Reach native speed with MacOS llama.cpp container inference

Reaching native performance on macOS llama.cpp container inference with API remoting

September 18, 2025 | Kevin Pouget

Related topics: APIs, Artificial intelligence, Containers, Developer Tools, Virtualization

Related products: Podman Desktop

Share:

Table of contents:

Containers are Linux, but they can still run on macOS with the help of a thin virtualization layer, inside a Linux virtual machine. But while the CPU and RAM are accessed at native speed, GPU acceleration has always been a challenge.

Earlier this year, we demonstrated how the enablement of Venus-Vulkan boosted GPU computing performance by 40x, reaching up to 75–80% of the native performance. In this post, we introduce a preview of our latest work, which closes the gap and brings `llama.cpp` AI inference to native speed in most use cases.

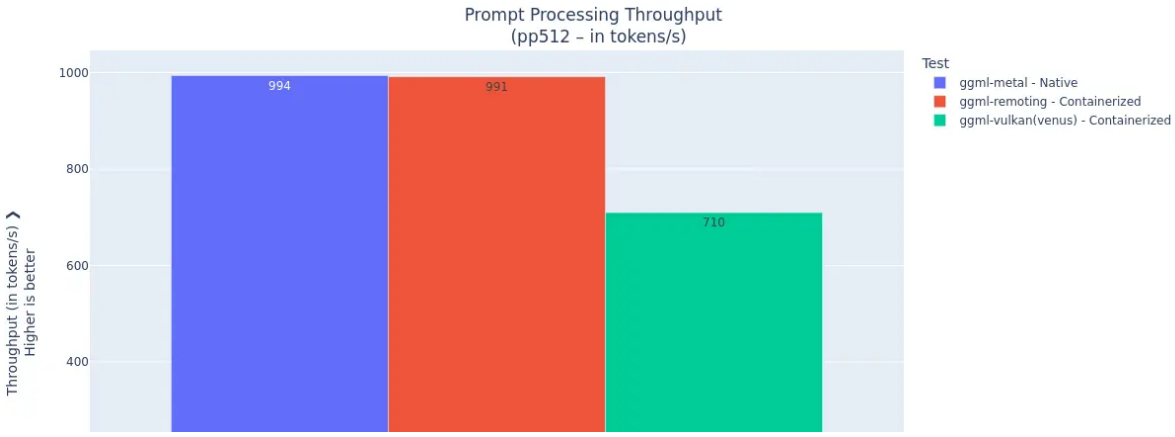


Figure 1: Token generation throughput performance comparison between native, API remoting, and Vulkan-Venus acceleration on the M4 Pro MacBook Pro 48 GB system.

The llama.cpp API remoting architecture

The challenge of running AI inference inside containers on macOS is that OCI containers do not run natively on macOS—they need the Linux kernel. So, Podman launches a Linux virtual machine, powered by the open source libkrun/krunkit projects.

Our API remoting accelerator runs on top of `libkrun`'s VirtIO `virt-gpu` support and leverages the same technique as Mesa Venus-Vulkan: forwarding API calls between the virtual machine and the host system. The Mesa project provides a general-purpose solution, where the calls to the Vulkan API are serialized with the Venus protocol, and forwards them to the host via the Virtio `virt-gpu`. On the host, the `virglrenderer` library deserializes the call parameters and invokes the MoltenVK Vulkan library, which is built on top of Apple Metal API.

On our acceleration module, we chose to focus on a narrower target: AI inference with `llama.cpp` GGML tensor library. The acceleration stack consists of four components:

- **ggml-remotingfrontend**, a custom GGML library implementation running in the container of the Linux virtual machine.
- `libkrun`'s **virtio-gpu** and its Linux driver (unmodified).
- **virglrenderer**, a modified version of the upstream library that supports loading a secondary library and forwarding calls to it.
- **ggml-remotingbackend**, a custom GGML library client running on the host. It receives call requests from the `virglrenderer` and invokes the `ggml-metal` library to drive the `llama.cpp` GPU

acceleration.

Figure 2 shows an overview of the architecture.

