Guida MoE DeepSeek DwarfStar esperienza locale

Ultimo aggiornamento: 2026-08-18

CAPIRE UN MODELLO MoE IN ESECUZIONE LOCALE

DeepSeek V4 Flash e DwarfStar (DS4)

Dalla teoria dei Mixture of Experts all’esperimento reale con 32 GB di RAM, RTX 5060 Ti 16 GB e SSD SATA

Esperimento svolto il 6 agosto 2026

Documento didattico personale

Come leggere questa guida

Il documento alterna tre livelli di informazione, tenuti volutamente distinti:

- Concetti generali: come funzionano i transformer, i Mixture of Experts, il prefill, il decode e la KV cache.
- Comportamento documentato di DwarfStar/DS4: loader specializzato, formato GGUF, backend CUDA e streaming degli esperti da SSD.
- Osservazioni del nostro esperimento: messaggi di log, errori incontrati, configurazioni provate e prestazioni misurate sulla macchina reale.

Indice

1. Il modello mentale in due minuti

Nel nostro caso il file GGUF occupava circa 86,7 GB decimali, cioè circa 81 GiB visualizzati da Linux. La GPU possedeva 16 GiB di VRAM e il PC 32 GiB di RAM. DS4 è riuscito ugualmente a produrre testo perché non ha cercato di tenere tutti gli esperti nella GPU o nella RAM: ha lasciato la maggior parte dei pesi sullo SSD e ha caricato solo gli esperti necessari, man mano che il router li richiedeva.

Questa soluzione risolve un problema di capacità, ma non quello della velocità. Lo SSD SATA è molto più lento della VRAM. Quando la cache degli esperti non contiene ciò che serve, DS4 deve leggere dal disco, preparare i dati, copiarli verso la GPU ed eseguire il calcolo. Nel nostro test ciò ha portato a circa 0,16 token al secondo: poco più di sei secondi per token generato.

Figura 1 — Differenza concettuale fra un blocco denso e un blocco Mixture of Experts.

2. Da testo a token: cosa fa un LLM durante l’inferenza

2.1 Il testo viene trasformato in token

Un modello linguistico non riceve direttamente parole e frasi. Un tokenizer scompone il testo in unità numeriche chiamate token. Un token può essere una parola intera, una parte di parola, un segno di punteggiatura o una sequenza frequente di caratteri.

La scomposizione esatta dipende dal tokenizer incluso nel modello. L’importante è capire che ogni token attraversa una successione di livelli matematici e produce un nuovo stato vettoriale, cioè una rappresentazione numerica del suo significato nel contesto.

2.2 Due fasi: prefill e generazione

L’inferenza ha due fasi molto diverse:

Il prefill può elaborare molti token in parallelo, ma nel nostro caso lo streaming degli esperti dal disco lo rendeva comunque lento. Durante il decode il ciclo viene ripetuto token per token: per una risposta di 16 token, il modello compie 16 iterazioni successive.

2.3 Attenzione e KV cache

L’attenzione permette a ogni token di utilizzare informazioni provenienti dai token precedenti. Per non ricalcolare da zero tutto il prompt a ogni nuova parola, il programma conserva una Key–Value cache, abbreviata in KV cache. È una memoria di lavoro della conversazione, distinta dai pesi del modello.

3. Modello denso e Mixture of Experts

3.1 Il modello denso

In un transformer denso, ogni token attraversa gli stessi grandi blocchi di pesi. In particolare, il blocco feed-forward (FFN) applica sempre la stessa rete a tutti i token. Se il modello cresce, crescono sia la memoria necessaria sia la quantità di calcolo effettuata per ciascun token.

Schema semplificato:

3.2 Il modello MoE

In un Mixture of Experts, il grande blocco feed-forward viene sostituito da molti blocchi più piccoli chiamati esperti. Un router assegna un punteggio agli esperti e sceglie i migliori k per il token corrente. Il risultato finale è una combinazione pesata delle uscite degli esperti scelti.

Formula concettuale

La formula non significa che gli esperti discutano fra loro. Sono reti neurali indipendenti che ricevono lo stesso stato del token. Il router decide quanto contribuire debba ciascuna uscita.

Figura 2 — Flusso semplificato di un token in un livello MoE.

3.3 Un esempio intuitivo

Immaginiamo il token “software” nella frase «Il software libero è conoscenza condivisa». In un determinato livello, il router potrebbe assegnare punteggi elevati a esperti che, durante l’addestramento, si sono specializzati in codice, terminologia tecnica o licenze. In un livello successivo, lo stesso token potrebbe essere inviato a una combinazione diversa, più utile per la struttura grammaticale o per il significato della frase.

4. Router, esperti condivisi e instradati

4.1 Il router

Il router è una piccola parte del modello che, dato lo stato di un token, produce un punteggio per ogni esperto. Vengono selezionati i migliori k. Questa operazione è chiamata routing o instradamento.

1. Il token entra nel livello con un vettore numerico.
2. Il router calcola i punteggi degli esperti.
3. Vengono scelti gli esperti Top-k.
4. Gli esperti selezionati elaborano il token.
5. Le uscite vengono combinate secondo i pesi del router.
6. Il token prosegue nel livello successivo.

4.2 Esperti condivisi

L’architettura DeepSeekMoE introduce anche esperti condivisi: reti sempre attive che catturano conoscenze comuni e riducono la duplicazione fra gli esperti instradati. Gli esperti condivisi funzionano come una base generale, mentre quelli routed vengono scelti dinamicamente. Questa distinzione è descritta nel lavoro DeepSeekMoE [3].

