

# Efficient Inference

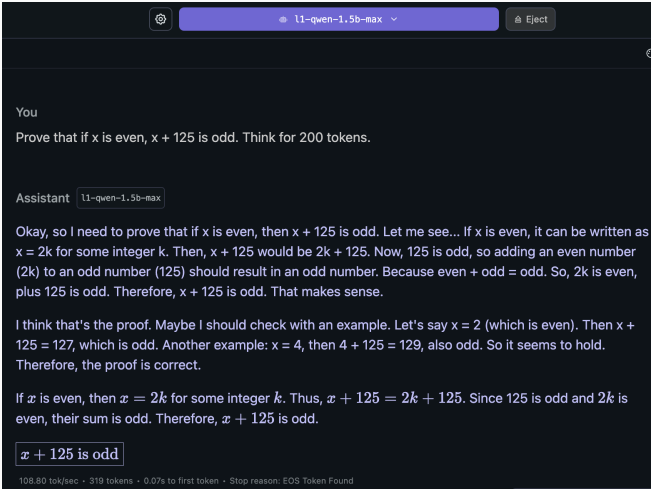
---

Sean Welleck | CMU Advanced NLP

March 27, 2025

Most slides by Hailey Schoelkopf from the NeurIPS 2024 LLM Inference Tutorial

# Efficient inference



The screenshot shows the LM Studio interface with the following content:

Model: l1-qwen-1.5b-max

You: Prove that if  $x$  is even,  $x + 125$  is odd. Think for 200 tokens.

Assistant (l1-qwen-1.5b-max):

Okay, so I need to prove that if  $x$  is even, then  $x + 125$  is odd. Let me see... If  $x$  is even, it can be written as  $x = 2k$  for some integer  $k$ . Then,  $x + 125$  would be  $2k + 125$ . Now, 125 is odd, so adding an even number ( $2k$ ) to an odd number (125) should result in an odd number. Because even + odd = odd. So,  $2k$  is even, plus 125 is odd. Therefore,  $x + 125$  is odd. That makes sense.

I think that's the proof. Maybe I should check with an example. Let's say  $x = 2$  (which is even). Then  $x + 125 = 127$ , which is odd. Another example:  $x = 4$ , then  $4 + 125 = 129$ , also odd. So it seems to hold. Therefore, the proof is correct.

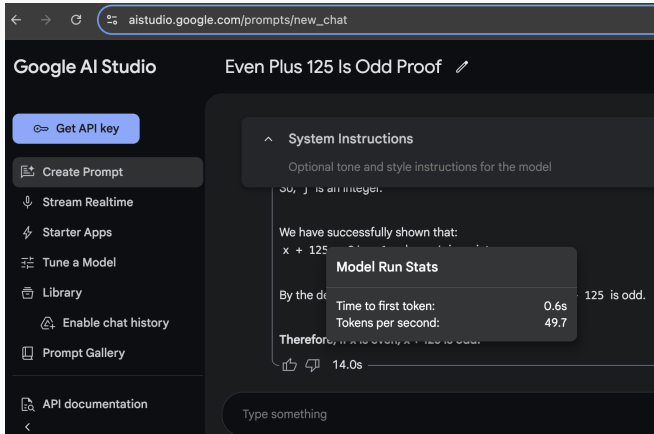
If  $x$  is even, then  $x = 2k$  for some integer  $k$ . Thus,  $x + 125 = 2k + 125$ . Since 125 is odd and  $2k$  is even, their sum is odd. Therefore,  $x + 125$  is odd.

**$x + 125$  is odd**

108.80 tok/sec · 319 tokens · 0.07s to first token · Stop reason: EOS Token Found

Using the L1 model on a Macbook with LM Studio

# Efficient inference



The screenshot shows the Google AI Studio interface. The browser address bar displays `aistudio.google.com/prompts/new_chat`. The main heading is "Even Plus 125 Is Odd Proof". On the left sidebar, there are several options: "Get API key", "Create Prompt", "Stream Realtime", "Starter Apps", "Tune a Model", "Library", "Enable chat history", "Prompt Gallery", and "API documentation".

The chat content includes:

- System Instructions:** Optional tone and style instructions for the model.
- Text: "So,  $j$  is an integer."
- Text: "We have successfully shown that:"
- Equation:  $x + 125$
- Text: "By the de... 125 is odd."
- Text: "Therefore, if  $x$  is even,  $x + 125$  is odd."

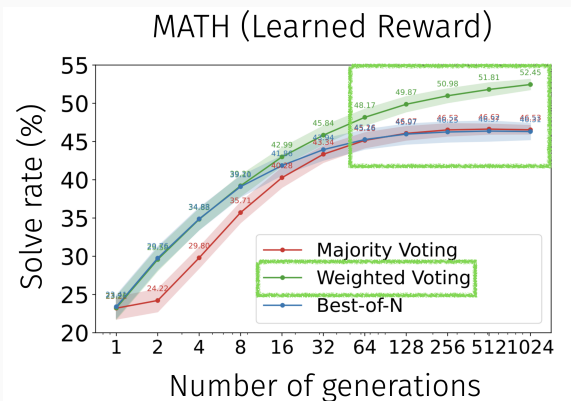
A "Model Run Stats" overlay is positioned over the chat, displaying:

Model Run Stats	
Time to first token:	0.6s
Tokens per second:	49.7

At the bottom of the chat, there is a thumbs up icon, a thumbs down icon, and a timer showing "14.0s". A text input field at the very bottom contains the placeholder text "Type something".

