

Large Language Model Applications

Deployment
Discussion

Project last stretch

- All students:
 - Submit a 10-14 minute video presentation on your project. **These are due Monday, April 20 at 11:59 PM, no exceptions.**
 - Write a project report. **These are due Thursday, April 30 at 11:59 PM. The usual late penalty applies.**
 - Complete participation activities in class on Tuesday, April 21 and Thursday, April 23. Rather than PollEverywhere, participation will involve you submitting feedback for 2+ other projects one each of the data.
- Applied track: demo session on Tuesday, April 21. **This will be in this classroom.**
 - Present a live demo to your classmates and instructors.
- Research track: spotlight talks on Thursday, April 23
 - ~6 projects will be selected to give live talks to the class.

Deployment

So you've finetuned a model, and want to build an app that makes inference calls to it. What do you do?

A naive solution

```
MODEL_ID = "meta-llama/Meta-Llama-3-8B-Instruct"

# Global variables to hold the loaded model and tokenizer
tokenizer = None
model = None

# Load the model exactly once when the server starts up
@app.on_event("startup")
async def load_model():
    global tokenizer, model
    print(f"Loading {MODEL_ID}...")

    tokenizer = AutoTokenizer.from_pretrained(MODEL_ID)

    # device_map="auto" automatically loads the model onto available GPUs
    # torch_dtype=torch.bfloat16 reduces memory usage
    model = AutoModelForCausalLM.from_pretrained(
        MODEL_ID,
        torch_dtype=torch.bfloat16,
        device_map="auto"
    )
    print("Model loaded successfully!")

# Define the expected JSON payload format
class PromptRequest(BaseModel):
    prompt: str
    max_new_tokens: int = 128
```

```
# Define the endpoint to handle incoming requests
@app.post("/generate")
async def generate(request: PromptRequest):
    if model is None or tokenizer is None:
        raise HTTPException(status_code=503, detail="Model is still loading.")

    # Tokenize the input and move it to the same device as the model (e.g., GPU)
    inputs = tokenizer(request.prompt, return_tensors="pt").to(model.device)

    # Generate the output
    with torch.no_grad():
        outputs = model.generate(
            **inputs,
            max_new_tokens=request.max_new_tokens,
            pad_token_id=tokenizer.eos_token_id,
            do_sample=True,
            temperature=0.6
        )

    # Decode only the newly generated tokens (ignoring the prompt)
    input_length = inputs.input_ids.shape[-1]
    generated_tokens = outputs[0][input_length:]
    response_text = tokenizer.decode(generated_tokens, skip_special_tokens=True)

    return {"response": response_text.strip()}
```

A naive solution

```
MODEL_ID = "meta-llama/Meta-Llama-3-8B-Instruct"

# Global variables to hold the loaded model and tokenizer
tokenizer = None
model = None

# Load the model exactly once when the server starts up
@app.on_event("startup")
async def load_model():
    global tokenizer, model
    print(f"Loading {MODEL_ID}...")

    tokenizer = AutoTokenizer.from_pretrained(MODEL_ID)

    # device_map="auto" automatically loads the model onto available GPUs
    # torch_dtype=torch.bfloat16 reduces memory usage
    model = AutoModelForCausalLM.from_pretrained(
        MODEL_ID,
        torch_dtype=torch.bfloat16,
        device_map="auto"
    )
    print("Model loaded successfully!")

# Define the expected JSON payload format
class PromptRequest(BaseModel):
    prompt: str
    max_new_tokens: int = 128
```

```
# Define the endpoint to handle incoming requests
@app.post("/generate")
async def generate(request: PromptRequest):
    if model is None or tokenizer is None:
        raise HTTPException(status_code=503, detail="Model is still loading.")

    # Tokenize the input and move it to the same device as the model (e.g., GPU)
    inputs = tokenizer(request.prompt, return_tensors="pt").to(model.device)

    # Generate the output
    with torch.no_grad():
        outputs = model.generate(
            **inputs,
            max_new_tokens=request.max_new_tokens,
            pad_token_id=tokenizer.eos_token_id,
            do_sample=True,
            temperature=0.6
        )

    # Decode only the newly generated tokens (ignoring the prompt)
    input_length = inputs.input_ids.shape[-1]
    generated_tokens = outputs[0][input_length:]
    response_text = tokenizer.decode(generated_tokens, skip_special_tokens=True)

    return {"response": response_text.strip()}
```

★ What makes this solution bad for a real-world deployment?

What makes the solution bad for a real-world deployment?

- A.  If two users were to ask for generations at the same time, the second will have to wait until generation finishes for the first.
- B. Reducing the model's precision from float32 (the default) to float16 will severely hurt performance.
- C. Tokenization is happening on the CPU rather than on the GPU.
- D.  The default batching behavior of `model.generate` is very inefficient since it doesn't batch queries together.

★ What traits do we want from a real-world LLM deployment?

★ What traits do we want from a real-world LLM deployment?

- Efficient continuous batching of users queries
- Rate limiting and user authentication
- Robustness to adversarial attacks
- Dynamic scaling up to handle utilization surges
- Whatever we can do to make models small and inference fast (without sacrificing quality)
 - Efficient decoding algorithms
 - Optimized attention kernels
 - Quantization

An alternative solution with vLLM

```
import uuid
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, Field
from vllm.engine.arg_utils import AsyncEngineArgs
from vllm.engine.async_llm_engine import AsyncLLMEngine
from vllm.sampling_params import SamplingParams

app = FastAPI(title="vLLM Llama 3 Server (Demo)")

# Define the expected JSON payload format with basic validation
class PromptRequest(BaseModel):
    prompt: str = Field(..., max_length=2000)
    max_new_tokens: int = Field(default=128, le=1024)

# --- 1. vLLM Engine Setup ---
MODEL_ID = "meta-llama/Meta-Llama-3-8B-Instruct"

# Configure the engine arguments
engine_args = AsyncEngineArgs(
    model=MODEL_ID,
    tensor_parallel_size=1,    # Change this if you have multiple GPUs
    max_model_len=4096,       # Limit context window to save VRAM
    disable_log_requests=True # Keeps the console clean
)
```

```
engine = AsyncLLMEngine.from_engine_args(engine_args)

# --- 2. The Non-Blocking Endpoint ---
@app.post("/generate")
async def generate(request: PromptRequest):
    # vLLM requires a unique ID for every request it processes
    request_id = str(uuid.uuid4())

    sampling_params = SamplingParams(
        temperature=0.6,
        max_tokens=request.max_new_tokens
    )

    # engine.generate returns an async generator, offloading the work
    # to the vLLM background loop so FastAPI remains unblocked.
    results_generator = engine.generate(request.prompt, sampling_params, request_id)

    final_output = None

    # Iterate through the generator until the sequence is finished
    async for request_output in results_generator:
        final_output = request_output

    if final_output is None:
        raise HTTPException(status_code=500, detail="Generation failed")

    return {"response": final_output.outputs[0].text.strip()}
```

What does vLLM handle for you?