4.3 Cosa significavano “43 × 6” nei nostri log

DS4 ha riportato un working set concettuale di «43 layers × 6 experts». Per il nostro runtime significa che un token attraversava 43 livelli con routing e, in ciascun livello, venivano richiesti 6 esperti. Si tratta quindi di 258 invocazioni di esperto per token.

5. Perché il MoE è efficiente nel calcolo ma enorme in memoria

Il vantaggio fondamentale del MoE è separare due grandezze che, in un modello denso, crescono insieme:

Questa è la ragione per cui DeepSeek V4 Flash può essere relativamente efficiente quando dispone di molta memoria veloce, ma può diventare lentissimo quando i pesi devono essere recuperati continuamente da un disco SATA.

6. Quantizzazione, imatrix e formato GGUF

6.1 Perché quantizzare

I pesi di un modello sono numeri. In alta precisione ogni numero occupa più bit; riducendo la precisione, il file e la memoria necessaria diminuiscono. La quantizzazione sostituisce numeri molto precisi con rappresentazioni compatte, accettando una certa perdita di accuratezza.

6.2 Quantizzazione asimmetrica della variante usata

Il file usato era `DeepSeek-V4-Flash-IQ2XXS-w2Q2K-AProjQ8-SExpQ8-OutQ8-chat-v2-imatrix-0731.gguf`. Il nome sintetizza la ricetta: le parti più voluminose, cioè gli esperti routed, sono compresse in modo molto aggressivo; componenti sensibili come proiezioni, esperti condivisi e output restano a precisioni più alte. Il README di DS4 sottolinea che non si tratta di una quantizzazione uniforme: è intenzionalmente asimmetrica [1].

6.3 Che cos’è una importance matrix

Una importance matrix, spesso abbreviata in imatrix, viene ricavata facendo passare dati rappresentativi nel modello e osservando quali pesi o direzioni sono più importanti per le attivazioni. Il quantizzatore può così distribuire meglio l’errore: protegge maggiormente le parti sensibili e accetta più approssimazione dove l’impatto è minore. Non rende la quantizzazione senza perdita, ma la rende più informata.

6.4 Il contenitore GGUF

GGUF è un formato binario pensato per distribuire e caricare modelli destinati all’inferenza. Contiene tensori quantizzati e metadati, come architettura, tokenizer e informazioni necessarie al loader. La specifica GGUF lo descrive come un formato estensibile, progettato per il caricamento rapido e per il deployment in un singolo file [4].

7. Che cos’è DwarfStar / DS4

DwarfStar, il cui repository e i binari usano il nome DS4, è un motore di inferenza nativo e molto specializzato. Non è una piattaforma universale come un framework generale: è progettato attorno a DeepSeek V4 Flash e ai file GGUF preparati per il progetto. Il README lo definisce esplicitamente un engine “narrow”, con caricamento, prompt rendering, stato KV e backend dedicati [1].

Figura 3 — I principali passaggi compiuti da DS4 per una richiesta.

8. Tutti i macro-step di DS4, dall’avvio all’output

8.1 Controllo degli argomenti

Il programma legge le opzioni e verifica che la combinazione sia supportata. Il nostro primo tentativo univa `--gpu-vram 14` e `--ssd-streaming`. DS4 lo ha rifiutato subito perché quel limite di VRAM attivava il percorso di placement, considerato incompatibile con lo streaming SSD in quella configurazione. Questo controllo ha evitato un caricamento errato o ambiguo.

Esempio osservato

8.2 Inizializzazione del backend CUDA

DS4 inizializza CUDA, seleziona il dispositivo visibile e rileva l'architettura. Nel nostro caso: RTX 5060 Ti, architettura Blackwell e target runtime sm_120. La compilazione aveva richiesto il target architetturale `sm_120a` perché alcuni kernel MXFP4 block-scaled utilizzavano istruzioni specifiche. NVIDIA distingue infatti i target generici da quelli con suffisso "a", che abilitano caratteristiche legate a una determinata architettura [5].

8.3 Apertura del GGUF e costruzione della mappa dei tensori

Il programma apre il file, legge metadata e indirizzi dei tensori e prepara una mappa tra componenti logici e intervalli del file. Non è necessario copiare subito tutti gli 81 GiB. Con lo streaming attivo, il log ha indicato una mappa iniziale limitata all'embedding dei token, circa 0,99 GiB di span.

8.4 Pianificazione della memoria

Prima di eseguire il prompt, DS4 calcola un piano che comprende:

- KV cache necessaria al contesto richiesto;
- buffer temporanei e workspace CUDA;
- pesi residenti che conviene tenere in VRAM;
- riserva temporanea per il prefill degli esperti;
- cache dinamica degli esperti per il decode.

Piano osservato con cache esperti da 4 GB

Il budget da 4 GB non era una cache generica. Prima veniva sottratta una riserva di 3,38 GiB necessaria al prefill; restavano circa 0,62 GiB per conservare gli esperti dinamici fra una richiesta e la successiva.

8.5 Tokenizzazione e prompt template

Il testo viene trasformato in token e racchiuso nel formato di conversazione previsto da DeepSeek. Nell'esperimento il prompt breve produceva 14 token elaborati nel prefill. Il template può includere ruoli, delimitatori, istruzioni di sistema e indicatori del thinking mode.

