How we improved AI inference on macOS Podman containers

June 5, 2025 | Kevin Pouget

Related topics: Containers, Developer Tools, Virtualization

Related products: Podman Desktop, Red Hat AI, Red Hat OpenShift AI

Share:

Table of contents:

Containers are technologies that allow the packaging and isolation of applications, along with their entire runtime environment. This eases the transition between environments (dev, test, production), but also helps enforce security policies with regards to the network access, file access, etc. In the world of AI, tools like Podman Desktop AI Lab and RamaLama rely on Podman containers to let users run large language models (LLM) locally, while Red Hat OpenShift AI runs them at scale on OpenShift Kubernetes clusters.

However, containers are Linux, and although they can run in different Linux distributions, they cannot run without a Linux kernel. The Podman solution to this challenge is (lightweight) virtual machines (VMs). A VM, launched by Podman machine, creates a virtual environment inside the macOS system, where a Linux environment runs and waits to create containers on demand. The macOS network and the home file system are passed to the VM, so that the virtualization layer is mostly transparent for the user.

Challenges of GPU access in virtual machines

When discussing AI, the question of performance is critical. And because performance gains in AI processing are largely due to GPU offloading, the question of GPU access within a VM is inextricably linked to performance.

With hardware-assisted virtualization, VMs are able to run natively on the host processor, allowing for near-native performance.

However, it is more complex to offload computations to physical GPUs, as they are not exposed to the VM directly. Indeed, an original goal of the virtual machine is to *isolate* the virtual system from the physical one; therefore, we must find a way to break this isolation and give the VM access to the physical GPU. There are various ways to do this:

- **Full device passthrough**. This technique requires a logical disconnection of the device from the host system (macOS) and passing the relevant memory mappings to the guest filesystem, so that the VM can access the device memory registers directly, as when running bare-metal. In the case of the GPU of a user workstation, this technique is impossible to follow as the host system relies on the GPU device to control the screen.
- **Hardware-assisted device passthrough.** This technique is similar to full device passthrough, except that the device has been designed with virtualization in mind, exposing multiple memory-mapped control interfaces. This allows the host system to remain in control of the main interface, while the VM can receive a secondary interface. This technique is impossible to follow on Apple Silicon as the hardware does not offer such secondary control interfaces.
- **Virtual device emulation.** This technique requires the hypervisor to expose a virtual device to the VM, with the same memory-mapped interface as a physical device. This way, the guest can load the actual driver of the physical device and transparently use its software stack. This technique is impossible to follow as the Apple Silicon drivers and software stack isn't available on Linux. Besides, the implementation of the virtual-device to physical-device bridge would not be feasible either.
- **Paravirtualized device.** This class of techniques involves a cooperation between the hypervisor and the guest system. The hypervisor exposes virtual devices with no direct hardware counterpart, and the guest loads virtualization-aware drivers and libraries.

This guest-host communication channel is used to offer the VM services that are implemented in the host system. This technique is used to share files between the two systems, but also give network access, share the desktop screen, enable copy-and-paste, etc. The technique can also be used to access hardware accelerators such as GPU by forwarding the compute requests and responses between the two systems.

Device paravirtualization is what has been implemented in the Podman Machine/ `libkrun` compute stack to benefit from GPU acceleration in the virtual machine. We describe this stack in the next section.

Running container GPU acceleration on macOS

Running GPU-accelerated containers on macOS involves a rather complex stack. Let us have a look at it.

