ChatGPT

Concurrent Inference with MLX on Apple Silicon (M4 Max)

Overview of MLX Concurrency and Parallelism

Apple's **MLX** framework enables efficient local inference on Apple Silicon GPUs, but achieving true concurrency (multiple queries at once) requires careful use of MLX-LM's features. By default, MLX-LM (the high-level language model package) processes one prompt at a time in its simple usage (e.g. `mlx_lm.generate`). Early versions did **not** natively schedule multiple requests in parallel – incoming queries would queue and run sequentially [1]. This means that to serve **multiple users** concurrently, you must either run separate model instances (e.g. in multiple processes) or use MLX's **batching** capabilities to combine requests.

Multi-threading vs. Multi-processing: MLX is built in Python with heavy GPU acceleration under the hood (Metal kernels). The intensive operations release the GIL, so Python threads can exploit the GPU concurrency to an extent. In practice, spinning up Python threads for separate `generate` calls can work, but without built-in scheduling they may still compete serially for the single GPU resource. Using **Python 3.14+**, which offers no-GIL multithreading, can improve true parallel execution in one process [2]. Alternatively, **multi-process** deployment (e.g. running multiple worker processes each with a loaded model) can handle concurrent clients, at the cost of higher memory use (each process has its own model copy). Apple's unified memory can hold multiple instances if you have sufficient RAM, but each instance will separately utilize the Neural Engine/GPU – this typically leads to context switching rather than real simultaneous speedup.

Continuous Batching (Dynamic Batching): The most effective approach to parallelism on a single Apple GPU is **dynamic batching** – i.e. combining multiple inference requests into one batched forward pass whenever possible. This approach, inspired by Google's **vLLM** on NVIDIA GPUs, keeps the hardware fully utilized by processing tokens for several queries together. MLX-LM introduced **continuous batching** support in late 2025, allowing the MLX server to merge incoming requests on the fly [3] [4]. Awni Hannun (one of MLX's developers) noted that even for single-client streams, high-throughput deployments benefit from continuous batching to better use the hardware [5]. In MLX-LM's latest version (v0.30.x), the built-in server can accept a **max concurrency** setting and will dynamically batch requests behind the scenes. For example, the `mlx_lm.server` now supports continuous batching mode, and the OpenAI-compatible server in the community (see **mlx-openai-server** below) exposes a `--max-concurrency` flag [6] to control how many requests are handled in parallel with batching.

Batching for Throughput vs. Latency

Batching is the key to optimizing throughput on Apple Silicon. By increasing batch size (processing N prompts together), you amortize the cost of each GPU kernel launch over N requests, often achieving near-linear speedups until hardware limits are hit. Community experiments have shown dramatic gains. For instance, using the **MLX ParaLLM** batching library on a 22B model, one user achieved a **5.8× throughput increase** (from ~17 tokens/s to ~101 tokens/s) by running 31 prompts concurrently [7]. Memory usage grew modestly (about **150 MB per additional request** in that case) [7]. Another MLX user reported **3–4× throughput gains** by continuously batching multiple chat queries on an M4 Max [3]. These results underscore that Apple's M-series GPU *can* generate tokens for many queries in parallel nearly as fast as for one query, up to the point of memory or bandwidth saturation.

Batch Prompting: To use batching, you can supply multiple tokenized prompts to MLX at once. MLX-LM now includes a `BatchGenerator` utility (and a high-level `batch_generate` function) to facilitate batched inference. For example, you can do:

```python
from mlx_lm import load, BatchGenerator
model, tokenizer = load("mlx-community/Llama-2-7B-4bit")
prompts = ["Question 1...", "Question 2...", "Question 3..."]  # multiple user
queries
# Initialize batch generator
gen = BatchGenerator(model, stop_tokens=set([tokenizer.eos_token_id]))
uids = gen.insert(prompts, max_tokens=[100]*len(prompts))  # insert all prompts
while responses := gen.next():
    for r in responses:
        print(f"Response for prompt {r.uid}: {r.text}")  # collect outputs as
they stream
```

Under the hood, `BatchGenerator` will handle padding the prompts and generating tokens for all requests together. This yields significantly higher **throughput** (tokens per second) by maximizing GPU utilization. The trade-off is **latency**: a given user's query might wait a bit for others to join the batch. In high-load scenarios (e.g. many concurrent users), the latency penalty per user is small compared to the throughput boost. But if you only have a single user at a time, using batch mode doesn't help – in fact, overly large batch sizes can hurt latency, as the model must generate extra tokens for empty slots once some prompts finish. The sweet spot is to batch only when multiple queries are pending. MLX's continuous batching logic in the server tries to do exactly this: it waits a short window for additional requests to arrive and batches them together [5]. One simple strategy is to use a short wait/queue time (e.g. 50–200ms) to accumulate queries before running a batch.

