

# llama.cpp – Parameter-Referenz

Für die Ausführung von Qwen3-30B-A3B auf 2× NVIDIA RTX 3090 (48 GB VRAM gesamt).

Hinweis zum Modell: In Deiner Anfrage stand Qwen3.6-35B-A3B. Ein Modell mit diesem Namen 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 ~3 Mrd. aktiven Parametern pro Token. Diese Referenz ist auf dieses Modell zugeschnitten.

## Ausgangslage

Zwei RTX 3090 liefern 48 GB GDDR6X-VRAM (24 GB pro Karte) auf Ampere-Architektur (Compute Capability 8.6, FP16/BF16 Tensor Cores). Im Gegensatz zu kleineren Setups ist hier genug VRAM für das vollständige Modell – sogar in Q8\_0 (~32 GB) bleibt noch Platz für einen großzügigen KV-Cache. Die Optimierung verschiebt sich von „VRAM sparen“ zu „Multi-GPU richtig konfigurieren“. Drei Stellschrauben sind dafür neu wichtig: `--split-mode`, `--tensor-split` und `--main-gpu`. Für PCIe-3090s ohne NVLink-Brücke ist Pipeline-Parallelismus (`-sm layer`, Default) meist die robusteste Wahl. Bei NVLink-Brücke kann `-sm tensor` (experimentell) die Token-Generierung um 30–50 % beschleunigen.

Empfohlene Quantisierung für 2× 3090: Q6\_K (~25 GB, sehr gute Qualität) oder Q8\_0 (~32 GB, nahezu lossless) – beides passt mit großem KV-Cache. Q4\_K\_M (~18 GB) lohnt nur, wenn Du extrem langen Kontext (≥128k) oder viele parallele Slots brauchst.

## 1. Performance-Optimierung (inkl. Multi-GPU)

Diese Parameter steuern Lastverteilung, Durchsatz und – speziell auf Multi-GPU-Setups – das Verteilen des Modells zwischen den beiden Karten.

Parameter	Default	Erklärung	Tipp für 2× 24 GB VRAM
<code>--n-gpu-layers (-ngl)</code>	auto / -1	Anzahl der Transformer-Layer auf der GPU. Restliche Layer laufen auf der CPU.	Mit 48 GB total passt Qwen3-30B-A3B (selbst in Q8_0, ~32 GB) komplett auf die GPUs. Setze <code>-ngl 99</code> – das lädt alles. Auto-Fit ( <code>--fit on</code> ) erledigt das in neueren Versionen automatisch.
<code>--split-mode (-sm)</code> <code>[none layer tensor]</code>	layer	Wie das Modell über mehrere GPUs verteilt wird. <code>layer</code> = Pipeline-Parallelismus, <code>tensor</code> = Tensor-Parallelismus, <code>none</code> = Single-GPU.	Default <code>layer</code> ist die kompatible Wahl für PCIe-3090s ohne NVLink – schnelles Prefill, KV-Cache pro Layer auf der jeweiligen GPU. <code>tensor</code> ist experimentell aber oft 30–50 % schneller bei der Token-Generierung – Voraussetzung: KV-Cache in f16 (keine Quantisierung). Bei zwei 3090 mit NVLink-Brücke unbedingt testen.
<code>--tensor-split (-ts)</code>	modusabhängig	Komma-separierte Verhältnisse, wie das Modell auf die GPUs aufgeteilt wird, z. B. 1,1 oder 3,1.	Bei zwei gleichen 3090 ohne Display-Belegung: <code>-ts 1,1</code> . Wenn GPU 0 zusätzlich das Display antreibt: <code>-ts 9,11</code> oder 4,5, damit die zweite Karte mehr Modell-Layer bekommt. Werte werden normalisiert (3,1 = 75/25).
<code>--main-gpu (-mg) INDEX</code>	0	Index der „Haupt-GPU“ – hält KV-Cache bei <code>-sm row</code> bzw. ist die einzige GPU bei <code>-sm none</code> .	Bei <code>-sm layer</code> (Default) kaum Einfluss. Bei <code>-sm none</code> wählst Du hiermit aus, welche Karte allein arbeitet. <code>--list-devices</code> zeigt die Indizes (CUDA0, CUDA1).