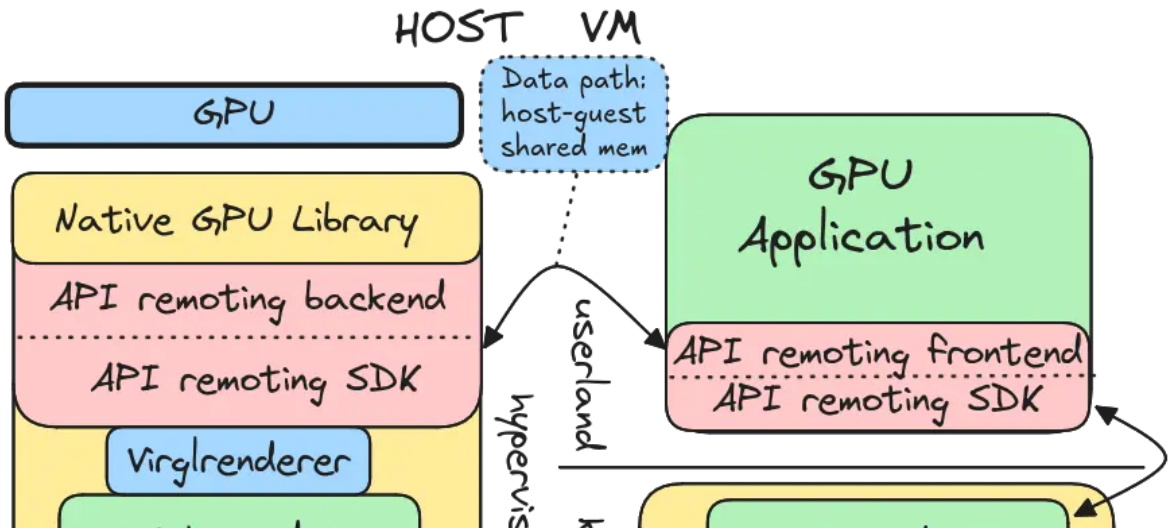
1. At the top of the stack, the user runs Podman Desktop or RamaLama, the user interface applications.
2. Below it, the user-interface applications rely on Podman machine to create the lightweight virtual machine with the right settings, so that the VM has access to the network and user home directory, and exposes the Podman remote control interface.
3. The virtual machine is managed by krunkit and libkrun, a lightweight virtual machine manager (VMM) based on Apple's low-level Hypervisor Framework.
4. `libkrun` exposes a `virtio-gpu` virtual device that the guest system can control with the Virtio-GPU Linux driver.
5. In the guest system, the Vulkan-virtio library of the MESA project implements the frontend of the API forwarding mechanism. The MESA project is an open source implementation of OpenGL, Vulkan and other graphics APIs. The MESA library implements the Vulkan API, and sends the Vulkan API calls via the Venus protocol through a guest-host shared memory page. The shared memory page has

been provided by the hypervisor, via the `virtio-gpu` virtual device. This shared memory communication link is the one used for the data exchanges during the inference queries (data path), as opposed to the guest-kernel <> hypervisor link, only used for passing small-size parameters.

6. In the host system, the virglrenderer of the MESA project implements the backend helper of the API forwarding mechanism. It receives the Venus messages sent by the frontend, and calls the Vulkan library accordingly.
7. The MoltenVK Chronos project implements the Vulkan API on macOS, and translates the API calls to the Apple Metal/MPS GPU acceleration libraries.
8. Finally, the Apple Metal/MPS GPU acceleration libraries control the physical GPU and drive computation acceleration offloading.

Now that we have seen the layers of the GPU acceleration stack, we need to add the cherry on top of it, without which the stack wouldn't be exercised: the AI inference engine. In Podman Desktop and RamaLama, llama.cpp is the engine that operates the AI model and it runs in a container. It is in charge of loading the model weights in the GPU, and performs the AI inference queries. `llama.cpp` and its `ggml-vulkan` backend puts in motion the full GPU acceleration stack described above to offer an API endpoint to the user.

Figure 1 shows an overview of the macOS GPU acceleration stack.



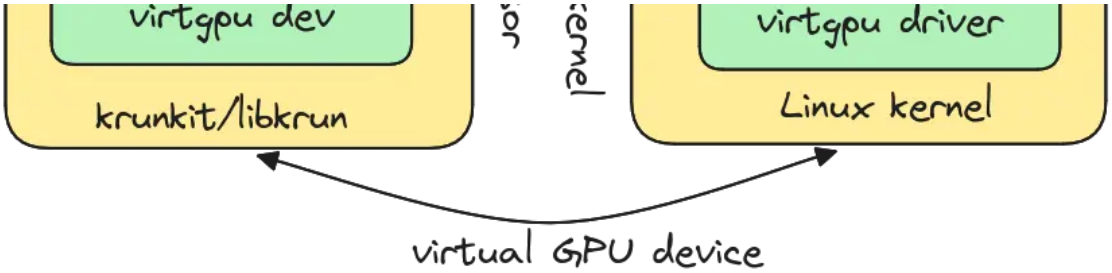


Figure 1: Overview of the GPU computing virtualization stack on macOS.