Batch Size Limits: The optimal batch size depends on model size and GPU memory. Each concurrent sequence consumes additional memory for its KV cache (see next section) and intermediate activations. In the earlier example, a 22B 4-bit model on an M1 Max hit diminishing returns around batch size ~30 [8] – beyond that, throughput gains flattened due to memory bandwidth limits. On an M4/M5 Max or Ultra with more memory and bandwidth, you may sustain gains to higher batch counts. It's wise to do some benchmarking on your specific model: measure tokens/sec vs. batch size and monitor memory (using `mx.metal.get_memory_info()` as shown in MLX docs [9]). As a rule, **batch for throughput, stream singly for lowest latency**. Many inference servers adaptively batch only when load is high.

Leveraging KV Cache for Faster Inference

Large Language Models produce each token by attending over all prior tokens. The **key-value cache** (KV cache) stores the computed key and value vectors for each past token so that the model doesn't recompute them from scratch on every new token. MLX-LM fully supports caching: by default it will reuse the cache when generating token-by-token (in functions like `stream_generate`). It uses a **rotating KV cache** window (default 4096 tokens) to prevent unbounded growth in memory and keep per-token latency stable [10]. This behaves like a rolling context: if you generate beyond the window size, older cache entries are dropped ("rotated out") to cap memory usage [11]. This design yields consistent performance even in long conversations (with older history dropped as needed).

For serving multiple queries, KV caching mostly helps **per-session** rather than across different users. Within a single chat session, keeping the cache in memory means **follow-up prompts are much faster** because the model doesn't re-process the entire conversation from scratch. For example, LM Studio noted that with MLX's prompt caching enabled, *follow-up responses in a vision-language model became "drastically faster"* since the text portion could reuse the cached encoding of prior turns [12]. In practice, this means that if user A sends a message, the model encodes it and stores the KV states; when user A sends the next message in the same session, MLX can reuse those states instead of recomputing everything. This yields lower time-to-first-token (TTFT) for subsequent turns.

For **different users/queries**, the KV caches are separate (each session has its own). However, you can still leverage caching in creative ways. MLX-LM introduced **prompt cache files** that let you persist encoded caches to disk [13]. If you have a repeated prefix or system prompt common to many queries, you could load its cache to avoid re-encoding it each time. This is similar to "prefix-batching" tricks: e.g. if all users share a long instruction prompt, encode it once and reuse that KV for everyone. MLX-LM's API doesn't (yet) provide a simple `past_key_values` parameter like HuggingFace Transformers, but advanced users have implemented workarounds. For instance, one developer created a persistent KV cache by storing per-token KV in a trie and merging caches for new prompts [14]. While such techniques are non-trivial, they hint at future improvements. Indeed, there is interest in **batch KV caching** – keeping caches across batched requests so that a new batch can include some prompts that already have partial cache filled [15] [16]. As of early 2026, MLX-LM's batch generator will build caches for the batch and discard them after use (no built-in persistent merge yet) [15]. For now, **best practice** is to use caching within each session for speed, and if you have recurring prompts, manually reuse those via the prompt cache mechanism.

In summary, **use KV caching by default** – MLX-LM does this automatically in generation functions. Ensure your generate calls do not disable the cache. In chat scenarios, keep the `model` object alive for the session so its cache remains warm. If you restart the model or process each request, you lose the cache benefit (so avoid unloading the model between turns). When using the MLX OpenAI server or similar, it will manage session caching for you (just like OpenAI's API, passing a conversation ID keeps context). Caching combined with batching yields the best of both: reuse past computations *and* do as much new computation in parallel as possible.

Built-in Capabilities vs. User Implementation

MLX-LM has rapidly evolved, and many concurrency features are now built-in or available in the official ecosystem:

- **Multi-threaded GPU execution:** MLX (the lower-level array library) will utilize the GPU's parallelism automatically for matrix ops. You don't manually spawn threads for that; it's inherent. For CPU-bound parts, MLX can also use multiple CPU threads (e.g. for tokenization or small ops). These are mostly under-the-hood optimizations.

