

# llama.cpp – Parameter-Referenz

Für die Ausführung von Qwen3-30B-A3B auf einer NVIDIA RTX 2080 Ti (11 GB VRAM).

Hinweis zum Modell: In Deiner Anfrage stand Qwen3.6-35B-A3B. Ein solches Modell existiert (Stand Mai 2026) nicht im offiziellen Qwen-Katalog. Gemeint ist mit hoher Wahrscheinlichkeit Qwen3-30B-A3B – ein Mixture-of-Experts-Modell mit 30 Mrd. Parametern total und ca. 3 Mrd. aktiven Parametern pro Token. Diese Referenz ist auf genau dieses Modell zugeschnitten.

## Ausgangslage

Die RTX 2080 Ti hat 11 GB GDDR6 VRAM und Turing-Architektur (Compute Capability 7.5, kein FP8, aber Tensor Cores für FP16/INT8). Qwen3-30B-A3B in Q4\_K\_M-Quantisierung belegt ca. 17–18 GB – passt also nicht vollständig in den VRAM. Die wichtigste Stellschraube ist daher das selektive Auslagern der MoE-Experten auf die CPU (`--n-cpu-moe` oder `--override-tensor`), während Attention-Layer auf der GPU bleiben. Mit aktiviertem Flash-Attention und Q8/Q4-quantisiertem KV-Cache sind 16k–32k Kontext realistisch.

## 1. Performance-Optimierung

Diese Parameter steuern die Verteilung zwischen GPU und CPU sowie den Durchsatz. Sie haben den größten Einfluss auf Tokens/Sekunde bei knappem VRAM.

Parameter	Default	Erklärung	Tipp für 11 GB VRAM
<code>--n-gpu-layers (-ngl)</code>	auto / -1	Anzahl der Transformer-Layer, die auf die GPU geladen werden. Restliche Layer laufen auf der CPU.	Bei MoE-Modellen wie Qwen3-30B-A3B reicht der VRAM nicht für alle Layer. Setze <code>-ngl 99</code> und kombiniere mit <code>--n-cpu-moe</code> oder <code>--override-tensor</code> , um Expert-Tensoren ( <code>ffn*_exps</code> ) auf die CPU auszulagern. So bleiben Attention-Layer auf der GPU – das bringt den größten Speed-Gewinn.
<code>--n-cpu-moe (-ncmoe) N</code>	0	Speziell für MoE-Modelle: hält die ersten N MoE-Schichten (Experten-FFN) auf der CPU im RAM.	Starte mit <code>-ncmoe 24</code> bis <code>-ncmoe 32</code> bei 11 GB VRAM und Qwen3-30B-A3B (Q4_K_M). Vorher mit <code>llama-bench</code> verschiedene Werte testen – jeder zusätzliche CPU-Layer spart ~250–350 MB VRAM.
<code>--override-tensor (-ot)</code>	–	Regex-Pattern, das einzelne Tensoren explizit einem Backend (z. B. CPU) zuweist.	Alternative zu <code>-ncmoe</code> mit feinerer Kontrolle, z. B. <code>'blk\.([0-9]*[02468])\.ffn_.*_exps\.=CPU'</code> für gerade Expert-Layer. Nützlich, wenn Du das letzte freie GB VRAM ausreizen willst.
<code>--threads (-t) N</code>	Anzahl CPU-Kerne	Threads für Token-Generierung (sequenzieller Anteil).	Bei MoE-Auslagerung sind CPU-Threads kritisch. Bei einem 8-Kern-Ryzen sind 6–8 Threads optimal; mehr bringt nichts, weil Memory-Bandwidth zum Engpass wird. Probiere <code>-t 6</code> bis <code>-t 8</code> .

