llama-bench — cosa fa, come funziona, come si lancia e come si interpretano

Ultimo aggiornamento: 2026-11-06

llama-bench — cosa fa, come funziona, come si lancia e come si interpretano i risultati

llama-bench è il benchmark ufficiale di llama.cpp: misura la velocità di prefill (elaborazione del prompt, in tok/s) e di decode (generazione token per token, in tok/s) di un modello GGUF, su una o più GPU, con i parametri di runtime che scegli tu. Non genera testo reale: usa token casuali per stressare il modello in condizioni deterministiche e confrontare configurazioni tra loro.

1. Come funziona

1. Carica il modello GGUF specificato con i parametri di offload che gli passi (`-ngl` layer in GPU, `-ncmoe` esperti MoE in CPU, `-sm` split mode, `-fa` flash attention, `-lm` modalità di caricamento).
2. Crea un contesto grande appena quanto serve al punto di test: `n_ctx = prompt + generazione + depth`. In questa build non esiste il flag `-c`: il contesto non si dichiara, si deriva dai punti `-pg`.
3. Per ogni punto di test `-pg <pp,tg>` fa due tipi di misura: Prefill (`pp`): elabora `pp` token casuali in chunk di `-b` batch (suddivisi in ubatch `-ub`), e misura il tempo totale → `pp / tempo = tok/s`. Decode (`tg`): genera `tg` token uno per volta, sincronizzando la GPU dopo ogni token (condizione peggiore e più realistica per la chat) → `tg / tempo = tok/s`.
4. Esegue una warmup (saltabile con `--no-warmup`) e poi `-r` ripetizioni (default 5) di ogni punto, tenendo solo i tempi cronometrati. Se usi `-d <depth>`, prima di ogni ripetizione ripristina un contesto già riempito con `depth` token, così la misura di decode parte da un contesto lungo (utile per misurare il decode a contesto 256K senza rigenerarlo ogni volta).
5. Per ogni punto riporta la media dei tok/s e la deviazione standard (±), in una tabella markdown (o csv/json/jsonl/sql con `-o`).

Il punto chiave: un solo comando può misurare molte combinazioni (multipli valori di batch, ubatch, tipi di KV cache, punti prefill/decode), perché ogni flag accetta liste separate da virgole.

2. Come si lancia

Prerequisiti: il binario `llama-bench` della stessa build di `llama-server` (in questo sistema: `/localIA/llama/llama.cpp/build/bin/llama-bench`), il file GGUF del modello, e la GPU libera (il benchmark usa VRAM e CPU a pieno: fermare prima eventuali server che usano la stessa GPU).

2.1 Comando minimo

```
llama-bench -m /percorso/modello.gguf
```

Con i default (5 ripetizioni, prompt 512 + generazione 128, batch 2048, ubatch 512, KV f16) stampa una tabella con le righe `pp512` e `tg128`.

2.2 I parametri che contano

Flag	Default	Cosa fa
`-m`	—	file GGUF da testare
`-pg <pp,tg>`	—	punti di test: prefill di `pp` token, generazione di `tg` token. Più punti = più righe. Es. `-pg 4096,0 -pg 65536,256`
`-r`	5	ripetizioni per punto (ognuna cronometrata)
`-b`	2048	batch size (token per passo)
`-ub`	512	ubatch (quantità di token che la GPU elabora in una volta sola)
`-ctk` / `-ctv`	f16	quantizzazione KV cache per key/value: `f16`, `q8_0`, `q4_0`…
`-ngl`	-1 (tutto)	layer offloadati in GPU
`-ncmoe`	0	esperti MoE lasciati in CPU (0 = tutti in GPU; 9999 = tutti tranne la parte che non sta)
`-sm`	layer	split mode: `none`, `layer`, `row`, `tensor`
`-fa`	auto	flash attention: `on`, `off`, `auto`
`-dev`	auto	dispositivo GPU: `CUDA0`, `CUDA1`, …
`-lm`	auto	load mode: `auto`, `none`, `mmap`, `mlock`, …
`-d`	0	depth: token di contesto pre-riempito prima della misura di decode
`-t`	8	thread CPU
`-o`	md	formato output: `md`, `csv`, `json`, `jsonl`, `sql`
`--no-warmup`	—	salta la warmup
`--progress`	—	progresso a video durante l'esecuzione

I flag a lista accettano valori multipli separati da virgola (es. `-b 1024,2048,4096`): il benchmark combina ogni valore con tutti gli altri e produce una riga per combinazione — è così che si fa una matrice di tuning in un solo lancio.

