LLM Performance Hub

LLM-Performance ist ein Systemproblem, kein GPU-Einkaufsproblem. Ob man ein einzelnes Modell auf einer lokalen Workstation betreibt oder einen Multi-Modell-Kubernetes-Cluster verwaltet — Inferenzgeschwindigkeit, Latenz und Effizienz werden durch das Zusammenspiel des gesamten Stacks bestimmt:

- Modellarchitektur und Quantisierungsstufe
- VRAM-Kapazität und Speicherbandbreite
- Kontextlänge und KV-Cache-Druck
- Runtime-Scheduling und Request-Batching
- CPU- und Treiber-Overhead
- Systemtopologie — PCIe-Bandbreite, NUMA-Grenzen, Netzwerkfabric

Diese Seite ist meine persönliche Referenz dafür, wie sich LLMs unter realen Workloads verhalten und wie man sie gezielt optimiert. Der Fokus liegt auf selbst gehosteten und On-Premise-Deployments — nicht auf verwalteter Cloud-Inferenz.

Was LLM-Performance wirklich bedeutet

Performance ist keine einzelne Zahl. Zwei Systeme können identischen Durchsatz haben und trotzdem ein völlig unterschiedliches Nutzererlebnis liefern, je nachdem wie die Latenz verteilt ist.

Throughput vs. Latenz

- **Throughput:** Aggregierte Tokens pro Sekunde über alle gleichzeitigen Sessions. Relevant für Deployments mit mehreren Nutzern.
- **Latenz:** Zeit bis zum ersten Token (TTFT) plus Dekodierzeit pro Token. Das ist das, was der Nutzer tatsächlich spürt.

Ein System, das rein auf Durchsatz optimiert ist, kann sich im interaktiven Betrieb träge anfühlen. Die richtige Balance hängt vom Anwendungsfall ab.

Wo der Flaschenhals üblicherweise sitzt

Engpässe treten in einer recht vorhersehbaren Reihenfolge auf. Den richtigen Layer zu identifizieren ist wertvoller als blind Hardware aufzurüsten:

1. **VRAM-Kapazität** — Passt das Modell überhaupt? Wenn nicht, hat man schon verloren.
2. **Speicherbandbreite** — Wie schnell können Gewichte pro Token in die Recheneinheiten gelesen werden?
3. **Runtime-Scheduling** — Wie effizient queued und batched der Serving-Layer Requests?
4. **KV-Cache-Druck** — Lange Kontexte füllen den VRAM schnell; Cache-Eviction ist teuer.
5. **CPU/Treiber-Overhead** — Oft übersehen, immer vorhanden sobald Parallelität ins Spiel kommt.

Inference-Runtimes und lokales Tooling

Meine Setups unterscheiden sich zwischen Arbeit und Zuhause — Benchmarks und Beobachtungen sind daher immer einem konkreten Kontext zugeordnet.

Arbeit — Kubernetes-Cluster (Quadro RTX 6000 Ada, 24 GB VRAM)

Im Arbeitsumfeld laufen llama.cpp-Instanzen in Kubernetes-Pods, ein Modell pro GPU, mit LiteLLM als einheitlichem API-Proxy und Open WebUI für den Teamzugang. Produktiv im Einsatz: Gemma 4 26B und Qwen3-Coder 30B.

- llama.cpp RPC Multi-GPU Setup Guide (Windows 11 & Linux Distributed Cluster)
- Agentic LLM Inference Tuning Reference für Qwen 3.6 und Gemma 4
- *Context-Scheduling und Slot-Management unter Parallellast* — in Vorbereitung

Zuhause — Eigene Workstation (RTX 5070, 12 GB VRAM)

Zu Hause ist der Toolstack experimenteller: llama.cpp für die rohe Inferenz, Jan.ai und LM Studio als lokale Frontends, OpenCode für KI-gestützte Entwicklungsworkflows. Die 12-GB-Grenze ist real — Quantisierungsentscheidungen wiegen hier deutlich schwerer als auf der Arbeit.