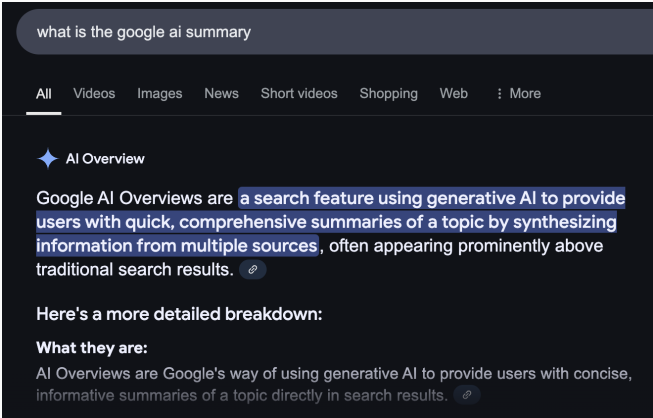
Using a LLM API

# Efficient inference



Running experiments involving a lot of inference

# Efficient inference



what is the google ai summary

All Videos Images News Short videos Shopping Web : More

✦ AI Overview

Google AI Overviews are a search feature using generative AI to provide users with quick, comprehensive summaries of a topic by synthesizing information from multiple sources, often appearing prominently above traditional search results. [🔗](#)

**Here's a more detailed breakdown:**

**What they are:**

AI Overviews are Google's way of using generative AI to provide users with concise, informative summaries of a topic directly in search results. [🔗](#)

Serving many customer requests

## Scope:

- Basics of efficient inference
- How can we make inference strategies faster?
- Which strategies are most efficient?

## Recap: Inference

- Generate a single sequence,  $y \sim p_{\theta}(y|x)$ 
  - $y_1 \sim p_{\theta}(\cdot|x)$
  - $y_2 \sim p_{\theta}(\cdot|y_1, x)$
  - $y_3 \sim p_{\theta}(\cdot|y_1, y_2, x)$
  - ...

## Recap: Inference

- Generate a single sequence,  $y \sim p_{\theta}(y|x)$ 
  - $y_1 \sim p_{\theta}(\cdot|x)$
  - $y_2 \sim p_{\theta}(\cdot|y_1, x)$
  - $y_3 \sim p_{\theta}(\cdot|y_1, y_2, x)$
  - ...
- Inference strategies that involve generating multiple sequences
  - Best-of- $N$ , voting
  - Tree search
  - Refinement
  - ...

See decoding lecture (#7) and advanced inference (#21) lecture



How do we measure “efficiency”?

- **Latency**
  - *How long does a user wait for a response?*

How do we measure “efficiency”?

- **Latency**
  - *How long does a user wait for a response?*
    - Startup overhead
    - Slow token generation

How do we measure “efficiency”?

- **Latency**
  - *How long does a user wait for a response?*
    - Startup overhead
    - Slow token generation
  - Time to first token, time per request

How do we measure “efficiency”?

- **Latency**

- *How long does a user wait for a response?*
  - Startup overhead
  - Slow token generation
- Time to first token, time per request

- **Throughput**

- *How many requests are completed per second?*

How do we measure “efficiency”?

- **Latency**

- *How long does a user wait for a response?*
  - Startup overhead
  - Slow token generation
- Time to first token, time per request

- **Throughput**

- *How many requests are completed per second?*
  - How well the GPU is utilized
  - Parallelism (e.g., batching, multiple devices)

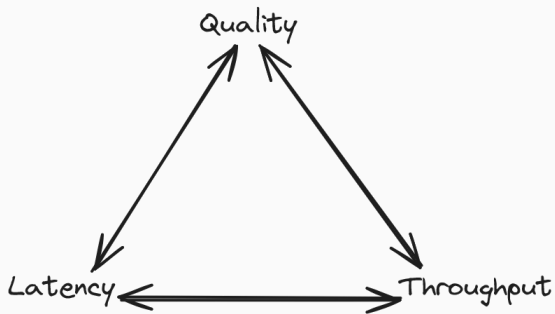
How do we measure “efficiency”?

- **Latency**

- *How long does a user wait for a response?*
  - Startup overhead
  - Slow token generation
- Time to first token, time per request

- **Throughput**

- *How many requests are completed per second?*
  - How well the GPU is utilized
  - Parallelism (e.g., batching, multiple devices)
- Tokens per second, requests per second



Latency, Throughput, and Quality often trade off at a given budget.

Example: queueing (waiting to be batched) can hurt latency

Key problem: efficiently execute operations on the given hardware

- Re-use computation
- Take advantage of unique hardware advantages
- Minimize bottlenecks



How do ML accelerator designs impact generation efficiency?

How do ML accelerator designs impact generation efficiency?

- How much data can we keep on-device?
  - **VRAM (GB)**: e.g., 80GB

How do ML accelerator designs impact generation efficiency?

- How much data can we keep on-device?
  - **VRAM (GB)**: e.g., 80GB
- How many operations/second can the device perform?
  - **FLOP/s**: e.g., 1,979 teraFLOP/s

How do ML accelerator designs impact generation efficiency?

