Layer-Streaming Offloading: Running 9B+ Language Models on 8GB Edge Devices with MLX

AtomGradient

March 2026

Abstract

We present **Layer-Streaming Offloading**, a technique that enables large language models exceeding device physical memory to run inference on Apple Silicon edge devices (iPad, iPhone) with 8 GB unified memory. By loading and releasing model layer weights on demand from NVMe storage during each forward pass, we achieve 60–88% peak memory reduction while maintaining output correctness. We demonstrate this on the Qwen3.5 model family (0.8B–27B parameters) across Mac Studio M2 Ultra, iPad Air M3, and iPhone 15 Pro Max. Our key findings: (1) a 9B model at 6-bit quantization (6.9 GB) causes OOM on 8 GB iPad under standard loading, but **runs at 0.39 TPS with adaptive hybrid residency**; (2) TPS scales linearly with resident layer ratio; (3) streaming overhead is significant—even 100% resident streaming is 36% slower than baseline on iPad; (4) over-aggressive residency causes iOS swap thrashing, dropping TPS by 14×; (5) optimal residency is device-dependent. Our implementation and benchmark data are publicly available.

1 Introduction

Running large language models on edge devices enables private, low-latency inference without cloud dependency. However, state-of-the-art models with billions of parameters produce weight files that exceed the physical memory of mobile devices. An iPhone 15 Pro Max or iPad Air M3 has 8 GB of unified memory, shared between the operating system, applications, and GPU compute. In practice, only approximately 5–6 GB is available for model weights after accounting for iOS system overhead.

This creates a hard boundary: models with weights exceeding ∼5 GB cannot be loaded using standard full-resident approaches. The conventional response is to use smaller models or more aggressive quantization, sacrificing quality for deployability. We propose an alternative: **layer-streaming offloading**, which loads model weights one layer at a time from NVMe storage during inference, keeping only a subset of layers resident in memory at any time.

Our contributions:

1. A **layer-streaming mechanism** for MLX that reduces peak memory by 60–88% with zero impact on output correctness.
2. **Adaptive hybrid residency** that auto-computes optimal resident layers based on available memory, model size, and KV cache needs, with dynamic shedding under memory pressure.
3. **Real device measurements** on iPad Air M3 and iPhone 15 Pro Max, including an **honest assessment of limitations**: streaming overhead, swap thrashing at over-allocation, and prediction model inaccuracy.
4. **Verification** that edge device inference is bandwidth-bound: iPad/iPhone TPS ratio = 1.90–1.93× matches the theoretical 2× memory bandwidth ratio.

2 Background

2.1 Hardware

We evaluate on three Apple Silicon devices spanning the performance spectrum:

All three devices use Apple's unified memory architecture (UMA), where CPU and GPU share the same memory pool. This eliminates PCIe transfer overhead but means GPU memory pressure directly competes with system memory.

Table 1: Test hardware specifications.

Device	Chip	GPU Cores	Memory	Bandwidth	NVMe
Mac Studio	M2 Ultra	76	192 GB	∼800 GB/s	∼7.4 GB/s
iPad Air M3	M3	10	8 GB	∼100 GB/s	∼2 GB/s
iPhone 15 Pro Max	A17 Pro	6	8 GB	∼50 GB/s	∼2 GB/s

2.2 MLX and Weight Loading

MLX is Apple's machine learning framework optimized for Apple Silicon. We discovered that MLX uses standard file I/O (`read()`/`lseek()`) rather than memory-mapped I/O for loading safetensors weight files. All weights must be fully resident in memory before inference. There is no built-in lazy loading or streaming mechanism, making application-level layer-streaming necessary.

2.3 Model Family

We use the Qwen3.5 model family, a hybrid architecture combining GatedDeltaNet (linear attention) and standard softmax attention at a ratio of 3:1 (`full_attention_interval`=4). This provides a uniform layer structure ideal for studying streaming behavior. Models range from 0.8B to 27B parameters at quantization levels from 4-bit to bf16.