8.6 Prefill

Durante il prefill DS4 attraversa tutti i livelli per tutti i token del prompt. In ogni livello MoE:

1. calcola l'attenzione e aggiorna lo stato dei token;
2. esegue il router per stabilire quali esperti servono;
3. verifica se i pesi degli esperti sono già nella cache GPU;
4. in caso di cache miss, legge gli intervalli necessari dal GGUF;
5. trasferisce o prepara i pesi per i kernel CUDA;
6. esegue gli esperti selezionati e combina le uscite;
7. salva nella KV cache le informazioni riutilizzabili nel decode.

Il log `gpu prefill layer 43/43` indicava il completamento dell'ultimo livello instradato. La velocità di prefill warm misurata è stata circa 0,19 token al secondo.

8.7 Decode: generazione autoregressiva

Dopo il prefill, DS4 predice un token, lo aggiunge alla sequenza e ripete il calcolo per il token successivo. La KV cache evita di ricostruire da zero l’intero passato, ma il routing MoE e l’eventuale caricamento degli esperti devono comunque essere eseguiti a ogni iterazione.

Ciclo autoregressivo

8.8 Sampling e rendering del testo

Il modello produce logits, cioè punteggi per tutti i token del vocabolario. Il sampler applica temperatura e altri filtri, quindi sceglie il token successivo. Nel test `--temp 0` rendeva la selezione sostanzialmente deterministica. Il tokenizer inverso converte l’identificatore in testo e DS4 lo stampa immediatamente.

9. SSD streaming e cache degli esperti

Figura 4 — Il percorso dei pesi dalla memoria di massa al calcolo CUDA.

9.1 Cache hit e cache miss

Una cache hit avviene quando l’esperto richiesto è già in VRAM: il calcolo può partire senza leggere lo SSD. Una cache miss richiede invece una lettura dal GGUF e una preparazione del peso. Se la cache è piccola rispetto alla varietà di esperti richiesta, i blocchi vengono espulsi poco dopo essere stati caricati e poi richiesti nuovamente.

9.2 Thrashing

Il thrashing è il comportamento in cui il sistema dedica gran parte del tempo a spostare dati dentro e fuori dalla cache, invece di eseguire calcolo utile. Con 4 GB di budget, DS4 ha dichiarato 94 esperti memorizzabili, circa 6,75 MiB ciascuno, e ha avvertito che il working set routed era molto più grande.

Ordine di grandezza osservato

9.3 Modalità cold e warm

Con `--ssd-streaming-cold`, DS4 evita o riduce il preload degli esperti frequenti per misurare una partenza fredda. Senza l’opzione, la modalità warm prepara meglio la cache iniziale. Nel nostro test warm il tempo totale è sceso da 3 minuti 57 secondi a 3 minuti 41 secondi e il prefill è passato da 0,17 a 0,19 token/s. La generazione è rimasta a 0,16 token/s.

9.4 Perché 7 GB non hanno migliorato il risultato

Aumentare il budget da 4 a 7 GB non ha prodotto una differenza misurabile nel prompt provato. Questo non dimostra che la cache più grande non serva mai. Può significare che, per quella sequenza e quel percorso di fallback, il collo di bottiglia dominante fosse altrove; oppure che il pattern di esperti non consentisse sufficiente riuso; oppure che altri limiti della VRAM compensassero il vantaggio.

10. Il nostro esperimento reale, passo per passo

10.1 La macchina effettiva

All’inizio pensavamo che il PC avesse 64 GB di RAM. Lo script di inventario ha mostrato invece 32 GB, formati da quattro moduli da 8 GB. Questo controllo è stato importante: l’ottimizzazione deve

partire da misure reali, non da supposizioni.

10.2 Compilazione: il significato di sm_120a

La compilazione `cuda-generic` ha generato codice per `sm_120`, ma `ptxas` ha rifiutato istruzioni MMA block-scaled MXFP4. Il Makefile del commit b030961 prevedeva una regola specifica: compilando con `CUDA_ARCH=sm_120a`, il target diventava `compute_120a / sm_120a` e veniva abilitato il percorso MXF4. La seconda compilazione è riuscita.

10.3 Modello scaricato

Lo script di DS4 ha scaricato il GGUF Q2 imatrix da Hugging Face e creato il collegamento `ds4flash.gguf`. Il file remoto è indicato come 86,7 GB [2]; Linux mostrava 81 GiB perché usa unità binarie per `du -h`.

10.4 Gli errori utili incontrati

10.5 Risultati

Figura 5 — Confronto delle velocità riportate da DS4 nelle prove da 16 token.

La risposta prodotta è stata: «Il software libero è conoscenza condivisa, non merce.» Il risultato più importante non è la velocità, ma la correttezza del percorso completo: modello aperto, router funzionante, esperti letti dallo SSD, calcolo CUDA completato e testo generato con exit status 0.

10.6 Interpretazione delle metriche di I/O

`/usr/bin/time -v` ha mostrato valori molto elevati in “File system inputs”. Questa metrica deriva dalla contabilità `ru_inblock` del sistema e non va trasformata automaticamente in byte senza conoscere il comportamento del kernel e del filesystem. È comunque un segnale coerente con intensa attività di lettura. Per una misura precisa dei MB/s sarebbero serviti strumenti come `iostat`, `pidstat -d`, `perf` o tracing specifico.