- How much data can we keep on-device?
  - **VRAM (GB):** e.g., 80GB
- How many operations/second can the device perform?
  - **FLOP/s:** e.g., 1,979 teraFLOP/s
- How long does it take to send operands from GPU memory (HBM) to the processor?
  - **Memory Bandwidth (GB/s):** e.g. 3.35 TB/s

- Loading inputs (activations) from memory

- Loading inputs (activations) from memory
  - **Memory Bandwidth**

- Loading inputs (activations) from memory
  - **Memory Bandwidth**
- Loading *weights* from memory

- Loading inputs (activations) from memory
  - **Memory Bandwidth**
- Loading *weights* from memory
  - **Memory Bandwidth**



- Loading inputs (activations) from memory
  - **Memory Bandwidth**
- Loading *weights* from memory
  - **Memory Bandwidth**
- Performing computation

- Loading inputs (activations) from memory
  - **Memory Bandwidth**
- Loading *weights* from memory
  - **Memory Bandwidth**
- Performing computation
  - **FLOP/s**

- Loading inputs (activations) from memory
  - **Memory Bandwidth**
- Loading *weights* from memory
  - **Memory Bandwidth**
- Performing computation
  - **FLOP/s**
- Communicating across devices

- Loading inputs (activations) from memory
  - **Memory Bandwidth**
- Loading *weights* from memory
  - **Memory Bandwidth**
- Performing computation
  - **FLOP/s**
- Communicating across devices
  - **Communication Speeds (GB/s)**

- Loading inputs (activations) from memory
  - **Memory Bandwidth**
- Loading *weights* from memory
  - **Memory Bandwidth**
- Performing computation
  - **FLOP/s**
- Communicating across devices
  - **Communication Speeds (GB/s)**
- ...

Time per operation can be modeled as<sup>1</sup>:

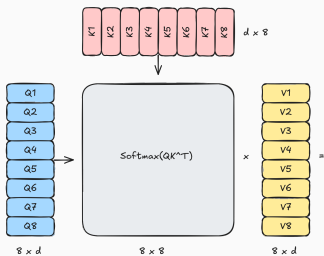
$$\text{Time} = \max \left( \frac{\text{Operation FLOP}}{\text{Device FLOP/s}}, \frac{\text{Data Transferred (GB)}}{\text{Memory Bandwidth (GB/s)}} \right)$$

Operations are either “compute-bound” or “memory-bound”

---

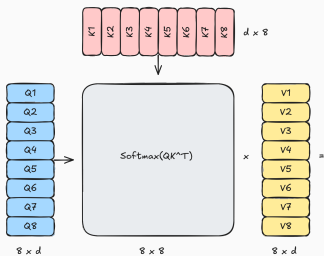
<sup>1</sup>[He, 2022]

# Basics | stages and KV cache

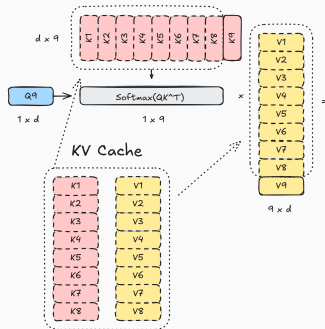


**Prefill Stage:** process prompt all at once. Keys and values retained and initialize the “KV Cache”.

# Basics | stages and KV cache



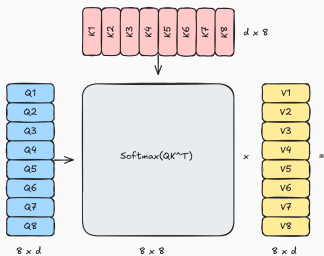
**Prefill Stage:** process prompt all at once. Keys and values retained and initialize the “KV Cache”.



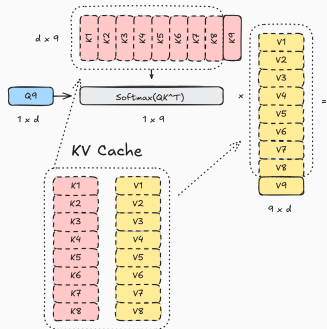
**Decode Stage:** use cached KV values to compute attention for current timestep. Append new K, V to KV cache



# Basics | stages and KV cache



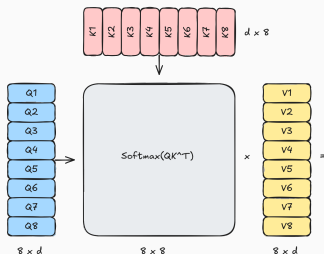
**Prefill Stage:** process prompt all at once. Keys and values retained and initialize the “KV Cache”.



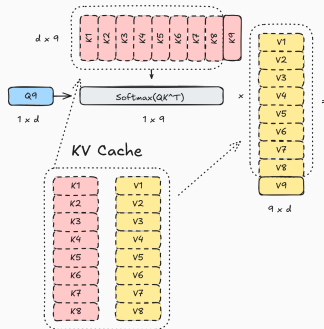
**Decode Stage:** use cached KV values to compute attention for current timestep. Append new K, V to KV cache

$$\text{Size} = (\text{batch} \cdot \text{n\_ctx}) \cdot (2 \cdot \text{n\_layer} \cdot \text{n\_heads} \cdot \text{head\_dim}) \cdot (\text{n\_bytes})$$

# Basics | stages and KV cache



**Prefill Stage:** process prompt all at once. Keys and values retained and initialize the “KV Cache”.



**Decode Stage:** use cached KV values to compute attention for current timestep. Append new K, V to KV cache

$$\text{Size} = (\text{batch} \cdot \text{n\_ctx}) \cdot (2 \cdot \text{n\_layer} \cdot \text{n\_heads} \cdot \text{head\_dim}) \cdot (\text{n\_bytes})$$