2.3 Esempio: un singolo punto di tuning

```
llama-bench -m modello.gguf -ngl 999 -fa on \
  -ctk q8_0 -ctv q4_0 -b 2048 -ub 256 -r 3 \
  -pg 4096,0 -pg 65536,256
```

Misure: prefill a 4096 token e decode di 256 token dopo un prompt da 65536 (contesto ~65800), con KV cache key a 8 bit e value a 4 bit, 3 ripetizioni.

2.4 Esempio: MoE su una GPU specifica (il caso di questo server)

```
GGML_CUDA_DISABLE_GRAPHS=1 llama-bench \
  -m /localIA/llama/models/Qwen3.8.35B_Moe/Qwen3.8-35B-A3B-IQ4_XS.gguf \
  -ngl 999 -ncmoe 9999 -sm layer -fa on -lm none -dev CUDA1 \
  -ctk q8_0 -ctv q4_0 -b 2048 -ub 256 -r 1 \
  -pg 4096,0 -pg 65536,0 -pg 262144,0 -pg 65536,256
```

Lettura: tutti i layer possibili in GPU (`-ngl 999`), tutti gli esperti MoE che non stanno in VRAM in CPU (`-ncmoe 9999`), solo la GPU `CUDA1`, contesto massimo ~262K derivato dal punto `pg 262144,0`.

Nota: `GGML_CUDA_DISABLE_GRAPHS=1` è utile quando si confrontano configurazioni che cambiano le dimensioni del batch (i CUDA graph possono mascherare differenze reali); per uso singolo non serve.

2.5 Matrice completa in un solo script

Un pattern pratico: un loop di shell che lancia `llama-bench` per ogni combinazione di (tipo KV, batch/ubatch) e appende i risultati a un log. Lo script usato su questo server (`/usr/local/bin/moe256k-matrix.sh`) è nell'allegato in fondo.

3. Come si interpretano i risultati

La tabella markdown ha queste colonne (header reali di questa build):

```
| model | size | params | backend | ngl | n_cpu_moe | n_ubatch | fa | dev | lm | test | t/s |
```

- model — nome e quantizzazione del modello (es. `qwen35moe 35B.A3B IQ4_XS - 4.25 bpw`); `bpw` = bit per peso.
- size — dimensione del file GGUF.
- params — numero di parametri totali.
- backend — backend compute (es. `CUDA`).
- ngl — layer offloadati in GPU (999 = "tutti quelli che stanno").
- n_cpu_moe — esperti MoE rimasti in CPU (9999 = "tutti quelli che non stanno in VRAM").
- n_ubatch — ubatch usato.
- fa — flash attention: `1` = on, `0` = off.
- dev — dispositivo (es. `CUDA1`).
- lm — load mode (`none`, `mmap`, …).
- test — il punto misurato (vedi sotto).
- t/s — velocità in token/secondo: media ± deviazione standard delle `-r` ripetizioni.

3.1 Il nome del test

- `pp4096` — prefill: elaborati 4096 token di prompt → tok/s di prefill su prompt da 4096.
- `tg128` — generazione: 128 token generati (dopo il prompt default di 512) → tok/s di decode a contesto corto.
- `pp65536+tg256` — le due fasi insieme: prefill da 65536 poi 256 token generati → la riga riporta la velocità della parte chiesta (il punto con `tg>0` misura il decode in un contesto lungo: il contesto totale è `pp+tg`).

- Sufisso `@ d123456` — la misura di decode è stata fatta con un contesto già riempito con 123456 token (flag `-d`): è il modo "puro" di misurare il decode a contesto lungo senza rigenerarlo.