11. Come leggere i messaggi di log

12. Cosa abbiamo dimostrato e cosa no

12.1 Cosa abbiamo dimostrato

- La build CUDA di DS4 può essere compilata e avviata su RTX 5060 Ti Blackwell usando il target corretto.
- Il GGUF Q2 imatrix di DeepSeek V4 Flash può essere eseguito con 32 GB di RAM e 16 GB di VRAM mediante SSD streaming.
- La swap da 48 GB era una rete di sicurezza ma non è stata utilizzata nei test riusciti.
- Il percorso di fallback CUDA ha mantenuto la correttezza nonostante la pressione sulla VRAM.
- La modalità warm ha ridotto il tempo di prefill, ma la generazione è rimasta circa 0,16 token/s.
- Aumentare il budget cache da 4 a 7 GB e ridurre il chunk dell’arena non ha prodotto un vantaggio misurabile nel prompt provato.

12.2 Cosa non abbiamo dimostrato

- Non abbiamo misurato con un profiler la percentuale esatta di tempo spesa in I/O, copie host–device e kernel CUDA.
- Non abbiamo confrontato lo stesso sistema con un SSD NVMe.
- Non abbiamo eseguito una suite di prompt, benchmark di qualità o test statistici ripetuti.
- Non abbiamo misurato contesti lunghi, coding agent, server concorrente o speculative decoding.
- Non abbiamo modificato in modo sostanziale il sorgente: abbiamo soprattutto trovato i parametri compatibili e verificato i fallback.
- Non possiamo concludere che 7 GB di cache siano inutili in generale; solo che non hanno migliorato questo caso.

12.3 La lezione tecnica principale

L'esperimento mostra anche il valore di un motore specializzato. DS4 conosce il layout preciso, la quantizzazione asimmetrica e il comportamento del modello; per questo riesce a costruire un percorso che un loader generico potrebbe non offrire o non ottimizzare nello stesso modo.

12.4 Come cambierebbe l'esperienza con hardware diverso

13. Glossario essenziale

Fonti e riferimenti

[1] antirez, "ds4 / DwarfStar README", repository GitHub. Consultato il 6 agosto 2026. Apri la fonte

[2] antirez, "deepseek-v4-gguf", repository Hugging Face; file Q2 imatrix da 86,7 GB. Consultato il 6 agosto 2026. Apri la fonte

[3] Dai et al., "DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models", arXiv:2401.06066, 2024. Apri la fonte

[4] ggml-org, "GGUF specification", documentazione ufficiale del formato. Apri la fonte

[5] NVIDIA, "CUDA Programming Guide — Compute Capabilities", target architecture-specific come compute_120a. Apri la fonte

Materiale dell'esperimento

- Inventario hardware generato con lo script `collect_ds4_info.sh`.
- Repository locale DS4 al commit b030961.
- Log di compilazione CUDA e messaggi `ptxas` del primo tentativo.
- Output runtime dei test cold/warm con cache esperti 4 GB e 7 GB.
- Misure di `/usr/bin/time -v`, `nvidia-smi`, `free` e stato della swap.

Promemoria finale

| Obiettivo del documento Conservare ciò che abbiamo imparato durante l'esperienza: non una raccolta di comandi, ma un modello mentale chiaro di cosa fa un MoE, perché un file da circa 87 GB può essere eseguito su una macchina più piccola, e quali passaggi compie DS4 dall'avvio alla generazione del testo. |

| --- |

| Nota di precisione I numeri “43 livelli × 6 esperti” e le dimensioni delle cache provengono dai log della build usata, commit b030961. Non vanno letti come una descrizione completa e universale di tutte le varianti di DeepSeek V4. Inoltre i test sono pochi e su un solo prompt: indicano il comportamento della nostra prova, non un benchmark scientifico. |

| --- |

| La frase più importante Un MoE non carica “un piccolo modello diverso” a ogni domanda. È un unico grande modello composto da molte sottoreti specializzate; per ogni token, un router sceglie solo alcune sottoreti da attivare. Il calcolo è sparso, ma i pesi complessivi devono restare accessibili. |

| --- |

| Testo visibile | Possibile scomposizione concettuale | Cosa vede il modello |

| --- | --- | --- |

| Il software libero | “Il” · “ software” · “ libero” | una sequenza di identificatori interi |

| conoscenza condivisa | “conosc” · “enza” · “ condivisa” | altri identificatori interi |

| RTX 5060 Ti | “RTX” · “ 5060” · “ Ti” | token tecnici o numerici |

| Fase | Che cosa elabora | Perché serve | Nel nostro test |

| --- | --- | --- | --- |

| Prefill | Tutti i token già presenti nel prompt | Costruisce le rappresentazioni e la KV cache del contesto | 0,17–0,19 token/s |

| Decode / generazione | Un nuovo token alla volta | Predice il token successivo riutilizzando la KV cache | circa 0,16 token/s |

| Analogia I pesi sono il sapere e le abilità apprese dal modello. La KV cache è il taccuino temporaneo della conversazione corrente. Aumentare il contesto rende più grande il taccuino, non il sapere permanente. |

| --- |

| stato_token → attenzione → FFN completo → nuovo stato_token |

| --- |

| y = somma [peso_router(i) × Esperto_i(x)] per i appartenenti ai Top-k |

| --- |

| Attenzione all’analogia Gli esperti non hanno etichette umane certe come “esperto di Python” o “esperto di diritto”. La specializzazione emerge statisticamente. Le etichette sono utili per capire il concetto, ma non descrivono necessariamente ciò che ogni rete ha imparato. |

| --- |

| Una distinzione importante 258 invocazioni non significa necessariamente 258 esperti globalmente diversi. Ogni livello possiede il proprio insieme di esperti. La cache contiene blocchi di pesi associati agli esperti dei diversi livelli; ciò che conta è se i blocchi richiesti sono già presenti o devono essere letti dallo SSD. |