- State-of-the-art serving throughput
- Efficient management of attention key and value memory with PagedAttention
- Continuous batching of incoming requests, chunked prefill, prefix caching
- Fast and flexible model execution with piecewise and full CUDA/HIP graphs
- Quantization: FP8, MXFP8/MXFP4, NVFP4, INT8, INT4, GPTQ/AWQ, GGUF, compressed-tensors, ModelOpt, TorchAO, and more
- Optimized attention kernels including FlashAttention, FlashInfer, TRTLLM-GEN, FlashMLA, and Triton
- Optimized GEMM/MoE kernels for various precisions using CUTLASS, TRTLLM-GEN, CuTeDSL
- Speculative decoding including n-gram, suffix, EAGLE, DFlash
- Automatic kernel generation and graph-level transformations using torch.compile
- Disaggregated prefill, decode, and encode

An even simpler version

```
python -m vllm.entrypoints.openai.api_server \
    --model meta-llama/Meta-Llama-3-8B-Instruct
```

- This gives you an OpenAI-style API

```
from openai import OpenAI

# 1. Initialize the client
# Point the base_url to your local vLLM server's /v1 endpoint
client = OpenAI(
    base_url="http://localhost:8000/v1",
    api_key="super-secret-key"
)

# 2. Make the request
response = client.chat.completions.create(
    model="meta-llama/Meta-Llama-3-8B-Instruct",
    messages=[
        {"role": "system", "content": "You are a helpful and concise assistant."},
        {"role": "user", "content": "Explain quantum computing in one sentence."}
    ],
    max_tokens=50,
    temperature=0.7
)
```

Let's go into more detail on some of these traits we want for efficient deployment.

- More efficient attention
- Quantization
- Prefix caching
- Clever batching
- Efficient decoding algorithms
- Multi-GPU parallelism

(all of these are handled for you by libraries like vLLM)

More efficient attention

- 2023 work

Add OPT

WoosukKwon committed on Feb 9, 2023

Initial commit

WoosukKwon committed on Feb 9, 2023

- LLaMA 1: February 2023

Efficient Memory Management for Large Language Model Serving with *PagedAttention*

Woosuk Kwon^{1,*} Zhuohan Li^{1,*} Siyuan Zhuang¹ Ying Sheng^{1,2} Lianmin Zheng¹ Cody Hao Yu³ Joseph E. Gonzalez¹ Hao Zhang⁴ Ion Stoica¹
¹UC Berkeley ²Stanford University ³Independent Researcher ⁴UC San Diego

Abstract

High throughput serving of large language models (LLMs) requires batching sufficiently many requests at a time. However, existing systems struggle because the key-value cache (KV cache) memory for each request is huge and grows and shrinks dynamically. When managed inefficiently, this memory can be significantly wasted by fragmentation and redundant duplication, limiting the batch size. To address this problem, we propose PagedAttention, an attention algorithm inspired by the classical virtual memory and paging techniques in operating systems. On top of it, we build vLLM, an LLM serving system that achieves (1) near-zero waste in KV cache memory and (2) flexible sharing of KV cache within and across requests to further reduce memory usage. Our evaluations show that vLLM improves the throughput of popular LLMs by 2-4× with the same level of latency compared to the state-of-the-art systems, such as FasterTransformer and Orca. The improvement is more pronounced with longer sequences, larger models, and more complex decoding algorithms. vLLM’s source code is publicly available at <https://github.com/vllm-project/vllm>.

1 Introduction

The emergence of large language models (LLMs) like GPT [5,

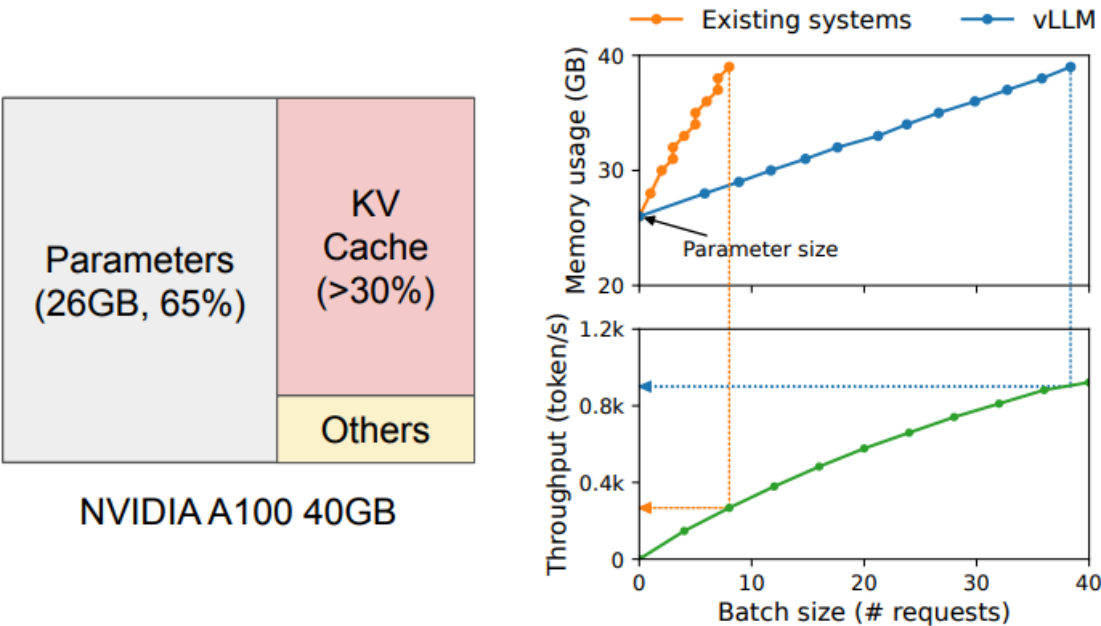
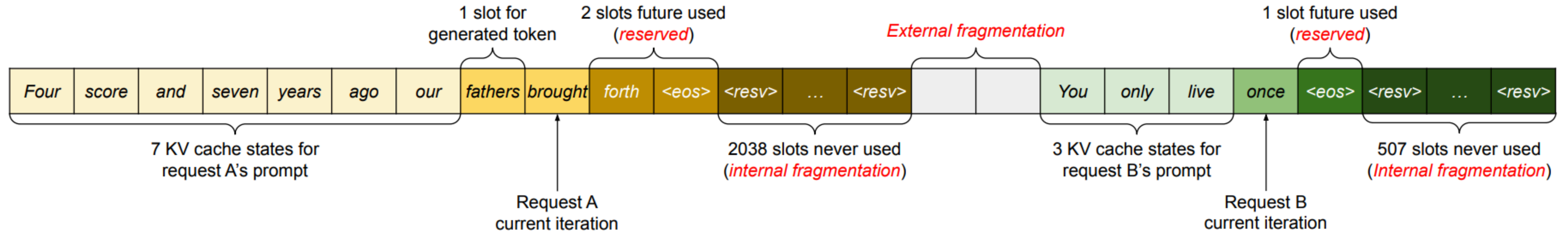


Figure 1. *Left:* Memory layout when serving an LLM with 13B parameters on NVIDIA A100. The parameters (gray) persist in GPU memory throughout serving. The memory for the KV cache (red) is (de)allocated per serving request. A small amount of memory (yellow) is used ephemeraally for activation. *Right:* vLLM smooths out the rapid growth curve of KV cache memory seen in existing systems [31, 60], leading to a notable boost in serving throughput.