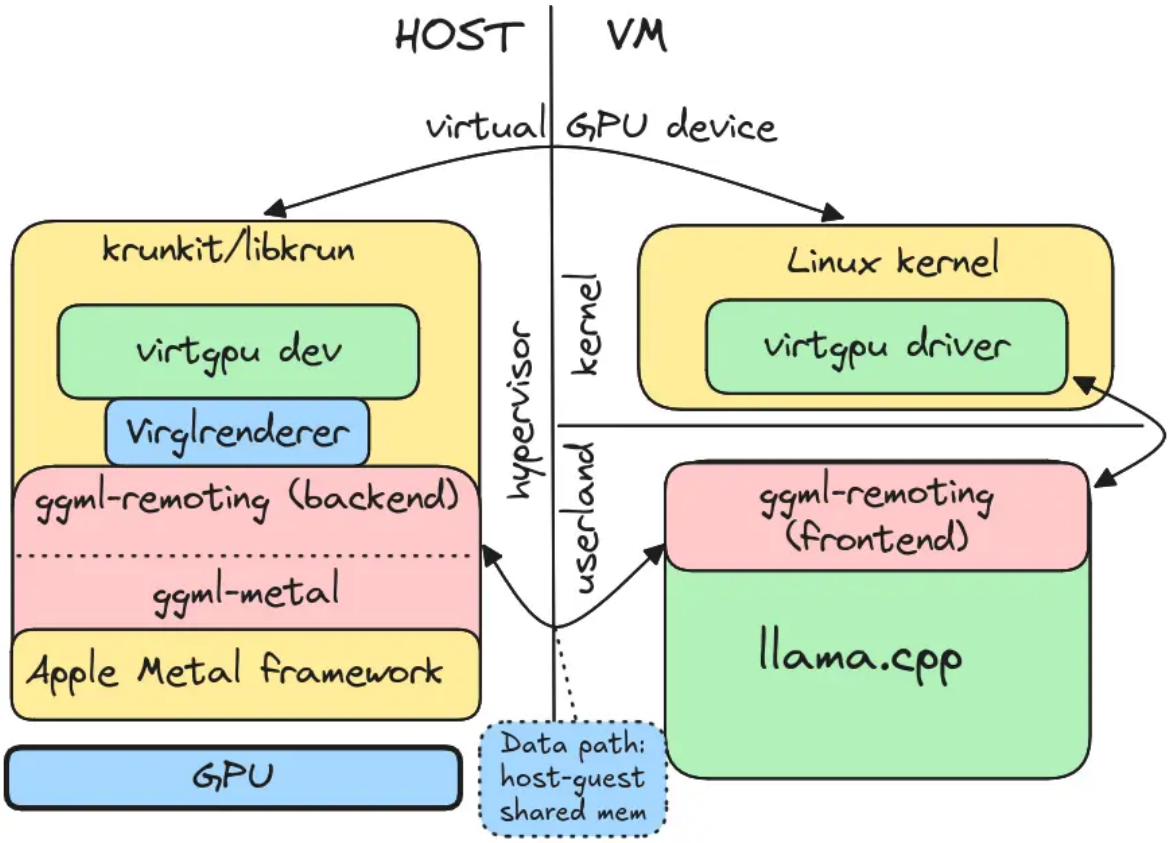


Figure 2: Overview of our `llama.cpp` API remoting architecture.

Considerations for API remoting

A frequent question about API remoting is whether it breaks VM/container isolation.

By nature, it creates a communication link between the VM container and the host system to access the GPU, so it does indeed breach part of the VM isolation. However, it's a matter of trade-offs between performance and strong isolation.

Key advantages in this proof-of-concept implementation:

- The VM hypervisor runs with simple user privileges. This is by design of the Podman machine/ `libkrun` stack and not specific to this work.
- The back end and the GPU only execute trusted code. It is

the `ggml-metal` back-end library, loaded by the hypervisor on the host, contains all the necessary GPU kernel code. This eliminates a whole class of risks coming from the execution of malicious kernels.

