Drei Rechner, ein Modell — llama.cpp RPC Cluster im Heimnetz

Ich habe drei Rechner zu Hause, die sonst meist im Leerlauf laufen. Die Idee: alle drei per llama.cpp RPC zu einem gemeinsamen VRAM-Pool zusammenschalten und damit Modelle betreiben, die auf einer einzelnen Karte nicht mehr passen. Dieses Dokument beschreibt, wie ich das umgesetzt habe — inklusive der Stellen, an denen es hakelig wurde.

Hardware-Übersicht

Master Node — Windows 11

- AMD Ryzen 7800 X3D, 32 GB RAM
- MSI GeForce RTX 5070 Ventus X2 OC, 12 GB VRAM

Worker Node 1 & 2 — Linux (Debian)

- Intel Core i5, 11. Generation, 16 GB RAM
- NVIDIA RTX 2070, 8 GB VRAM

Gepoolter VRAM: 12 + 8 + 8 = 28 GB

Was 28 GB VRAM wirklich bedeuten

Bevor man anfängt, lohnt sich ein ehrlicher Blick auf die Zahlen. Ein 70B-Modell in Q4_K_M braucht allein für die Gewichte rund 43 GB. Das übersteigt den Pool deutlich.

Komponente	Bedarf (70B Q4_K_M)	Gepoolter VRAM	Differenz
Modellgewichte	~43 GB	28 GB	?15 GB
KV-Cache (8K Kontext)	~5 GB	0 GB	?5 GB

Komponente	Bedarf (70B Q4_K_M)	Gepoolter VRAM	Differenz
KV-Cache (32K Kontext)	~25 GB	0 GB	?25 GB

Die fehlenden 15 GB Gewichte landen im System-RAM des Master-Nodes. Der KV-Cache ebenfalls. Das hat Konsequenzen: Die Inferenzgeschwindigkeit liegt realistisch bei **8–12 Tokens pro Sekunde** — nicht mehr. Wer schnelle Antworten erwartet, wird enttäuscht sein. Für experimentelle Zwecke und zum Verstehen der Architektur ist es trotzdem interessant.

Wer in diesem Setup produktiv mit großen Kontexten arbeiten will, kommt an RAG nicht vorbei — kleines Kontextfenster (z.B. 8K) halten, relevante Passagen per Retrieval einbinden, VRAM für die Gewichte freihalten.

OS-Wahl für die Worker Nodes

Ich habe Debian auf beiden Intel-Maschinen installiert. Gründe:

- Sehr stabile NVIDIA-Treiberunterstützung
- Gut dokumentierte CUDA-Einrichtung
- Kein Overhead durch Desktop-Umgebung nötig

Ubuntu funktioniert genauso — der Installationsweg ist nahezu identisch.

Linux Worker Nodes einrichten

Pakete und Build-Toolchain

```
sudo apt install build-essential g++ cmake git curl libcurl4-openssl-dev pciutils
```

NVIDIA-Treiber installieren

APT-Quellen um non-free erweitern:

```
sudo sed -i 's/main/main non-free contrib/g' /etc/apt/sources.list
sudo apt update
```

Kernel-Header, DKMS und nvidia-detect installieren:

```
sudo apt install linux-headers-$(uname -r) build-essential dkms nvidia-detect
```

Prüfen ob die Karte erkannt wird:

```
nvidia-detect
```

Wenn ja, Treiber installieren:

```
sudo apt install nvidia-driver nvidia-kernel-dkms
sudo reboot
```

Nach dem Neustart verifizieren:

```
nvidia-smi
```

CUDA Toolkit

```
sudo apt install nvidia-cuda-toolkit
```

Compute Capability der GPU ermitteln — das braucht man beim Build:

```
nvidia-smi --query-gpu=compute_cap --format=csv
```

---

llama.cpp auf den Linux Nodes bauen

Ich kompiliere llama.cpp selbst — einerseits für maximale Kompatibilität, andererseits weil ich wissen will, was da läuft. Auf dem i5 mit acht parallelen Prozessen dauert der Build ca. 6 Minuten.

Schritt 1: Normaler CUDA-Build

```
git clone https://github.com/ggml-org/llama.cpp
cd llama.cpp
cmake -B build -DGGML_CUDA=ON
cmake --build build --config Release -j 8
```

Den `-j 8`-Parameter an die eigene Kernzahl anpassen.

Schritt 2: RPC-fähigen Build erstellen

Jetzt nochmal kompilieren, diesmal mit RPC-Support. Das ist ein separater Build-Ordner:

```
mkdir build-rpc-cuda
cd build-rpc-cuda
cmake .. -DGGML_CUDA=ON -DGGML_RPC=ON
cmake --build . --config Release
```

Das erzeugt das `rpc-server`-Binary, das auf den Worker Nodes laufen muss.