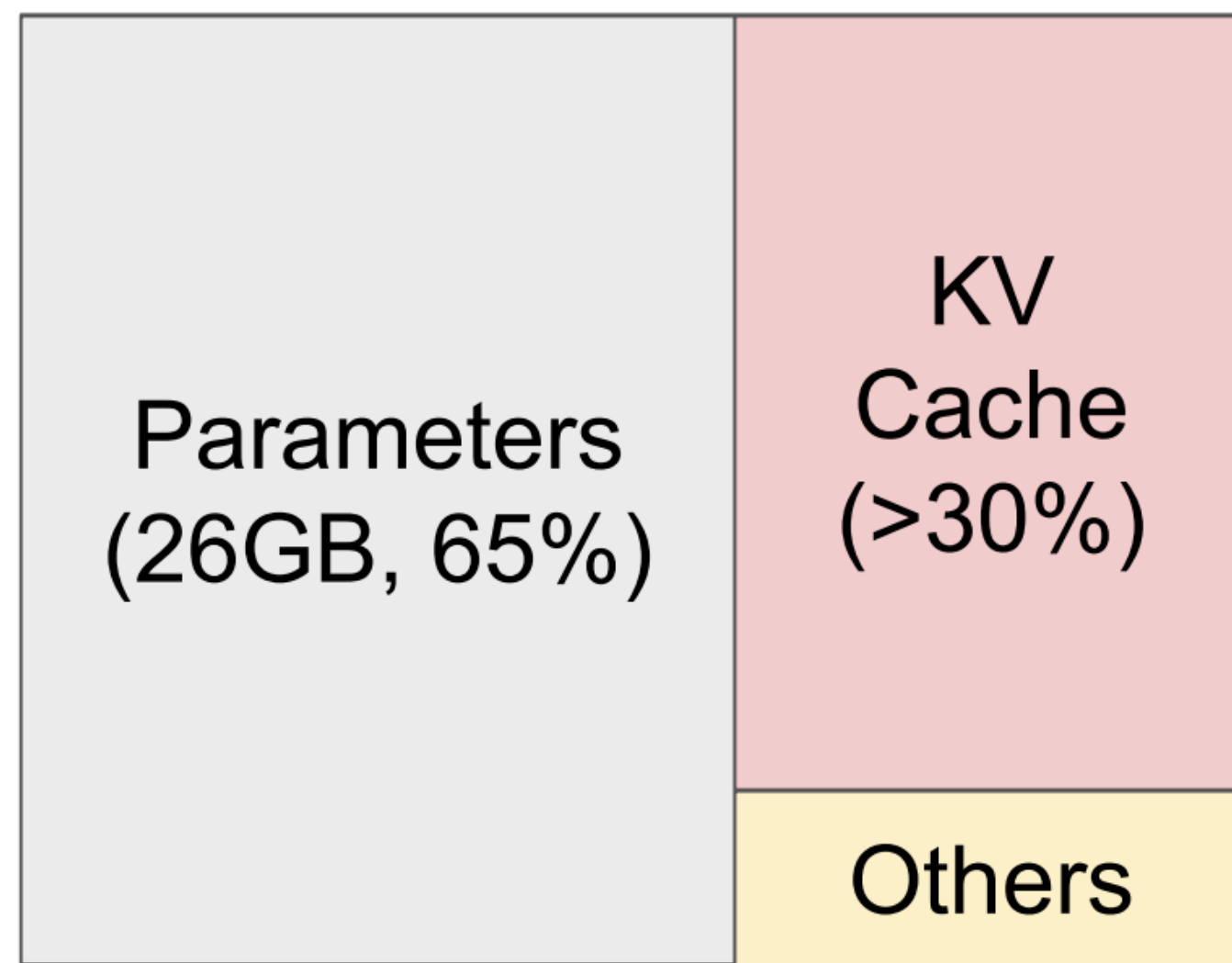
the cost per request—of LLM serving systems is becoming more important.

At the core of LLMs lies an autoregressive Transformer model [53]. This model generates words (tokens), *one at a time*, based on the input (prompt) and the previous sequence