Current limitations and considerations:

- The back-end library lives within the hypervisor address space. A crash within the back-end library could bring down the hypervisor. This extends the risk of a vulnerability in the trusted code that can be exploited by a malicious model. The mature and sound `llama.cpp` code base, along with vulnerability management techniques and patching, would reduce this.
- The back-ends of multiple containers run in the same address space. (**Note:** While the current implementation only allows one container to run at a time, this limitation is slated to be lifted in future releases.) But from the GPU point of view, the different containers are *not* isolated:
  - In terms of execution, invalid operations from one container can crash another container.
  - In terms of security, one container *might* be able to access the GPU data of another container. This threat is mitigated by the fact that the containers do not provide the GPU kernel code; they can only trigger existing `ggml-metal` kernels. However, the threat is not fully eliminated; vulnerabilities could still exploit it.

Overall, we can say that this API remoting design isn't multitenant safe.

API remoting benchmarks

We validated the stability and performance of our acceleration stack by running the `llama.cpp`'s `llama-bench` benchmark over various model families, sizes, and quantizations on different Mac systems. We looked at the performance of prompt processing `pp512`, which measures how fast the AI engine processes the input text (relevant when

feeding large documents), and token generation (`tg128`), which measures how fast the AI generates text (important for the user experience).

Testing with various Mac hardware

Figures 3 and 4 show the prompt processing and token generation performance benchmarks with different Mac systems.

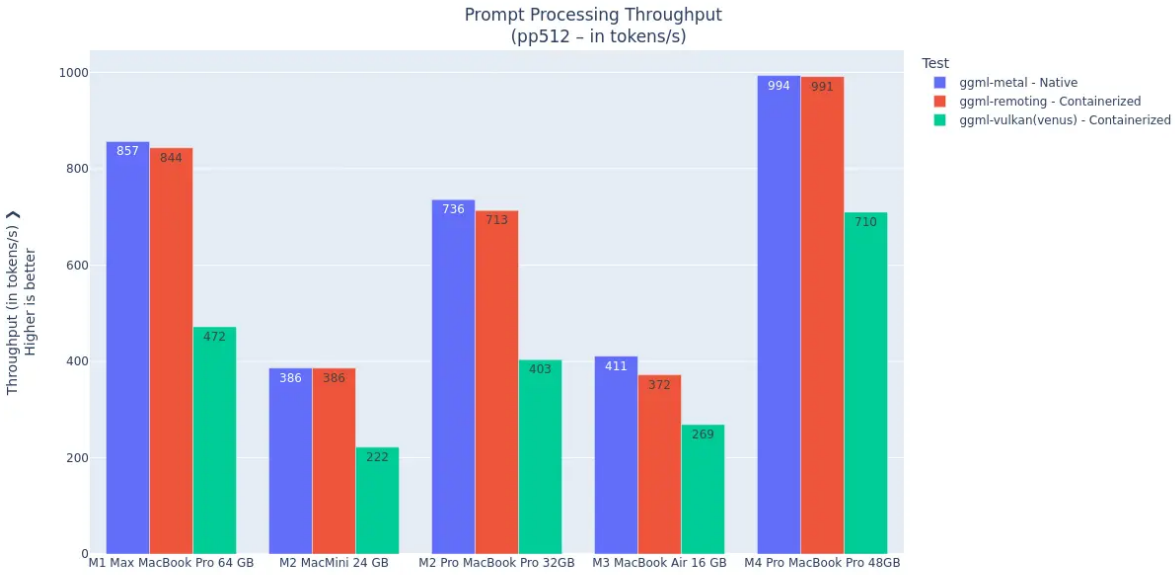


Figure 3: Prompt processing performance with various Mac hardware.

