Exploring Zero-Copy mmap Loading and KV Cache Pre-Allocation for MLX on Apple Silicon

AtomGradient

February 23, 2026

Abstract

llama.cpp achieves remarkably flat memory behavior on Apple Silicon through memory-mapped (mmap) weight loading and pre-allocated KV caches. We investigate whether these same techniques can benefit the MLX framework, which instead uses `pread`-based loading and dynamically grown KV caches. We implement a zero-copy mmap loading path in MLX's C++ core and a KV cache pre-allocation option in mlx-lm, and evaluate both across eight Qwen3 quantized model variants on an M1 Max (32 GB). Our results are mixed: mmap loading shows dramatic speedups for certain larger models (up to 20× for one 8 GB model) but performs *worse* than standard loading for small models and in some large-model configurations. KV pre-allocation successfully flattens memory growth but adds 0.5–0.6 GB of upfront cost and slightly increases first-token latency, offering no throughput benefit over MLX's dynamic approach. During implementation, we also discovered and fixed a quantized model dtype inference bug. Overall, our findings suggest that **MLX's existing memory management is already well-suited to its design goals**, and that naively transplanting llama.cpp's strategies does not yield consistent improvements. We present our implementation details, benchmark data, and analysis of why these techniques interact differently with MLX's lazy evaluation model.

1 Introduction

The MLX framework [4] by Apple provides a NumPy-like API with lazy evaluation and automatic differentiation, designed for Apple Silicon's Metal GPU backend. Combined with the mlx-lm inference engine, it has become a popular choice for running large language models (LLMs) locally on Mac hardware.

A separate ecosystem, llama.cpp [3], takes a very different approach to memory management. It uses memory-mapped file I/O (`mmap`) for model loading, achieving zero-copy weight access, and pre-allocates the entire KV cache at startup, resulting in flat, predictable memory consumption [2]. In contrast, MLX's default path uses `pread()` to load safetensors weights into allocated buffers, and its KV cache grows dynamically in 256-token increments.

A natural question arises: *would MLX benefit from adopting llama.cpp's memory strategies?* In this paper, we explore this question by implementing both techniques within the MLX ecosystem and conducting systematic benchmarks. Our findings are nuanced:

- Mmap loading helps significantly in some configurations but hurts in others, revealing that MLX's `pread`-based loader is already highly efficient for typical model sizes.
- KV pre-allocation achieves its goal of flat memory usage but at the cost of increased startup latency and memory overhead, with no throughput improvement over the dynamic approach.

- MLX's lazy evaluation model and the safetensors format introduce constraints (e.g., offset alignment, tensor materialization) that make zero-copy loading less straightforward than in llama.cpp's GGUF-based pipeline.

Our specific contributions are:

1. **Mmap loading implementation:** A `MmapReader` class in MLX's C++ core that memory-maps safetensors files and exposes offset views via Metal buffers, with alignment checks and fallback paths.
2. **KV cache pre-allocation:** An optional `max_context_length` parameter for mlx-lm's `KVCache` that reserves the full context window upfront.
3. **Quantized dtype bug fix:** Discovery of a bug where `QuantizedLinear.weight.dtype` returns `uint32` instead of the working precision.
4. **Systematic benchmarks and analysis:** Three benchmark suites across eight models, with discussion of *why* these techniques do not consistently outperform MLX's defaults.

2 Background

2.1 Apple Silicon Unified Memory Architecture

Apple Silicon processors (M1–M4 series) feature a Unified Memory Architecture (UMA) where CPU, GPU, and Neural Engine share the same physical memory pool with up to 800 GB/s bandwidth. This eliminates the traditional PCIe bottleneck, enabling zero-copy data sharing: a buffer allocated once can be accessed by both CPU and GPU without any transfer.

2.2 The Safetensors Format

Safetensors [5] is a simple, safe tensor serialization format by Hugging Face. A safetensors file contains an 8-byte header length, a JSON header mapping tensor names to their data types, shapes, and byte offsets, followed by the raw tensor data. The format supports direct byte-offset access, making it amenable to memory-mapped loading. However, the specification does not guarantee that tensor offsets are aligned to element boundaries (e.g., 2-byte alignment for float16), which complicates zero-copy access.