KV Cache storage can exceed the weights' storage size, especially for long-context and large batches!

```
class Attention(nn.Module):
    def forward(
        bsz, seqlen, _ = x.shape
        xq, xk, xv = self.wq(x), self.wk(x), self.wv(x)

        xq = xq.view(bsz, seqlen, self.n_local_heads, self.head_dim)
        xk = xk.view(bsz, seqlen, self.n_local_kv_heads, self.head_dim)
        xv = xv.view(bsz, seqlen, self.n_local_kv_heads, self.head_dim)

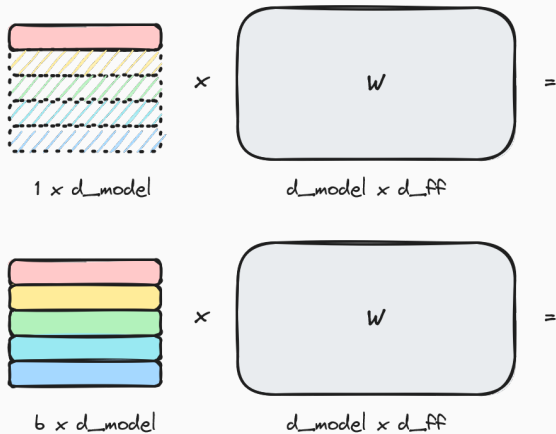
        xq, xk = apply_rotary_emb(xq, xk, freqs_cis=freqs_cis)

        self.cache_k = self.cache_k.to(xq)
        self.cache_v = self.cache_v.to(xq)

        self.cache_k[:,bsz, start_pos : start_pos + seqlen] = xk
        self.cache_v[:,bsz, start_pos : start_pos + seqlen] = xv
```

KV cache in Llama 3 inference code

# Basics | batching



Inputs to a model can be batched together and computed simultaneously.

### LLM APIs, batch inference, RL

- Combine multiple user prompts into a batch
- Process many inputs at once for evaluation or training

# Basics | when does batching matter?

## LLM APIs, batch inference, RL

- Combine multiple user prompts into a batch
- Process many inputs at once for evaluation or training

**Throughput** is a main concern

- Tokens per second across all sequences

# Basics | when does batching matter?

## LLM APIs, batch inference, RL

- Combine multiple user prompts into a batch
- Process many inputs at once for evaluation or training

## Single-user, interactive use

- Single-user chat
- Streaming outputs
  - Batching can increase wait time before first token

**Throughput** is a main concern

- Tokens per second across all sequences

# Basics | when does batching matter?

## LLM APIs, batch inference, RL

- Combine multiple user prompts into a batch
- Process many inputs at once for evaluation or training

**Throughput** is a main concern

- Tokens per second across all sequences

## Single-user, interactive use

- Single-user chat
- Streaming outputs
  - Batching can increase wait time before first token

**Latency** is a main concern

- Time to first token
- Time for a single sequence



- Latency, throughput, quality
- Hardware constraints: storage, communication costs, computation costs
- KV cache
- Batching

## Efficient inference

- Generating a single token
- Generating a full sequence
- Generating multiple sequences

## Efficient inference

- **Generating a single token**
- Generating a full sequence
- Generating multiple sequences

For a single decoding step, how do we work around hardware constraints?

For a single decoding step, how do we work around hardware constraints?

- **Memory Bandwidth** ↓: shrink the data we need to move

For a single decoding step, how do we work around hardware constraints?

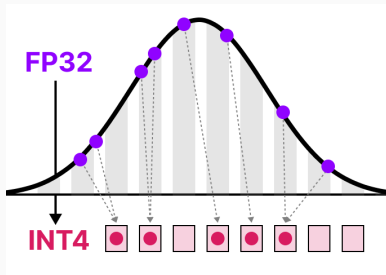
- **Memory Bandwidth** ↓: shrink the data we need to move
- **FLOP/s** ↑: better utilize the hardware

For a single decoding step, how do we work around hardware constraints?

- **Memory Bandwidth** ↓: shrink the data we need to move
- **FLOP/s** ↑: better utilize the hardware
- **FLOP** ↓: use fewer operations

Memory Bandwidth ↓: reduce data transferred

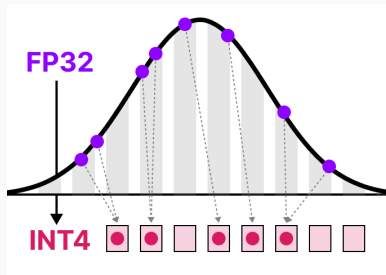
- Quantize weights or activations





Memory Bandwidth ↓: reduce data transferred

- Quantize weights or activations



- Compress or distill model

(bytes per parameter) · (total parameters)

## Memory Bandwidth ↓: reduce data transferred

Benchmarks run on an 8xA100-80GB, power limited to 330W with a hybrid cube mesh topology. Note that all benchmarks are run at *batch size=1*, making the reported tokens/s numbers equivalent to "tokens/s/user". In addition, they are run with a very small prompt length (just 5 tokens).

Model	Technique	Tokens/Second	Memory Bandwidth (GB/s)
Llama-2-7B	Base	104.9	1397.31
	8-bit	155.58	1069.20
	4-bit (G=32)	196.80	862.69
Llama-2-70B	Base	OOM	
	8-bit	19.13	1322.58
	4-bit (G=32)	25.25	1097.66

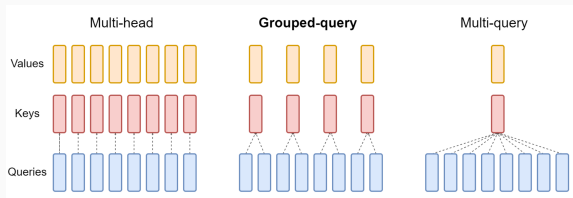
gpt-fast benchmarks

# Efficiency | single-token

Memory Bandwidth ↓: reduce data transferred

The KV cache is a key source of memory bandwidth overhead

$$(\text{batch} \cdot n_{\text{ctx}}) \cdot (2 \cdot n_{\text{layer}} \cdot n_{\text{heads}} \cdot \text{head\_dim}) \cdot (n_{\text{bytes}})$$