- **Batch Generation:** As of MLX-LM 0.30+, batch inference is an official feature. The `mlx_lm.generate` function remains for single prompts, but MLX-LM provides examples for batched usage [17] and a `BatchGenerator` class in its API. The development community even contributed a non-breaking PR to integrate ideas from **mlx_parallm** (the third-party batching library) into MLX-LM natively [18]. The result is that you can call MLX's batch generate and get optimized performance similar to ParaLLM, without patching MLX yourself. Under the hood, this includes a **BatchedKVCache** and auto-padding of prompts [19] [20] to handle different lengths. Initial releases lacked some features (e.g. repetition penalty, streaming of batch outputs) [21], but those are being added over time. The MLX team has been fixing batch-related bugs in recent releases (e.g. handling of caches, edge cases for specific models [22] [23]).

- **Continuous Batching in Server:** The MLX-LM package includes a CLI server (`mlx_lm.server`) primarily for interactive use, but it was not originally designed for high-concurrency. Recognizing this, the MLX developers (Awni et al.) have recently added continuous batching *to the server component* as well [4]. This means if you launch the server and enable concurrency, it will accumulate concurrent `generate` requests and process them in shared batches. In the v0.30.3 release notes, for example, there's mention of a new *"server benchmark for continuous batching"* and associated fixes [24]. In short, **MLX-LM now has built-in support for concurrent serving** – you just need to configure it and use the right interface.

- **Queueing and Scheduling:** While MLX-LM does the heavy lifting, it doesn't itself provide a full HTTP API or multi-client queue out-of-the-box (beyond the basic server CLI). However, community projects fill this gap. Notably, `mlx-openai-server` (an OpenAI-compatible FastAPI server for MLX) implements request queuing and concurrency controls on top of MLX-LM [25] [26]. It supports text, multimodal, and other model types, and allows setting `--max-concurrency` for parallelism, plus a `--queue-timeout` for pending requests [6]. This tool will manage incoming HTTP requests, hold them in a queue, and feed them to MLX in batches. Similarly, the **vLLM-MLX** project provides a drop-in OpenAI-like server that uses MLX under the hood and advertises continuous batching and multi-modal support [27]. These are effectively ready-made inference servers leveraging MLX's capabilities.

What **still requires user effort**? If you have very custom needs, you might need to modify or extend MLX-LM. For example, if you wanted to implement a bespoke scheduling algorithm (say prioritizing certain users or mixing models in one process), you'd handle that at the application level. Also, running **multiple models** concurrently (e.g. a LLaMA 2 and a Mistral model at the same time) is up to you – you could either load both into one process (if memory allows) and run one while the other is idle, or run separate processes for each model. MLX won't automatically distribute queries to different models; that logic lives in your server/

application. Another area is **caching across sessions** – if you want to persist KV cache beyond process lifetime (for example, warm-start common prefixes after a restart), you'd implement loading of prompt cache files yourself. The core MLX-LM gives you the tools (save/load cache), but not a turnkey solution for sharing caches between users or runs.

Summary: Out-of-the-box, MLX-LM now handles *intra-model concurrency* (batching, parallel token generation, KV caching) for a single model. What you as the developer must handle is the *inter-client orchestration* – i.e. accepting multiple user requests, deciding when to batch them, and returning responses to the right user. Fortunately, projects like `mlx-openai-server` handle most of this, so you don't have to reinvent it. You primarily need to choose the right approach and tune the settings (batch size, concurrency level, etc.) for your deployment.

Best Practices for Development and Deployment

Development Tips (Modifying or Using MLX/MLX-LM):
1. **Stay Updated:** Make sure you're using the latest version of `mlx` and `mlx-lm`. The MLX ecosystem is evolving fast (as of Jan 2026, we're seeing frequent improvements to batching and caching). New releases often bring performance boosts or bug fixes for concurrency. For example, sliding-window batch fixes and continuous batching were added in recent versions [22] [24]. Upgrading can save you from doing manual workarounds.
2. **Use the Official API for Batching:** Instead of writing your own multi-thread loops calling `generate` in parallel, use MLX-LM's batch generation functions. They are optimized in C++/Metal and ensure proper cache usage. The code snippet above using `BatchGenerator` illustrates how to feed multiple prompts. If you find the high-level API lacking (say you need streaming returns for batched queries), consider contributing to MLX-LM or checking GitHub discussions – continuous improvements are often guided by user feedback.
3. **Leverage Async I/O if needed:** If building your own server in Python (e.g. with FastAPI or Flask), write your endpoints to handle requests asynchronously. You can offload the MLX generation call to a background thread or task queue. For example, `async def generate_text(...)` could `await` a threadpool executor running `mlx_lm.generate`. This way, your server can accept other requests while one is generating. Python's `concurrent.futures` or `anyio` libraries are useful here. This pattern is essentially what the OpenAI-compatible servers use internally.
4. **Monitor Resource Usage:** During development, use MLX's debugging utilities to profile memory and speed. The guide snippet in the DZone article shows how to check available memory and MLX's allocated/peak memory [9]. Also measure token throughput and latency with different configurations (quantization, batch sizes, etc.). This will inform how to configure your deployment.