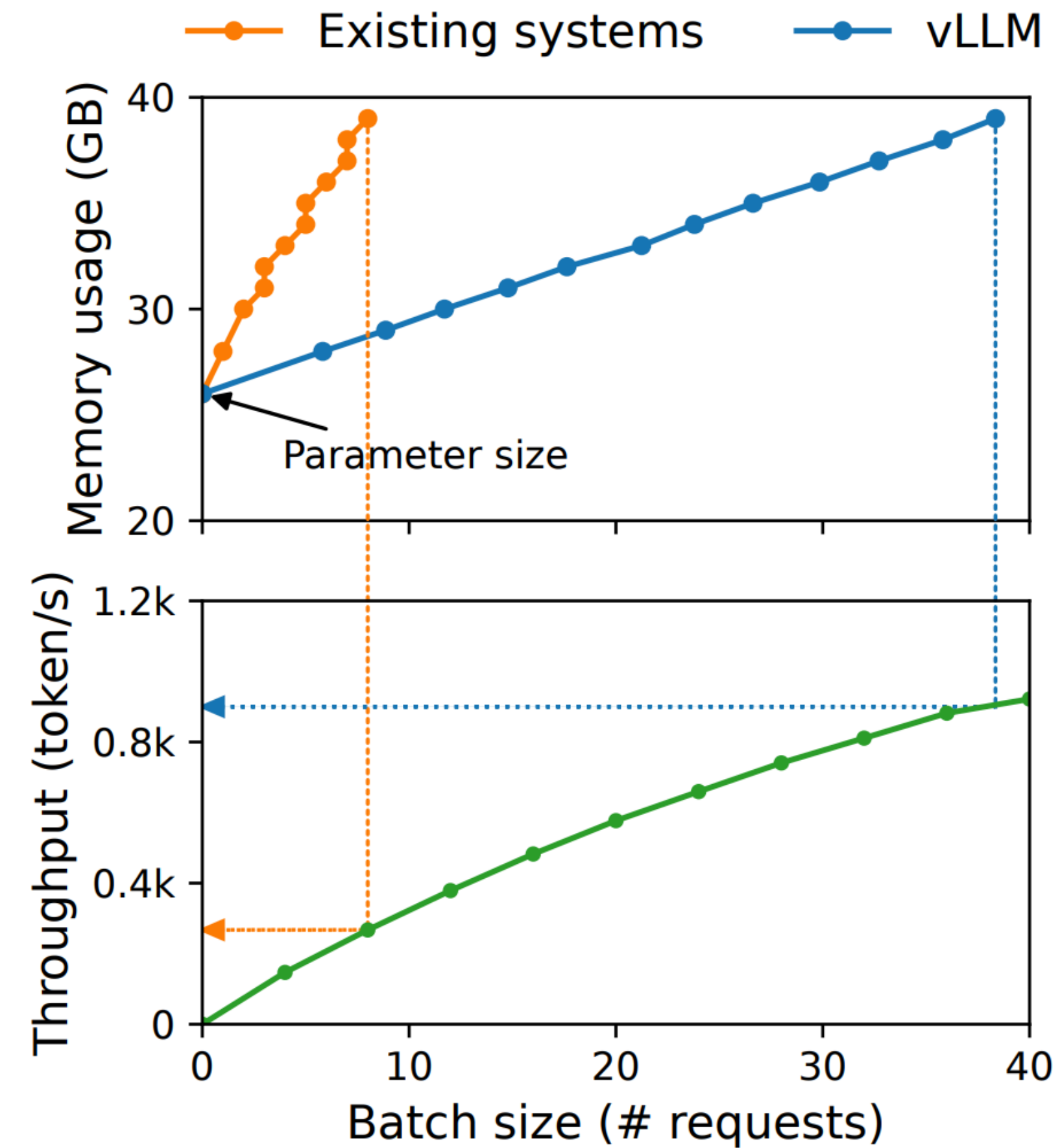
Main Motivation: Memory



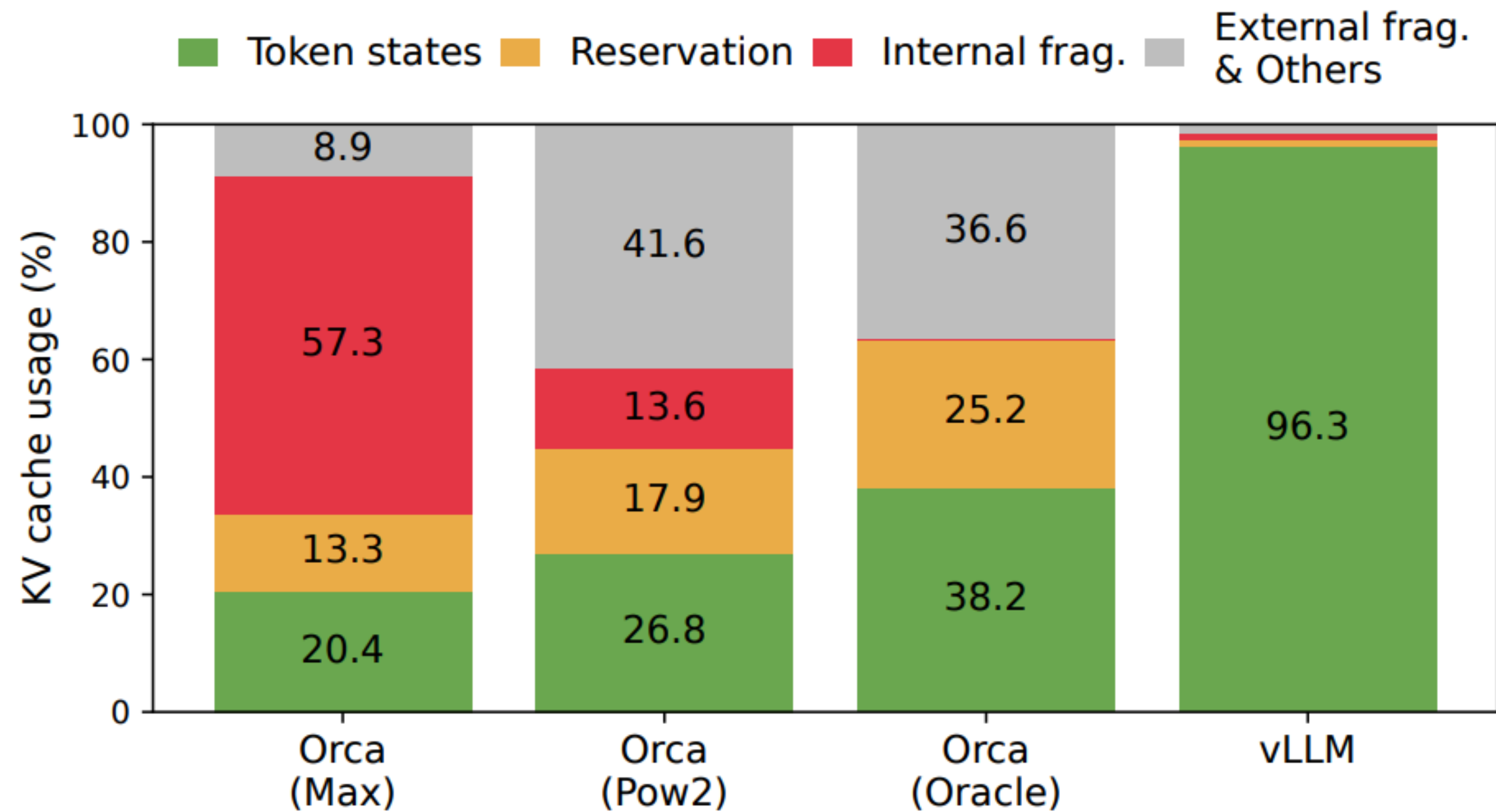
Main Motivation: Memory



NVIDIA A100 40GB



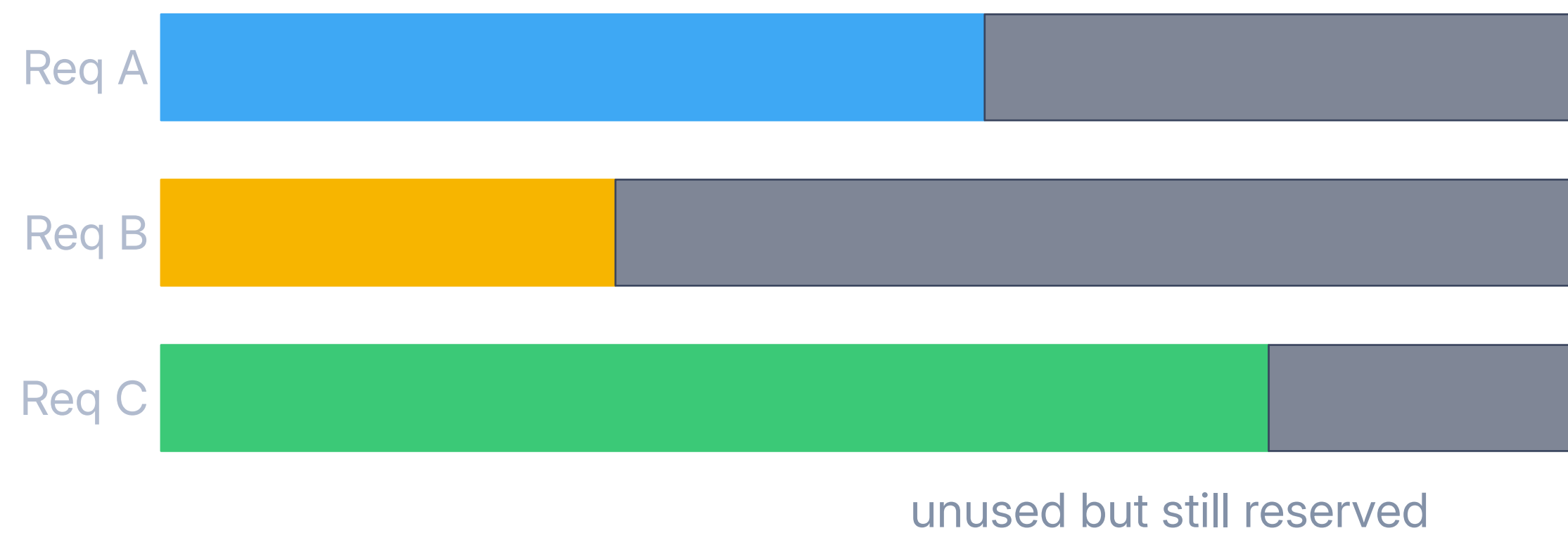
Main Motivation: Memory



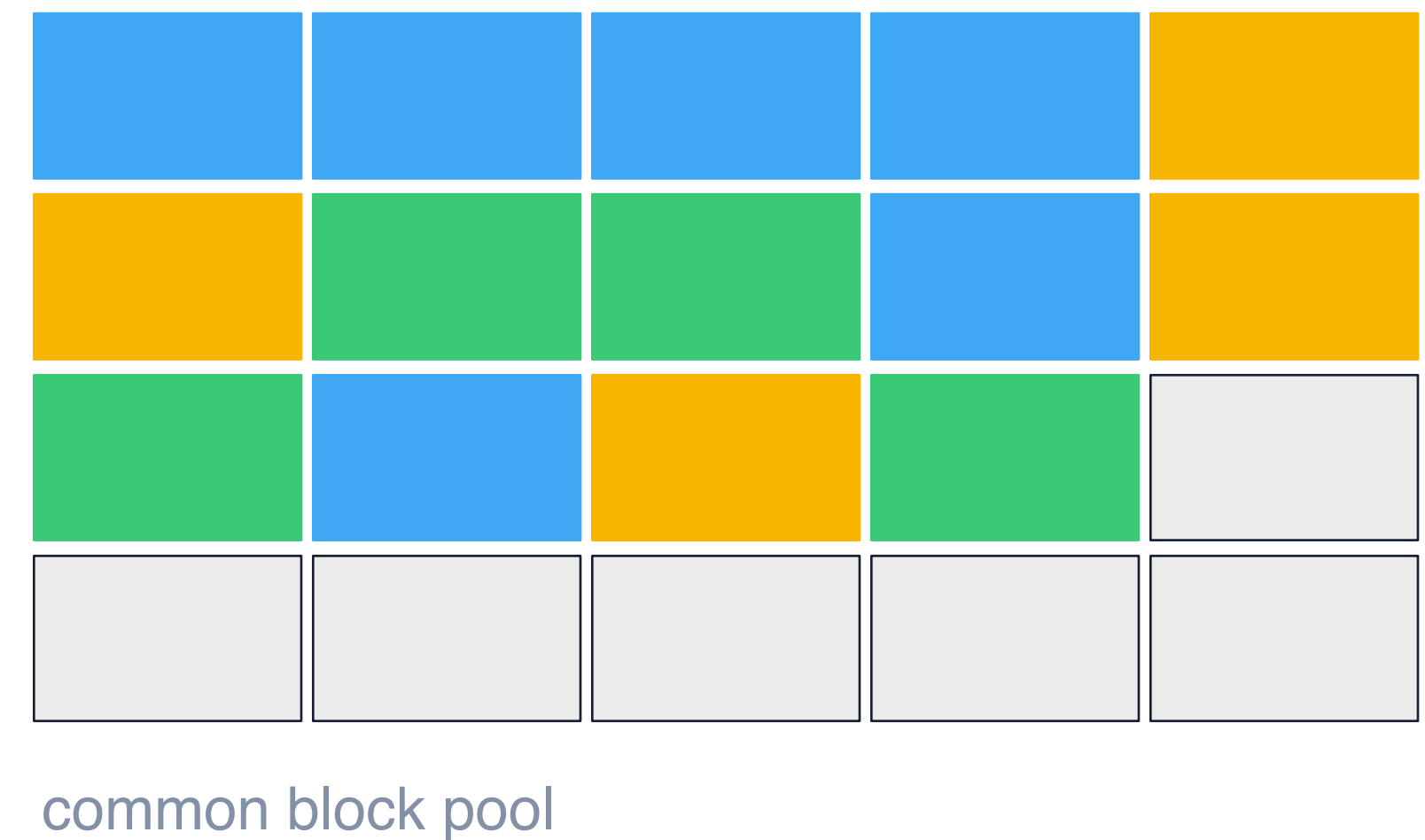
Paged Attention

Improve memory usage

Without paging



With paging



Quantization

- Most language models are trained with 32-bit floating point precision
- One easy way to halve the model size is to quantize down to 16-bit precision
- Some information is lost since multiple 32-bit numbers may map to the same 16-bit one