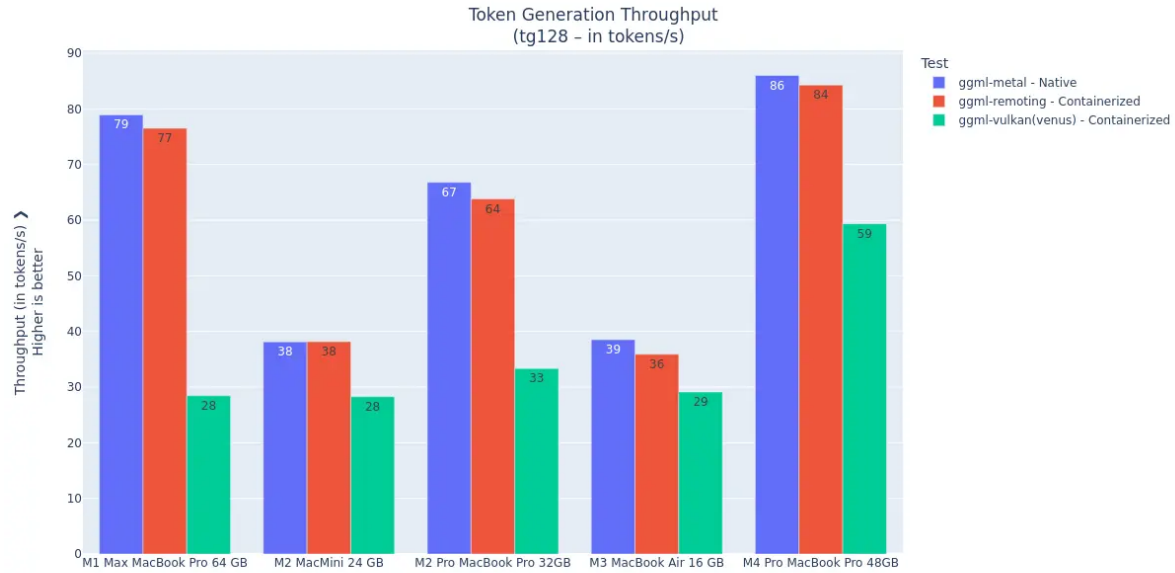


Figure 4: Token generation performance with various Mac hardware.

We can see that overall, the `llama.cpp` API remoting performance is mostly on par with native Metal performance. In the following tests, we only used the M4 Pro MacBook Pro 48 GB system.

The lower performance of the M2 Mac Mini and M3 MacBook Air systems most likely comes from the bottleneck of their RAM bandwidth, which is limited to 100 GB/s, while the other systems have more than double.

Testing with various model families

Figures 5 and 6 show the prompt processing and token generation performance benchmarks with different model families: granite3.3, llama3.2, mistral, phi, qwen (all from the Ollama repository, with the latest tag) on the M4 Pro MacBook Pro 48 GB system.

We see that with the `llama.cpp` API remoting acceleration, prompt processing and token generation are done at the native speed.

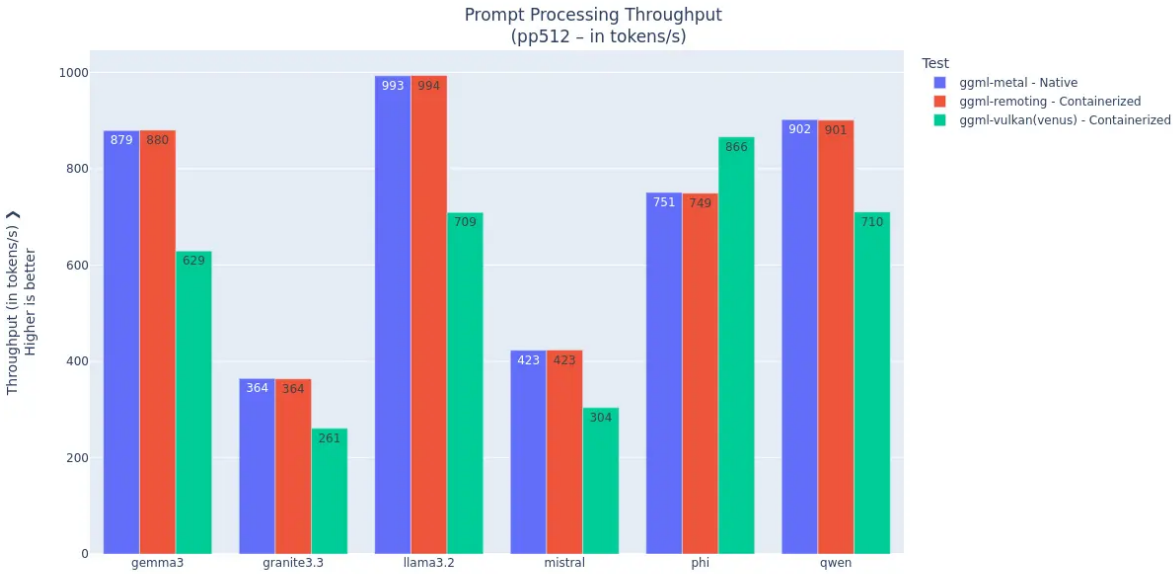
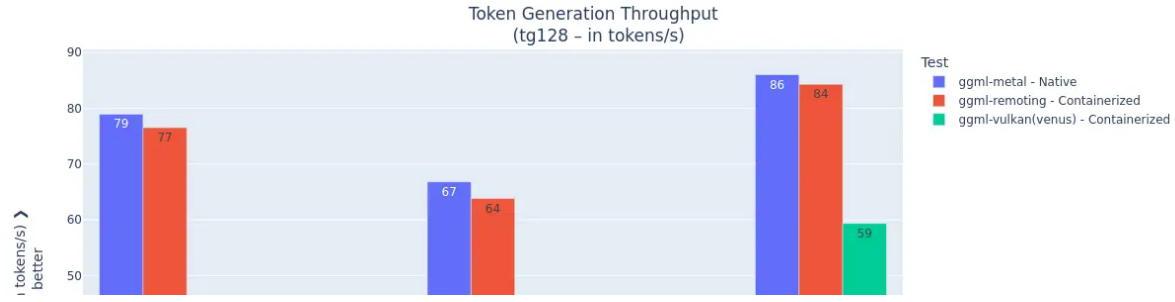