Deployment Recommendations (Inference Server Setup):
1. **Use a Purpose-Built Server**: Rather than writing a server from scratch, it's recommended to use an existing solution like **mlx-openai-server** or **vLLM-MLX**. These provide a **REST API** (matching OpenAI's format) which makes integration easy, and they implement queuing, batching and streaming properly. For example, `mlx-openai-server` can be launched with a single command and will serve **multiple users with queuing and concurrency**:

```
mlx-openai-server launch --model-path ./MyModel-mlx --model-type lm \
    --max-concurrency 4 --queue-timeout 300
```

Here `--max-concurrency 4` allows up to 4 requests to be processed in parallel (batched if possible) [26], and additional requests will wait in the queue (up to 300s) [26]. This server also supports *multimodal* models (`--model-type multimodal`) if you plan to serve image+text models like **Phi** or others. Using such a server ensures you benefit from **built-in continuous batching** without extra effort – as one developer announced, *"MLX-LM's server's new continuous batching is ~50% faster than llama.cpp at 100 parallel inferences"* in their tests [28]. In short, you get high throughput and token streaming over an HTTP API out-of-the-box.

1. **Quantize and Optimize Models:** For best performance on Apple Silicon, use MLX's quantization support. MLX-LM can convert models to 4-bit or 8-bit in seconds [29], drastically reducing memory per model. This enables larger batch sizes and/or running multiple model instances. Quantized models often *increase* throughput due to lower memory bandwidth usage [30]. Ensure you quantize to a format MLX supports natively (e.g., MXFP4 or 4-bit integer). Also consider using the smallest model that meets your users' needs (e.g., a 7B model will obviously infer faster and allow more concurrency than a 30B model on the same chip). In practice, an M4 Max can comfortably handle a quantized 13B or 30B model with moderate batch sizes [31] [32], but pushing beyond that (e.g., multiple 30B models concurrently) could exhaust memory.

2. **Configure Batch and Timeout Policies:** In deployment, you might tweak how aggressive the batching is. For instance, if you care about latency, you might set a very short batch collect interval (so that the server doesn't wait long to group requests). If throughput is the priority (e.g., an offline data generation job), you could allow slight delays to form larger batches. Similarly, tune `--max-concurrency` to your scenario – setting it equal to your batch size limit. Start with perhaps 4 or 8, then increase if you see the GPU is underutilized and memory is still available. The anecdotal evidence suggests beyond 32 concurrent streams, returns diminish on current M-series Macs [8], but your mileage may vary.

3. **Use Session Stickiness for Chats:** If deploying a chatbot service, make sure each user's consecutive messages go to the same model instance and reuse the same session context (cache). If you scale out with multiple processes or machines, implement a routing so that user X always hits the same backend for the duration of a conversation. This way the KV cache is warm and response time will be much better on later turns. MLX's rotating cache will handle memory, so you can safely allow long conversations – just be mindful that very large histories (>>4k tokens) will lead to older parts being dropped [10] (which is usually acceptable). If truly long contexts are needed, consider models fine-tuned for that or use MLX's support for segmented long context (there are specialized techniques outside scope here).

4. **Testing Under Load:** Before serving real users, simulate multiple concurrent requests to see how your setup behaves. Measure the average latency per request and throughput (tokens/sec or requests/sec) as concurrency rises. Check that the GPU usage is high (near 100% utilization) – if not, you might not be batching effectively. Also ensure the system isn't running out of memory; if you see swapping or OOM killer events, reduce batch size or quantize further. MLX's unified memory means GPU and CPU share RAM, so leaving some headroom is wise (don't use *all* available RAM for the model, or the OS might compress memory and slow things down).