- OpenCode LLM Benchmarks: IndexNow & Migration Mapping
- The AI Coding Loop: Verification-Driven Development with AI Agents

- *Jan.ai vs. LM Studio: Frontend-Verhalten und Parameter-Zugang im Vergleich* — in Vorbereitung
- *12 GB VRAM: Modellauswahl — was läuft, was nicht* — in Vorbereitung

---

Hardware-Grenzen, die wirklich zählen

Rohe GPU-Rechenleistung ist selten der limitierende Faktor. Systemarchitektur — PCIe-Bandbreite, VRAM-Kapazität, Speicherbandbreite — bestimmt, was überhaupt möglich ist, bevor man auch nur eine Modelldatei anfasst. Die beiden Setups, mit denen ich täglich arbeite, repräsentieren zwei sehr unterschiedliche Constraint-Profile: 24 GB auf der Arbeit (Server-GPU, ECC, gute Bandbreite) und 12 GB zu Hause (Consumer-GPU, schneller pro Core, engeres VRAM-Budget).

- *PCIe-Bandbreite und Multi-GPU-Durchsatz in der Praxis* — in Vorbereitung
- *24 GB VRAM: praktische Modellfit- und KV-Cache-Grenzen für große Modelle* — in Vorbereitung
- *12 GB VRAM: Quantisierungsentscheidungen und ihre Konsequenzen* — in Vorbereitung

---

Benchmarks & Modellvergleiche

Benchmarks sind nur dann nützlich, wenn sie eine konkrete Frage beantworten. Leaderboard-Scores ohne Workload-Kontext sind Rauschen.

Arbeit — Quadro RTX 6000 Ada (24 GB VRAM)

Der Arbeitscluster betreibt Gemma 4 26B und Qwen3-Coder 30B produktiv. Bei 24 GB geht es nicht nur darum, ob das Modell passt — sondern wie viel Kontext-Headroom nach dem Laden der Gewichte noch übrig bleibt und wie sich das Modell unter realer gleichzeitiger Teamnutzung schlägt.

- Agentic LLM Inference Tuning Reference für Qwen 3.6 und Gemma 4
- *Gemma 4 26B und Qwen3-Coder 30B unter realistischer Parallellast* — in Vorbereitung
- *Quadro RTX 6000 Ada: Durchsatz unter Multi-Modell-Kubernetes-Last* — in Vorbereitung

Zuhause — RTX 5070 (12 GB VRAM)

12 GB sind eine echte Einschränkung. Die meisten interessanten Modelle brauchen aggressive Quantisierung, und das KV-Cache-Budget schrumpft bei längeren Kontexten schnell. Der Vorteil: Die RTX 5070 hat für ihre Klasse eine hervorragende Speicherbandbreite, und die interaktive Single-User-Latenz ist sehr gut.

- Qwen 3.6 27B und 35B MTP vs. Standard auf 16-GB-GPU — Speculative Decoding: Geschwindigkeit vs. Kontextkosten
- *RTX 5070 12 GB: Welche Modelle laufen wirklich gut?* — in Vorbereitung
- *Quantisierungsstufen: Auswirkung auf Qualität und Geschwindigkeit bei 12 GB VRAM* — in Vorbereitung

Modellgeschwindigkeit & Qualität

- OpenCode LLM Benchmarks: IndexNow & Migration Mapping
- *Qwen3-Coder 30B vs. andere Coding-Modelle auf realen Aufgaben* — in Vorbereitung
- *Modellvergleich: Gemma 4 vs. Qwen3 für agentische Workflows* — in Vorbereitung

Spezialisierte Tests

- **Coding:** The AI Coding Loop: Verification-Driven Development with AI Agents
- **Strukturierter Output:** *Zuverlässigkeit von JSON-/Structured-Output über Modellfamilien hinweg* — in Vorbereitung
- **Zusammenfassung:** *Zusammenfassungsqualität bei variierenden Kontextlängen* — in Vorbereitung

---

Inferenz-Optimierung

Techniken, die Latenz reduzieren oder den Durchsatz verbessern, ohne die Ausgabequalität zu beeinträchtigen.