Figure 5: Prompt processing performance with various model families.



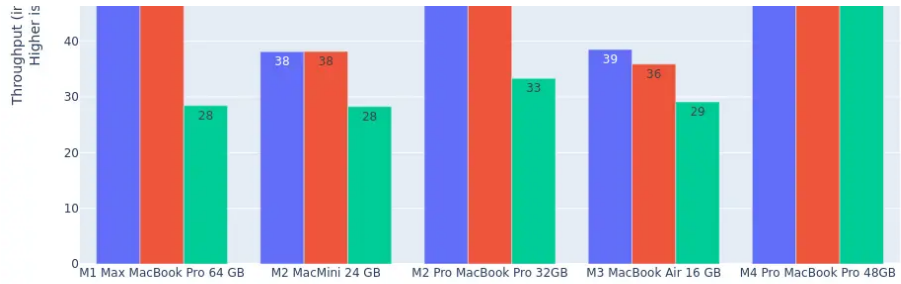


Figure 6: Token generation performance with various model families.

Testing with various model sizes

Figures 7 and 8 show the prompt processing and token generation performance benchmarks with different sizes of the `llama2`, `llama3.1`, and `llama3.2` models on the M4 Pro MacBook Pro 48 GB system.

We see that with the `llama.cpp` API remoting acceleration, prompt processing was done at native speed, and the token generation is near native. The highest difference (95% of native) is observed with the *smallest* model, where the processing time gets low and the API remoting overhead becomes visible:

