Lokale Installation und Betrieb von LLMs mit vLLM

vLLM ist ein hochdurchsatzorientiertes Inference-Framework für Hugging-Face-Modelle. Im Vergleich zu llama.cpp liegt der Fokus auf kontinuierlichem Batching und PagedAttention — das macht vLLM besonders interessant wenn mehrere Requests gleichzeitig bedient werden sollen. Für Einzelnutzer im Homelab ist llama.cpp oft die einfachere Wahl; vLLM lohnt sich wenn der Throughput wichtiger ist als unkomplizierte Konfiguration.

Voraussetzungen

- NVIDIA GPU mit CUDA-Support (Ampere oder neuer empfohlen)
- Mindestens 8 GB VRAM für kleinere Modelle; für 7B-Klasse realistisch 10–12 GB
- Python 3.10–3.12
- CUDA 12.1 oder neuer

CUDA und Treiber prüfen:

```
nvcc --version
nvidia-smi
```

System vorbereiten und uv installieren

`uv` ist ein moderner Python-Paketmanager von Astral — deutlich schneller als pip, zuverlässigere Dependency-Resolution, empfehlenswert für alle Python-basierten AI-Projekte.

Linux:

```
sudo apt update && sudo apt upgrade -y
sudo apt install curl
curl -LsSf https://astral.sh/uv/install.sh | sh
```

Shell neu laden damit `uv` im PATH liegt:

```
source ~/.bashrc
# oder
source ~/.zshrc
```

Windows:

```
powershell -ExecutionPolicy ByPass -c "irm
https://astral.sh/uv/install.ps1 | iex"
```

---

vLLM installieren

Eine virtuelle Umgebung anlegen und vLLM darin installieren:

```
uv venv vllm-env --python 3.12
source vllm-env/bin/activate
uv pip install vllm
```

Auf Windows:

```
uv venv vllm-env --python 3.12
vllm-env\Scripts\activate
uv pip install vllm
```

Die Installation zieht PyTorch mit CUDA-Support automatisch — das sind mehrere GB, also etwas Geduld einplanen.

Installation verifizieren:

```
python -c "import vllm; print(vllm.__version__)"
```

Erstes Modell starten

vLLM lädt Modelle direkt von Hugging Face. Beim ersten Start wird das Modell heruntergeladen und im Cache gespeichert (`~/.cache/huggingface`).

Beispiel mit Qwen3 4B (passt gut auf 12 GB VRAM):

```
vllm serve Qwen/Qwen3-4B-Instruct \
  --host 0.0.0.0 \
  --port 8000
```

Beispiel mit Gemma 3 12B auf 24 GB VRAM (Arbeitsumgebung):

```
vllm serve google/gemma-3-12b-it \
  --host 0.0.0.0 \
  --port 8000 \
  --dtype bfloat16
```

Wenn der Server läuft, ist die API unter `http://localhost:8000` erreichbar und OpenAI-kompatibel — LiteLLM, Open WebUI und andere Clients können direkt damit arbeiten.

API testen

Verfügbare Modelle anzeigen:

```
curl http://localhost:8000/v1/models
```

Chat-Request absetzen:

```
curl http://localhost:8000/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "Qwen/Qwen3-4B-Instruct",
    "messages": [
      {"role": "user", "content": "Was ist der Unterschied zwischen
vLLM und llama.cpp?"}
    ]
  }'
```

Wichtige Start-Parameter

Parameter	Beschreibung	Beispiel
`--host`	Bind-Adresse	`0.0.0.0` für Netzwerkzugriff

Parameter	Beschreibung	Beispiel
`--port`	Port	`8000`
`--max-model-len`	Maximale Kontextlänge	`8192`
`--gpu-memory-utilization`	VRAM-Auslastung (0–1)	`0.85`
`--dtype`	Datentyp	`bfloat16` oder `float16`
`--quantization`	Quantisierungsmethode	`awq`, `gptq`
`--tensor-parallel-size`	Multi-GPU-Parallelität	`2` für zwei GPUs

`--gpu-memory-utilization` ist praktisch wenn vLLM zu viel VRAM belegt und OOM wirft. Default ist 0.9 — auf 0.8 oder 0.85 reduzieren wenn andere Prozesse auch VRAM brauchen.

`--max-model-len` begrenzt den KV-Cache. Bei 12 GB VRAM macht es Sinn, diesen Wert explizit zu setzen:

```
vllm serve Qwen/Qwen3-4B-Instruct \
  --host 0.0.0.0 \
  --port 8000 \
  --max-model-len 8192 \
  --gpu-memory-utilization 0.85
```

Quantisierte Modelle verwenden

Wer auf VRAM-Budget achten muss, nutzt AWQ- oder GPTQ-quantisierte Varianten — die meisten großen Modellreihen haben diese direkt auf Hugging Face:

```
vllm serve Qwen/Qwen3-7B-Instruct-AWQ \
  --quantization awq \
  --host 0.0.0.0 \
  --port 8000
```

Als systemd-Service betreiben

Für dauerhaften Betrieb auf einem Linux-Server:

```
[Unit]
Description=vLLM Inference Server
After=network.target

[Service]
Type=simple
User=DEIN_USER
WorkingDirectory=/home/DEIN_USER
ExecStart=/home/DEIN_USER/vllm-env/bin/vllm serve Qwen/Qwen3-4B-
Instruct \
  --host 0.0.0.0 \
  --port 8000 \
  --max-model-len 8192
Restart=on-failure
RestartSec=10

[Install]
WantedBy=multi-user.target
```

Speichern als `/etc/systemd/system/vllm.service`, dann:

```
sudo systemctl daemon-reload
sudo systemctl enable vllm
sudo systemctl start vllm
sudo systemctl status vllm
```

Fehlerbehebung

CUDA out of memory beim Start `--gpu-memory-utilization` reduzieren oder `--max-model-len` kleiner setzen. vLLM reserviert beim Start den gesamten kalkulierten KV-Cache — das ist kein Bug, sondern das Design.

Modell lädt aber Requests schlagen fehl Prüfen ob der `--max-model-len` mit der Kontextlänge des Modells kompatibel ist. Manche Modelle haben eine maximale Länge die vLLM nicht automatisch erkennt.

Langsame erste Anfrage Normal — vLLM kompiliert beim ersten Request CUDA-Kernel. Ab der zweiten Anfrage läuft es mit voller Geschwindigkeit.

---

vLLM vs. llama.cpp — wann was?

	vLLM	llama.cpp
Modellformat	Safetensors (HF)	GGUF
Stärke	Hoher Durchsatz, Parallelität	Feingranulare Kontrolle, Offline
VRAM-Overhead	Höher	Niedriger
Konfiguration	Einfacher für HF-Modelle	Flexibler für GGUF-Ökosystem
Produktionstauglich	Ja	Ja (llama-server)

Für meinen Arbeitscluster bleibt llama.cpp die erste Wahl — das GGUF-Ökosystem, die LiteLLM-Integration und der niedrigere VRAM-Overhead passen besser. vLLM kommt ins Spiel wenn ein Modell nur als Safetensors verfügbar ist oder wenn ich unter echter Parallellast testen will.

Revision #2

Created 2026-04-10 15:49:56 UTC by Carsten

Updated 2026-07-05 21:28:33 UTC by Carsten