| --- |

| Grandezza | Significato | Nel MoE |

| --- | --- | --- |

| Parametri totali | Tutti i pesi di tutti gli esperti e degli altri componenti | Può essere molto grande |

| Parametri attivi per token | Solo i pesi effettivamente usati per quel token | Resta molto più piccolo |

| Memoria necessaria | Luogo in cui i pesi totali devono essere disponibili | Non scompare: VRAM, RAM, SSD o nodi remoti |

| Calcolo per token | Moltiplicazioni realmente eseguite | Dipende soprattutto dagli esperti selezionati |

| Analogia della biblioteca Una biblioteca può contenere milioni di libri, ma per rispondere a una domanda il bibliotecario consulta solo pochi volumi. Il lavoro di lettura è limitato; lo spazio necessario per conservare l'intera biblioteca resta enorme. Se i libri richiesti sono in un deposito lontano, il tempo di trasporto domina la ricerca. |

| --- |

| Rappresentazione concettuale | Spazio per peso | Vantaggio | Rischio |

| --- | --- | --- | --- |

| FP16 | 16 bit | Buona fedeltà | Molto spazio |

| Q8 | circa 8 bit | Compressione moderata | Piccola perdita |

| Q4 | circa 4 bit | Compressione forte | Perdita maggiore |

| Q2 / IQ2 | circa 2 bit | Compressione estrema | Richiede tecniche e scelte molto attente |

| Sigla nel nome | Interpretazione didattica |

| --- | --- |

| IQ2XXS / Q2_K | Quantizzazione estrema di parti degli esperti routed |

| AProjQ8 | Proiezioni di attenzione conservate a circa 8 bit |

| SExpQ8 | Esperti condivisi a circa 8 bit |

| OutQ8 | Proiezione di uscita a circa 8 bit |

| imatrix | Quantizzazione guidata da una matrice di importanza/calibrazione |

| GGUF non significa compatibilità universale DS4 non è un lettore generico di qualsiasi GGUF. Il progetto dichiara di aspettarsi layout, metadata e mix di quantizzazioni specifici dei file pubblicati per DwarfStar [1]. Il contenitore è standardizzato, ma il significato esatto dei tensori e le ottimizzazioni del motore restano specifici. |

| --- |

| Componente | Responsabilità concettuale |

| --- | --- |

| Loader GGUF | Legge metadata e tensori nel layout atteso |

| Tokenizer e prompt renderer | Converte testo e messaggi nel formato del modello |

| Backend CUDA | Esegue i kernel sulla GPU NVIDIA |

| Memory planner | Decide quanto spazio usare per KV, cache e pesi residenti |

| SSD streaming | Recupera gli esperti routed quando non entrano in memoria |

| Graph executor | Organizza le operazioni per prefill e decode |

| Sampling | Trasforma i logits in una scelta del token successivo |

| CLI / server / agent | Espone il motore all’utente o ad altri programmi |

| ds4: --ssd-streaming is not compatible with multi-GPU placement |

| --- |

| expert budget 4.00 GiB = 3.38 GiB prefill headroom + 0.62 GiB dynamic cache KV + buffers + resident model + expert cache + prefill reserve = 5.03 GiB planned |

| --- |

| prompt → prefill → token 1 → token 2 → token 3 → ... → stop |

| --- |

| 94 esperti in cache 43 livelli × 6 esperti = 258 invocazioni per token → frequenti sostituzioni e nuove letture |

| --- |

| Conclusione prudente I dati mostrano “nessun miglioramento misurabile nel nostro test”. Senza un profiler non possiamo attribuire una percentuale precisa del tempo al disco, alle copie, ai kernel Q8 o ai fallback MMQ. |

| --- |

| Componente | Configurazione rilevata |

| --- | --- |

| Sistema operativo | Linux Mint 22.2, kernel Linux 7.0 |

| CPU | AMD Ryzen 7 5700G, 8 core / 16 thread |

| RAM | 32 GiB visibili, 4 × 8 GB DDR4-2133 |

| GPU | NVIDIA GeForce RTX 5060 Ti, 16.311 MiB VRAM, Blackwell |

| Driver / CUDA | Driver 595.84; toolkit CUDA 13.2 installato per la compilazione |

| Disco modello | Crucial BX500 SATA da 480 GB, filesystem XFS su /localIA |

| Swap di sicurezza | 48 GB sul secondo SSD; mai utilizzata nei test riusciti |

| Tentativo | Risultato | Lezione |

| --- | --- | --- |

| CUDA_ARCH=native → sm_120 | Errore su `mma with block scale` / `mxf4` | Il target generico non esponeva le istruzioni richieste |

| CUDA_ARCH=sm_120a | Build completata | Serviva il target architecture-specific |

| DeepSeek-V4-Flash-IQ2XXS-w2Q2K-AProjQ8-SExpQ8-OutQ8-chat-v2-imatrix-0731.gguf Dimensione locale: circa 81 GiB |

| --- |

| Messaggio | Causa concettuale | Correzione |

| --- | --- | --- |

| SSD streaming incompatibile con placement | `--gpu-vram` attivava un percorso di placement diverso | Rimosso il limite interno e usato `CUDA_VISIBLE_DEVICES=0` |

| 2.00 GiB too small; needs 3.38 GiB | Il budget non copriva la riserva di prefill | Aumentato a 4 GB |

| q8 fp16 cache budget exhausted | Non c’era margine per altre copie FP16 | DS4 ha continuato con kernel Q8 |