- `151.59 t/s` ⇔ 1 token every `6.6ms`
- `145.29 t/s` ⇔ 1 token every `6.7ms`

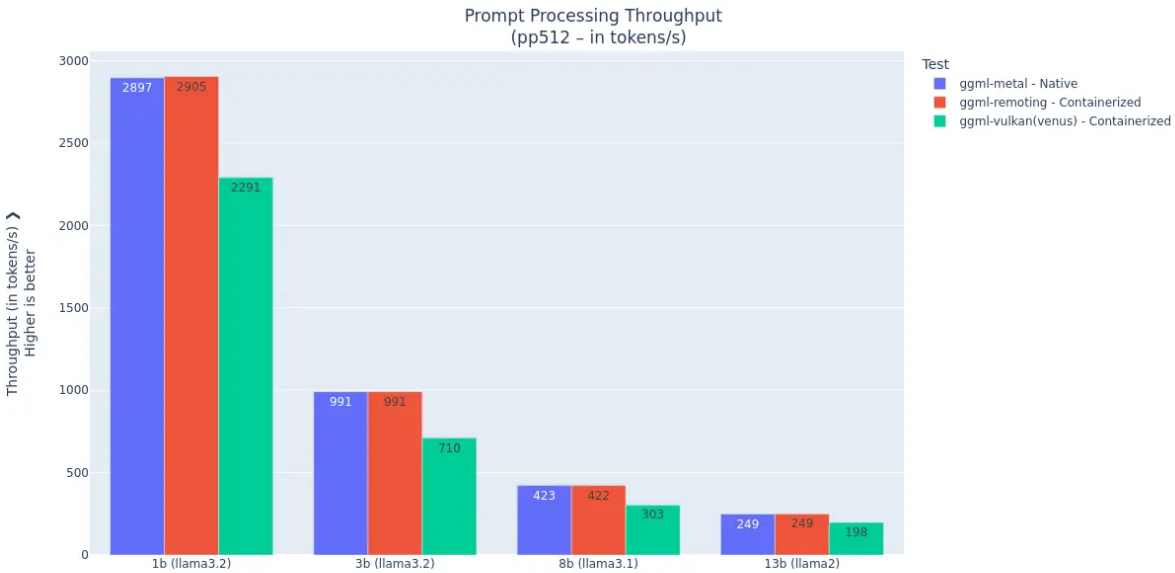


Figure 7: Prompt processing performance with Llama model sizes

Figure 7: Prompt processing performance with Llama model sizes.

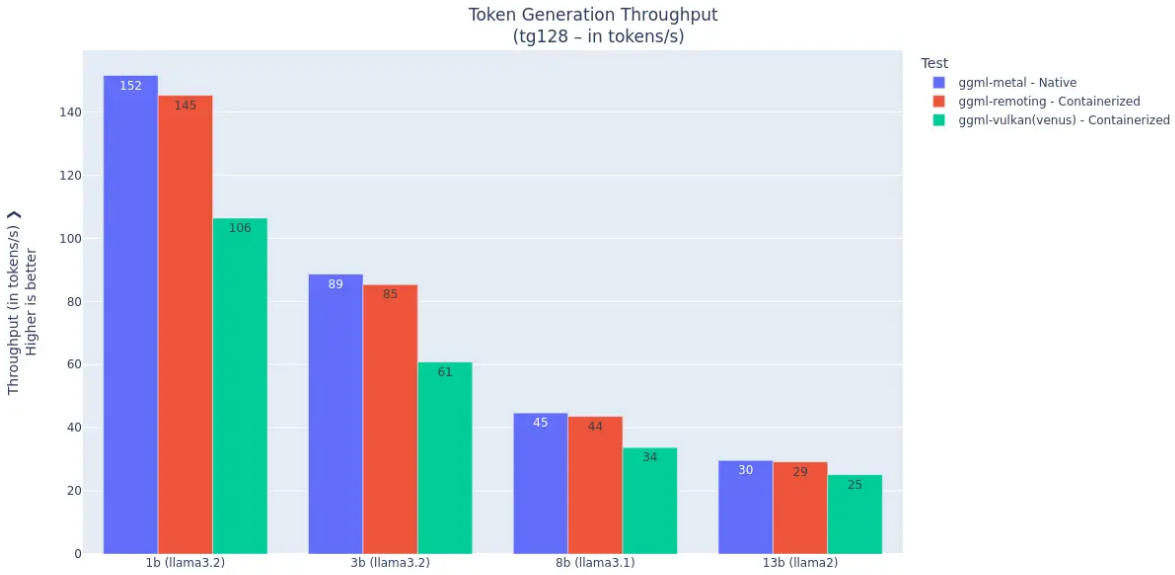


Figure 8: Token generation performance with various model families.

In the next section, I'll walk you through the steps to get the `llama.cpp` API remoting running on your system and how to run the back end to validate the performance.

Try with Podman Desktop

Install this extension in Podman Desktop:

```
quay.io/crcont/podman-desktop-remoting-ext:v0.1.3_b6298-remot
```

Copy snippet

Then select the following menus in the `llama.cpp` API remoting status bar:

1. **Restart Podman Machine with API remoting support:** This restarts the Podman machine with the API remoting binaries.
2. **Launch an API Remoting accelerated Inference Server:**
   a. Select the model.
   b. Enter a host port.
   c. Wait for the inference server to start. The first launch takes a bit

of time, as it pulls the RamaLama remoting image, and `llama.cpp` needs to precompile and cache its GPU kernels.