The hybrid architecture has an important implication for memory budgeting: only 1/4 of layers maintain full KV cache, while 3/4 use constant-size recurrent state, reducing KV memory to ∼25% of a pure-attention model.

3 Method

3.1 Layer-Streaming Mechanism

Algorithm 1 describes the streaming forward pass. For each decoder layer, weights are loaded from a pre-split safetensors file, the layer's forward computation is executed, outputs are synchronized, and weights are replaced with tiny placeholder arrays to release GPU memory.

Algorithm 1 Layer-Streaming Forward Pass

Require: Input tokens x, layer files $\{f_0, f_1, \ldots, f_{L-1}\}$, resident set $\mathcal{R}$
$h \leftarrow \text{EmbedTokens}(x)$ {Always resident}
masks $\leftarrow \text{CreateMasks}(h, \text{cache})$
for $i = 0$ **to** $L - 1$ **do**
  if $i \notin \mathcal{R}$ **then**
    Load weights from f_i into layer i {NVMe → GPU}
    `mx.eval(layer[i].parameters())`
  end if
  $h \leftarrow \text{Layer}_i(h, \text{mask}, \text{cache}[i])$
  `mx.eval`$(h, \text{cache}[i])$ {Force computation before unload}
  if $i \notin \mathcal{R}$ **then**
    Replace layer i weights with placeholders
    `mx.clear_cache()`
  end if
end for

return LMHead(Norm(h))

3.2 Weight Pre-Splitting

We preprocess each model into per-layer safetensors files: `non_layer.safetensors` (embedding, norm, LM head—always resident) and `layer_XXXX.safetensors` (per-layer, loaded on demand). Vision tower weights from multimodal checkpoints are excluded. For Qwen3.5-9B-6bit, this produces 33 files: 1 non-layer (1.54 GB) and 32 layers (∼168 MB each).

3.3 Partial Layer Residency

For a memory budget B, non-layer overhead O, and per-layer weight size w, the maximum number of resident layers is:

$$N_{\text{resident}} = \left\lfloor \frac{B - O}{w} \right\rfloor \tag{1}$$

Only $L - N_{\text{resident}}$ layers are streamed. We evaluate three placement strategies: `first_n`, `last_n`, and `interleaved`.

3.4 Adaptive Hybrid Residency

Rather than requiring a manually-specified budget, we introduce an adaptive algorithm that auto-computes the optimal resident count:

$$N_{\text{resident}} = \left\lfloor \frac{M_{\text{avail}} - O - w - R_{\text{runtime}} - \text{KV}_{\text{reserve}}}{w} \times 0.9 \right\rfloor \tag{2}$$

where M_{avail} is `os_proc_available_memory()`, O is non-layer weight size, w is average layer size, $R_{\text{runtime}} = 150\,\text{MB}$, and $\text{KV}_{\text{reserve}}$ depends on priority mode:

- **maxTPS**: KV reserve for 512 tokens (maximize resident layers)
- **balanced**: KV reserve for 1024 tokens
- **maxContext**: KV reserve for full expected context

A 0.9 safety factor prevents over-allocation. During generation, memory pressure is checked every 64 tokens; if available memory drops below 300 MB, 25% of resident layers are shed from highest index first. iOS memory warnings trigger full emergency shedding.

4 Experiments

4.1 Baseline: Full-Resident Performance

Table 2: Mac Studio M2 Ultra baseline (full-resident, 200 tokens, 3 runs averaged).

Model	Quant	Layers	PPL↓	TPS↑	Mem (GB)
Qwen3.5-0.8B	8-bit	24	6.185	302.7	0.80
Qwen3.5-0.8B	bf16	24	6.185	219.2	1.47
Qwen3.5-2B	6-bit	24	5.398	214.3	1.49
Qwen3.5-2B	8-bit	24	5.361	203.0	1.93
Qwen3.5-2B	bf16	24	5.366	129.4	3.58
Qwen3.5-4B	4-bit	32	4.959	148.1	2.34
Qwen3.5-4B	6-bit	32	4.759	117.1	3.32
Qwen3.5-4B	bf16	32	4.746	66.2	7.94
Qwen3.5-9B	6-bit	32	4.485	77.9	6.90
Qwen3.5-9B	8-bit	32	4.455	67.3	8.97
Qwen3.5-27B	4-bit	64	4.010	35.0	14.34

4.2 Memory Reduction with Streaming

Table 3: Memory reduction with full layer-streaming (0% resident layers).