---

llama.cpp auf Windows installieren

Ich habe winget verwendet — unkompliziert und ausreichend für den Master-Node:

```
winget install llama.cpp
```

Wichtig: Master und Worker müssen denselben llama.cpp-Stand (Git-Commit) verwenden. Das RPC-Protokoll ist noch in aktiver Entwicklung; Versionsunterschiede führen zu Abstürzen beim Tensor-Laden. Bei winget immer prüfen, welcher Stand installiert wird, und die Linux-Worker ggf. auf denselben Commit zurücksetzen.

Netzwerk

Kein WLAN. Verteilte Inferenz reagiert sehr empfindlich auf Latenz und Jitter — WLAN macht das Ergebnis unbrauchbar. Gigabit-Ethernet per Kabel ist das absolute Minimum.

Das RPC-Protokoll überträgt Modell-Tensoren und Hidden States **unverschlüsselt**. Die Ports (Standard: 50052, ich nutze 21000) dürfen ausschließlich im lokalen Netz erreichbar sein. Niemals nach außen öffnen.

Firewall auf den Linux-Nodes: den gewählten Port für die IP des Master-Nodes freigeben, alles andere sperren.

Worker Nodes starten

Auf beiden Linux-Maschinen ins Build-Verzeichnis wechseln und den RPC-Server starten:

```
./rpc-server --host 0.0.0.0 --port 21000
```

`--host 0.0.0.0` lässt den Server auf allen Interfaces lauschen. Das ist nur dann vertretbar, wenn die Firewall den Zugriff auf die IP des Masters beschränkt.

Master Node: Inferenz starten

Der Master orchestriert das Sharding — die Remote-Worker erscheinen als zusätzliche CUDA-Devices.

Meine Worker laufen auf `192.168.0.91` und `192.168.0.92`:

```
./llama-cli.exe -m llama-3-70b-q4_k_m.gguf \
  --rpc 192.168.0.91:21000,192.168.0.92:21000 \
  --n-gpu-layers 100 \
  --ctx-size 8192
```

- `--rpc`: Adressen der Worker-Nodes, kommagetrennt
- `--n-gpu-layers 100`: So viele Layer wie möglich in den gepoolten VRAM; der Rest landet im System-RAM des Masters
- `--ctx-size 8192`: Bei 28 GB VRAM vernünftiger Ausgangswert; größere Kontexte kosten schnell alles was noch übrig ist

---

Performance-Flags

Diese Flags bringen messbare Verbesserungen:

- `--flash-attn` — Reduziert Speicherbedarf und Rechenaufwand des Attention-Mechanismus. Immer einschalten.
- `--mlock` — Verhindert, dass das OS Modellgewichte auf die Auslagerungsdatei schreibt. Ohne dieses Flag gibt es Stottern bei längerem Betrieb.

- `--no-mmap` — Auf den Worker-Nodes empfehlenswert wenn schnelle NVMe vorhanden ist: erzwingt vollständiges Laden des Shards in RAM beim Start.
- `--threads` — Für die CPU-Fallback-Verarbeitung auf dem Master: 4–8 Threads sind optimal, unabhängig von der Gesamtkernzahl. Mehr Threads erzeugen Kontextwechsel-Overhead der die Performance verschlechtert.

Fehlerbehebung

"RPC Failed to Connect" oder Worker meldet "0 MiB free"

Verbindung klappt, aber der Worker meldet keinen freien VRAM:

1. CUDA-Backend auf dem Worker ist nicht korrekt initialisiert — `nvidia-smi` prüfen
2. Falsches `CUDA_DOCKER_ARCH` beim Build gesetzt
3. `nvidia-cuda-toolkit` unvollständig installiert

Absturz beim Tensor-Laden

Fast immer ein Versions-Mismatch. llama.cpp-Version auf Master und Worker angleichen — identischer Git-Commit, nicht nur dieselbe Release-Nummer.

Screenshots

Hier ein paar Aufnahmen aus dem laufenden Betrieb — NVTOP und NVITOP zeigen schön, wie die VRAM-Last auf die Worker verteilt wird.

(Screenshots aus dem Wiki-Upload — siehe Originalartikel)

Fazit

Das Setup läuft. 70B-Modelle sind auf 28 GB VRAM machbar, aber man muss die Erwartungen kalibrieren: 8–12 TPS, kein großes Kontextfenster, der Master-RAM ist der eigentliche Flaschenhals. Für das Experimentieren mit Modellen, die sonst gar nicht laufen würden, ist es trotzdem ein sinnvolles Setup.

Wer ernsthaft mit großen Kontexten und diesem VRAM-Budget arbeiten will, sollte RAG in Betracht ziehen — kleines Fenster halten, Retrieval macht den Rest.

Revision #10

Created 2026-04-08 12:36:07 UTC by Carsten

Updated 2026-07-05 20:36:50 UTC by Carsten