3.2 Cosa guardare

1. Confrontare solo righe con lo stesso punto di test. La velocità dipende fortemente dalla lunghezza: `pp262144` è quasi sempre molto più lento di `pp4096` (più contesto = più KV da gestire a ogni step); `tg` a contesto lungo è più lento che a contesto corto.
2. Prefill (pp): domina l'attesa prima della prima risposta. Se vuoi risposte rapide a prompt lunghi, guarda le righe `pp` grandi: batch/ubatch più alti in genere aumentano il prefill fino a un limite, poi saturano o peggiorano.
3. Decode (tg): domina la lettura della risposta. Dipende da: dimensione del contesto (più contesto = più lento), quantizzazione KV (q8_0/q4_0 = più veloce di f16, con piccolo costo di qualità), flash attention, quanti layer/experti stanno in VRAM.
4. La deviazione standard (±): se è grande rispetto alla media, la misura è instabile (altra attività sulla GPU, thread CPU contesi, swap). Con GPU dedicata e `-r ≥ 3` dovrebbe essere piccola.
5. Scelta della combo: per la stessa riga `tg` (stesso contesto), la combo con più tok/s vince per decode; per `pp`, vince quella con più tok/s a prefill. Spesso non c'è un vincitore unico: batch grandi favoriscono il prefill, ubatch piccoli possono favorire il decode a contesto corto. Si sceglie un compromesso per l'uso reale (es. 2048/256 è tipicamente un buon punto medio).

3.3 Esempio di lettura (misura reale, Qwen 35B MoE su 3060)

```
| qwen35moe 35B.A3B IQ4_XS - 4.25 bpw | 18.27 GiB | 35.51 B | CUDA | 999 | 9999 | 256 | 1 | CUDA1 | none |
pp512  | 296.79 ± 0.00 |
| ...                                                                                                  | tg128  | 28.45 ± 0.00  |
```

Lettura: prefill a prompt 512 a ~297 tok/s; decode a contesto ~640 a ~28 tok/s (misura breve, senza MTP/speculazione). Le righe `pp` più grandi e le `tg` a contesto lungo della matrice completa danno il quadro completo.

3.4 Avvertenze

- I numeri dipendono da build e flag: non confrontare risultati presi con flag diversi o su versioni diverse di llama.cpp.
- `llama-bench` misura l'inference grezza: non include speculazione MTP, che in produzione può alzare il decode; i numeri qui sono quindi una baseline (valore minimo garantito).
- Il benchmark usa token casuali: la velocità è stabile e ripetibile, ma non riflette il pattern di un testo reale (che di solito è simile o leggermente migliore).
- Se due configurazioni hanno differenze < 5% con ± simile, di solito non vale la pena scegliere una in base a quelle misure: guarda la VRAM/RAM usata e la qualità (per la KV).

Allegato — script matrice usato su questo server

`/usr/local/bin/moe256k-matrix.sh` (log in `/tmp/moe256k-matrix.log`):

```
#!/usr/bin/env bash
# Matrice llama-bench: tipi di KV x batch/ubatch
# Misure: prefill 4K/64K/256K + decode 256 token a contesto 64K
LB=/localIA/llama/llama.cpp/build/bin/llama-bench
M=/localIA/llama/models/Qwen3.8.35B_Moe/Qwen3.8-35B-A3B-IQ4_XS.gguf
export GGML_CUDA_DISABLE_GRAPHS=1
LOG=/tmp/moe256k-matrix.log
```

```
: > "$LOG"

run() { # kt vt b ub
  echo "===== ctk=$1 ctv=$2 b=$3 ub=$4 =====" >> "$LOG"
  $LB -m "$M" -ngl 999 -ncmoe 9999 -sm layer -fa on -lm none -dev CUDA1 \
      -ctk "$1" -ctv "$2" -b "$3" -ub "$4" -r 1 \
      -pg 4096,0 -pg 65536,0 -pg 262144,0 -pg 65536,256 \
      >> "$LOG" 2>&1
  echo "DONE ctk=$1 ctv=$2 b=$3 ub=$4 rc=$?" >> "$LOG"
}

run f16  f16  2048 256          # riferimento: KV piena
run q8_0 q8_0 2048 256         # riferimento: KV 8/8
for B in "1024 128" "2048 256" "4096 256" "8192 512"; do
  read -r BB UB <<< "$B"
  run q8_0 q4_0 "$BB" "$UB"    # combo operativa: KV 8/4 x 4 batch
done
echo ALL-DONE >> "$LOG"
```

Avvio: `nohup bash /usr/local/bin/moe256k-matrix.sh >/tmp/moe256k-matrix-run.out 2>&1 &` — il prefill da 256K pesa decine di minuti a punto: pianificare di conseguenza (con `-r 1` e 6 combo, 3–4,5 ore).