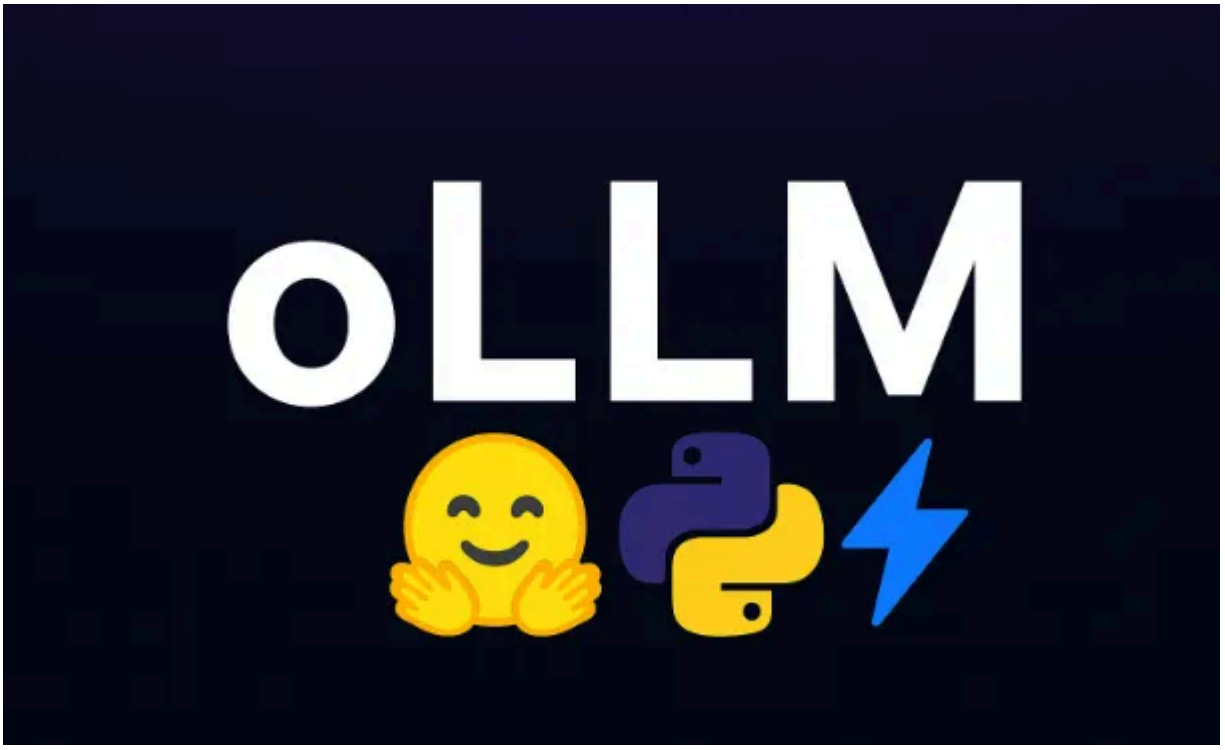
AI Advances · Follow publication

Member-only story

I Ran an 80-Billion-Parameter AI Model on My 8GB GPU -My Experience with oLLM

5 min read · Oct 6, 2025

Himanshu Soni Follow



Original Image from oLLM Repo. credits: https://github.com/Mega4alik/ollm

As someone who creates AI-related content and experiments with new tools for my projects, I wanted to explore **large language models (LLMs)** locally to test prompts, analyze documents, and create small prototypes.

But there was a problem: **my GPU.**

I'm running an **RTX 3060 Ti (8GB VRAM)**, a solid mid-range GPU for gaming and light AI work. Every time I tried loading a model beyond 13 billion parameters, I hit the same dreaded message.

> *"CUDA out of memory."*

Even after trying quantization and memory-optimized libraries, I couldn't get large models like Llama-2-70B or Falcon-40B to load without crashing. I was stuck using smaller, trimmed-down versions that didn't perform nearly as well.

And honestly, it was discouraging. The world of large models felt completely out of reach for people like me who didn't own expensive GPUs or cloud subscriptions.

Until one night, while scrolling through AI research posts, I came across a small Python library that completely changed my setup: **oLLM**.