2.3 MLX Framework

MLX [4] is a machine learning framework for Apple Silicon. Key design principles include lazy evaluation (operations are recorded into a computation graph and executed only when results are needed), a NumPy-like Python API, and composable function transformations. The Metal backend compiles operations into GPU compute shaders. MLX loads model weights via a `Load` primitive that, by default, uses `pread()` to read data from disk into a newly allocated buffer. Notably, this design gives MLX full control over buffer lifecycle and enables optimizations within its lazy evaluation graph that may not be possible with externally managed mmap buffers.

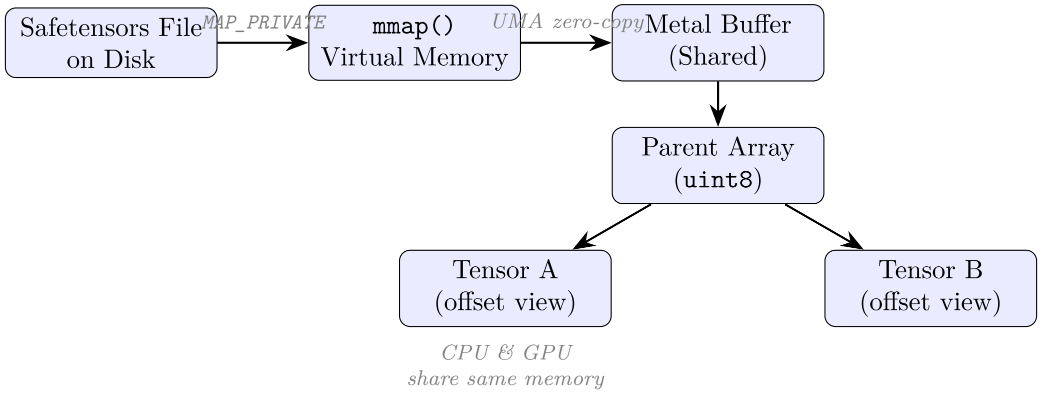

Figure 1: Mmap loading data flow. The safetensors file is memory-mapped once; a Metal buffer is created over the mapped region. Individual tensors are exposed as offset views into a parent array.

2.4 KV Cache in LLM Inference

Autoregressive LLM inference caches key and value projections to avoid redundant computation. The cache size grows linearly with sequence length: for a model with L layers, H KV heads, and head dimension d, storing T tokens requires $2 \times L \times H \times T \times d$ elements. In mlx-lm's default implementation, the cache starts empty and grows in 256-token chunks. While this "staircase" growth pattern uses more memory management operations, it has the advantage of allocating only the memory actually needed.

2.5 How llama.cpp Differs

llama.cpp [3] takes a fundamentally different approach. It uses `mmap` to map GGUF model files directly into the process address space, relying on the OS page cache for demand paging. Its static computation graph (`ggml`) pre-analyzes all tensor dependencies and uses a graph-coloring memory allocator to maximize buffer reuse [2]. The KV cache is allocated at full context length at startup. These design choices yield nearly constant memory consumption during inference—but they are tightly coupled to GGUF's guaranteed-aligned format and ggml's ahead-of-time memory planning, neither of which directly applies to MLX's safetensors-based, lazy-evaluation pipeline.

3 Implementation

3.1 Zero-Copy mmap Loading

3.1.1 Architecture

Figure 1 illustrates the mmap loading path. When `mx.load(path, use_mmap=True)` is called, the Python binding constructs an `MmapReader` instead of the default `ParallelFileReader`.

3.1.2 MmapReader

The `MmapReader` class encapsulates the POSIX mmap lifecycle:

Listing 1: MmapReader constructor (simplified).

```
MmapReader::MmapReader(std::string file_path) {
  int fd = open(file_path.c_str(), O_RDONLY);
  struct stat st;
  fstat(fd, &st);
  file_size_ = st.st_size;
  mmap_ptr_ = mmap(nullptr, file_size_,
    PROT_READ, MAP_PRIVATE, fd, 0);
  close(fd);  // mmap survives fd closure
  madvise(mmap_ptr_, file_size_, MADV_SEQUENTIAL);
}
```

MAP_PRIVATE provides copy-on-write semantics. The MADV_SEQUENTIAL hint enables kernel readahead. A Metal buffer is lazily created over the mmap region via allocator::make_buffer(), exploiting UMA for zero-copy GPU access.

