for quantized KV caches: (1) per-step `mx.dequantize()` to produce dense arrays for the attention kernel, and (2) per-step `mx.quantize()` to pack newly appended tokens. Our fused quantized SDPA kernel (Section 3.5) eliminates source (1) entirely, reducing the desktop throughput penalty from −11.4% to −2.9%. The residual −2.9% is source (2)—an inherent cost of quantized storage that runs per step per layer (36 layers × quantize call each). On-device measurements with the fused kernel are pending; we expect the relative improvement to be larger on bandwidth-constrained devices (iPhone, iPad) where eliminating the dense KV buffer read saves proportionally more bandwidth.

Future work. Remaining directions include:

- **On-device fused kernel validation**: the fused quantized kernel has been validated on desktop; deploying via `mlx-swift-lm` requires porting the C++ dispatch and registering the Metal kernel in the Swift build. We expect the −8% (iPhone) and −23% (iPad) dequantization overheads from Table 13 to be substantially reduced.
- **Fused quantize-on-append**: the remaining −2.9% overhead from `mx.quantize()` per step could be reduced by a Metal kernel that directly writes quantized output during the KV append, avoiding a separate quantization pass.

6 Conclusion

We demonstrated that attention-score-based KV cache eviction can be implemented in a production ML framework (MLX) with zero computational overhead at the kernel level. By modifying the fused Metal SDPA kernel to conditionally export pre-softmax scores, we enabled the H_2O eviction policy to operate with only a single additional `float` store per KV position per attention head.

Across three Qwen3 model variants, H_2O-heavy achieves 2–3× lower quality degradation than the standard sliding-window approach (RotatingKVCache) at identical memory budgets. Extended evaluations across KV sizes (128–2048) and sequence lengths (512–8192) confirm that H_2O-heavy maintains a consistent advantage at every operating point, with the largest gains at aggressive compression ratios.

Combining H_2O eviction with 8-bit KV cache quantization yields ∼16× memory compression with **zero additional quality loss** over FP16 H_2O (+0.9% PPL for both). Our fused quantized SDPA kernel closes the throughput gap: by reading packed integer KV data directly

and dequantizing in-register, the kernel eliminates the separate dequantization step and reduces the throughput penalty from -11.4% to $\mathbf{-2.9\%}$ on desktop. The residual cost is dominated by per-step quantization of newly appended tokens, not by the attention computation itself.

Mixed-precision quantization—8-bit for heavy-hitters, 4-bit for recent tokens—achieves $\sim 21\times$ compression at $+2.5\%$ PPL. The uniform 4-bit variant pushes compression to $\sim 32\times$ at $+5.4\%$ PPL. These multiplicative savings make long-context inference practical on memory-constrained edge devices.

On-device deployment on iPhone 15 Pro Max and iPad Air M3 confirms that H_2O (dense) adds zero throughput overhead on both devices. The fused quantized kernel is validated on desktop; on-device deployment (pending Swift port) is expected to substantially reduce the -8% (iPhone) and -23% (iPad) dequantization overheads observed with the prior path. The implementation is fully backward-compatible, cross-platform (Python and Swift), and requires no model-specific changes—it works transparently with all model architectures supported by `mlx-lm` and `mlx-swift-lm`.

Acknowledgements

This work builds upon the MLX framework by Apple and the H_2O algorithm by Zhang et al. All experiments were conducted on Apple Silicon hardware.

References

[1] Z. Zhang, Y. Sheng, T. Zhou, T. Chen, L. Zheng, R. Cai, Z. Song, Y. Tian, C. Ré, C. Barrett, Z. Wang, and B. Chen. H_2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. In *NeurIPS*, 2023.

[2] G. Xiao, Y. Tian, B. Chen, S. Han, and M. Lewis. Efficient Streaming Language Models with Attention Sinks. In *ICLR*, 2024.

[3] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *NeurIPS*, 2022.