The Hardware Wall Every AI Enthusiast Knows

If you've ever tried to run massive LLMs locally, you already know the problem:

- Models like GPT-3, Llama 3.1, or Qwen3 are **tens of billions of parameters** big.
- They need GPUs like **A100s or H100s**, which are enterprise-grade hardware.
- Even cloud solutions are expensive if you just want to experiment casually.

Quantization helped reducing model size by lowering precision but it always came at a cost: **reduced accuracy** and **loss of reasoning depth**.

So for most of us, the playground of large-scale models stayed locked behind expensive hardware.

That was, until I decided to try running **Qwen3-Next-80B** on my humble RTX 3060 Ti.

Discovering oLLM: The Turning Point

What caught my attention was oLLM's promise:

> *"Run 80B+ parameter models on consumer GPUs using SSD offloading no quantization required."*

Repo Link : https://github.com/Mega4alik/ollm

It sounded impossible.

But after digging deeper, I realized what made it work **smart memory management** and **SSD streaming**.

Instead of cramming the entire model into GPU VRAM, oLLM cleverly **streams model weights and the attention cache directly from SSD to GPU** in real-time.

That means only the parts needed at the moment get loaded, drastically reducing VRAM usage while maintaining full precision.

Comparison Traditional VS oLLM

It was basically saying:

> "You don't need a supercomputer. You just need a fast SSD."

My Setup

Here's what I used for my tests:

- **GPU:** NVIDIA RTX 3060 Ti (8GB VRAM)
- **CPU:** Ryzen 7 5800X
- **RAM:** 32 GB DDR4
- **Storage:** 1TB Samsung 980 Pro NVMe SSD
- **OS:** Windows 11

My Step-by-Step Installation Journey

Here's exactly how I got oLLM up and running no shortcuts, no magic, just what worked for me.

Step 1: Check the Hardware

Before starting, I made sure my setup met the basics:

- At least **8GB VRAM** (for inference)
- **NVMe SSD** (Gen3 or Gen4 preferred SATA SSDs are too slow)
- Python 3.10 or newer

Speed matters more than size here, SSD bandwidth is the real hero.

Step 2: Install Python and Git

I already had Python 3.11 installed, but if you don't, grab it from python.org.

Step 3: Create a Virtual Environment

This kept my dependencies clean:

```
python -m venv ollm-env
ollm-env\Scripts\activate
```

Step 4: Install PyTorch (with GPU Support)

I checked my CUDA version:

```
nvidia-smi

# Then installed the matching PyTorch build (for CUDA 12.1):
pip install torch torchvision torchaudio --index-url https://download.pytorch.o

# If you don't have a GPU, install the CPU version (it works, just slower):
pip install torch torchvision torchaudio
```

Step 5: Install oLLM

This part was simple:

```
pip install ollm

# Or, for the latest GitHub version:

pip install git+https://github.com/OptimalScale/ollm.git
```

Step 6: Download and Load a Model

I logged into my Hugging Face account and chose a model:

```
huggingface-cli login
```

Then I used Python to load the model:

```
from ollm import oLLM
from transformers import AutoTokenizer

model_name = "Qwen/Qwen3-Next-80B"
model = oLLM.from_pretrained(
    model_name,
    device_map="auto",
    past_key_values_path="./kv_cache_qwen80b"
)
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

Step 7: Generate My First Output

Once loaded, I tried generating text:

```
prompt = "Explain in simple terms how SSD offloading allows large LLMs to run o
inputs = tokenizer(prompt, return_tensors="pt").to("cuda")
outputs = model.generate(**inputs, max_new_tokens=100)
print(tokenizer.decode(outputs[0], skip_special_tokens=True))
```

When I saw the first output, it was honestly surreal.
My 8GB GPU had just handled an **80-billion-parameter** model.

Step 8: Tuning Offloading

I experimented with parameters to make it smoother:

```
model = oLLM.from_pretrained(
    "Qwen/Qwen3-Next-80B",
    device_map="auto",
    torch_dtype="auto",
    low_cpu_mem_usage=True,
    past_key_values_path="./kv_cache_qwen80b"
)
```

This reduced CPU usage and ensured the KV-cache was efficiently offloaded to SSD.

Step 9: Cleaning Cache

After multiple runs, the cache folder grew to hundreds of GBs, so I cleaned it:

```
rmdir /s /q kv_cache_qwen80b
```

That's just SSD hygiene.

Step 10: Verify GPU Load

Finally, I ran:

```
nvidia-smi
```

It showed stable usage around **7.5GB VRAM** the rest was streamed dynamically from the SSD.

Comparison of Models training on oLLM

It wasn't blazing fast about one token every two seconds — but it worked **without quantization and without losing accuracy.**

When oLLM Truly Shines

I now use oLLM for:

- Long document analysis (100k+ tokens)
- Offline code and research summarization
- RAG pipeline experimentation

For content creators, developers, and researchers who want to *feel* the power of large models without cloud costs, this is a dream come true.

The Trade-offs I Noticed

It's not perfect here's what I learned:

Issues Fixes

Despite these, the benefits far outweigh the issues.

Final Thoughts

Running an **80-billion-parameter model on my 8GB GPU** still feels unreal.

oLLM doesn't just make large models accessible it **changes the narrative**. It proves that innovation in software can outsmart hardware limitations.

For me, this isn't just about performance. It's about **access** giving independent creators, students, and small developers the power to experiment with models once reserved for data centers.

So if you've been sitting on your gaming PC thinking you're missing out on the big AI revolution you're not.

With oLLM, your GPU isn't too small it's just waiting to be used smarter.

AI | Data Science | Machine Learning | Deep Learning

Responses (13)

Bgerby

What are your thoughts?

ML
6 days ago

What is the token per second?

24 1 reply Reply

Rodaddy
3 days ago

Does this work with apple silicon or is the a Cuda thing? I can see a great use for use'n 24GB of my total memory for this and allowing to offload to my macs very fast internal ssd as you did. I guess I can go pull it and find out, just thought I'd ask

12 Reply

Paul Proctor
4 days ago

Running a large model out of system ram has always been doable. You are basically running most of the model on cpu though, which means it will be very slow. If you want a 70b model to run fast, you need a stack of gpus with enough vram to hold the... more

6 Reply

See all responses

More from Himanshu Soni and AI Advances

In Towards AI by Himanshu Soni

I Ran an AI Model on my CPU, and It's the Future Here's Why.

For years, the world of artificial intelligence has been dominated by one expensive truth: if you want to run powerful AI models, you need...

Oct 6 9

In AI Advances by Fareed Khan

How to Build an Advanced Agentic E-commerce WhatsApp Bot with Hyperstack AI Studio

Meta Webhooks, RAG-Powered Q&A, and Secure, Stateful Logic

Sep 26 577 7

In AI Advances by Kuriko Iwai

Mastering Autoencoders (AEs) for Advanced Unsupervised Learning

Explore the core mechanics of AEs with essential regularization techniques and various layer architectures

Oct 2 448 3

Himanshu Soni

Can Sanskrit End AI Hallucinations? The Ancient Language with a Modern Solution

AI models like ChatGPT and Google Translate often “hallucinate” because of language ambiguity. Could Sanskrit’s precision and context-rich...

Oct 1 3 1

See all from Himanshu Soni

See all from AI Advances

Recommended from Medium

In AI Advances by Nikhil Anand

I wasted months running slow LLMs before learning this

Why your LLM is running at just 10% of its potential speed

6d ago 512 9

In Generative AI by Thomas Reid

Google puts another nail in the RAG coffin with URL Context Grounding

Eliminate model hallucinations when processing online data

Oct 2 303 10

In Coding Nexus by Civil Learning

RIP Fine-Tuning: How Stanford's ACE Framework Teaches AI to Learn Without Retraining

For years, people have used 'fine-tuning" as the go-to method to enhance large language models' performance. This involves supplying the...

5d ago 217 3

In Level Up Coding by Vivedha Elango

Why Your RAG System Fails Complex Questions? (And How Structure Fixes Everything)

Understanding the Retrieval and Structuring (RAS) Paradigm for Precise Reasoning and Domain Expertise with Implementation Examples

6d ago 310 6

Max Petrusenko

The Smartphone That Makes Police Officers Sweat (And Why You Need One)

How a $400 Google Pixel running a secret operating system became the most feared device in law enforcement

Oct 7 3.1K 45

Bytefer

NVIDIA’s Open-Source ASR Model Kills Whisper, With 1 Hour of Audio in Just 19 Seconds and Higher...

Open source and commercially available, super fast and high quality, the best choice for local AI!

Sep 30 123

See more recommendations