| arena alloc failed for moe gate mmq | Impossibile allocare un chunk contiguo da 1792 MiB | Fallback non fatale; provato chunk più piccolo senza guadagno |

| Configurazione | Prefill | Generazione | Tempo totale | RSS massimo | Swap |

| --- | --- | --- | --- | --- | --- |

| 1 token, cold, cache 4 GB | 0,20 t/s | metrica non significativa | 1:13,98 | circa 1,0 GiB | 0 |

| 16 token, cold, cache 4 GB | 0,17 t/s | 0,16 t/s | 3:57,16 | circa 22,3 GiB | 0 |

| 16 token, warm, cache 4 GB | 0,19 t/s | 0,16 t/s | 3:41,44 | circa 22,1 GiB | 0 |

| 16 token, warm, cache 7 GB | 0,19 t/s | 0,16 t/s | 3:41,18 | circa 22,1 GiB | 0 |

| Riga di log | Traduzione concettuale |

| --- | --- |

| CUDA backend initialized … sm_120 | La GPU è stata riconosciuta e il backend può eseguire kernel CUDA. |

| expert budget 4.00 GiB = 3.38 + 0.62 | Il budget è diviso tra riserva temporanea di prefill e cache dinamica. |

| initial cuda model map restricted to token embedding | All’avvio è residente/mappata solo una parte essenziale del modello. |

| CUDA host registration skipped | Il programma non è riuscito a registrare quella memoria host come pinned; continua con un percorso alternativo. |

| loading model tensors 10.67 GiB cached | La cache dei tensori residenti in GPU è cresciuta fino a quel valore. |

| q8 fp16 cache budget exhausted | DS4 evita altre copie FP16 per non saturare la VRAM e usa kernel Q8. |

| model arena alloc failed … 1792 MiB | Non c’è un blocco libero sufficiente per quell’arena; viene usato un fallback. |

| gpu prefill layer 43/43 | Il prefill ha completato l’ultimo livello previsto. |

| prefill: 0.19 t/s, generation: 0.16 t/s | Velocità media delle due fasi. |

| Exit status: 0 | Il processo ha terminato correttamente. |

| Errore fatale o avviso? Un messaggio con “failed” non è sempre fatale. Nel nostro caso l’allocazione MMQ è fallita, ma DS4 disponeva di un fallback e ha prodotto correttamente il testo. La prova decisiva è il comportamento successivo e l’exit status finale. |

| --- |

| Capacità e velocità sono problemi diversi DS4 ha risolto la capacità: il modello non entrava in RAM e VRAM, ma poteva restare sullo SSD. La velocità è rimasta limitata perché un MoE richiede molti accessi a esperti diversi e la memoria di massa è lontana, in latenza e banda, dalla VRAM. |

| --- |

| Upgrade | Effetto probabile | Perché |

| --- | --- | --- |

| SSD NVMe veloce | Riduzione dei tempi di cache miss e prefill | Più banda e minore latenza rispetto al SATA |

| 64–128 GB di RAM | Più pagine del modello e cache disponibili senza pressione | Meno dipendenza dal disco, a seconda dell’implementazione |

| GPU con più VRAM | Più pesi ed esperti residenti, meno fallback | La VRAM evita copie e letture nel ciclo di decode |

| Sistema con memoria unificata ampia | Meno separazione rigida fra RAM e VRAM | Il modello può risiedere in un unico spazio veloce |

| Più GPU | Possibile suddivisione dei pesi | Richiede supporto di tensor/expert parallelism e interconnessioni adeguate |

| Termine | Definizione |

| --- | --- |

| Attenzione | Meccanismo che permette a un token di usare informazioni dai token precedenti. |

| Backend | Implementazione che esegue le operazioni su una piattaforma, per esempio CUDA o Metal. |

| Cache hit / miss | Esperto già presente nella cache / esperto da recuperare altrove. |

| Decode | Fase autoregressiva che genera un token alla volta. |

| Esperto | Rete feed-forward specializzata all’interno di un livello MoE. |

| GGUF | Contenitore binario di tensori e metadata per l’inferenza. |

| Imatrix | Dati di calibrazione che guidano la quantizzazione in base all’importanza. |

| KV cache | Memoria temporanea delle chiavi e dei valori dell’attenzione per il contesto corrente. |

| Logits | Punteggi non normalizzati associati ai possibili token successivi. |

| MoE | Mixture of Experts: architettura che attiva solo un sottoinsieme di esperti per token. |

| Prefill | Elaborazione iniziale dell’intero prompt e costruzione della KV cache. |

| Quantizzazione | Riduzione della precisione numerica dei pesi per risparmiare spazio e memoria. |

| Router | Componente che sceglie gli esperti da attivare. |

| Sampling | Regole che trasformano i logits nella scelta del token successivo. |

| Thrashing | Sostituzione continua dei dati in cache, con molto movimento e poco calcolo utile. |