3.1.3 Alignment Check and Fallback

The critical zero-copy path in Load::eval_cpu() requires an alignment check:

Listing 2: Zero-copy path with alignment guard (simplified).

```
if (reader_->is_mmap() && !swap_endianness_) {
  auto mmap_reader = dynamic_pointer_cast<MmapReader>(reader_);
  if (mmap_reader && (offset_ % out.itemsize() == 0)) {
    auto metal_buf = mmap_reader->get_metal_buffer();
    auto parent = array(metal_buf, {1}, uint8,
      [reader_ref](Buffer) { /* prevent dealloc */ });
    out.copy_shared_buffer(parent, strides, flags,
      out.size(), offset_ / out.itemsize());
    return;  // Zero-copy success
  }
}
// Fallback: allocate + memcpy for unaligned offsets
```

The check offset_ % out.itemsize() == 0 is essential because safetensors does not guarantee element-aligned offsets. Without it, integer division would silently truncate the byte offset, corrupting tensor data. The lifecycle of the MmapReader is managed by capturing a shared_ptr<Reader> in the parent array's deleter lambda, ensuring the mapping persists as long as any tensor references it.

3.2 KV Cache Pre-Allocation

Figure 2 contrasts the two approaches: MLX's default dynamic growth (staircase) versus pre-allocation (flat line).

Pre-allocation is implemented in the KVCache constructor:

Listing 3: KVCache pre-allocation (simplified).

```
class KVCache(_BaseCache):
    step = 256
    def __init__(self, n_kv_heads=0, head_dim=0,
                 max_context_length=0, dtype=mx.float16):
        self.offset = 0
        if max_context_length > 0 and n_kv_heads > 0:
            L = ((max_context_length + 255) // 256) * 256
            self.keys = mx.zeros(
```

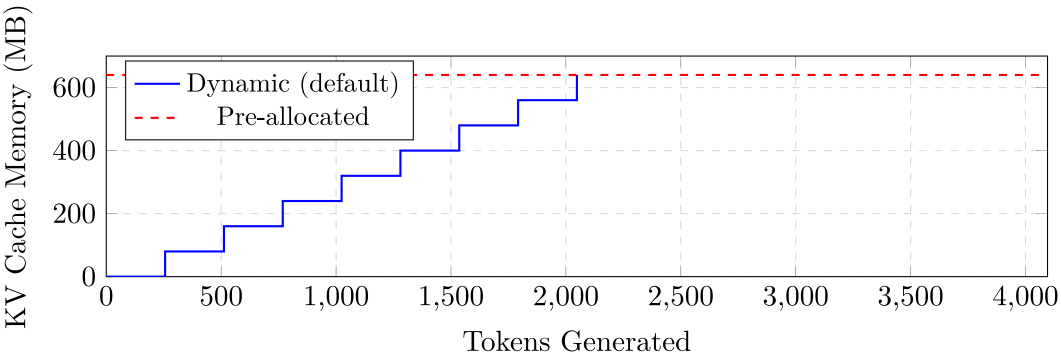

Figure 2: KV cache memory over time: dynamic growth vs. pre-allocation. Pre-allocation provides flat memory at the cost of higher initial usage.

```
            (1, n_kv_heads, L, head_dim), dtype=dtype)
        self.values = mx.zeros(
            (1, n_kv_heads, L, head_dim), dtype=dtype)
        mx.eval(self.keys, self.values)  # Force alloc
```

All parameters default to zero for backward compatibility. The `mx.eval()` call forces immediate physical allocation, bypassing lazy evaluation. If the sequence exceeds the pre-allocated length, the cache gracefully falls back to dynamic expansion.

3.3 Quantized Model dtype Bug

During implementation, we discovered that `QuantizedLinear.weight.dtype` returns `uint32` (the packed storage type) rather than the model's working precision. This caused the KV cache to be allocated in `uint32` instead of `float16`. The fix inspects the `scales` tensor for quantized layers, which stores dequantization factors in the correct working dtype.