Improvement of the macOS container GPU computing stack performance

The performance of the macOS container GPU computing stack has greatly improved in recent times, thanks to the introduction of the Vulkan API forwarding in `libkrun`, and the improvements of the `ggml-vulkan` backend of `llama.cpp`.

Figure 2 depicts this evolution. In this test, we used RamaLama to launch an inference server, but the core improvements really come from `libkrun` and `llama.cpp`.

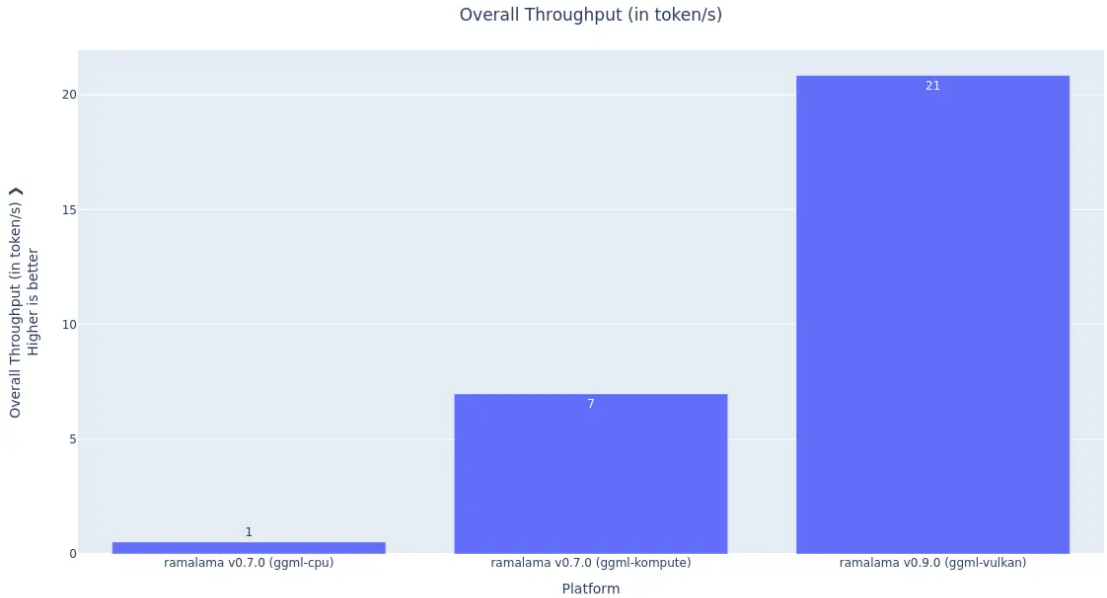


Figure 2: Comparison on the inference throughput (in token/s, higher is better) for different RamaLama versions and back ends (running inside a virtualized Linux container).

Here are the takeaways:

- In `RamaLama v0.7.0 (ggml-cpu)`, we explicitly disabled the use of the GPU acceleration in the container, to reproduce the performance available before `libkrun` GPU acceleration.
  - The inference throughput was 0.52 tokens per second.
- In `RamaLama v0.7.0 (ggml-kompute)`, RamaLama runs with a llama.cpp image based on the `ggml-kompute` back end.
  - The inference throughput was 6.97 tokens per second.
  - This is 13x better than without the GPU acceleration.
- In `RamaLama v0.9.0 (ggml-vulkan)`, RamaLama runs with a `llama.cpp` image based on the `ggml-vulkan` back end.
  - The inference throughput is 20.84 tokens per seconds.
  - This is 2.99x better than with the `llama.cpp/ ggml-kompute` image.

The combination of the `libkrun` GPU acceleration and the `llama.cpp` / `vulkan` improvements give an overall 40x improvement of the previous macOS container GPU computing performance.

Comparison of the container performance and native performance

Now, let us have a look at the performance comparison between the container and native execution.

Figure 3 shows the comparison of the RamaLama performance (running inside a VM container) against the native `llama.cpp` performance, with the `ggml-metal` and `ggml-vulkan` back end.