Parameter	Default	Erklärung	Tipp für 11 GB VRAM
<b>--threads-batch (-tb) N</b>	= --threads	Threads für Prompt-Processing (parallelisierbar, Matrix-Multiplikationen).	Kann höher sein als --threads – nimm hier alle physischen Kerne (kein SMT/Hyperthreading), z. B. -tb 12 auf einem 12-Kerner für schnelles Prompt-Eval.
<b>--batch-size (-b) N</b>	2048	Logische Batch-Größe für Prompt-Processing in Tokens.	Größere Batches = schnelleres Prefill, aber mehr VRAM. Bei 11 GB VRAM ist -b 2048 i. d. R. okay, da MoE-Experten auf CPU liegen. Bei langem Kontext (≥32k) ggf. auf -b 1024 reduzieren.
<b>--ubatch-size (-ub) N</b>	512	Physikalische Micro-Batch-Größe – wirklich gleichzeitig verarbeitete Tokens auf der GPU.	Hebt den Prompt-Eval-Durchsatz spürbar. Mit Flash-Attention sind -ub 1024 oder -ub 2048 auf 11 GB realistisch. Mit llama-bench -ub 512, 1024, 2048 den Sweet-Spot finden.
<b>--flash-attn (-fa) [on off auto]</b>	auto	Aktiviert Flash-Attention – speichersparender Attention-Kernel.	Immer aktivieren bei knappem VRAM: -fa on. Reduziert KV-Cache-Speicherbedarf deutlich und beschleunigt lange Kontexte. Pflicht bei 11 GB.
<b>--mlock</b>	aus	Sperrt Modellgewichte gegen Swapping in den RAM.	Nur sinnvoll, wenn das System unter Memory-Druck steht und CPU-Layer ins Swap geraten könnten. Bei dediziertem Inferenz-Server eher überflüssig.
<b>--no-mmap</b>	mmap aktiv	Deaktiviert Memory-Mapping; Modell wird vollständig in den RAM geladen.	Empfohlen bei MoE-Offload auf CPU – stellt sicher, dass Expert-Tensoren tatsächlich im RAM liegen und nicht von der SSD nachgeladen werden. Spürbarer Speed-Up.
<b>--numa [distribute isolated numactl]</b>	aus	NUMA-Optimierung für Mehrsocket- oder Mehr-CCD-Systeme.	Auf einem Single-Socket-Desktop irrelevant. Auf Ryzen mit mehreren CCDs kann --numa distribute testweise probiert werden.
<b>--cont-batching</b>	an (Server)	Continuous Batching für llama-server – verbindet wartende Anfragen.	Bei Single-User-Inferenz egal; bei Multi-User-Server unbedingt aktiviert lassen.

## 2. Kontext-Management

Diese Parameter bestimmen, wie viel Text das Modell gleichzeitig „im Kopf“ behält und wie der KV-Cache gespeichert wird. KV-Cache-Quantisierung ist neben MoE-Offload der zweite große VRAM-Hebel bei langem Kontext.

Parameter	Default	Erklärung	Tipp für 11 GB VRAM
<code>--ctx-size (-c) N</code>	0 (Modellvorgabe)	Maximale Kontextlänge in Tokens. 0 = vom Modell vorgegeben.	Bei 11 GB VRAM und Qwen3-30B-A3B sind 16k–32k Kontext realistisch, wenn KV-Cache quantisiert ist und Experten auf der CPU liegen. Beginne mit <code>-c 16384</code> und erhöhe, falls VRAM frei bleibt.
<code>--cache-type-k (-ctk)</code>	f16	Quantisierungstyp für den Key-Cache. Optionen: f32, f16, bf16, q8_0, q4_0, q4_1, iq4_nl, q5_0, q5_1.	Wichtigster VRAM-Hebel. <code>-ctk q8_0</code> halbiert den K-Cache mit kaum messbarem Qualitätsverlust. <code>-ctk q4_0</code> spart noch mehr – akzeptabel für Chat, aber Qualität langsam spürbar. Erfordert <code>-fa on</code> .
<code>--cache-type-v (-ctv)</code>	f16	Quantisierungstyp für den Value-Cache.	Analog zu <code>-ctk</code> . Übliche Kombination bei 11 GB: <code>-ctk q8_0 -ctv q8_0</code> . Aggressiver: <code>q4_0/q4_0</code> bei sehr langem Kontext. V-Cache ist etwas qualitätssensitiver als K-Cache.
<code>--keep N</code>	0	Anzahl initialer Tokens (System-Prompt), die bei Kontextverschiebung erhalten bleiben.	Bei Chat-Anwendungen mit fixem System-Prompt <code>--keep -1</code> (alle initialen Tokens behalten), damit die Persona nicht abgeschnitten wird.
<code>--n-predict (-n) N</code>	-1 (unbegrenzt)	Maximale Anzahl generierter Tokens pro Anfrage.	Für Reasoning-Modelle wie Qwen3 ruhig hochsetzen ( <code>-n 4096</code> bis <code>-n 32768</code> ), da der Thinking-Token-Block sehr lang werden kann.
<code>--rope-scaling [none linear yarn]</code>	linear	RoPE-Skalierung zur Erweiterung des Kontexts über die Trainingslänge hinaus.	Nur ändern, wenn Du den Kontext bewusst über die Default-Länge ausdehnst (z. B. YaRN). Sonst auf Modellvorgabe lassen.
<code>--rope-freq-base N</code>	Modellvorgabe	RoPE-Basisfrequenz 0. Beeinflusst Positionscodierung.	Nur experimentell anpassen; Qwen3 hat optimierte Defaults. Falsche Werte zerstören die Output-Qualität.
<code>--cache-reuse N</code>	0	Minimale Chunk-Größe für KV-Cache-Wiederverwendung über KV-Shifting.	Bei wiederholten Prompts mit gemeinsamem Prefix (RAG, Chat-History) deutlich schneller: <code>--cache-reuse 256</code> oder <code>2048</code> . Spart Prefill-Kosten.
<code>--swa-full</code>	aus	Erzwingt vollen KV-Cache bei Sliding-Window-Attention-Modellen.	Nur relevant für SWA-Modelle (z. B. gpt-oss); für Qwen3-30B-A3B nicht nötig.