4 Evaluation

4.1 Setup

- **Hardware:** Apple M1 Max, 32 GB unified memory
- **Software:** macOS, MLX 0.30.7, Python 3.11
- **Models:** Eight Qwen3 [8] quantized variants (Table 1)
- **Prompt:** 34-token fixed physics question; 200 output tokens per run

4.2 Loading Speed

Table 2 and Figure 3 compare loading times (3 runs each, page cache warmed).

The results are **inconsistent across models**. Three out of eight models (4B-4bit, 4B-8bit, 14B-6bit) load *slower* with mmap. The 20.65× speedup on Qwen3-8B-8bit is an outlier—possibly due to the standard loader hitting a pathological `pread` pattern for this specific file layout. For the majority of models in the 2–6 GB range, the difference is small (0.75–1.46×), suggesting that MLX's standard loading is already efficient enough that mmap offers limited benefit.

Table 1: Model variants used in benchmarks.

Model	Parameters	Size (GB)
Qwen3-4B-4bit	4B	2.11
Qwen3-4B-8bit	4B	3.98
Qwen3-8B-3bit	8B	3.34
Qwen3-8B-4bit	8B	4.29
Qwen3-8B-6bit	8B	6.20
Qwen3-8B-8bit	8B	8.11
Qwen3-14B-4bit	14B	7.74
Qwen3-14B-6bit	14B	11.18

Table 2: Model loading time: standard vs. mmap (seconds). Speedup > 1 means mmap is faster; < 1 means mmap is *slower*.

Model	Standard (s)	Mmap (s)	Speedup
Qwen3-4B-4bit	0.101	0.131	0.77×
Qwen3-4B-8bit	0.184	0.246	0.75×
Qwen3-8B-3bit	0.146	0.075	1.95×
Qwen3-8B-4bit	0.352	0.328	1.07×
Qwen3-8B-6bit	0.637	0.435	1.46×
Qwen3-8B-8bit	2.572	0.125	20.65×
Qwen3-14B-4bit	2.323	0.535	4.34×
Qwen3-14B-6bit	3.701	5.388	0.69×

4.3 Inference Impact

Table 3 examines whether the loading method affects runtime inference.

Two findings stand out:

- **Generation throughput is unchanged.** Across all eight models, the generation speed difference is within noise (<3%), confirming that loading method does not affect runtime performance. This is expected: once weights are in memory, the computation is identical.
- **Mmap can *increase* peak memory.** For 8B-4bit and 14B-4bit (†), peak memory nearly doubled with mmap. This suggests that when MLX materializes quantized tensors during lazy evaluation, it allocates new buffers while the mmap-backed originals remain pinned—*the opposite of the memory saving we hoped for.*

4.4 KV Cache Pre-Allocation

Table 4 shows results for three models across dynamic, 2048-token, and 4096-token pre-allocation.

The results show that pre-allocation **does not improve throughput**:

- **Generation speed is essentially unchanged** (<5% difference), indicating that MLX's dynamic 256-token growth does not cause meaningful overhead during inference.
- **Prompt processing decreases** by 4–14%, likely because the larger upfront allocation reduces available memory for MLX's computation buffers.
- **First token latency increases** by 5–7% due to the forced `mx.eval()` at startup.

