Best LLM Speed Trick: ExLlamaV2 vs llama.cpp Benchmarks (50-85% Faster)

February 14, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** ExLlamaV2 is 50-85% faster than llama.cpp for token generation on NVIDIA GPUs, and 2-2.5x faster for prompt processing. But it only works on NVIDIA, requires the model to fit entirely in VRAM, and has a smaller ecosystem. llama.cpp runs on everything (CPU, AMD, Apple Silicon, Intel Arc), supports CPU offloading for oversized models, and powers Ollama and LM Studio. If you have an NVIDIA GPU and your model fits in VRAM, ExLlamaV2 is the faster option. For everything else, llama.cpp wins by default.

📚 **More on this topic:** Model Formats Explained: GGUF vs GPTQ vs AWQ vs EXL2 · llama.cpp vs Ollama vs vLLM · VRAM Requirements · Text Generation WebUI Guide · Planning Tool

You're running a local model on an NVIDIA GPU, and you want it faster. You've heard ExLlamaV2 is the speed king. You've also heard llama.cpp is "good enough" and runs everything. Which one should you actually use?

This is a straightforward comparison. Both are inference engines that run quantized models on your GPU. They use different model formats (EXL2 vs GGUF) and different architectures (custom CUDA kernels vs cross-platform compute). The speed gap is real. So are the tradeoffs.

What Each Tool Actually Is

llama.cpp

The foundation of local LLM inference. Georgi Gerganov built it to run LLaMA on a MacBook in early 2023. It has since grown into the most widely used inference engine in local AI, with 95,000+ GitHub stars and over 1,000 contributors.

llama.cpp uses the **GGUF format** and runs on every platform: NVIDIA CUDA, AMD ROCm, Apple Metal, Intel SYCL, Vulkan, and pure CPU. Ollama and LM Studio both use llama.cpp as their inference backend. If you've ever run a model through Ollama, you've already used llama.cpp.

Its key strength is flexibility. CPU offloading lets you run models that don't fit in VRAM by splitting layers between GPU and system RAM. No other major inference engine does this.

ExLlamaV2

A speed-focused inference engine built by a solo developer (turboderp) specifically for NVIDIA GPUs. It uses the **EXL2 format**, a quantization scheme where different layers get different bit precisions based on how sensitive they are to quantization error.

ExLlamaV2 achieves its speed through hand-written CUDA kernels optimized for NVIDIA's tensor cores. It doesn't try to support every platform. It does one thing: run quantized models as fast as possible on NVIDIA hardware.

The tradeoff is obvious. NVIDIA GPUs only, model must fit entirely in VRAM, and the ecosystem is smaller. No Ollama support, no LM Studio support. You access it through TabbyAPI or text-generation-webui.

Speed Benchmarks — Head to Head

These benchmarks come from oobabooga's testing (Llama 2 13B on RTX 3090) and matt-c1's llama.cpp speed measurements (Llama 3 8B on RTX 4090). Both were done with models fully loaded into VRAM.

Token Generation Speed

Backend + Format	13B Model (RTX 3090)	Notes
ExLlamaV2 (EXL2 4.25 bpw)	**~57 tok/s**	Custom CUDA kernels
ExLlamaV2 (GPTQ 4-bit)	**~64 tok/s**	GPTQ also supported
llama.cpp (GGUF Q4_K_M)	~31 tok/s	Full GPU offload
llama.cpp (GGUF Q4_K_S)	~35 tok/s	Slightly faster, slightly lower quality
bitsandbytes 4-bit	~23 tok/s	For comparison

ExLlamaV2 generates tokens **50-85% faster** than llama.cpp on the same hardware at the same quantization level. On a 13B model, that's the difference between 31 tok/s and 57 tok/s. Both are readable speeds, but the ExLlamaV2 experience is noticeably snappier.

Prompt Processing (Prefill) Speed

This is where the gap gets dramatic. Prompt processing is how fast the engine ingests your input before generating a response.

Backend	3,200-Token Prompt (8B, RTX 4090)	Ratio
ExLlamaV2	~14,000 tok/s	**1.0x (baseline)**
llama.cpp (flash attention on)	~7,500 tok/s	~1.9x slower
llama.cpp (flash attention off)	~3,500 tok/s	~4x slower

ExLlamaV2 processes prompts **2-2.5x faster** than llama.cpp with flash attention enabled. Without flash attention, the gap is even worse. This matters most with long system prompts, RAG contexts, or any workload that feeds large amounts of text into the model before generating.