bfloat16



float32: IEEE single-precision



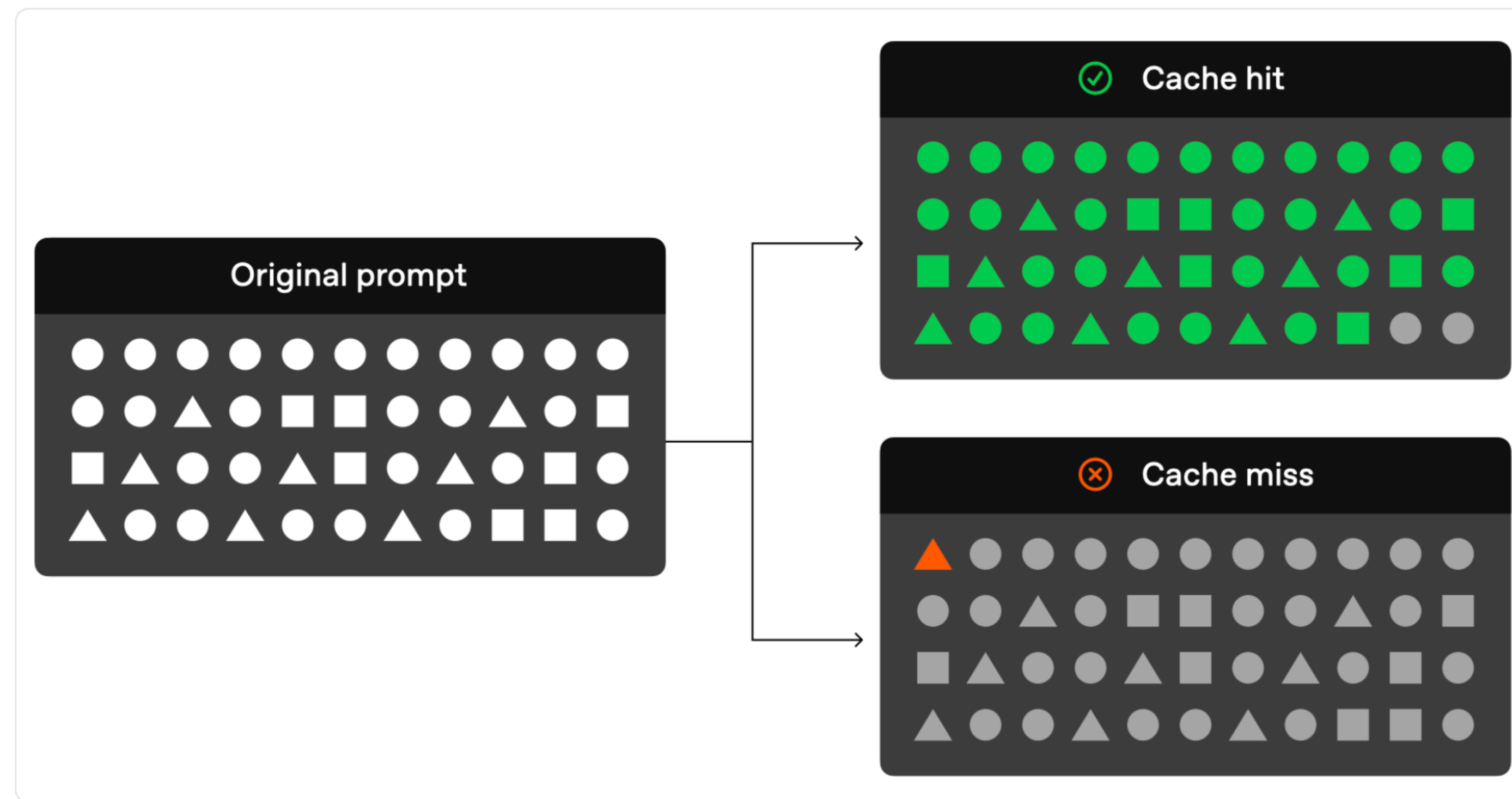
float16: IEEE half-precision



Model	Params	32-bit	FP16	8-bit	4-bit
Llama 2-Chat	7B	56	28	14	7
	13B	104	52	26	13
	70B	336	168	84	42

Table 1: Estimated GPU memory requirements (in Gigabyte) for Llama 2-Chat models at inference using various precision levels and parameter sizes (Kaplan et al., 2020; Hoffmann et al., 2022).

Prefix caching



Prefix caching

Pricing

 Copy Page

Flagship models

Our latest models

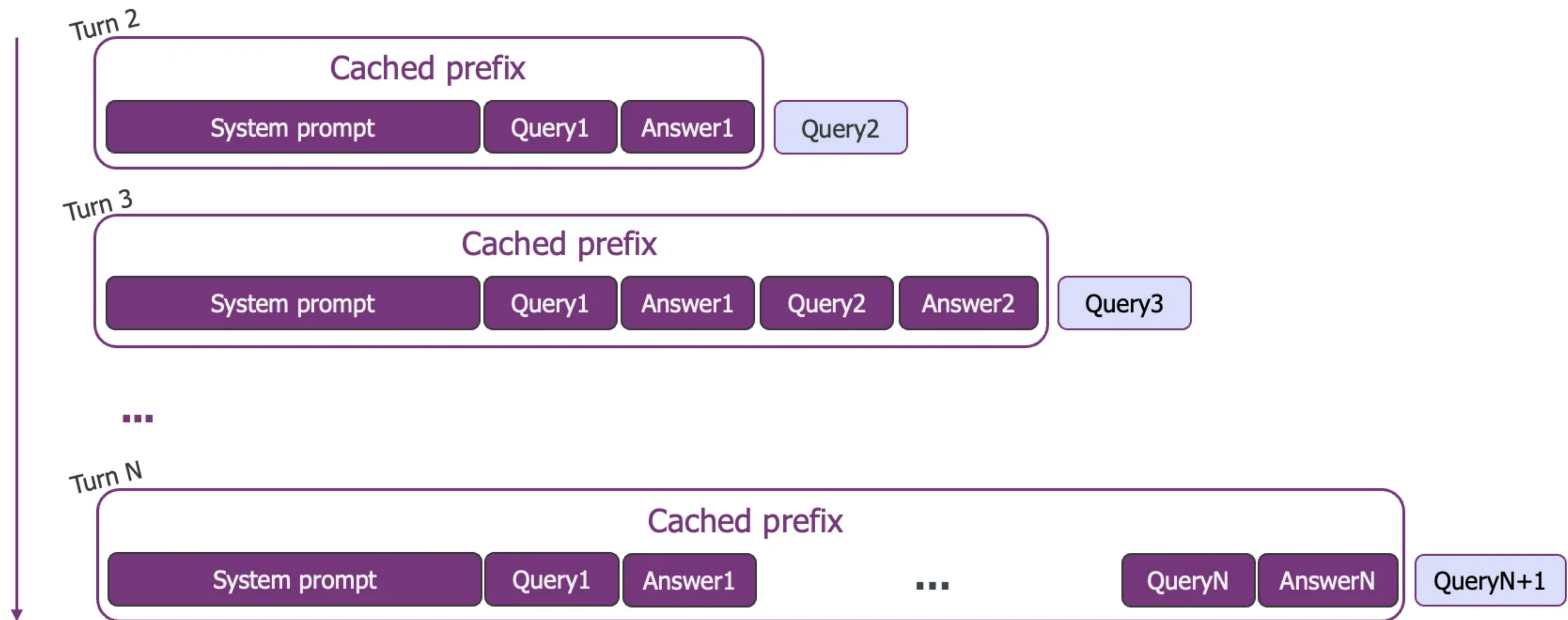
Prices per 1M tokens.

- Standard
- Batch
- Flex
- Priority

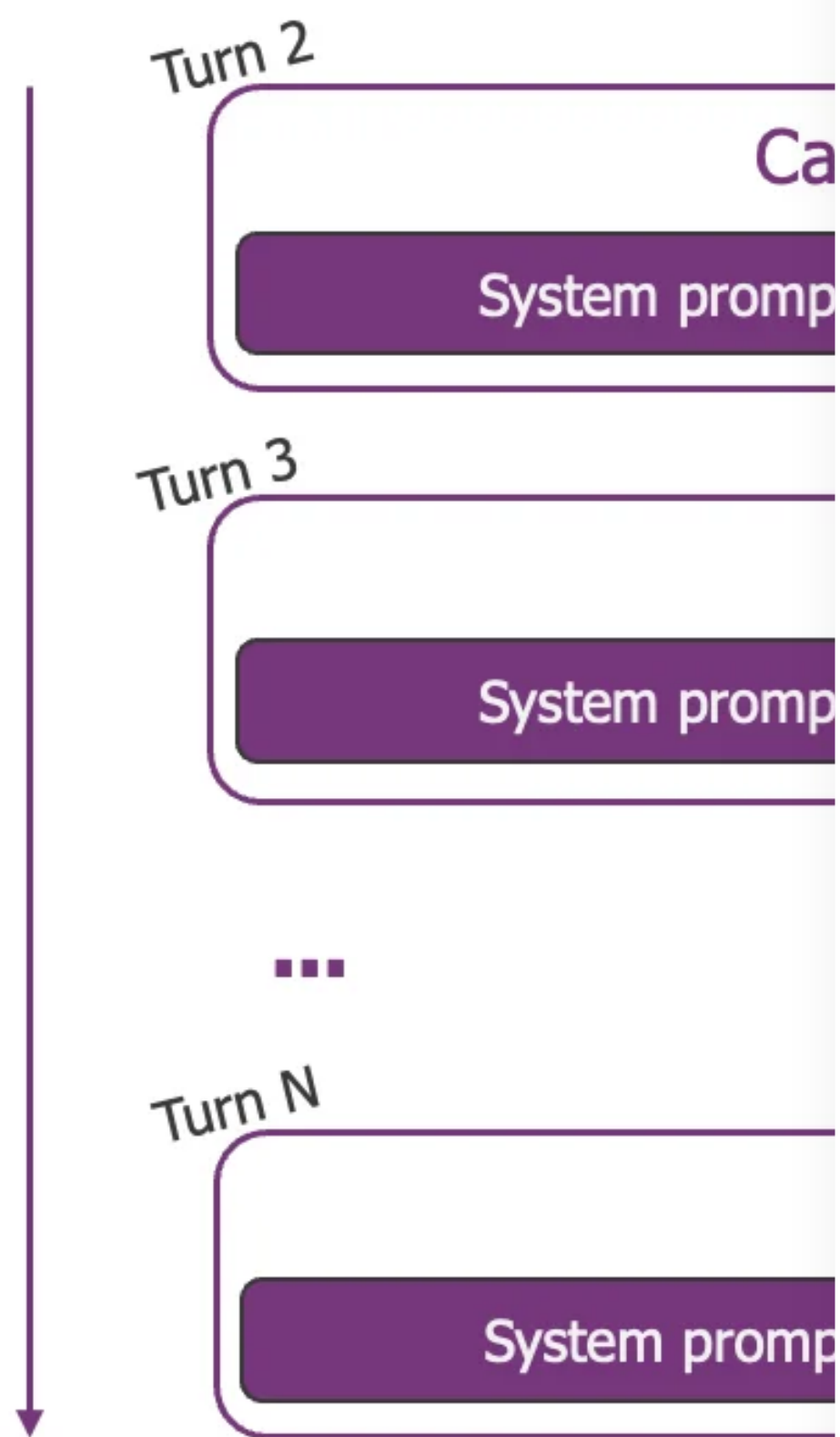
Model	<u>Short context</u>			<u>Long context</u>		
	Input	Cached input	<u>Output</u>	Input	Cached input	<u>Output</u>
gpt-5.4	\$2.50	\$0.25	\$15.00	\$5.00	\$0.50	\$22.50
gpt-5.4-mini	\$0.75	\$0.075	\$4.50	-	-	-
gpt-5.4-nano	\$0.20	\$0.02	\$1.25	-	-	-
gpt-5.4-pro	\$30.00	-	\$180.00	\$60.00	-	\$270.00