Model	Normal (GB)	Streaming (GB)	Reduction	Fits 8 GB?
Qwen3.5-0.8B-8bit	0.74	0.29	60%	Yes
Qwen3.5-4B-4bit	2.34	0.47	80%	Yes
Qwen3.5-9B-8bit	8.86	2.29	74%	Yes
Qwen3.5-27B-4bit	14.09	1.70	88%	Yes

4.3 TPS vs. Residency Tradeoff

Table 4: Streaming TPS with partial residency on Mac Studio (50 tokens, Qwen3.5-9B-8bit).

Config	Resident	TPS	Mem (GB)	I/O %
Full Resident	32/32	68.3	8.97	0%
Hybrid 75%	24/32	5.9	7.44	70%
Hybrid 50%	16/32	3.2	5.73	77%
Hybrid 25%	8/32	2.2	4.02	80%
Stream 0%	0/32	1.7	2.31	81%

4.4 Strategy Comparison

For architectures with uniform layer sizes (like Qwen3.5), the placement strategy is irrelevant. All three strategies (first_n, last_n, interleaved) perform within 3% at 50% residency, confirming that only the *count* of resident layers matters.

4.5 Device Baselines and Streaming

Table 5: Decode TPS on real 8 GB devices (baseline, full-resident loading).

Model	Weights	iPad M3	iPhone A17	Ratio	Status
0.8B-8bit	0.80 GB	103.2	53.5	1.93×	Fits
2B-8bit	1.93 GB	43.8	23.0	1.90×	Fits
4B-4bit	2.34 GB	35.5	18.4	1.93×	Fits
4B-6bit	3.32 GB	25.2	13.2	1.91×	Fits
9B-6bit	**6.90 GB**	**OOM**	—	—	**Crash**

The 9B-6bit model (6.90 GB) crashes with OOM on iPad M3, killed by the OS after 24.7 s. Despite weights being smaller than 8 GB physical memory, iOS system overhead of ∼2–3 GB leaves only ∼5–6 GB available. **Layer-streaming is necessary for all models with weights > 5 GB on 8 GB devices.**

Table 6: Layer-streaming on real 8 GB devices (0% resident, 10 tokens).

Model	iPad Base	iPad Stream	iPhone Base	iPhone Stream	Savings
0.8B-8bit	103.2 / 799 MB	5.8 / 293 MB	53.5 / 792 MB	4.1 / 293 MB	63%
4B-4bit	35.5 / 2330 MB	2.3 / 436 MB	18.4 / 2317 MB	1.8 / 436 MB	81%
9B-6bit	**OOM**	**0.28 / 1780 MB**	—	—	—

4.6 Hybrid and Adaptive Residency: 9B-6bit on 8 GB Devices

Table 7: Hybrid residency for Qwen3.5-9B-6bit on real 8 GB devices. *iPhone at 5 GB: iOS swap thrashing degrades TPS *below* full-stream baseline.

Budget	Resident	Streamed	iPad TPS	iPhone TPS	Memory	Speedup
Full stream	0/32	32	0.28	0.27	1.78 GB	—
2.0 GB	2/32	30	0.30	0.29	2.12 GB	+7%
3.0 GB	8/32	24	0.37	0.34	3.12 GB	+32%
3.5 GB	11/32	21	0.39	0.39	3.63 GB	+43%
4.0 GB	**14/32**	**18**	**0.45**	**0.45**	**4.13 GB**	**+61%**
5.0 GB	20/32	12	0.57	0.21*	5.13 GB	iPad +103%

The adaptive algorithm auto-selects residency via `os_proc_available_memory()`. Table 8 shows the three priority modes on iPad M3.