Why the Speed Gap Exists

llama.cpp is built for portability. Its compute kernels work across CUDA, Metal, ROCm, Vulkan, and CPU. That means generic code paths that work everywhere but aren't optimized for any specific GPU architecture.

ExLlamaV2 targets NVIDIA tensor cores specifically. Its CUDA kernels are hand-tuned for the way NVIDIA hardware actually moves data through its processing units. It sacrifices portability for throughput.

The analogy: llama.cpp is a car that drives on any road. ExLlamaV2 is a race car that only works on a track, but it's faster on that track.

Speed by VRAM Tier (RTX 3090)

The benchmarks above focus on a single model size. Here's what the speed gap looks like across the model sizes that matter at each VRAM tier, all on an RTX 3090:

Model	VRAM Used	ExLlamaV2 (EXL2)	llama.cpp (GGUF)	Ollama
7B Q4	~5 GB	~80-100 tok/s	~45-55 tok/s	~43-53 tok/s
14B Q4	~9 GB	~40-55 tok/s	~22-30 tok/s	~20-28 tok/s
32B Q4	~20 GB	~20-28 tok/s	~12-17 tok/s	~11-16 tok/s
70B Q3	~40 GB (offload)	Can't run	~3-5 tok/s	~3-5 tok/s

The pattern holds across all sizes: ExLlamaV2 is roughly 1.5-1.8x faster for generation when the model fits entirely in VRAM. The 70B row shows the tradeoff — ExLlamaV2 can't offload to system RAM, so llama.cpp is your only option for models that exceed your VRAM.

Ollama wraps llama.cpp with a small overhead from its server layer. The difference is typically 5-10% — not worth worrying about for most users.

GGUF vs EXL2: The Format Difference

These two backends use different quantization formats. They're not interchangeable — you need the right format for the right engine.

GGUF (llama.cpp)

Every layer is quantized to the same precision. You pick a quant level (Q4_K_M, Q5_K_M, Q6_K, etc.) and the entire model uses it. The "K-quant" variants allocate slightly more bits to attention layers and fewer to feed-forward layers, but it's coarse-grained.

EXL2 (ExLlamaV2)

Each layer gets a different bit precision based on how sensitive it is to quantization. You set a target average (e.g., 4.65 bits per weight) and the calibration process distributes bits where they matter most. A layer that degrades badly at 4-bit might get 6 bits, while a robust layer gets 3 bits.

Quality Comparison

At equivalent sizes, quality is nearly identical:

Format	MMLU Accuracy (8B model)	Perplexity (13B model)
FP16 baseline	65.20%	—
EXL2 8.0 bpw	65.20%	—
GGUF Q8_0	65.23%	—
EXL2 4.25 bpw	64.64%	4.32
GGUF Q4_K_M	64.64%	4.33
EXL2 3.5 bpw	60.28%	—
GGUF Q3_K_M	62.89%	—

Above 4 bits per weight, the formats are equivalent in measured quality. Below 3.5 bpw, GGUF's I-Quants hold up better than EXL2. One Llama 3 70B subjective quality test found EXL2 scored slightly higher in blind evaluations, though the margin was small.

The practical takeaway: **don't choose your backend based on quality differences.** At 4+ bpw, both formats deliver the same model quality. Choose based on speed and hardware compatibility.

Where to Find Models

Both formats are widely available on HuggingFace for popular models. Look for:

- **GGUF**: bartowski, TheBloke (legacy), and official model repos
- **EXL2**: turboderp, LoneStriker, bartowski (some models)

GGUF has significantly more models available since it's the default for Ollama's entire library. EXL2 coverage is good for major models but thinner for niche ones.

When to Use llama.cpp

Your model doesn't fit in VRAM. This is the biggest one. llama.cpp can offload layers to system RAM with the `-ngl` flag. ExLlamaV2 can't. If you have a 24GB GPU and want to run a 70B model, llama.cpp is your only option between these two. It'll be slow (a few tok/s with heavy CPU offloading), but it works.

You're not on NVIDIA. AMD GPUs via ROCm, Apple Silicon via Metal, Intel Arc via SYCL, or pure CPU. ExLlamaV2 doesn't support any of these.

You use Ollama or LM Studio. These tools use llama.cpp internally and only support GGUF. If Ollama is your workflow and you don't want to change, the choice is made for you.

You want the simplest setup. `ollama run qwen3:8b` is one command. ExLlamaV2 requires Python environment management, CUDA toolkit matching, and manual model downloads.

You need the widest model selection. Every model on Ollama's registry is GGUF. HuggingFace has GGUF versions of virtually everything. EXL2 coverage is good but not as comprehensive.