Regional processing (data residency) endpoints are charged a 10% uplift for `gpt-5.4`, `gpt-5.4-mini`, `gpt-5.4-nano`, and `gpt-5.4-pro`. See our [Your data](#) guide for supported regions and processing details.

Prefix caching



Prefix cache



I Know What You Asked: Prompt Leakage via KV-Cache Sharing in Multi-Tenant LLM Serving

Guanlong Wu
SUSTech
santiscowgl@gmail.com

Zheng Zhang
ByteDance Inc.
zhang2heng@buaa.edu.cn

Yao Zhang
ByteDance Inc.
zhangyao.crypto@bytedance.com

Weili Wang
SUSTech
12032870@mail.sustech.edu.cn

Jianyu Niu*
SUSTech
niu jy@sustech.edu.cn

Ye Wu
ByteDance Inc.
wuye.2020@bytedance.com

Yinqian Zhang*†
SUSTech
yinqianz@acm.org

Abstract—Large Language Models (LLMs), which laid the groundwork for Artificial General Intelligence (AGI), have recently gained significant traction in academia and industry due to their disruptive applications. In order to enable scalable applications and efficient resource management, various multi-tenant LLM serving frameworks have been proposed, in which the LLM caters to the needs of multiple users simultaneously. One notable mechanism in recent works, such as SGLang and vLLM, is sharing the Key-Value (KV) cache for identical token sequences among multiple users, saving both memory and computation.

This paper presents the first investigation on security risks associated with multi-tenant LLM serving. We show that the state-of-the-art mechanisms of KV cache sharing may lead to new side channel attack vectors, allowing unauthorized reconstruction of user prompts and compromising sensitive user information among mutually distrustful users. Specifically, we introduce our attack, PROMTPEEK, and apply it to three scenarios where the adversary, with varying degrees of prior knowledge, is capable of reverse-engineering prompts from other users. This study underscores the need for careful resource management in multi-tenant LLM serving and provides critical insights for future security enhancement.

I. INTRODUCTION

The rise of Large Language Models (LLMs) like GPT [25] or Llama [44] has enabled a variety of new applications.

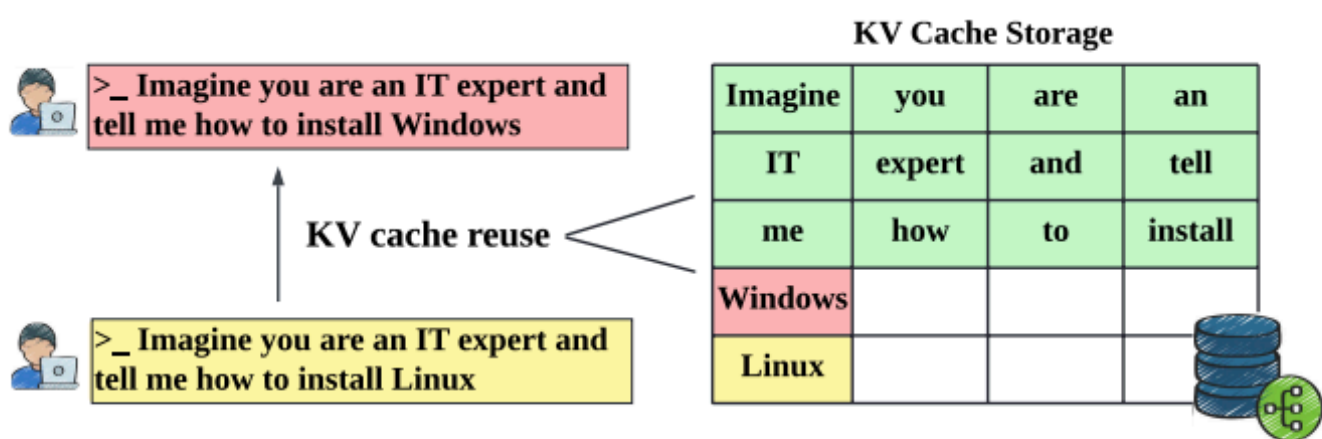


Figure 1: KV cache reuse.

capacity, the extensive size of the KV cache [12] restricts the ability to serve concurrent requests, becoming a critical bottleneck in multi-tenant scenarios.

One promising solution proposed by recent work (e.g., vLLM, SGLang) [32], [61], [53] is to share the KV cache across the requests to reduce both computation and memory usage. The rationale is that identical tokens in different requests can generate the same KV cache if their preceding tokens are also identical. Figure 1 illustrates an instance of KV cache sharing. When the first user submits the query “Imagine you are an IT expert and tell me how to install Windows”, the KV cache for each token is computed and stored on the LLM

Clever batching

Continuous Batching

- High-level goal:
 - maximize GPU utilization—every token position in the inputs batch performs useful computation
- When a request finishes generating its current token, the scheduler can immediately insert a new request into the vacant slot.
- HuggingFace blog post below is excellent if you want to learn more.