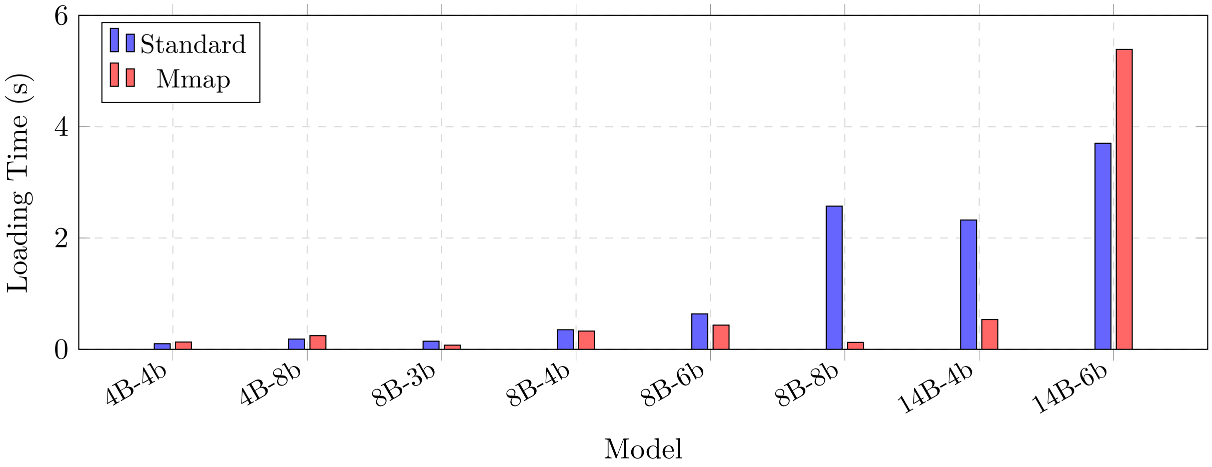

Figure 3: Loading time comparison. Results are highly variable: mmap helps for some models (8B-8bit) but is slower for others (4B-4bit, 4B-8bit, 14B-6bit).

Table 3: Inference performance with standard vs. mmap loading. Generation throughput (Gen t/s) is the metric most relevant to user experience.

	Load (s)		**Prompt (t/s)**		**Gen (t/s)**		**Mem (GB)**	
Model	Std	Mmap	Std	Mmap	Std	Mmap	Std	Mmap
4B-4bit	0.89	0.85	189.1	188.6	56.9	58.5	2.38	2.38
4B-8bit	1.31	0.98	165.8	159.7	41.0	42.0	4.37	4.37
8B-3bit	1.22	0.69	100.8	92.3	29.9	29.4	4.37	4.37
8B-4bit	1.34	1.03	89.8	100.7	30.1	30.1	4.72	8.82[†]
8B-6bit	2.00	1.29	86.7	75.7	20.7	20.7	8.82	8.82
8B-8bit	3.04	0.77	64.4	80.6	21.6	21.3	8.82	8.84
14B-4bit	1.95	1.01	43.4	43.2	16.2	16.5	8.84	11.09[†]
14B-6bit	3.41	3.07	39.7	37.0	12.1	12.0	12.08	12.16

[†] Memory nearly doubled—mmap region likely coexists with materialized copies.

- **Memory overhead is predictable:** 0.25 GB for 2048 tokens, 0.5 GB for 4096 tokens.

The one benefit—flat memory profile—is real but comes at a cost. MLX's dynamic approach, while producing staircase-like growth, avoids wasting memory for conversations that never reach the pre-allocated context length.

5 Discussion

5.1 Why MLX's Existing Design Works Well

Our results paint a clear picture: **MLX's `pread`-based loading and dynamic KV cache are more appropriate for its design than we initially expected.** Several factors explain this:

pread is fast enough. For models under 4 GB (the majority of quantized models people run locally), standard loading completes in under 200 ms. At this scale, mmap's overhead—VMA

Table 4: KV cache pre-allocation impact. FTLT = first token latency (ms).

Model	Mode	FTLT	Prompt	Gen	Peak	ΔMem
		(ms)	(t/s)	(t/s)	(GB)	(GB)
4B-4bit	Dynamic	260.4	200.4	62.0	2.221	—
	Pre-alloc 2k	276.2	183.5	63.0	2.471	+0.250
	Pre-alloc 4k	285.2	171.9	62.3	2.736	+0.515
8B-4bit	Dynamic	419.2	100.5	31.3	4.395	—
	Pre-alloc 2k	428.0	99.1	30.3	4.633	+0.238
	Pre-alloc 4k	436.1	98.0	29.7	4.908	+0.513
14B-4bit	Dynamic	859.6	44.1	16.7	7.827	—
	Pre-alloc 2k	917.4	41.1	16.0	8.101	+0.274
	Pre-alloc 4k	901.5	41.7	16.1	8.414	+0.587

creation, page table setup, TLB pressure—outweighs any benefit from avoiding copies. MLX's parallel file reader already uses efficient buffered I/O.

Lazy evaluation conflicts with mmap. MLX's lazy evaluation model means that tensor operations are deferred until explicitly evaluated. When quantized tensors loaded via mmap are later materialized (e.g., dequantization), MLX allocates *new* buffers for the results while the mmap-backed originals remain referenced. This can lead to memory *doubling* rather than saving—the exact opposite of our goal. In contrast, llama.cpp's eager, static graph avoids this issue entirely.