When to Use ExLlamaV2

You have an NVIDIA GPU and the model fits in VRAM. This is the core use case. If a 32B model at Q4 fits in your 24GB of VRAM, ExLlamaV2 will generate tokens 50-85% faster than llama.cpp.

Speed is your top priority. For interactive chat, coding assistants, or any workload where faster responses improve your experience, the speed difference is worth the setup complexity.

You process long prompts regularly. The 2-2.5x prompt processing advantage matters for RAG, long system prompts, or any workflow that feeds large contexts into the model. If you're stuffing 8,000 tokens of context into every request, ExLlamaV2 processes that input twice as fast.

You have multiple NVIDIA GPUs. ExLlamaV2 uses tensor parallelism (both GPUs work on every layer simultaneously). llama.cpp uses pipeline parallelism (GPUs take turns processing layers). On a 2x RTX 3090 setup, ExLlamaV2's approach is significantly faster. In some llama.cpp multi-GPU benchmarks, users found a single GPU was actually faster than two.

You use text-generation-webui or TabbyAPI. Both support ExLlamaV2 natively.

Setup: llama.cpp (via Ollama)

The easiest path. If you already have Ollama, you're already using llama.cpp.

```
# Install Ollama
curl -fsSL https://ollama.com/install.sh | sh

# Pull and run a model
ollama run qwen3:8b
```

For direct llama.cpp usage (more control, no Ollama wrapper):

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

# Run with full GPU offload
./build/bin/llama-cli -m model.gguf -ngl 99 -fa -c 8192 -p "Your prompt"
```

```
# Start an OpenAI-compatible API server
./build/bin/llama-server -m model.gguf -ngl 99 -fa -c 8192 --host 0.0.0.0 --port 8080
```

Key flags:

- `-ngl 99` — Offload all layers to GPU (reduce number if model doesn't fit)
- `-fa` — Enable flash attention (significant speed boost, always use this)
- `-c 8192` — Context window size

Setup: ExLlamaV2 (via TabbyAPI)

TabbyAPI is the recommended frontend. It provides an OpenAI-compatible API server backed by ExLlamaV2.

Step 1: Install TabbyAPI

```
git clone https://github.com/theroyallab/tabbyAPI
cd tabbyAPI

# Create Python environment (3.10-3.12)
python -m venv venv
source venv/bin/activate

# Install dependencies (match your CUDA version)
pip install -r requirements.txt
```

Step 2: Download an EXL2 Model

Find EXL2 models on HuggingFace. For a 24GB GPU, a 32B model at ~4.0 bpw fits well:

```
# Using huggingface-cli
pip install huggingface_hub
huggingface-cli download turboderp/Qwen2.5-32B-Instruct-exl2 --revision 4.0bpw --local-dir model
```

Step 3: Configure and Start

Edit `config.yml`:

```
model:
  model_dir: models/
  model_name: Qwen2.5-32B-4.0bpw

network:
  host: 0.0.0.0
  port: 5000

developer:
  cache_mode: Q4   # Q4 KV cache saves VRAM with minimal quality loss
```

