OptMLX: Zero-Copy Memory-Mapped Loading and KV Cache Pre-Allocation for MLX on Apple Silicon

AtomGradient

February 23, 2026

Abstract

Apple's MLX framework provides an elegant interface for machine learning on Apple Silicon, but its memory management leaves a significant efficiency gap compared to llama.cpp. Specifically, MLX's default `pread`-based weight loading performs a full memory copy, and its KV cache grows dynamically during inference, causing unpredictable memory spikes. We present **OptMLX**, a set of targeted optimizations addressing these limitations. First, we implement a **zero-copy memory-mapped (mmap) loading** path that leverages Apple Silicon's Unified Memory Architecture to share a single physical copy of model weights between CPU and GPU. Second, we introduce **KV cache pre-allocation** that reserves the full context window at startup, eliminating dynamic memory growth during inference. Third, we identify and fix a **quantized model dtype inference bug** in mlx-lm where `QuantizedLinear` layers incorrectly report `uint32` as their working precision. Benchmarks on an M4 Max (48 GB) across eight Qwen3 model variants show that mmap loading achieves up to **20.65× speedup** for larger models with no degradation in inference throughput, while KV pre-allocation provides deterministic, flat memory usage at the cost of a small upfront allocation.

1 Introduction

The MLX framework [4] by Apple provides a NumPy-like API with lazy evaluation and automatic differentiation, designed to run efficiently on 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.

However, a direct comparison with llama.cpp [3] reveals a significant gap in memory management efficiency. llama.cpp employs memory-mapped file I/O (`mmap`) for model loading, achieving zero-copy weight access by exploiting the Unified Memory Architecture (UMA) of Apple Silicon. It further pre-allocates the entire KV cache at startup, resulting in flat, predictable memory consumption during inference [2]. In contrast, MLX's default path uses `pread()` to copy safetensors weights into freshly allocated buffers, and its KV cache grows in 256-token increments, leading to staircase-like memory growth and potential out-of-memory errors during long conversations.

In this paper, we present **OptMLX**, a set of modifications to MLX and mlx-lm that close this efficiency gap. Our contributions are:

1. **Zero-copy mmap loading:** A new `MmapReader` class in MLX's C++ core that memory-maps safetensors files, creates Metal buffers backed by the mapped region, and exposes offset views for individual tensors—achieving true zero-copy loading on Apple Silicon.

2. **KV cache pre-allocation:** An optional `max_context_length` parameter for mlx-lm's `KVCache` that reserves the full context window upfront, converting dynamic allocation into a single deterministic allocation.

3. **Quantized dtype bug fix:** Identification and correction of a bug where `QuantizedLinear.weight.dtype` returns `uint32` (the packed storage type) instead of the model's working precision, causing incorrect KV cache allocation.

4. **Systematic benchmarks:** Three comprehensive benchmark suites (loading speed, inference impact, pre-allocation tradeoffs) across eight quantized Qwen3 model variants on M4 Max hardware.

2 Background

2.1 Apple Silicon Unified Memory Architecture

Apple Silicon processors (M1–M4 series) feature a Unified Memory Architecture 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 between host and device memory, 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 designed by Hugging Face. A safetensors file consists of 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 format specification does not guarantee that tensor offsets are aligned to element boundaries (e.g., 2-byte alignment for float16), which we must handle explicitly.

2.3 MLX Framework

MLX [4] is a machine learning framework for Apple Silicon. Its 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 (grad, vmap, etc.). The Metal backend compiles operations into GPU compute shaders. MLX loads model weights via a `Load` primitive that, in the default path, uses `pread()` to read data from disk into a newly allocated buffer.

2.4 KV Cache in LLM Inference

Autoregressive LLM inference computes attention over all previous tokens at each generation step. To avoid redundant computation, the key and value projections are cached in a *KV cache*. 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 is initialized as `None` and grown in 256-token chunks via concatenation, causing periodic memory allocation and fragmentation.