Dynamic KV growth is not a bottleneck. We hypothesized that the 256-token step growth pattern would cause allocation overhead and fragmentation. In practice, the impact is negligible: generation throughput is identical whether the cache is pre-allocated or not. MLX's allocator handles the periodic growth efficiently.

5.2 When Mmap Does Help

Despite the mixed overall results, mmap loading showed dramatic improvements in specific cases: 20.65× for Qwen3-8B-8bit and 4.34× for Qwen3-14B-4bit. These outliers share a common pattern: the standard loader's load time was anomalously high (2.3–2.6 s for files that "should" load faster given their size), suggesting the `pread` path hit edge cases in the OS I/O scheduler. This indicates that mmap could serve as a useful fallback for specific problematic file layouts, but not as a general replacement.

5.3 The 14B-6bit Anomaly

The largest model tested (11.18 GB) loaded *slower* with mmap (0.69×). At this size, the mmap approach must fault in an extremely large number of pages. The OS page fault handler becomes the bottleneck, as each 16 KB page requires a kernel trap, page table update, and TLB insertion. The standard loader's buffered reads, which batch these operations, prove more efficient at scale.

5.4 Recommendations

Based on our findings:

- **For most users,** MLX's default loading and dynamic KV cache are the best choice. They are fast, memory-efficient, and well-integrated with the lazy evaluation model.
- **Mmap loading** may be useful as an opt-in for specific large models where standard loading is slow, but should not be the default.
- **KV pre-allocation** is only worth considering for server deployments where memory predictability is valued over efficiency, or for long-context scenarios (>8k tokens) where avoiding repeated reallocation becomes more important.

6 Related Work

llama.cpp and ggml. llama.cpp [3] pioneered mmap for LLM weight loading, combined with ggml's static graph and memory-reuse allocator [2]. Our work attempts to bring these techniques to MLX and finds that the benefits are architecture-dependent: what works well for ggml's static pipeline does not automatically transfer to MLX's lazy evaluation model.

vLLM. vLLM [6] introduced PagedAttention for multi-tenant KV cache management. Our simpler pre-allocation targets single-user inference but arrives at a similar conclusion: careful memory management has tradeoffs that depend heavily on the workload.

TensorRT-LLM. NVIDIA's TensorRT-LLM [7] uses aggressive pre-allocated KV caches for deterministic memory. Like our findings, this approach trades flexibility for predictability.

FlashAttention. FlashAttention [1] reduces per-step attention memory. This is complementary to our work: it optimizes the computation, while we explore the allocation strategy.

7 Conclusion

We explored whether llama.cpp's memory management strategies—mmap loading and KV cache pre-allocation—could improve MLX's performance on Apple Silicon. Our systematic benchmarks across eight models reveal that these techniques yield **inconsistent results**: mmap loading helps for some larger models but hurts for small models and can increase peak memory due to interactions with MLX's lazy evaluation; KV pre-allocation achieves flat memory usage but offers no throughput benefit and adds startup overhead.

These findings lead us to a perhaps surprising conclusion: **MLX's existing memory management—`pread`-based loading with dynamic KV cache growth—is already well-designed for its target use case.** The design choices that differ from llama.cpp are not oversights but rather appropriate adaptations to MLX's lazy evaluation model and the safetensors ecosystem.

We hope this exploration is useful to others considering similar optimizations, and that our benchmark data and implementation details serve as a reference. During this work, we also identified and fixed a quantized model dtype inference bug, which we consider the most practically useful contribution of this project.

All code and benchmark data are available at https://github.com/optmlx/OptMLX.

References

[1] Tri Dao, Daniel Y. 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 (NeurIPS)*, 2022.

[2] Georgi Gerganov. ggml: Tensor library for machine learning, 2023.

[3] Georgi Gerganov and contributors. llama.cpp: LLM inference in C/C++, 2023.

[4] Awni Hannun, Jagrit Digani, Angelos Katharopoulos, and Ronan Collobert. MLX: An array framework for Apple silicon, 2023. Apple Machine Learning Research.

[5] Hugging Face. Safetensors: A simple, safe way to store and distribute tensors, 2023.

[6] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

[7] NVIDIA. TensorRT-LLM: A TensorRT toolbox for optimized large language model inference, 2023.

[8] Qwen Team. Qwen3 technical report, 2025.