In conclusion, **MLX on Apple Silicon can indeed handle concurrent inference** with the right approach. Use MLX-LM's built-in batching and caching to exploit parallelism, and wrap it in a robust inference server that queues and merges requests. All model types supported by MLX (LLaMA, Mistral, Phi, etc.) can benefit – the library abstracts the model details so that batching, caching, and quantization work uniformly. As of January 2026, the combination of MLX-LM 0.30+ and community servers delivers **production-grade local inference**: high throughput (hundreds of tokens/sec) [33] [34], reasonable latencies, and support for multi-user workloads with privacy and efficiency. By following these best practices, you can serve multiple users on your M4 Max Mac with impressive performance, tapping into Apple Silicon's full potential for ML inference.

Sources

- Apple Machine Learning Research blog – *Exploring LLMs with MLX on M5 GPU* (performance and quantization) [29] [31]
- MLX-LM GitHub – Release notes and issues on batch and continuous batching [24] [5]
- MLX-LM Documentation – Python API usage and examples (generation, batch generation, streaming) [17] [35]
- Reddit r/LocalLLaMA – Community experiments with MLX batching and throughput [7], and announcement of vLLM-MLX (OpenAI API with continuous batching) [3]
- ChatPaper arXiv summary – *Comparative study of LLM runtimes on Apple Silicon* (MLX's rotating cache and memory behavior) [10]
- LM Studio Blog – *Unified MLX engine architecture* (prompt caching benefits for follow-up queries) [12]
- **mlx-openai-server** project (FastAPI server for MLX) – documentation of features (multimodal support, queuing, concurrency) [25] [6]
- Medium article – *MLX_LM Continuous Batching* by Melkor (implementing continuous batch via multithreading) [2]

---

1 Parallel requests on Apple Silicon Macs with mlx-vlm? : r/LocalLLM
https://www.reddit.com/r/LocalLLM/comments/1pic6br/parallel_requests_on_apple_silicon_macs_with/

2 MLX_LM Continuous Batching. Apple Silicon is one of the easiest… | by Melkor | Medium
https://medium.com/@clnaveen/mlx-lm-continuous-batching-e060c73e7d98

3 27 vLLM-MLX: Native Apple Silicon LLM inference - 464 tok/s on M4 Max : r/LocalLLaMA
https://www.reddit.com/r/LocalLLaMA/comments/1qeley8/vllmmlx_native_apple_silicon_llm_inference_464/

4 Continuos batching on mlx_lm server! MLX is ready to fly! Thanks ...
https://x.com/ivanfioravanti/status/1996396612172816746

5 Support batching in mlx_lm.server · Issue #499 · ml-explore/mlx-lm
https://github.com/ml-explore/mlx-lm/issues/499

6 25 26 mlx-openai-server · PyPI
https://pypi.org/project/mlx-openai-server/

7 8 MLX batch generation is pretty cool! : r/LocalLLaMA
https://www.reddit.com/r/LocalLLaMA/comments/1fodyal/mlx_batch_generation_is_pretty_cool/

9 Fine-Tuning LLMs Locally Using MLX LM
https://dzone.com/articles/fine-tuning-llms-locally-using-mlx-lm-guide

10 11 13 30 33 34 Production-Grade Local LLM Inference on Apple Silicon: A Comparative Study of MLX, MLC-LLM, Ollama, llama.cpp, and PyTorch MPS
https://chatpaper.com/paper/208032

12 Introducing the unified multi-modal MLX engine architecture in LM Studio | LM Studio Blog
https://lmstudio.ai/blog/unified-mlx-engine

14 15 16 Plans for batch KV cache? · Issue #548 · ml-explore/mlx-lm - GitHub
https://github.com/ml-explore/mlx-lm/issues/548

17 35 mlx-lm · PyPI
https://pypi.org/project/mlx-lm/

18 19 20 21 willccbb/mlx_parallm: Fast parallel LLM inference for MLX - GitHub
https://github.com/willccbb/mlx_parallm

22 23 24 ml-explore/mlx-lm v0.30.3 on GitHub
https://newreleases.io/project/github/ml-explore/mlx-lm/release/v0.30.3

28 MLX batched/continous inference with structured outputs - Reddit
https://www.reddit.com/r/LocalLLaMA/comments/1qjpjt6/mlx_batchedcontinous_inference_with_structured/

29 31 32 Exploring LLMs with MLX and the Neural Accelerators in the M5 GPU - Apple Machine Learning Research
https://machinelearning.apple.com/research/exploring-llms-mlx-m5