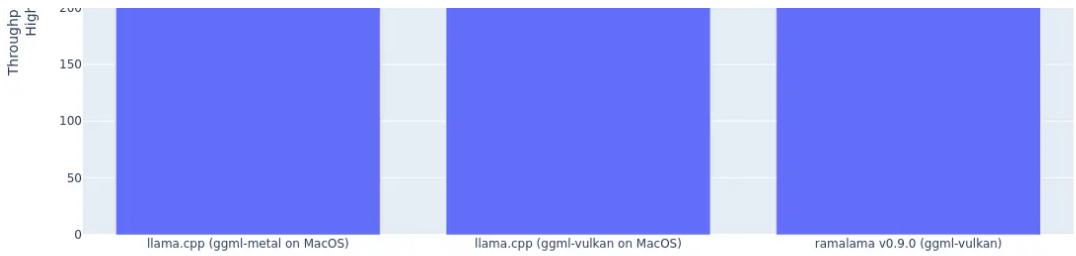


Figure 3: Comparison on the inference throughput (in token/s, higher is better) for llama.cpp running on macOS with the Metal and Vulkan backends, and RamaLama running inside a Linux virtualized container.

In this plot, we see the following interesting comparisons:

- `RamaLama v0.9.0 (ggml-vulkan)` and `llama.cpp (ggml-vulkan on macOS)` are on par, which is a great result. This indicates that `libkrun` Vulkan API forwarding has minimal performance overhead. It is great, because it means that the GPU computation can pass the VM isolation without inducing any significant performance overhead. This low overhead can be explained by multiple factors:
  - The GPU computing API has a coarse granularity, meaning that the API forwarding mechanism is sparsely involved.
  - The GPU computing kernel execution time is orders of magnitude higher than the share-memory communication overhead.
- `llama.cpp (ggml-vulkan on macOS)` performs at 77% of llama.cpp (ggml-metal on macOS). This means that, when running **natively**, the `llama.cpp` `ggml-vulkan` backend performs slower than the `llama.cpp` `ggml-metal` back end. This difference can also be explained by multiple factors:
  - The MoltenVK Vulkan library operates on top of the Apple Metal API, so the Vulkan kernels have to be transpiled to Metal kernels.
  - The MoltenVK Vulkan library does not support the coop matrix feature (see this presentation from the Vulkanised 2025 developer conference).

To complete the performance evaluation, let us have a look at the

To complete the performance evaluation, let us have a look at the inference performance for various model sizes: `1b`, `3b`, `8b` and `13b`. See Figure 4.