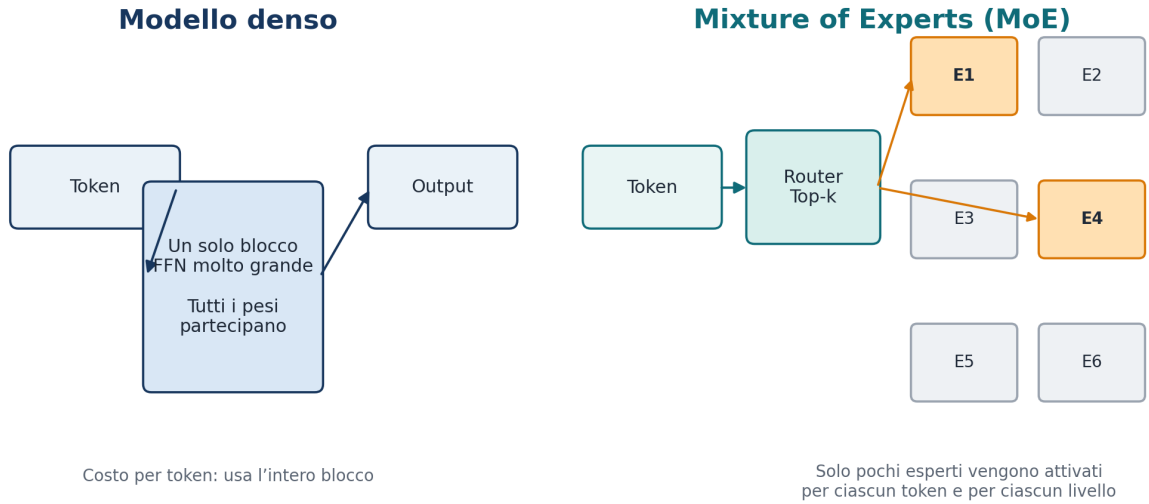
| Token | Unità numerica di testo elaborata dal modello. |

| VRAM | Memoria locale della GPU, molto veloce ma limitata. |

| Ciò che resta dell’esperienza Abbiamo visto un sistema completo adattarsi a una macchina troppo piccola per il modello: la quantizzazione ha ridotto i pesi, il MoE ha limitato il calcolo attivo, GGUF ha reso il modello distribuibile, DS4 ha pianificato la memoria e lo SSD streaming ha fornito gli esperti mancanti. Il prezzo è stato il movimento continuo dei dati, che ha trasformato un problema impossibile in un problema possibile ma lento. |

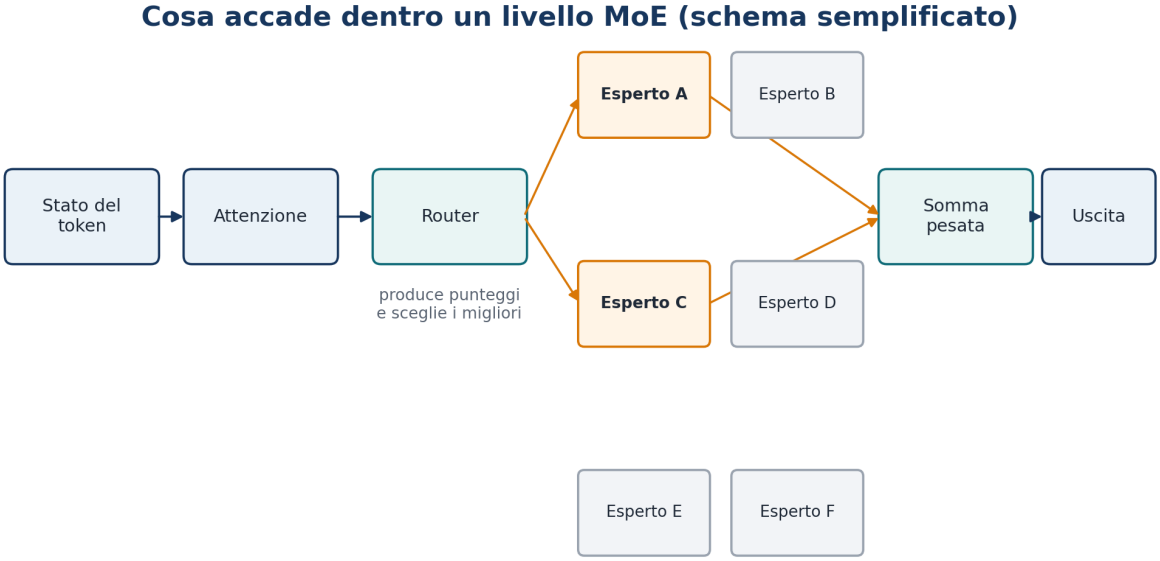
| --- |

Immagini importate



Il vantaggio del MoE è computazionale; tutti i pesi devono comunque essere disponibili da qualche parte.

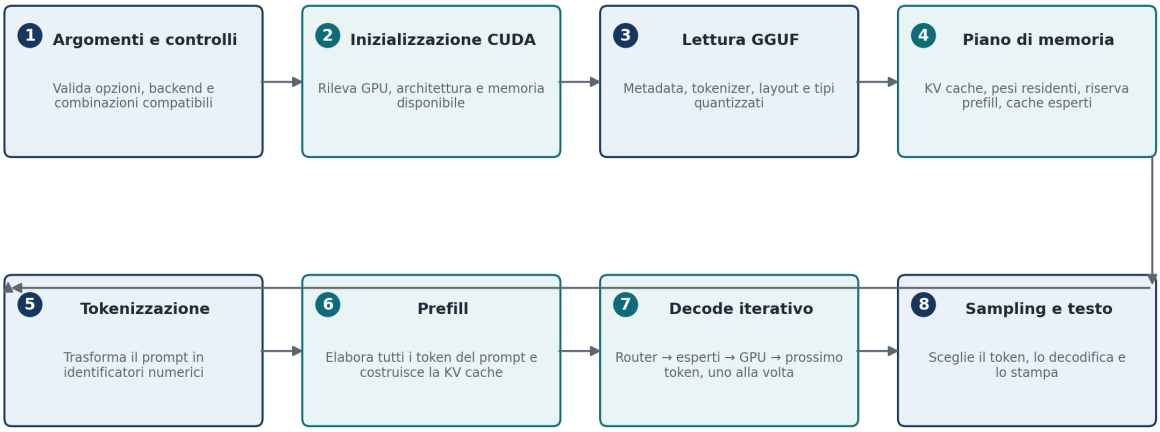
Immagine importata



Gli esperti sono reti feed-forward. L’attenzione non scompare: resta un calcolo comune a tutti i token.