Parameter	Default	Erklärung	Tipp für 2× 24 GB VRAM
<b>--device (-dev)</b>	alle sichtbar	Explizite Auswahl der Devices, z. B. -dev CUDA0,CUDA1.	Alternative zu CUDA_VISIBLE_DEVICES. Nützlich, um eine GPU bewusst auszulassen oder die Reihenfolge zu fixieren.
<b>--threads (-t) N</b>	Anzahl CPU-Kerne	Threads für Token-Generierung (sequenzieller Anteil).	Mit allen Layern auf GPU werden Threads weniger kritisch. -t 6 bis -t 8 reicht meistens; mehr bringt nur Overhead. CPU-RAM-Bandbreite ist hier nicht mehr der Flaschenhals.
<b>--threads-batch (-tb) N</b>	= --threads	Threads für Prompt-Processing (parallelisierbar).	Nimm alle physischen Kerne (kein SMT), z. B. -tb 12 auf einem 12-Kerner. Beschleunigt Prefill spürbar bei langen Prompts.
<b>--batch-size (-b) N</b>	2048	Logische Batch-Größe für Prompt-Processing in Tokens.	Mit 48 GB VRAM kannst Du großzügig sein: -b 4096 oder -b 8192 beschleunigen das Prefill langer Prompts deutlich. Speicherkosten sind moderat im Vergleich zu KV-Cache.
<b>--ubatch-size (-ub) N</b>	512	Physikalische Micro-Batch-Größe – tatsächlich gleichzeitig verarbeitete Tokens auf der GPU.	Bei 48 GB ist -ub 2048 oft der Sweet-Spot, manchmal sogar -ub 4096. Mit llama-bench -ub 512, 1024, 2048, 4096 empirisch ermitteln – Skalierung wird oberhalb 2048 meist flach.
<b>--flash-attn (-fa) [on off auto]</b>	auto	Flash-Attention – speichersparender Attention-Kernel.	Auf Ampere (3090, Compute 8.6) voll unterstützt. -fa on beschleunigt lange Kontexte und ist Voraussetzung für KV-Cache-Quantisierung.
<b>--parallel (-np) N</b>	1	Anzahl paralleler Slots für gleichzeitige Anfragen (llama-server).	Mit 48 GB kannst Du -np 4 bis -np 8 für Multi-User-Setups laufen lassen. Jeder Slot belegt eigenen KV-Cache-Anteil.
<b>--cont-batching</b>	an (Server)	Continuous Batching – kombiniert wartende Anfragen.	Bei -np > 1 immer aktiviert lassen. Steigert Durchsatz bei Multi-User-Workloads erheblich.
<b>--no-mmap</b>	mmap aktiv	Deaktiviert Memory-Mapping; Modell wird komplett in den RAM geladen.	Da das Modell ohnehin vollständig auf GPU liegt, hat das wenig Effekt. Bei knappem System-RAM kann --no-mmap sogar schaden.
<b>--mlock</b>	aus	Sperrt Modellgewichte gegen Swapping in den RAM.	Irrelevant, wenn alle Layer auf GPU – die Karte cached selbst. Nur bei hybridem CPU/GPU-Setup sinnvoll.
<b>--numa [distribute isolate numactl]</b>	aus	NUMA-Optimierung für Mehrsocket- oder Mehr-CCD-Systeme.	Auf Single-Socket-Workstations egal. Auf Threadripper/EPYC oder Multi-CCD-Ryzen testweise --numa distribute.

## 2. Kontext-Management

Diese Parameter bestimmen, wie viel Text das Modell „im Kopf“ behält und wie der KV-Cache gespeichert wird. Mit 48 GB VRAM kannst Du i. d. R. auf KV-Cache-Quantisierung verzichten und volle f16-Qualität fahren.