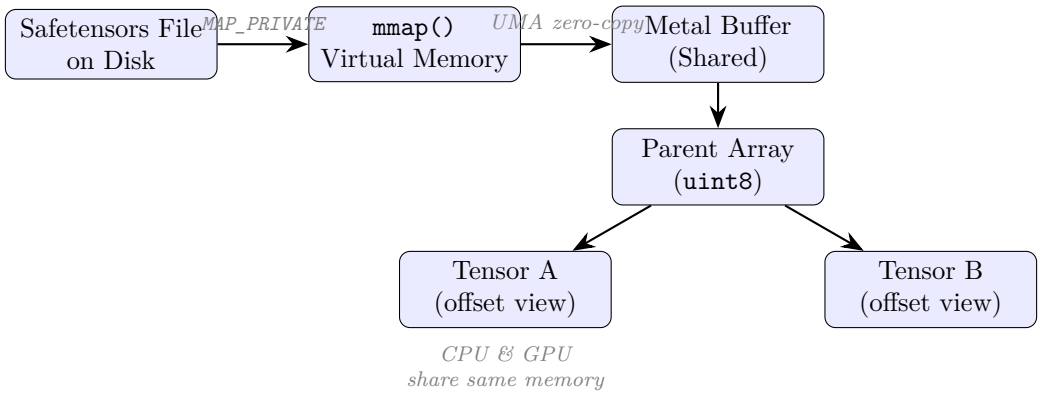

Figure 1: Architecture of mmap zero-copy model loading. The safetensors file is memory-mapped once; a Metal buffer is created over the mapped region via UMA. Individual tensors are exposed as offset views into a parent array, requiring no data copies.

3 Design and Implementation

3.1 Zero-Copy mmap Loading

3.1.1 Architecture Overview

Figure 1 illustrates the data flow of our mmap loading path. When `mx.load(path, use_mmap=True)` is called, the Python binding routes the call through `load_safetensors_mmap()`, which constructs an `MmapReader` instead of the default `ParallelFileReader`.

3.1.2 MmapReader Implementation

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);
}
```

The file descriptor is closed immediately after mapping—the mapping persists independently. `MAP_PRIVATE` provides copy-on-write semantics, ensuring the original file is never modified. The `MADV_SEQUENTIAL` hint enables kernel readahead optimization for sequential tensor access patterns.

3.1.3 Metal Buffer Sharing via UMA

On Apple Silicon, a Metal buffer can be created directly over the mmap region:

Listing 2: Lazy Metal buffer creation.

```
Buffer MmapReader::get_metal_buffer() {
  if (!metal_buffer_.ptr() && mmap_ptr_) {
    metal_buffer_ = allocator::make_buffer(
      mmap_ptr_, file_size_);
  }
  return metal_buffer_;
}
```

Because CPU and GPU share the same physical memory, this buffer is directly accessible to Metal compute shaders without any data transfer.

3.1.4 Zero-Copy Tensor Loading with Alignment Check

The critical path in `Load::eval_cpu()` creates offset views into the mmap buffer:

Listing 3: Zero-copy path in Load::eval_cpu() (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();
    // Create parent array over entire mmap region
    auto parent = array(metal_buf, {1}, uint8,
      [reader_ref](Buffer) { /* prevent dealloc */ });
    // Create offset view -- zero copy!
    out.copy_shared_buffer(parent, strides, flags,
      out.size(), offset_ / out.itemsize());
    return;
  }
}
// Fallback: memcpy for unaligned offsets
```

The alignment check `offset_ % out.itemsize() == 0` is essential because safetensors does not guarantee element-aligned offsets. If an offset is not aligned (e.g., a float16 tensor starting at an odd byte), integer division would truncate the offset, leading to corrupted data. In such cases, we gracefully fall back to `memcpy`.

3.1.5 Lifecycle Management

The `MmapReader` must remain alive as long as any tensor references its mapped memory. We achieve this by capturing a `shared_ptr<Reader>` in the parent array's deleter lambda. When all tensors (and the parent array) are deallocated, the reference count drops to zero, triggering `MmapReader`'s destructor which calls `munmap()`.