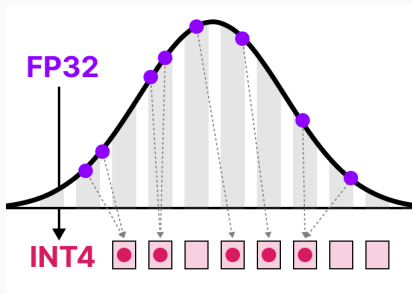
Architectural tweaks such as Grouped-Query Attention [Ainslie et al., 2023] reduce the number of Key + Value attention heads to shrink the required KV Cache size

# Efficiency | single-token

Memory Bandwidth ↓: reduce data transferred

The KV cache is a key source of memory bandwidth overhead

$$(\text{batch} \cdot \text{n\_ctx}) \cdot (2 \cdot \text{n\_layer} \cdot \text{n\_heads} \cdot \text{head\_dim}) \cdot (\text{n\_bytes})$$



As with model weights, elements of the KV cache can be *quantized* to reduce memory overheads

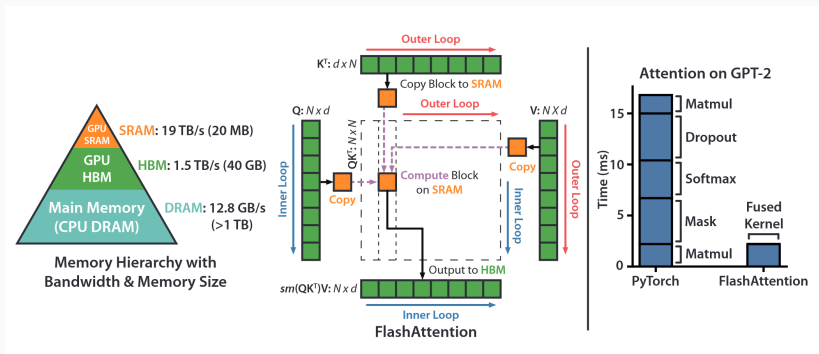
FLOP/s  $\uparrow$ : improve hardware utilization

(FLOP per second)  $\cdot$  (total operation FLOP)

# Efficiency | single-token

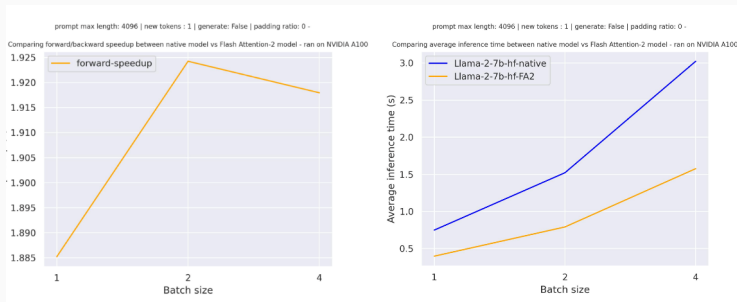
FLOP/s  $\uparrow$ : improve hardware utilization

(FLOP per second)  $\cdot$  (total operation FLOP)



Flash Attention [Dao et al., 2022] performs the same operations, but optimizes the implementation to achieve far greater speed

# Efficiency | single-token



Flash attention 2 benchmarking

([https://huggingface.co/docs/transformers/v4.34.0/perf\\_infer\\_gpu\\_one](https://huggingface.co/docs/transformers/v4.34.0/perf_infer_gpu_one))

FLOP ↓: reduce operations required

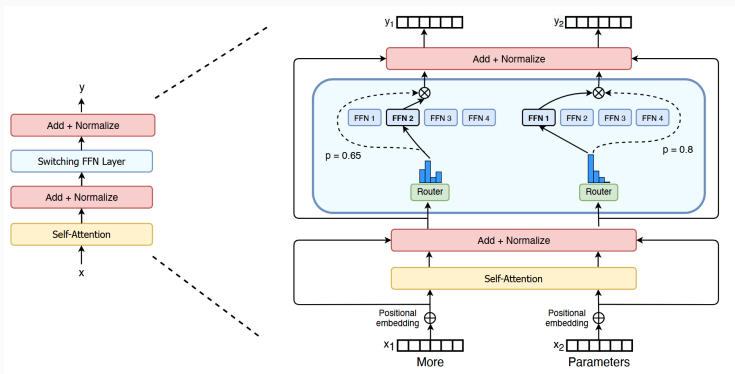
$$(\text{FLOP per second}) \cdot (\text{total operation FLOP})$$



# Efficiency | single-token

FLOP ↓: reduce operations required

(FLOP per second) · (total operation FLOP)



Mixture-of-Experts models use fewer FLOP per token than equi-parameter dense models [Fedus et al., 2022]

## Efficient inference

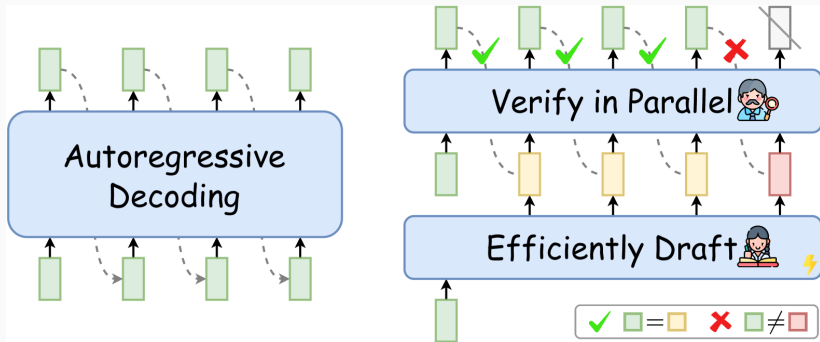
- Generating a single token
- **Generating a full sequence**
- Generating multiple sequences

Generation of long outputs is bottlenecked by sequential next-token prediction. But not all tokens are created equal!