Parameter	Default	Erklärung	Tipp für 2× 24 GB VRAM
<b>--ctx-size (-c) N</b>	0 (Modellvorgabe)	Maximale Kontextlänge in Tokens. 0 = vom Modell vorgegeben.	Mit 48 GB sind 64k–128k Kontext realistisch, je nach Quantisierung. Bei Qwen3-30B-A3B Q4_K_M und Q8-KV: -c 131072 für die volle native 128k-Länge. Bei Q8_0-Gewichten eher -c 65536.
<b>--cache-type-k (-ctk)</b>	f16	Quantisierung des Key-Cache. Optionen: f32, f16, bf16, q8_0, q4_0, q4_1, iq4_nl, q5_0, q5_1.	Mit 48 GB kannst Du f16 fahren und volle Qualität behalten. -ctk q8_0 halbiert den K-Cache, falls Du sehr lange Kontexte (≥64k) brauchst. Wichtig: Bei -sm tensor aktuell nur f16 erlaubt.
<b>--cache-type-v (-ctv)</b>	f16	Quantisierung des Value-Cache.	Analog zu -ctk. Behalte f16, solange Du nicht aus VRAM-Gründen quantisieren musst. V-Cache reagiert qualitätssensitiver auf Quantisierung als K-Cache.
<b>--keep N</b>	0	Anzahl initialer Tokens (System-Prompt), die bei Kontextverschiebung erhalten bleiben.	Bei Chat-Systemen mit festem System-Prompt: -keep -1 (alle initialen Tokens behalten).
<b>--n-predict (-n) N</b>	-1 (unbegrenzt)	Maximale generierte Tokens pro Anfrage.	Qwen3 ist ein Reasoning-Modell mit langen Thinking-Blöcken. -n 16384 oder -n 32768 ist sinnvoll, damit Antworten nicht abgeschnitten werden.
<b>--rope-scaling [none linear yarn]</b>	linear	RoPE-Skalierung zur Erweiterung über die Trainingslänge hinaus.	Qwen3 hat nativ 128k Kontext – meist keine Skalierung nötig. YaRN nur, wenn Du wirklich darüber hinauswillst.
<b>--rope-freq-base N</b>	Modellvorgabe	RoPE-Basisfrequenz 0. Beeinflusst Positionscodierung.	Defaults nicht ändern. Qwen3 hat optimierte Werte; falsche Einstellungen zerstören die Qualität.
<b>--cache-reuse N</b>	0	Minimale Chunk-Größe für KV-Cache-Wiederverwendung via KV-Shifting.	Bei wiederholten Prompts mit gemeinsamem Prefix (RAG, Chat): --cache-reuse 256 oder 2048. Spart Prefill-Kosten enorm.
<b>--kv-unified / --no-kv-unified</b>	auto	Einheitlicher KV-Buffer über alle Slots (bei -np > 1).	Bei Multi-User-Server (-np 4+) lassen, spart VRAM. Bei Single-User keine Bedeutung.

### 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 hervorgehoben.

Parameter	Default	Erklärung	Tipp für 2× 24 GB VRAM
<code>--temp N</code>	0.80	Temperatur – skaliert die Logits.	Qwen3 Thinking-Mode: <code>--temp 0.6</code> . Code/Fakten: 0.2–0.4. Brainstorming: 0.7–0.9.
<code>--top-k N</code>	40 (0 = aus)	Behält nur die K wahrscheinlichsten Tokens.	Qwen3-Empfehlung: <code>--top-k 20</code> . Niedriger = fokussierter.
<code>--top-p N</code>	0.95 (1.0 = aus)	Nucleus-Sampling: kleinste Menge mit kumulierter Wahrscheinlichkeit $\geq P$ .	Qwen3-Empfehlung: <code>--top-p 0.95</code> . Für deterministische Tasks runter auf 0.85.
<code>--min-p N</code>	0.05 (0.0 = aus)	Mindest-Wahrscheinlichkeit relativ zum wahrscheinlichsten Token.	Qwen3-Empfehlung: <code>--min-p 0</code> kombiniert 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.	Selten genutzt. 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> . Werte >1.2 führen zu seltsamem Vokabular.
<code>--repeat-last-n N</code>	64	Fenster für Repeat-Penalty in Tokens.	Bei langen Outputs (Reasoning) auf 256 bis 512 erhöhen.
<code>--presence-penalty N</code>	0.00	Konstante Strafe pro bereits gesehenem Token.	Community-Tipp für Qwen3-30B-A3B: <code>--presence-penalty 1.5</code> hilft gegen MoE-Loops.
<code>--frequency-penalty N</code>	0.00	Häufigkeitsabhängige Strafe.	Sparsam einsetzen (0.1–0.3). Hohe Werte verzerren den Stil.
<code>--mirostat [0 1 2]</code>	0 (aus)	Adaptive Sampling-Strategie mit konstanter Perplexity.	Alternative zu Top-K/Top-P. <code>--mirostat 2 --mirostat-lr 0.1</code> <code>--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.	Optionaler Ersatz für Repeat-Penalty. 0.8 oft besser als klassische Penalty.
<code>--seed N</code>	-1 (zufällig)	Seed für den Random Number Generator.	Für reproduzierbare Outputs (Benchmarks): <code>--seed 42</code> .

## Beispiel-Startbefehle für 2× RTX 3090

Variante A – Pipeline-Parallelismus (Default, robust, kein NVLink nötig):

```
llama-server \  
--model /models/Qwen3-30B-A3B-Q6_K.gguf \  
--host 0.0.0.0 --port 8080 \  
--n-gpu-layers 99 \  
--split-mode layer \  
--tensor-split 1,1 \  
--flash-attn on \  
--ctx-size 65536 \  
--batch-size 4096 \  
--ubatch-size 2048 \  
--cache-type-k f16 \  
--cache-type-v f16 \  
--threads 8 \  
--threads-batch 12 \  
--parallel 4 \  
--cont-batching \  
--cache-reuse 256 \  
--n-predict 16384 \  
--temp 0.6 --top-k 20 --top-p 0.95 --min-p 0 \  
--repeat-penalty 1.1 --presence-penalty 1.5
```