Immagine importata

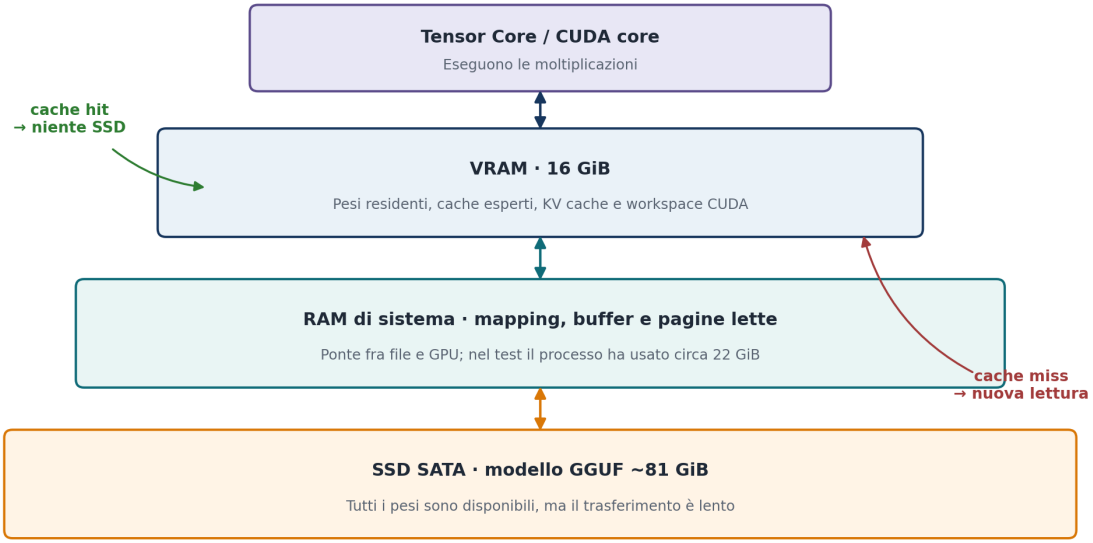
DwarfStar / DS4: macro-pipeline di un’inferenza locale



Il ciclo 7–8 si ripete fino al limite di token o a un token di arresto.

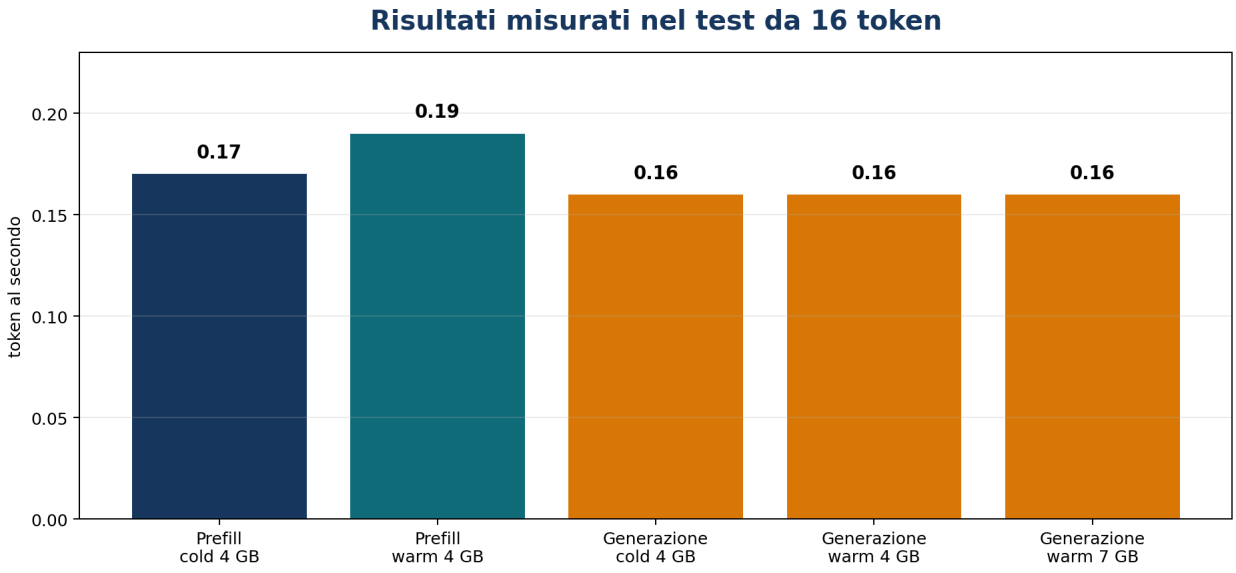
Immagine importata

Gerarchia di memoria durante lo streaming degli esperti



Quando la cache è troppo piccola, gli esperti vengono espulsi e riletti continuamente: è il “thrashing”.

Immagine importata



Prompt breve, contesto 512, RTX 5060 Ti 16 GB, SSD SATA; un solo esperimento per configurazione.

Immagine importata