... The **cow** jumped over the moon . <EOS>

How can we spend less time on “easier” tokens?

# Efficiency | single-generation | speculative decoding

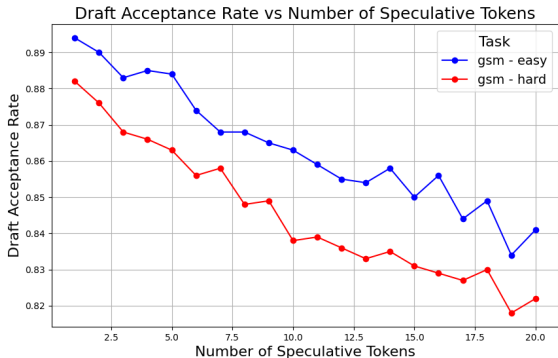


**Speculative decoding** uses a smaller **draft** model to produce “guesses” for the next N tokens cheaply, which are then “accepted” or “rejected” in parallel by the main model [Xia et al., 2024]

## Efficiency | single-generation | speculative decoding

```
1 def speculative_decode(tgt_m, drf_m, tok, inp: torch.Tensor, max_tok:
2   int, n_spec: int = 5, t: float = 1.0):
3   gen = inp; max_len = inp.shape[1] + max_tok
4   while gen.shape[1] < max_len:
5       tok_left = max_len - gen.shape[1]
6       spec_size = min(n_spec, tok_left - 1)
7       if spec_size > 0:
8           spec_id, spec_lprob = generate(drf_m, tok, gen, spec_size, t)
9           tgt_lprob = tgt_m(spec_id) # forwarding tgt model
10          rejs = compute_ll_rejs(tgt_lprob, spec_lprob)
11          if len(rejs) > 0:
12              accepted = spec_id[:, :rejs[0]]
13              adj_probs = compute_adjusted_dist(tgt_lprob, spec_lprob)
14              next_tok = Categorical(adj_probs)
15          else:
16              accepted = spec_id
17              next_tok = Categorical(tgt_lprob.exp())
18          gen = torch.cat([gen, accepted, next_tok])
```

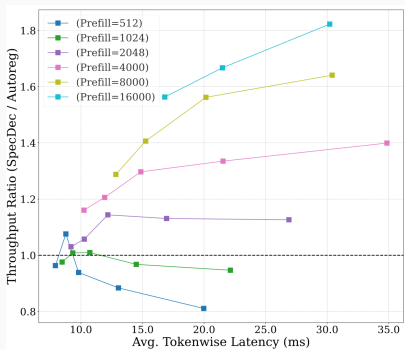
```
1 def compute_ll_rejs(tgt_lprob: torch.Tensor, spec_lprob: torch.Tensor,
2   spec_tok_id: torch.Tensor) -> torch.Tensor:
3   llrs = tgt_lprob[spec_tok_id] - spec_lprob[spec_tok_id]
4   uniform_lprobs = torch.log(torch.rand_like(llrs))
5   rej_idx = torch.nonzero((llrs <= uniform_lprobs))
6   return rej_idx
7
8 def compute_adjusted_dist(tgt_lprob: torch.Tensor, spec_lprob:
9   torch.Tensor, rej_idx: torch.Tensor) -> torch.Tensor:
10  adj_dist = torch.clamp(
11    torch.exp(tgt_lprob[rej_idx]) - torch.exp(spec_lprob[rej_idx]),
12    min=0
13  )
14  adj_dist = torch.div(adj_dist, adj_dist.sum())
15  return adj_dist
```



Draft model acceptance rates are distribution-dependent<sup>2</sup>

<sup>2</sup><https://github.com/cmu-l3/neurips2024-inference-tutorial-code/tree/main/section3>

# Efficiency | single-generation | speculative decoding



Speculative decoding can harm throughput at low context but improves both throughput and latency at long context lengths [Chen et al., 2024]



## Efficient inference

- Generating a single token
- Generating a full sequence
- **Generating multiple sequences**
  - Batched generation settings
  - Meta-generation strategies: best-of- $N$ , tree search, ...

Key idea:

- Leverage *redundancy* across generations to *re-use* computation

## Shared Prefix

You are ChatGPT, a large language model trained by OpenAI, based on the GPT-4 architecture.

Knowledge cutoff: 2023-04

Current date: 2023-11-16

Image input capabilities: Enabled

When you send a message containing Python code to python, it will be executed in a stateful Jupyter notebook environment. Python will respond...

## Unique Suffixes

Hi, can you write a...

Tell me a funny...

Who is Alan Turing?

Debug this Python...

Ignore all previous...