- Qwen 3.6 MTP vs. Standard: Speculative Decoding in der Praxis — Bringt MTP wirklich etwas?
- Agentic Inference Parameter Tuning für Qwen und Gemma — Praxistaugliche Einstellungen für agentische Produktiv-Workflows

- *Speculative Decoding in llama.cpp: reale Gewinne auf Ada- vs. Consumer-GPUs* — in Vorbereitung

---

Optimierungs-Playbook

Tuning funktioniert am besten als inkrementeller Prozess. Jeder Schritt sollte validiert werden, bevor man zum nächsten übergeht.

Schritt 1: Modell zum Laufen bringen

- Die richtige Quantisierungsstufe wählen — Q4_K_M, Q6_K, Q8_0 stehen jeweils für einen anderen Qualitäts-/Geschwindigkeits-Kompromiss.
- Das Kontextfenster auf das tatsächlich Notwendige reduzieren.
- CPU-Offloading ist das letzte Mittel; es zerstört den Durchsatz.

Schritt 2: Latenz stabilisieren

- Prefill-Kosten reduzieren — kürzere System-Prompts wirken sich stärker aus als erwartet.
- Structured-Output-Schemas früh validieren, um teure Wiederholungsgenerierungen zu vermeiden.
- TTFT separat von der Gesamtgenerierungszeit messen; sie haben unterschiedliche Ursachen.

Schritt 3: Durchsatz steigern

- Den parallelen Slot-Count in llama.cpp auf das eigene Request-Muster abstimmen.
- LiteLLMs Load-Balancing nutzen, um Requests auf mehrere Modellinstanzen zu verteilen.
- vLLM oder SGLang in Betracht ziehen, wenn llama.cpp-Scheduling zum Engpass wird.

Für Hosting-Strategie und Infrastrukturentscheidungen: LLM Hosting in 2026: Local, Self-Hosted and Cloud Infrastructure Compared.

Häufige Fragen

Warum ist mein LLM langsam, obwohl ich eine leistungsstarke GPU habe? Meistens liegt es an der Speicherbandbreite. LLM-Dekodierung ist speicherbegrenzt — die GPU wartet öfter auf ankommende Gewichte als dass sie wirklich rechnet. Quantisierung hilft, weil sie die Datenmenge reduziert, die pro Token bewegt werden muss.

Was ist wichtiger — VRAM-Größe oder GPU-Modell? VRAM-Kapazität kommt zuerst. Wenn Modell oder KV-Cache nicht in den VRAM passen, bricht die Performance in dem Moment zusammen, in dem der Runtime anfängt auf System-RAM auszulagern. Jenseits des reinen Fits ist dann die Speicherbandbreite entscheidend.

Warum verschlechtert sich die Performance, wenn mehr Nutzer gleichzeitig arbeiten? Request-Queuing, KV-Cache-Contention und Scheduler-Sättigung verstärken sich gegenseitig. Jeder zusätzliche parallele Request verbraucht VRAM und Speicherbandbreite, auf die alle anderen Sessions ebenfalls angewiesen sind. Die Degradationskurve ist selten linear.

Fazit

LLM-Performance-Optimierung folgt vorhersehbaren Prinzipien, sobald man die Engpass-Hierarchie verstanden hat. Die Hardware spielt eine weit kleinere Rolle als das korrekte Identifizieren des tatsächlich limitierenden Layers.

Erst messen ? Engpass identifizieren ? diesen Layer beheben ? wiederholen.

Revision #6

Created 2026-07-03 08:57:22 UTC by Carsten

Updated 2026-07-08 08:21:14 UTC by Carsten