### 3. Sampling-Konfiguration

Diese Parameter steuern, wie das nächste Token aus der Wahrscheinlichkeitsverteilung gewählt wird. Sie ändern nichts am Speicherbedarf, sind aber qualitätsentscheidend. Die Qwen3-Empfehlungen sind in den Tipps farblich hervorgehoben.

Parameter	Default	Erklärung	Tipp für 11 GB VRAM
<code>--temp N</code>	0.80	Temperatur – skaliert die Logits. Höher = kreativer, niedriger = deterministischer.	Qwen3-Empfehlung (Thinking-Mode): <code>--temp 0.6</code> . Für Code und Faktenfragen 0.2–0.4, für Brainstorming 0.7–0.9.
<code>--top-k N</code>	40 (0 = aus)	Behält nur die K wahrscheinlichsten Tokens vor dem Sampling.	Qwen3-Empfehlung: <code>--top-k 20</code> . Niedrigere Werte machen Output fokussierter. 0 deaktiviert den Filter komplett (nicht empfohlen ohne Min-P).
<code>--top-p N</code>	0.95 (1.0 = aus)	Nucleus-Sampling: kleinste Token-Menge mit kumulierter Wahrscheinlichkeit $\geq P$ .	Qwen3-Empfehlung: <code>--top-p 0.95</code> . Bei sehr deterministischen Tasks runter auf 0.85.
<code>--min-p N</code>	0.05 (0.0 = aus)	Mindest-Wahrscheinlichkeit relativ zum wahrscheinlichsten Token.	Robusterer Filter als Top-P. Qwen3 empfiehlt <code>--min-p 0</code> in Kombination mit Top-K=20 und Top-P=0.95. Allgemein gilt 0.05 als guter Default.
<code>--typical N</code>	1.00 (1.0 = aus)	Locally-Typical-Sampling – filtert Tokens nach Informationsgehalt-Abweichung.	Selten genutzt. Werte um 0.9 können Wiederholungen reduzieren, aber meistens Top-K + Min-P bevorzugen.
<code>--repeat-penalty N</code>	1.00 (1.0 = aus)	Strafe für kürzlich gesehene Tokens.	Qwen3-Empfehlung: <code>--repeat-penalty 1.1</code> . Zu hohe Werte (>1.2) führen zu seltsamem Vokabular. Über <code>--repeat-last-n</code> regelst Du das Fenster.
<code>--repeat-last-n N</code>	64	Wie viele letzte Tokens für Repeat-Penalty berücksichtigt werden.	Standardwert 64 ist konservativ. Bei längeren Outputs auf 256 bis 512 erhöhen.
<code>--presence-penalty N</code>	0.00	Konstante Strafe pro Token, das schon mal vorkam (OpenAI-Stil).	Werte um 1.0–1.5 helfen MoE-Modellen, weniger zu loopen. Praxis-Tipp aus der Community für Qwen3-30B-A3B: <code>--presence-penalty 1.5</code> .
<code>--frequency-penalty N</code>	0.00	Häufigkeitsabhängige Strafe – wächst mit der Anzahl bisheriger Vorkommen.	Sparsam einsetzen (0.1–0.3). Übertriebene Werte verzerren den Stil.
<code>--mirostat [0 1 2]</code>	0 (aus)	Adaptive Sampling-Strategie, die die Perplexity konstant hält.	Alternative zu Top-K/Top-P/Min-P. <code>--mirostat 2 --mirostat-lr 0.1 --mirostat-ent 5.0</code> liefert oft sehr konsistenten Output.
<code>--dry-multiplier N</code>	0.0	DRY-Sampler (Don't Repeat Yourself) – moderne Wiederholungs-Strafe.	Optional Ersatz für Repeat-Penalty. Werte um 0.8 oft besser als klassische Penalty – probieren bei langen Generationen.
<code>--seed N</code>	-1 (zufällig)	Seed für den Random Number Generator.	Für reproduzierbare Outputs (Benchmarks, Debugging) fixen Wert setzen, z. B. <code>--seed 42</code> .