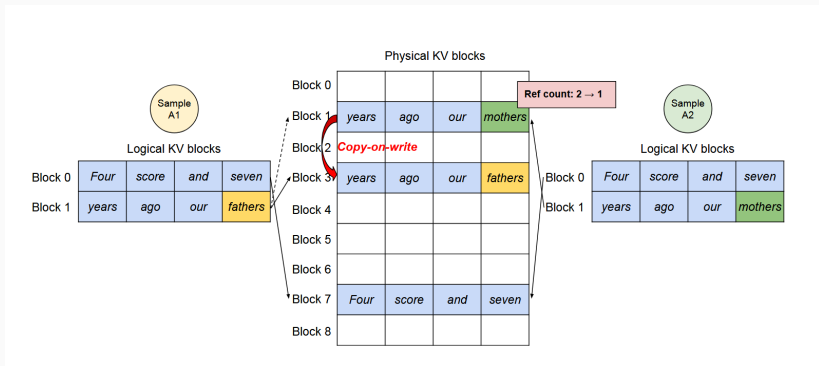
## Shared Prefix Setting

Common deployment and parallel generation scenarios have redundant **shared prefix** content in prompts<sup>3</sup>

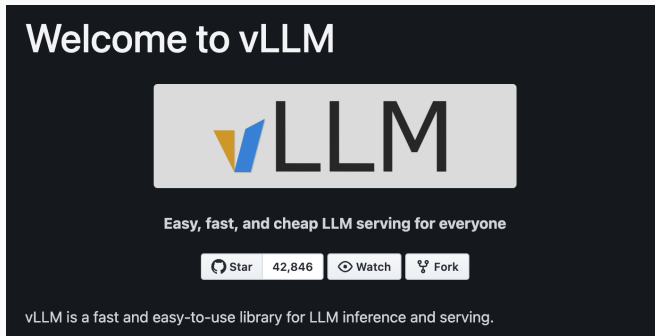
---

<sup>3</sup>Figure from [Juravsky et al., 2024]

# Efficiency | meta-generators | KV Cache reuse

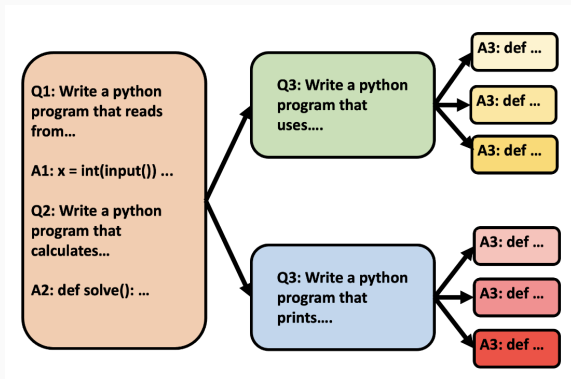


PagedAttention [Kwon et al., 2023] prevents redundant storage costs by mapping KV cache blocks to physical “pages” of VRAM



PagedAttention VLLM [Kwon et al., 2023]: fast inference library, originally built for PagedAttention

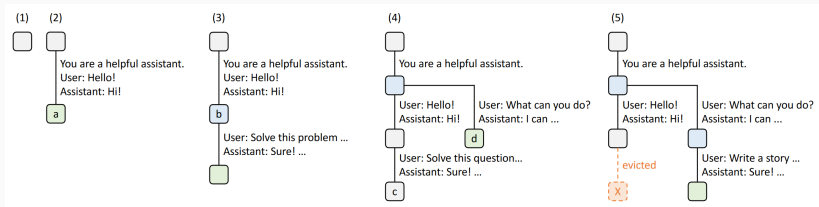
KV Cache reuse is not limited to single-level shared prefixes!



Multiple levels of prefix sharing can arise frequently: for example, combining a long few-shot prompt with Best-of-N generation<sup>4</sup>

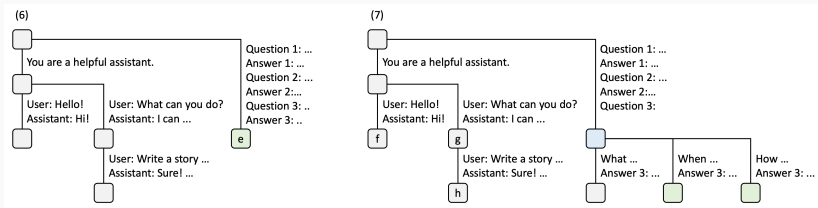
<sup>4</sup>Figure from [Juravsky et al., 2024]

# Efficiency | meta-generators | KV Cache reuse



RadixAttention enables complex prefix sharing patterns [Zheng et al., 2024], evicting least-recently-used KV cache blocks from memory when needed

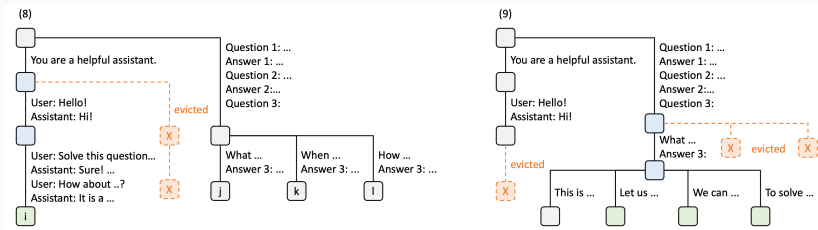
# Efficiency | meta-generators | KV Cache reuse



RadixAttention enables complex prefix sharing patterns [Zheng et al., 2024], evicting least-recently-used KV cache blocks from memory when needed



# Efficiency | meta-generators | KV Cache reuse



RadixAttention enables complex prefix sharing patterns [Zheng et al., 2024], evicting least-recently-used KV cache blocks from memory when needed

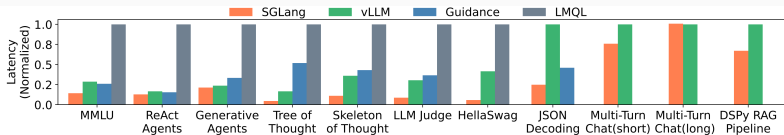


Figure 6: Normalized latency on Llama-7B models. Lower is better.