Table 8: Adaptive residency for Qwen3.5-9B-6bit on iPad M3 (8 GB).

Mode	Resident	TPS	Memory	Note
Full stream	0/32	0.28	1,780 MB	Baseline
balanced	2/32	0.26	5,603 MB	Minimal benefit
maxContext	**22/32**	**0.39**	**5,471 MB**	**Best: +39%**
maxTPS	24/32	0.03	5,802 MB	Swap thrashing
Full resident	—	—	—	Need 7,220 MB

Critical finding: maxTPS allocated 24 layers (5.8 GB) based on 6.5 GB available at startup. During generation, iOS swap pressure caused a **14× slowdown** vs. full streaming. The `available_memory` API reports a snapshot, not a guarantee.

4.7 Adaptive Residency: 4B-4bit Sweep (Memory-Sufficient)

When the model fits entirely in memory, streaming is not required—but what is its cost?

Table 9: 4B-4bit residency sweep on 8 GB devices. Baseline = standard (non-streaming) loading.

Resident	iPad TPS	vs Base	iPhone TPS	vs Base	Memory
0/32 (full stream)	2.3	6%	1.8	10%	436 MB
8/32 (25%)	3.9	11%	3.0	16%	916 MB
16/32 (50%)	5.2	15%	3.8	21%	1,395 MB
24/32 (75%)	6.5	18%	4.7	25%	1,875 MB
28/32 (90%)	10.3	29%	6.0	33%	2,114 MB
32/32 (stream 100%)	22.7	64%	5.9	32%	2,293 MB
Baseline (no stream)	**35.5**	**100%**	**18.4**	**100%**	**2,330 MB**

Key finding: Even with all 32 layers resident (no actual I/O), streaming is **36% slower on iPad** (22.7 vs. 35.5) and **68% slower on iPhone** (5.9 vs. 18.4) due to per-layer eval synchronization. **For models that fit in memory, standard loading is always better.**

4.8 Bandwidth Scaling

The iPad/iPhone TPS ratio across all tested models falls in 1.90–1.93×, matching the theoretical memory bandwidth ratio of 100/50 = 2×. This confirms that decode throughput on Apple Silicon edge devices is **bandwidth-bound**: performance scales linearly with memory bandwidth regardless of GPU core count.

5 Discussion

5.1 Practical Implications

For 8 GB devices, the optimal configuration depends on the use case:

- **Interactive (>10 TPS)**: Qwen3.5-4B-4bit with 90% residency (28/32): 12.6/6.0 TPS (iPad/iPhone) at 2.1 GB, freeing 200+ MB for KV cache.
- **Quality-optimized**: Qwen3.5-4B-6bit (PPL 4.759) at 25.2/13.2 TPS fully resident at 3.3 GB.
- **Maximum quality**: Qwen3.5-9B-6bit (PPL 4.485) via hybrid streaming. 0.45 TPS on iPad at 4.1 GB (14/32 resident). Suitable for background generation.
- **Hero result**: Qwen3.5-27B-4bit (PPL 4.010) via full streaming. Estimated 0.1–0.2 TPS. Demonstrates feasibility, not practicality.

5.2 I/O Bottleneck Analysis

Layer-streaming is fundamentally I/O-bound. Per-layer profiling shows forward computation takes ∼0.06 ms on Mac Studio, while loading a 200 MB layer takes ∼14 ms—a ratio of 1:230. On iPad with ∼2 GB/s NVMe, load time increases to ∼100 ms per layer (ratio 1:1660).

Sub-layer streaming (keeping attention weights resident, streaming only MLP) provides <2% improvement because total I/O volume remains nearly identical.