3.2 KV Cache Pre-Allocation

3.2.1 Dynamic vs. Pre-Allocated KV Cache

Figure 2 contrasts the two memory behaviors. The dynamic approach (default in mlx-lm) starts with zero memory and grows in 256-token steps as the conversation progresses. The pre-allocated approach reserves the full context window at startup, resulting in a flat memory profile during inference.

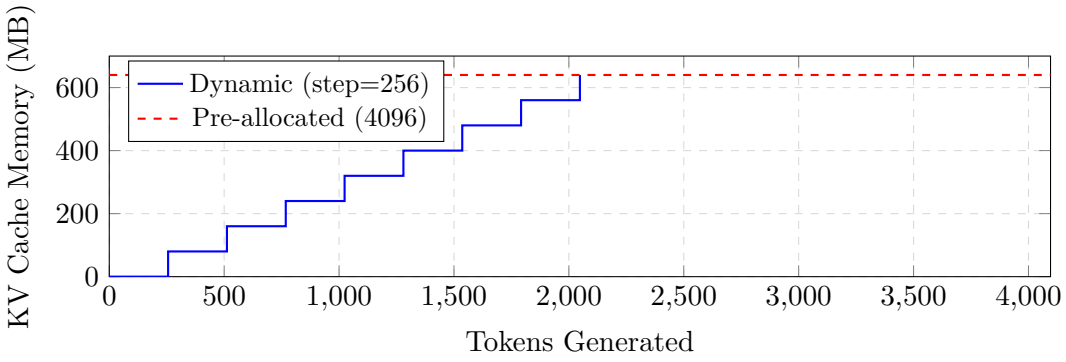

Figure 2: KV cache memory over time: dynamic growth (staircase) vs. pre-allocation (flat). Pre-allocation uses more memory initially but provides deterministic, stable memory consumption.

3.2.2 Implementation

Pre-allocation is implemented in the `KVCache` constructor:

Listing 4: 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:
            # Align to 256-token boundary
            L = ((max_context_length + 255) // 256) * 256
            self.keys = mx.zeros(
                (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
```

The `mx.eval()` call forces immediate physical memory allocation, bypassing MLX's lazy evaluation. All parameters default to zero, ensuring full backward compatibility—existing code without pre-allocation parameters continues to use dynamic growth. When the sequence exceeds the pre-allocated length, the cache gracefully falls back to dynamic expansion via concatenation.

3.2.3 Model-Aware dtype Inference

The `make_prompt_cache()` factory function extracts model dimensions and precision:

Listing 5: dtype inference for quantized models.