3. Play with the model that you launched, for example, with RamaLama:

```
ramalama chat --url http://127.0.0.1:1234
```

Copy snippet

See the Benchmarking section for comparing the performance of API remoting against Venus-Vulkan and native Metal on your system.

Try API remoting with RamaLama

1. Download the API remoting libraries:

```
curl -L -Ssf https://github.com/crc-org/llama.cpp/
```

Copy snippet

2. Ensure that you have the Podman machine and `krunkit` available (see the Prerequisites part of the tarball INSTALL.md), and RamaLama 0.12.
3. Prepare the `krunkit` binaries to run with the API remoting acceleration.

```
bash ./update_krunkit.sh
```

Copy snippet

4. Restart the Podman machine with the API remoting acceleration. You can pass an optional machine name to the script if you don't want to restart the default machine.

```
bash ./podman_start_machine.api_remoting.sh [MACHI
```

Copy snippet

5. Now you can use RamaLama with the remoting image:

```
export CONTAINERS_MACHINE_PROVIDER=libkrun # ensu
ramalama run --image quay.io/crcont/remoting:v0.1
```

Copy snippet

Again, see the Benchmarking section below for comparing the performance of API remoting against Venus-Vulkan and native Metal on your system.

Run the benchmark with RamaLama

To benchmark the API remoting performance, the easiest way is to use the RamaLama `llama.cpp` benchmark.

1. First, on Podman Desktop, stop any other API remoting inference server by selecting **Stop the API Remoting Inference Server.** Two API remoting containers cannot run simultaneously in this preview version.
2. Select **Show RamaLama benchmark commands.** This will show the following commands for RamaLama 0.12:

```
# API Remoting performance
ramalama bench --image quay.io/crcont/remoting:v0
```

Copy snippet

3. To compare it against the current container performance, launch the benchmark with the default image:

```
# Venus/Vulkan performance
ramalama bench llama3.2
```

Copy snippet

4. And to compare it with native performance, launch RamaLama without a container:

without a container:

```
brew install llama.cpp
# native Metal performance
ramalama --nocontainer bench llama3.2
```

Copy snippet

If you want to share your experience or performance on crc-org/llama.cpp, also include:

- The version of the tarball (`llama_cpp-api_remoting-b6298-remoting-0.1.6_b5`)
- The name of the container image (`quay.io/crcont/remoting:v0.12.1-apir.rc1_apir.b6298-remoting-0.1.6_b5`)
- Or the version of the Podman Desktop extension (`v0.1.3_b6298-remoting-0.1.6_b5`).
- The output of this command:

```
system_profiler SPSoftwareDataType SPHardwareDataT
```

Copy snippet

Conclusion

This project was started to evaluate the suitability of API remoting to improve the performance of containerized AI inside MacOS virtual machines. The initial investigations confirmed that the core components were available in the stack: host-guest memory sharing and the ability for the guest to trigger code execution on the host. The VirtIO Virt-GPU implementation, which spans between the Linux guest kernel and the hypervisor, already provides these capabilities.

So, the proof-of-concept development effort first focused on extracting and adapting the relevant code from Mesa Venus and Virglrenderer to make it reusable in more lightweight projects. The second focus was on

forwarding the GGML API calls between the guest and the host. The last focus was to optimize the interactions to improve the performance.

Next steps

The next steps for this work are to submit the changes upstream to the `llama.cpp` and `virglrenderer` projects. Once the patches are accepted, `libkrun/krunkit` and RamaLama/Podman Desktop will be extended to ship the API remoting libraries and enable them on demand.

Another step will be to review the use of the `llama.cpp` API remoting for RamaLama micro-VM, where `libkrun` is used in Linux systems to improve the isolation level of AI containers.

Finally, we are considering turning this API remoting project into a framework that could be used to enable new workloads to run with GPU acceleration inside virtual machines, such as PyTorch/MPS containers running on Apple macOS.

Related Posts

How we improved AI inference on macOS Podman containers

Introducing GPU support for Podman AI Lab

How RamaLama runs AI models in isolation by default

How RamaLama makes working with AI models boring

LLM Compressor is here: Faster inference with vLLM

Distributed inference with vLLM

Recent Posts

A case study in Kubelet regression in OpenShift