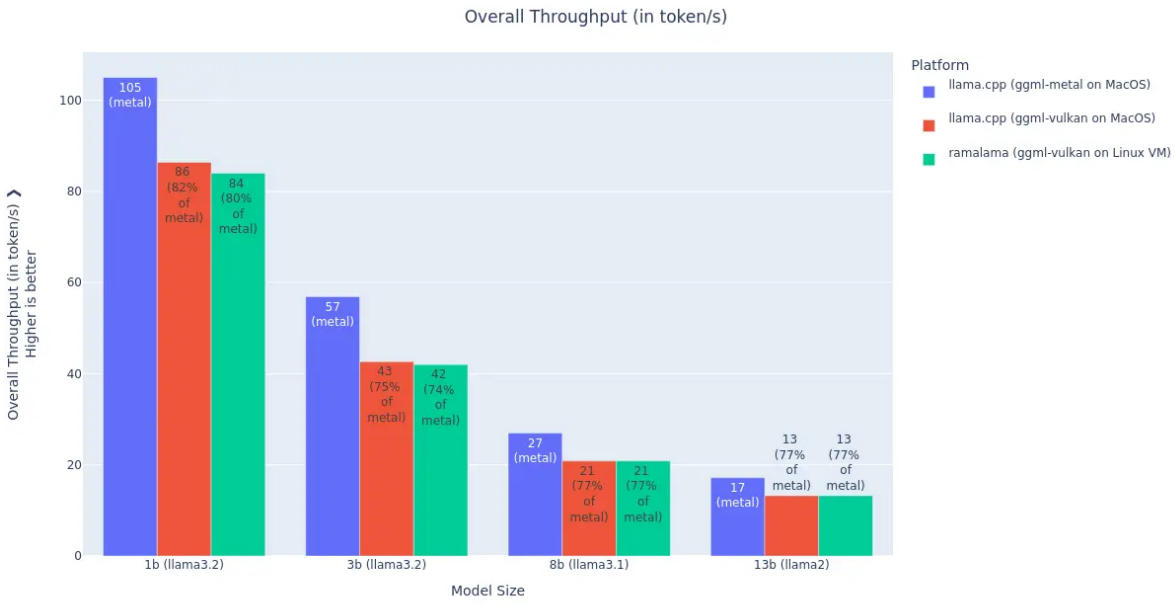


Figure 4: Comparison on the inference throughput (in token/s, higher is better) for llama.cpp running on macOS and RamaLama, with different AI model sizes.

We can see that the overall throughput varies significantly, according to the model size. This is expected, as the number of computations to perform increases when the number of weights in the model increases. But what is interesting to observe here is that the performance gap between the native Metal execution and the Vulkan back end remains stable:

- between 75% and 77% of the Metal performance for `llama-cpp ggml-vulkan` on macOS.
- between 74% and 80% of the Metal performance for `RamaLama v0.9.0` on Podman containers.

In the last section, after the information about the system under test and the test harness, we detail our current and future steps to stabilize the overall stack, open it to new workload, and investigate how to further improve the performance.

Information about the system under test and the test harness

Harness

Here's a breakdown of our system setup and test tools:

- The inference server is:
  - `RamaLama 0.7.0` and `0.9.0`
  - `llama.cpp b5581(2025-06-03)`
  - `krunkit 0.1.4`
- The inference server is running in this system:
  - `Apple MacBook M4 Pro, 48GB`
  - `macOS 15.5, Darwin 24.5.0`
- Unless otherwise specified, the model used is llama3.1:8b.
- For the multi-model test, the models are llama3.2:1b, llama3.2:3b, llama3.1:8b, and llama2:13b.
- The load generator is openshift-psap/llm-load-test.
- The metric shown in the plots above is the overall throughput, which includes the time-to-first token and the time-per-output-token.
- The test harness is openshift-psap/topsail mac_ai project.
- The tests have been executed under the control of TOPSAIL CI automation, in a transparent, reviewable and reproducible fashion. The tests artifacts are available at the following addresses:
  - Test 1
    - `RamaLama v0.7.0, ggml-kompute backend`
    - `RamaLama v0.7.0, ggml-cpu backend`
  - Test 2
    - `RamaLama v0.9.0, ggml-vulkan backend`
  - Test 3
    - `llama.cpp native, ggml-metal backend`
    - `llama.cpp native, ggml-vulkan backend`
    - `RamaLama v0.9.0, ggml-vulkan backend`

    - Test 4
        - `llama.cpp native, ggml-metal backend`
        - `llama.cpp native, ggml-vulkan backend`
        - `RamaLama v0.9.0, ggml-vulkan backend`
        - `llama3.2:1b`, `llama3.2:3b`, `llama3.1:8b` and `llama2:13b` from ollama library
- The plots above have been manually generated **based on the CI test artifacts** (only to improve the naming and readability).

Ongoing and future work

As you can see from the performance measurements, AI inference on macOS containers got a speedup of 40x thanks to the introduction of Vulkan GPU acceleration in the `libkrun` project and the optimization of the `ggml-vulkan` back end of `llama.cpp` inference server. The main ongoing action on this topic is the upstream integration of the `virtio-gpu` shared memory page negotiation in the Linux `virtio` driver. Once merged, this will allow the use of Vulkan API forwarding in `libkrun` VM from 100% upstream code.

Podman Desktop AI Lab and RamaLama are also continuously improving with new GPU accelerated features built on top of `llama.cpp` that will be directly available to Apple Silicon Mac users.

We are also investigating how to improve the Vulkan GPU acceleration, as well as how to use API forwarding in other environments to enable new AI workloads in macOS containers.

And on the performance evaluation side, we will integrate the performance test harness in a continuous performance testing environment, to ensure that the macOS containers GPU acceleration performance do not degrade when switching to new releases of the compute stack components.

Last updated: June 10, 2025