```
# Start the server
python main.py
```

TabbyAPI is now serving at `http://localhost:5000/v1/` with full OpenAI API compatibility. Point Open WebUI, SillyTavern, or any OpenAI-compatible client at it.

Setup Complexity: Honest Assessment

ExLlamaV2/TabbyAPI setup is harder than Ollama. You need to match your CUDA toolkit version with your PyTorch version with your Python version. If any of these are wrong, you get cryptic import errors. Budget 15-30 minutes for first-time setup vs 2 minutes for Ollama.

VRAM Usage Comparison

At equivalent quantization, both backends use similar VRAM for model weights. The difference is in KV cache handling.

KV Cache Options

Backend	KV Cache Precisions	VRAM Savings
ExLlamaV2	FP16, Q8, **Q4**	Q4 cache = ~25% of FP16

Backend	KV Cache Precisions	VRAM Savings
llama.cpp	FP16, Q8	Q8 cache = ~50% of FP16

ExLlamaV2's Q4 KV cache is a meaningful advantage for long-context workloads. At 16K context on a 32B model, the KV cache alone can consume several gigabytes. Q4 cache cuts that substantially with minimal quality loss.

Practical VRAM (Llama 2 13B, RTX 3090)

Format	VRAM Used
EXL2 4.0 bpw	7.88 GB
EXL2 4.65 bpw	9.03 GB
GGUF Q4_K_M (full GPU offload)	~8.5 GB

Comparable for model weights. The cache settings are where you gain or lose VRAM headroom.

Multi-GPU: A Clear Winner

If you have two NVIDIA GPUs, this is the single biggest reason to use ExLlamaV2.

Approach	How It Works	Result
ExLlamaV2 (tensor parallelism)	Both GPUs process every layer simultaneously	~33% speed increase over single GPU
llama.cpp (pipeline parallelism)	GPUs take turns processing layers sequentially	Often **slower** than a single GPU

llama.cpp's multi-GPU implementation splits layers across cards, so GPU 1 processes its layers, then GPU 2 processes its layers. The idle time during handoffs wastes a large portion of the potential speedup. Multiple users have reported that a single GPU outperforms their multi-GPU llama.cpp setup.

ExLlamaV2 distributes computation within each layer across all GPUs. Both cards work on every token. With NVLink (available on RTX 3090), the improvement is even larger.

If you bought a second GPU specifically for speed, use ExLlamaV2. Using llama.cpp with two GPUs is often worse than using one.

Can You Use Both?

Yes, and many power users do.

Common dual setup:

- **Ollama (llama.cpp)** for daily use — quick queries, Open WebUI chat, trying new models
- **TabbyAPI (ExLlamaV2)** for focused work: long coding sessions, RAG-heavy workflows, anything where 50%+ faster generation improves your experience

They don't conflict. Ollama runs on port 11434, TabbyAPI runs on port 5000. Switch between them by changing which endpoint your frontend points at. The only constraint is VRAM — you can't run both engines with loaded models simultaneously unless you have enough VRAM for both.

A practical workflow: keep Ollama as your default. When you sit down for a long session where speed matters, stop Ollama (`ollama stop`), start TabbyAPI, and work at full speed.

A Note on ExLlamaV3

ExLlamaV3 is now available (v0.0.22 as of February 2026) with a new **EXL3** quantization format based on trellis coding. The headline claim: Llama 3.1 70B remains coherent at 1.6 bits per weight, fitting under 16GB of VRAM with extreme compression.

ExLlamaV3 also adds tensor parallelism, expert parallelism for MoE models, continuous batching, and speculative decoding. TabbyAPI already supports it.

It's still early. Performance on Ampere GPUs (RTX 3090) has known issues, and the ecosystem is months behind ExLlamaV2. Worth watching, but ExLlamaV2 remains the stable recommendation for now.

What About vLLM?

If you're serving models to multiple users, neither ExLlamaV2 nor llama.cpp is the right tool. vLLM is built for high-throughput serving with continuous batching — it handles many simultaneous requests efficiently. But for single-user local inference, vLLM adds complexity without meaningful speed gains. Stick with ExLlamaV2 for speed or llama.cpp for flexibility.

Bottom Line

Factor	llama.cpp (GGUF)	ExLlamaV2 (EXL2)
Generation speed	Baseline	**50-85% faster**
Prompt processing	Baseline	**2-2.5x faster**
Hardware support	Everything	NVIDIA only
CPU offloading	Yes	No
Multi-GPU	Poor (pipeline)	**Good (tensor parallel)**
Setup difficulty	Easy (Ollama)	Moderate (TabbyAPI)
Model availability	Widest	Good, not as wide
Ecosystem size	95,000+ GitHub stars	4,400 GitHub stars
KV cache options	FP16, Q8	FP16, Q8, **Q4**

Use llama.cpp (usually via Ollama) if any of these apply: you're not on NVIDIA, your model doesn't fit in VRAM, you want the simplest possible setup, or you use Ollama/LM Studio.

Use ExLlamaV2 (via TabbyAPI) if all of these apply: you have an NVIDIA GPU, your model fits in VRAM, and you want the fastest inference available on consumer hardware.

For most readers of this site, llama.cpp via Ollama is the right starting point. But if you're on NVIDIA and tired of watching tokens crawl out one at a time, ExLlamaV2 is the upgrade. The speed difference isn't subtle. 31 tok/s vs 57 tok/s on the same model and same GPU. You'll feel it immediately.

Related Guides

- Model Formats Explained: GGUF vs GPTQ vs AWQ vs EXL2 — deep dive on quantization formats
- llama.cpp vs Ollama vs vLLM: When to Use Each — the three-way comparison
- VRAM Requirements for Local LLMs — how much VRAM you need
- Text Generation WebUI Guide — supports both ExLlamaV2 and llama.cpp
- Best Used GPUs for Local AI — hardware recommendations
- Quantization Explained — how quantization works

- Local AI Planning Tool — VRAM Calculator

Get notified when we publish new guides.

Subscribe — free, no spam

Source: https://insiderllm.com/guides/exllamav2-vs-llamacpp-speed-comparison/