Variante B – Tensor-Parallelismus (experimentell, ideal bei NVLink, schnellere Token-Generierung):

```
GGML_CUDA_P2P=1 llama-server \  
--model /models/Qwen3-30B-A3B-Q6_K.gguf \  
--n-gpu-layers 99 \  
--split-mode tensor \  
--tensor-split 1,1 \  
--flash-attn on \  
--ctx-size 32768 \  
--cache-type-k f16 --cache-type-v f16 \  
--batch-size 4096 --ubatch-size 2048 \  
--parallel 2 --cont-batching
```

## Empfohlene Tuning-Reihenfolge

- Schritt 1 – Baseline-Benchmark: `llama-bench -m model.gguf -ngl 99 -fa 1 --split-mode layer --tensor-split 1,1`. Liefert die Default-Performance.
- Schritt 2 – Split-Mode vergleichen: Denselben Bench mit `--split-mode tensor` wiederholen (vorher `GGML_CUDA_P2P=1` exportieren). Wenn Token-Generierung (tg128)  $\geq 30$  % schneller ist und keine Stabilitätsprobleme auftreten: tensor verwenden.
- Schritt 3 – Single-GPU-Referenz: `CUDA_VISIBLE_DEVICES=0 llama-bench -m model.gguf -ngl 99`. Manche Workloads sind auf einer GPU sogar schneller, wenn das Modell rein passt (kein Cross-GPU-Sync nötig).
- Schritt 4 – Micro-Batch optimieren: `llama-bench -ngl 99 -fa 1 -ub 512,1024,2048,4096`. Bei 48 GB ist 2048 oft der Sweet-Spot.
- Schritt 5 – Tensor-Split anpassen: Wenn GPU 0 das Display antreibt, asymmetrisch splitten (z. B. `-ts 4,5`), damit GPU 1 mehr Modell-Layer übernimmt.
- Schritt 6 – Sampling justieren: Erst nach Stabilität der Performance an Temperatur, Top-K, Penalties drehen.

## Grober VRAM-Bedarf bei Qwen3-30B-A3B auf 2× 3090

Komponente	Schätzung	Bemerkung
Modellgewichte Q4_K_M	~18 GB	viel Spielraum, sehr lange Kontexte möglich
Modellgewichte Q6_K	~25 GB	empfohlen – sehr gute Qualität
Modellgewichte Q8_0	~32 GB	nahezu lossless
KV-Cache @ 32k Ctx, f16	~7 GB	voller Qualität, kein Verlust
KV-Cache @ 64k Ctx, f16	~14 GB	großzügiger Kontext
KV-Cache @ 128k Ctx, f16	~28 GB	nur mit Q4_K_M-Gewichten realistisch
KV-Cache @ 128k Ctx, Q8_0	~14 GB	kombinierbar mit Q6_K-Gewichten
Compute-Buffer (ub=2048, fa on)	~1.5–2 GB	skaliert mit --ubatch-size
Headroom (Treiber, Display, OS)	~1–2 GB	v. a. auf GPU 0 mit Display

## Quellen

1. llama.cpp Multi-GPU-Dokumentation: [github.com/ggml-org/llama.cpp/blob/master/docs/multi-gpu.md](https://github.com/ggml-org/llama.cpp/blob/master/docs/multi-gpu.md)
2. 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)
3. Diskussion „Fine grained control of GPU offloading“ #7678: [github.com/ggml-org/llama.cpp/discussions/7678](https://github.com/ggml-org/llama.cpp/discussions/7678)
4. r/LocalLLaMA – „Dual GPU llama.cpp speedup“: [reddit.com/r/LocalLLaMA/comments/1tfIngz](https://reddit.com/r/LocalLLaMA/comments/1tfIngz)
5. r/LocalLLaMA – „llama.cpp performance breakthrough for multi-GPU setups“: [reddit.com/r/LocalLLaMA/comments/1q4s8t3](https://reddit.com/r/LocalLLaMA/comments/1q4s8t3)
6. KnightLi – „Is 2× V100 16GB Faster Than One 32GB Card?“ (Multi-GPU-Tradeoffs): [knightli.com/en/2026/05/09/llama-cpp-multi-gpu-offload-performance](https://knightli.com/en/2026/05/09/llama-cpp-multi-gpu-offload-performance)
7. r/LocalLLaMA – „Running Qwen3-Coder-30B-A3B with llama.cpp“: [reddit.com/r/LocalLLaMA/comments/1r6mwsd](https://reddit.com/r/LocalLLaMA/comments/1r6mwsd)