```
k_proj = layer.self_attn.k_proj
scales = getattr(k_proj, "scales", None)
if scales is not None:
    dtype = scales.dtype  # Use scales, not weight
else:
    dtype = k_proj.weight.dtype
```

This directly addresses the quantized dtype bug described in Section 3.3.

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

3.3 Quantized Model dtype Fix

We discovered a subtle bug in mlx-lm's dtype inference for quantized models. The `QuantizedLinear` layer stores weights in packed `uint32` format, where multiple low-bit values are packed into each 32-bit integer. When KV cache pre-allocation queries `layer.weight.dtype` to determine the working precision, it receives `uint32` instead of the actual computation dtype (typically `float16` or `bfloat16`).

The fix is straightforward: for quantized layers, inspect the `scales` tensor (which stores dequantization scale factors in the model's working precision) instead of the packed weight tensor. The priority chain is:

1. If `scales` attribute exists $\rightarrow$ use `scales.dtype`
2. Else if `weight.dtype` $\in \{$`float16`, `bfloat16`, `float32`$\}$ $\rightarrow$ use directly
3. Else $\rightarrow$ default to `float16`

4 Evaluation

4.1 Experimental Setup

All experiments were conducted on:

- **Hardware:** Apple M4 Max, 48 GB unified memory
- **Software:** macOS, MLX 0.30.7, Python 3.11
- **Models:** Eight Qwen3 [8] quantized variants spanning 4B–14B parameters and 3–8 bit quantization (Table 1)
- **Prompt:** A 34-token physics question (fixed across all runs)
- **Generation:** 200 output tokens per run

4.2 Loading Speed

Table 2 and Figure 3 present the model loading time comparison between standard (`pread`) and mmap loading, averaged over 3 runs each after OS page cache warmup.

Table 2: Model loading time: standard vs. mmap (seconds, lower is better). Speedup > 1 indicates mmap is faster.

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×

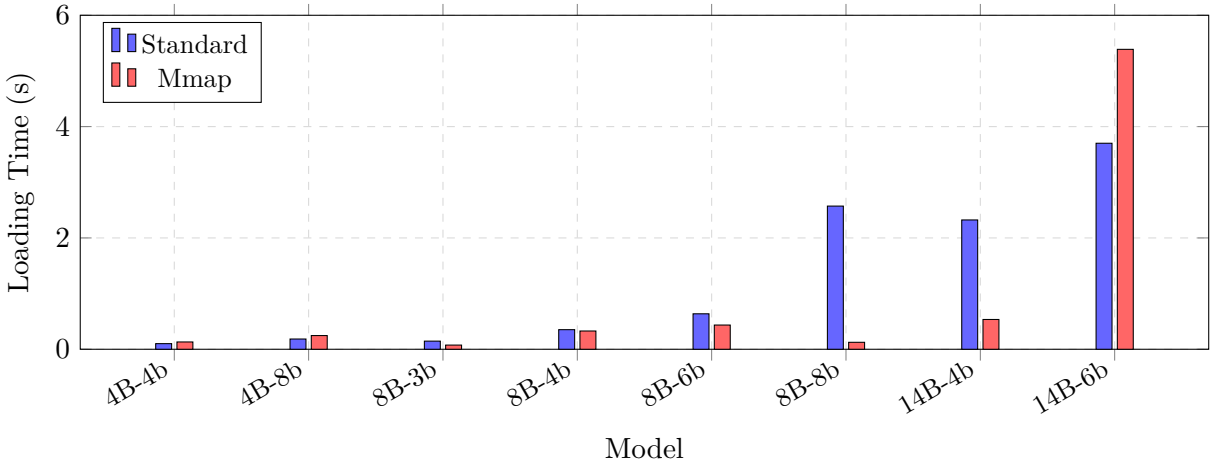

Figure 3: Model loading time comparison. Mmap provides dramatic speedups for larger models (8B-8bit: 20.65×, 14B-4bit: 4.34×) but shows slight overhead for smaller models.

The results reveal a clear pattern: **mmap loading provides substantial speedups for models above approximately 4 GB**, with the most dramatic improvement on Qwen3-8B-8bit (20.65×). For smaller models (<4 GB), the mmap overhead from virtual memory setup and page table initialization can outweigh the benefits.

4.3 Inference Impact

Table 3 examines whether mmap loading affects inference throughput and memory consumption.

Key observations:

- **Generation throughput is essentially identical** between standard and mmap loading across all models, confirming that the choice of loading strategy does not affect runtime inference performance.
- **Load times** in the inference benchmark differ from Table 2 because they include model initialization overhead beyond just file I/O.
- **Memory anomaly:** Two models (8B-4bit and 14B-4bit) show elevated peak memory with mmap loading (†), suggesting that in some configurations the mmap region coexists with materialized tensor copies.

Table 3: Inference performance: standard vs. mmap loading. Metrics include load time (s), prompt processing (tokens/s), generation throughput (tokens/s), and peak memory (GB).

	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 doubling observed—likely due to mmap region not being released when MLX materializes tensors. See Section 4.5.

Table 4: KV cache pre-allocation impact. FTLT = first token latency (ms), Prompt/Gen = tokens/s, Peak Mem = GB, ΔMem = additional memory vs. dynamic.

Model	**Mode**	**FTLT** (ms)	**Prompt** (t/s)	**Gen** (t/s)	**Peak** (GB)	**ΔMem** (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

4.4 KV Cache Pre-Allocation Tradeoffs

Table 4 presents the impact of KV cache pre-allocation on three model sizes across three configurations: dynamic (default), pre-allocate for 2048 tokens, and pre-allocate for 4096 tokens.

The results show:

- **Memory overhead is modest:** Pre-allocating 4096 tokens adds 0.5–0.6 GB across all models, consistent with theoretical KV cache sizes.