# Beispiel-Startbefehl für Qwen3-30B-A3B auf RTX 2080 Ti

Konservative Basis-Konfiguration mit Q4\_K\_M-Quantisierung, 16k Kontext, Q8-KV-Cache und MoE-Offload. Werte für -ncmoe und -ub vorher mit llama-bench tunen.

```
llama-server \
--model /models/Qwen3-30B-A3B-Q4_K_M.gguf \
--host 0.0.0.0 --port 8080 \
--n-gpu-layers 99 \
--n-cpu-moe 28 \
--flash-attn on \
--no-mmap \
--ctx-size 16384 \
--batch-size 2048 \
--ubatch-size 1024 \
--cache-type-k q8_0 \
--cache-type-v q8_0 \
--threads 8 \
--threads-batch 12 \
--threads-http 4 \
--cont-batching \
--cache-reuse 256 \
--n-predict 8192 \
--temp 0.6 \
--top-k 20 \
--top-p 0.95 \
--min-p 0 \
--repeat-penalty 1.1 \
--presence-penalty 1.5
```

## Empfohlene Tuning-Reihenfolge

- Schritt 1 – Sweet-Spot für MoE-Offload finden: llama-bench -m model.gguf -ngl 99 -ncmoe 20,24,28,32,36 -fa 1. Wähle den höchsten Wert, der noch ~1 GB VRAM-Headroom für den KV-Cache lässt.
- Schritt 2 – Micro-Batch optimieren: llama-bench -ngl 99 -ncmoe <X> -fa 1 -ub 512,1024,2048. Prompt-Eval skaliert oft linear bis 1024, danach diminishing returns.
- Schritt 3 – KV-Cache-Quantisierung wählen: Q8/Q8 für Standardbetrieb, Q4/Q4 nur wenn Du wirklich 32k+ Kontext brauchst. Immer mit -fa on.
- Schritt 4 – Threads anpassen: Da Experten auf der CPU laufen, sind Threads kritisch. Teste 6, 8, 10, 12 – mehr als physische Kerne (ohne SMT) bringt selten etwas.
- Schritt 5 – Sampling justieren: Erst Performance-Setup einfrieren, dann Sampling für Deinen Use-Case anpassen (Code, Chat, Reasoning).

## Grober VRAM-Bedarf bei Qwen3-30B-A3B (Q4\_K\_M)

Komponente	Schätzung	Bemerkung
Modellgewichte total	~17–18 GB	passt nicht ganz in 11 GB
Aktive GPU-Layer (Attention)	~5–6 GB	nach MoE-Offload auf CPU
KV-Cache @ 16k Ctx, FP16	~3.5 GB	Default ohne Quantisierung
KV-Cache @ 16k Ctx, Q8_0	~1.8 GB	empfohlen für 11 GB

Komponente	Schätzung	Bemerkung
KV-Cache @ 16k Ctx, Q4_0	~0.9 GB	aggressiv, leichter Qualitätsverlust
Compute-Buffer (ubatch=1024, fa on)	~0.8–1.2 GB	skaliert mit --ubatch-size
Headroom (Treiber, Display, OS)	~0.5–1 GB	Reserve einplanen

## Quellen

1. llama-server(1) Manpage – Debian unstable: [manpages.debian.org/unstable/llama.cpp-tools/llama-server.1.en.html](https://manpages.debian.org/unstable/llama.cpp-tools/llama-server.1.en.html)
2. llama.cpp GitHub Repository – ggml-org: [github.com/ggml-org/llama.cpp](https://github.com/ggml-org/llama.cpp)
3. Diskussion „Optimal parameters for parallel inference using llama-server #18308“: [github.com/ggml-org/llama.cpp/discussions/18308](https://github.com/ggml-org/llama.cpp/discussions/18308)
4. r/LocalLLaMA – „8GB VRAM – Qwen3-30B-A3B & gpt-oss-20b t/s mit llama.cpp“: [reddit.com/r/LocalLLaMA/comments/1nyxmci](https://reddit.com/r/LocalLLaMA/comments/1nyxmci)
5. r/LocalLLaMA – „Memory Tests using Llama.cpp KV cache quantization“: [reddit.com/r/LocalLLaMA/comments/1dalkm8](https://reddit.com/r/LocalLLaMA/comments/1dalkm8)
6. Hugging Face – Qwen3-30B-A3B-GGUF: [huggingface.co/Mungert/Qwen3-30B-A3B-GGUF](https://huggingface.co/Mungert/Qwen3-30B-A3B-GGUF)