5.3 Streaming Overhead (Limitation)

A critical and surprising finding: the streaming code path introduces significant overhead *even when no streaming occurs*. With all 32 layers resident, streaming achieves only 64% of baseline TPS on iPad and 32% on iPhone (Table 9). This overhead stems from:

- **Per-layer eval synchronization**: Each layer requires `mx.eval()` to force computation before the (potential) weight unload, preventing MLX's lazy evaluation from batching operations across layers.
- **Memory pressure monitoring**: The `checkMemoryPressure()` call every 64 tokens adds small but measurable overhead.
- **Weight loading checks**: Even for resident layers, the load/skip branching disrupts the optimized forward pass.

iPhone is disproportionately affected (68% overhead vs. 36% on iPad) because its fewer GPU cores (6 vs. 10) and lower bandwidth (50 vs. 100 GB/s) make the per-layer eval synchronization cost a larger fraction of total time.

5.4 Swap Thrashing (Limitation)

The adaptive maxTPS mode for 9B allocated 24/32 layers (5.8 GB) based on 6.5 GB reported available at startup. During generation, this left insufficient headroom. iOS began swap-paging, and TPS dropped to 0.03—**14× slower than full streaming** (0.28 TPS). The `os_proc_available_memory()` API reports a snapshot, not a guarantee; system services reclaim memory unpredictably. Our 0.9 safety factor was insufficient for this scenario.

5.5 TPS Prediction Accuracy (Limitation)

Our bandwidth-based formula $\text{TPS} = 1/(T_{\text{compute}} + N_{\text{streamed}} \times T_{\text{io}})$ showed prediction errors from 50% to over 4000%. The model doesn't account for: safetensors parsing overhead, per-layer eval synchronization cost, GPU compute efficiency vs. theoretical, and iOS memory management overhead. A more accurate model requires empirical calibration coefficients fitted to measured data per device.

5.6 Other Limitations

1. **Single model family**: All experiments use Qwen3.5. MoE architectures with non-uniform layers may benefit from strategic placement.
2. **No prefill streaming**: Current implementation streams during decode only.
3. **No prefetching**: Layers load synchronously. Overlapping I/O with compute could improve TPS.

6 Related Work

LLM on mobile devices. MLC-LLM [2] and llama.cpp [3] enable on-device inference through aggressive quantization. Our work is complementary: layer-streaming allows running models too large even for quantized deployment.

Offloading for LLMs. FlexGen [4] offloads to CPU/disk for throughput-oriented batch inference on GPUs. Our approach targets latency-sensitive single-sequence inference on unified memory architectures where CPU and GPU share the same memory pool.

MLX ecosystem. MLX [1] provides a NumPy-like framework optimized for Apple Silicon. We extend mlx-lm with per-layer weight management, demonstrating that the framework's lazy evaluation model enables efficient streaming despite lacking built-in support.

7 Conclusion

Layer-streaming offloading enables language models exceeding device memory to run inference on 8 GB Apple Silicon edge devices. Our experiments establish that: (1) streaming reduces peak memory by 63–81% on real devices; (2) a 9B model runs at 0.39 TPS via adaptive hybrid residency on iPad; (3) TPS scales linearly with resident ratio; (4) **streaming has significant overhead**—even 100% resident is 36–68% slower than baseline, so standard loading should be used when the model fits; (5) **over-aggressive residency causes swap thrashing**, dropping TPS by 14×—the available memory API is not a reliable allocator. The technique trades throughput for deployability: a 9B model on an 8 GB device at 0.39 TPS is slow but previously impossible.

Acknowledgements

Built with MLX by Apple. Experiments on Mac Studio M2 Ultra, iPad Air M3, and iPhone 15 Pro Max.

References

[1] Awni Hannun et al. MLX: An array framework for Apple silicon. `https://github.com/ml-explore/mlx`, 2024.

[2] MLC Team. MLC-LLM: Universal LLM deployment on edge devices. `https://github.com/mlc-ai/mlc-llm`, 2023.

[3] Georgi Gerganov. llama.cpp: LLM inference in C/C++. `https://github.com/ggerganov/llama.cpp`, 2023.

[4] Ying Sheng et al. FlexGen: High-throughput generative inference of large language models with a single GPU. In *ICML*, 2023.