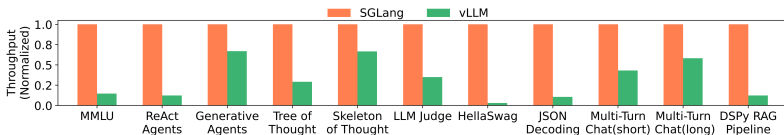
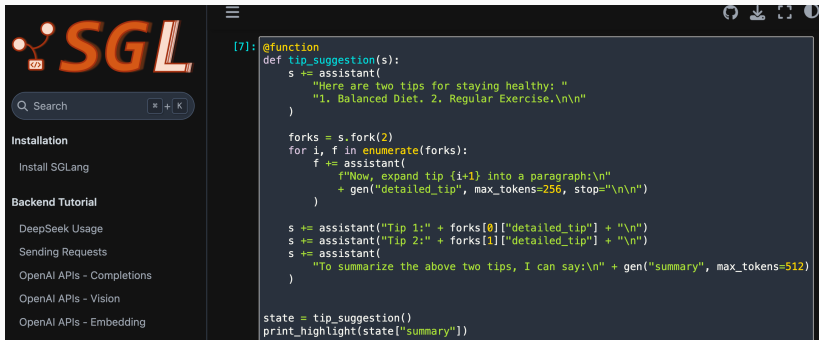


Figure 7: Normalized throughput on Mixtral-8x7B models with tensor parallelism. Higher is better.

SGLang [Zheng et al., 2024] latency and throughput comparison.



The screenshot shows the SGLang web interface. On the left is a navigation menu with the SGL logo and a search bar. The main area is a code editor displaying a Python function named `tip_suggestion(s)`. The function uses `assistant` to generate two tips, then forks two parallel processes to expand each tip into a paragraph. Finally, it uses `assistant` again to generate a summary of the two tips. The code is as follows:

```
[7]: @function
def tip_suggestion(s):
    s += assistant(
        "Here are two tips for staying healthy: "
        "1. Balanced Diet. 2. Regular Exercise.\n\n"
    )

    forks = s.fork(2)
    for i, f in enumerate(forks):
        f += assistant(
            f"Now, expand tip {i+1} into a paragraph:\n"
            + gen("detailed_tip", max_tokens=256, stop="\n\n")
        )

    s += assistant("Tip 1:" + forks[0]["detailed_tip"] + "\n")
    s += assistant("Tip 2:" + forks[1]["detailed_tip"] + "\n")
    s += assistant(
        "To summarize the above two tips, I can say:\n" + gen("summary", max_tokens=512)
    )

state = tip_suggestion()
print_highlight(state["summary"])
```

SGLang [Zheng et al., 2024]

*Which meta-generators are most efficient?*

- **Parallelizable**: trajectories can be run in parallel; not sequentially bottlenecked
- **Prefix-shareable**: long inputs are presented as identical shared prefix content, whose KV Caches can be reused across many model calls

**Token budget** is not the only indicator of meta-generator efficiency!

- Basics: latency, throughput, bottlenecks, KV cache and batching
- Speeding up:
  - Single-token generation
  - Full sequence generation
  - Multi-sequence generation

**Token budget** is not the only indicator of inference cost!

- Using a LLM on your laptop
  - LM Studio: <https://lmstudio.ai/>
  - Ollama: <https://ollama.com/>

- Using a LLM on your laptop
  - LM Studio: <https://lmstudio.ai/>
  - Ollama: <https://ollama.com/>
- Writing code to call LLMs on many types of devices (phone, etc.)
  - MLC LLM: <https://llm.mlc.ai/>

- Using a LLM on your laptop
  - LM Studio: <https://lmstudio.ai/>
  - Ollama: <https://ollama.com/>
- Writing code to call LLMs on many types of devices (phone, etc.)
  - MLC LLM: <https://llm.mlc.ai/>
- Writing code to call LLMs on GPU(s)
  - VLLM: <https://docs.vllm.ai/>
  - SGLang: <https://docs.sglang.ai/>





Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebrón, F., and Sanghai, S. (2023).

**Gqa: Training generalized multi-query transformer models from multi-head checkpoints.**



Chen, J., Tiwari, V., Sadhukhan, R., Chen, Z., Shi, J., Yen, I. E.-H., and Chen, B. (2024).





**Magicdec: Breaking the latency-throughput tradeoff for long context generation with speculative decoding.**




Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and R'e, C. (2022).


**Flashattention: Fast and memory-efficient exact attention with io-awareness.**

*ArXiv preprint, abs/2205.14135.*

-  Fedus, W., Zoph, B., and Shazeer, N. (2022).  
**Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity.**
-  He, H. (2022).  
**Making deep learning go brrrr from first principles.**
-  Juravsky, J., Brown, B., Ehrlich, R., Fu, D. Y., Ré, C., and Mirhoseini, A. (2024).  
**Hydragen: High-throughput llm inference with shared prefixes.**
-  Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. (2023).  
**Efficient memory management for large language model serving with pagedattention.**

 Xia, H., Yang, Z., Dong, Q., Wang, P., Li, Y., Ge, T., Liu, T., Li, W., and Sui, Z. (2024).

**Unlocking efficiency in large language model inference: A comprehensive survey of speculative decoding.**

 Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., and Sheng, Y. (2024).

**Sglang: Efficient execution of structured language model programs.**