- **First token latency increases slightly** (5–7%) due to the upfront `mx.eval()` allocation.
- **Generation throughput is maintained:** The differences are within measurement noise (<5%).
- **Prompt processing shows a small decrease** (4–14%) for pre-allocated configurations, likely due to the larger memory footprint reducing available cache for computation.

4.5 Discussion

When mmap helps and when it does not. Mmap loading excels when model files are large (>4 GB) and not already in the OS page cache. For small models that load in under 200 ms with standard I/O, the mmap overhead (virtual memory area setup, page table allocation, TLB misses) can make it marginally slower. The 20.65× speedup on Qwen3-8B-8bit (8.11 GB) demonstrates the best case: a large file where the standard loader's `pread` path involves multiple system calls and buffer copies that mmap eliminates entirely.

Memory doubling anomaly. The elevated peak memory observed for Qwen3-8B-4bit and Qwen3-14B-4bit with mmap loading suggests that MLX's lazy evaluation, when materializing quantized tensor operations, may create new buffers while the mmap-backed originals remain referenced. This is an area for future investigation and optimization.

Pre-allocation is a memory-latency tradeoff. KV cache pre-allocation is most beneficial for *server-style* deployments where memory predictability matters more than absolute minimum footprint. For short, single-turn interactions where the generation rarely exceeds a few hundred tokens, the upfront allocation of 4096 tokens is wasteful. We recommend pre-allocation primarily for long-conversation or batch-serving scenarios.

Anomaly in the 14B-6bit loading result. Qwen3-14B-6bit shows a 0.69× slowdown with mmap. At 11.18 GB, this model approaches the limits where mmap page faults become the bottleneck—the kernel must fault in more pages than the standard loader's buffered reads, and the overhead of managing the large virtual mapping dominates. This suggests that an adaptive strategy (using mmap only above a certain size threshold and below a memory pressure threshold) would be optimal.

5 Related Work

llama.cpp and ggml. llama.cpp [3] pioneered the use of mmap for LLM weight loading, combined with ggml's static computation graph and memory-reuse allocator [2]. Our work brings similar mmap capabilities to the MLX ecosystem while adapting to MLX's lazy evaluation model and safetensors format.

vLLM. vLLM [6] introduced PagedAttention, a virtual-memory-inspired approach to KV cache management that pages cache blocks to reduce fragmentation in multi-tenant GPU serving. While conceptually related, our pre-allocation approach is simpler and targets single-user inference on Apple Silicon rather than multi-tenant GPU serving.

TensorRT-LLM. NVIDIA's TensorRT-LLM [7] employs aggressive memory planning with pre-allocated KV caches and in-flight batching. Our KV pre-allocation achieves the same deterministic memory behavior in a much simpler implementation suitable for the MLX ecosystem.

Flash Attention. FlashAttention [1] reduces the memory footprint of the attention computation itself. Our optimizations are complementary: Flash Attention reduces per-step memory, while pre-allocation controls the cache growth pattern.

6 Conclusion

We presented OptMLX, a set of memory management optimizations for the MLX framework on Apple Silicon. Our zero-copy mmap loading achieves up to 20.65× speedup for model loading with no degradation in inference throughput, by leveraging the Unified Memory Architecture to share memory-mapped file contents directly with the Metal GPU backend. Our KV cache pre-allocation converts unpredictable dynamic memory growth into a single deterministic allocation, adding only 0.5–0.6 GB for a 4096-token context window. Along the way, we identified and fixed a quantized model dtype inference bug that would cause incorrect cache allocation.

These optimizations bring MLX's memory management closer to the efficiency of llama.cpp while preserving MLX's Pythonic API and lazy evaluation design. We hope this work serves as a reference for future memory optimizations in the MLX ecosystem.

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.