Paged Attention

Static v.s. **Continuous batching**

T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
S_1	S_1	S_1	S_1				
S_2	S_2	S_2					
S_3	S_3	S_3					
S_4	S_4	S_4					

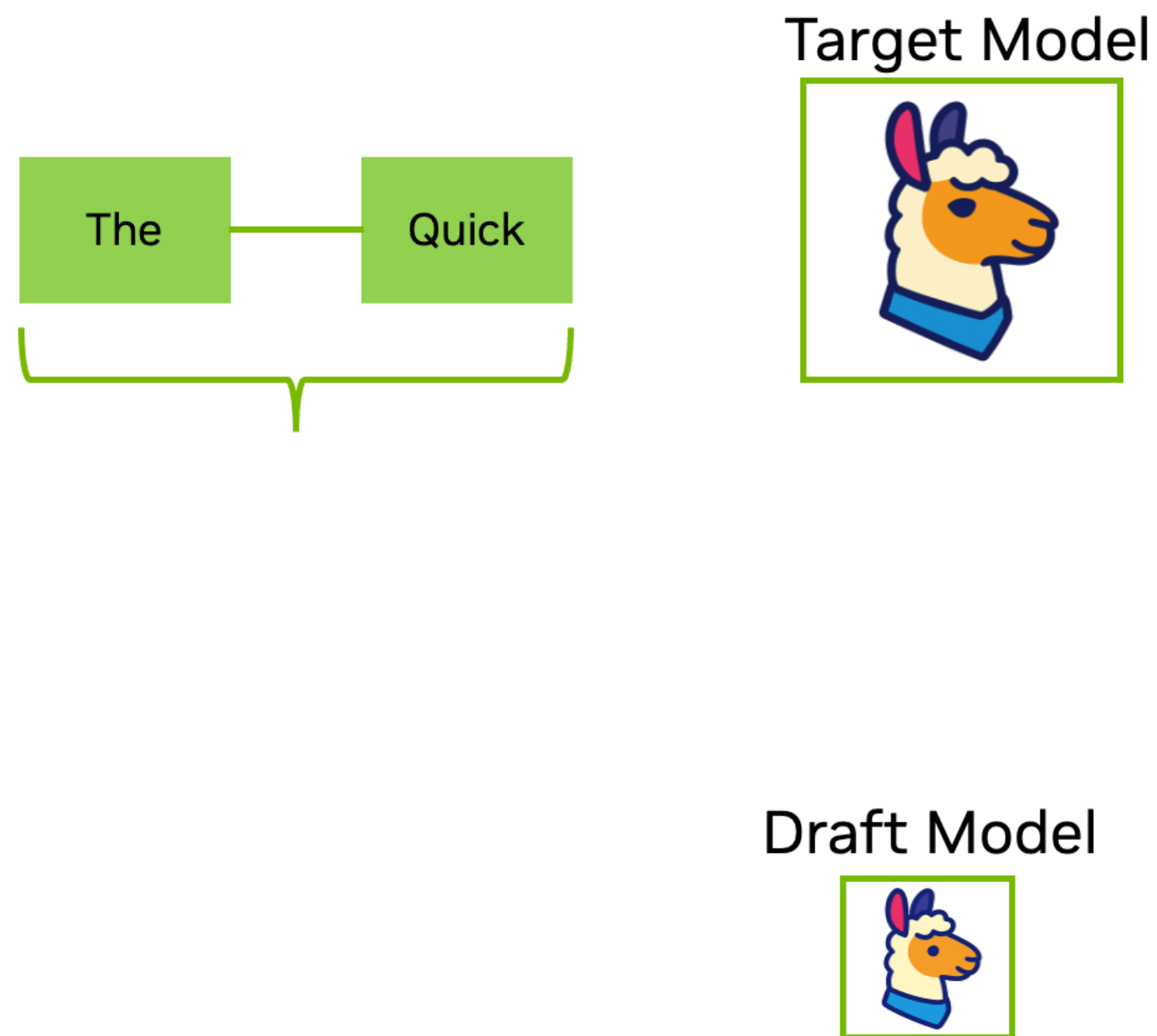
T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
S_1	S_1	S_1	S_1	S_1	END		
S_2	S_2	S_2	S_2	S_2	S_2	S_2	END
S_3	S_3	S_3	S_3	END			
S_4	S_4	S_4	S_4	S_4	S_4	END	

T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
S_1	S_1	S_1	S_1				
S_2	S_2	S_2					
S_3	S_3	S_3					
S_4	S_4	S_4					

T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
S_1	S_1	S_1	S_1	S_1	END	S_6	S_6
S_2	S_2	S_2	S_2	S_2	S_2	S_2	END
S_3	S_3	S_3	S_3	END	S_5	S_5	S_5
S_4	S_4	S_4	S_4	S_4	S_4	END	S_7

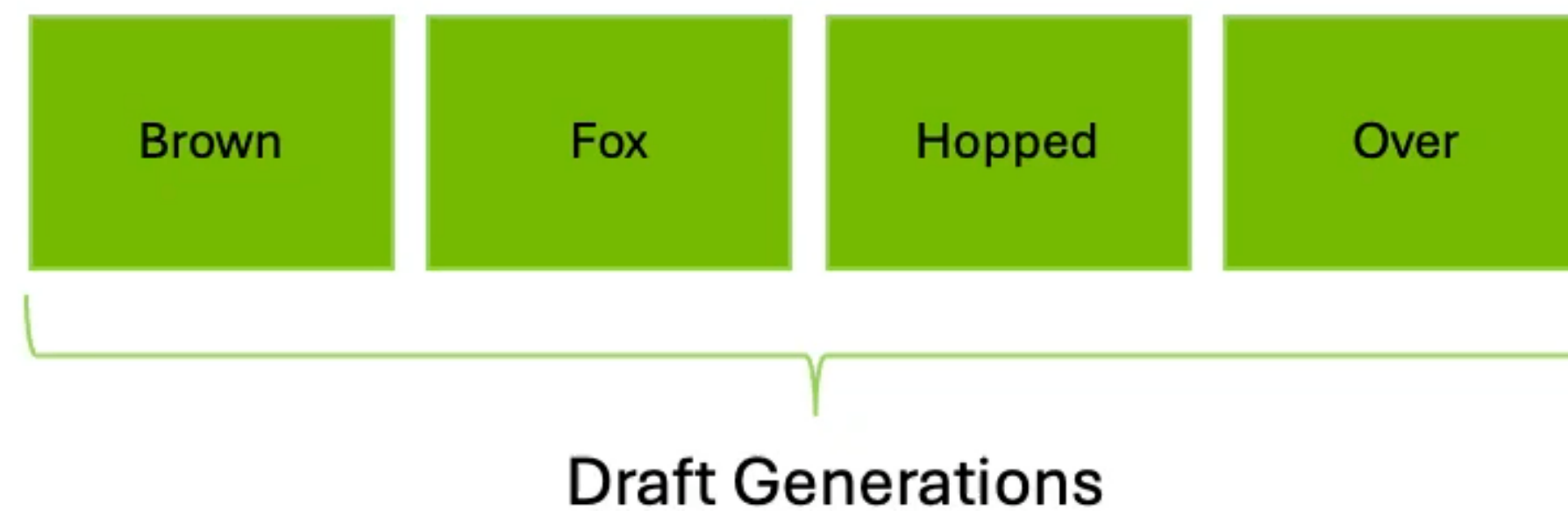
Efficient decoding algorithms

e.g. speculative decoding (beta)



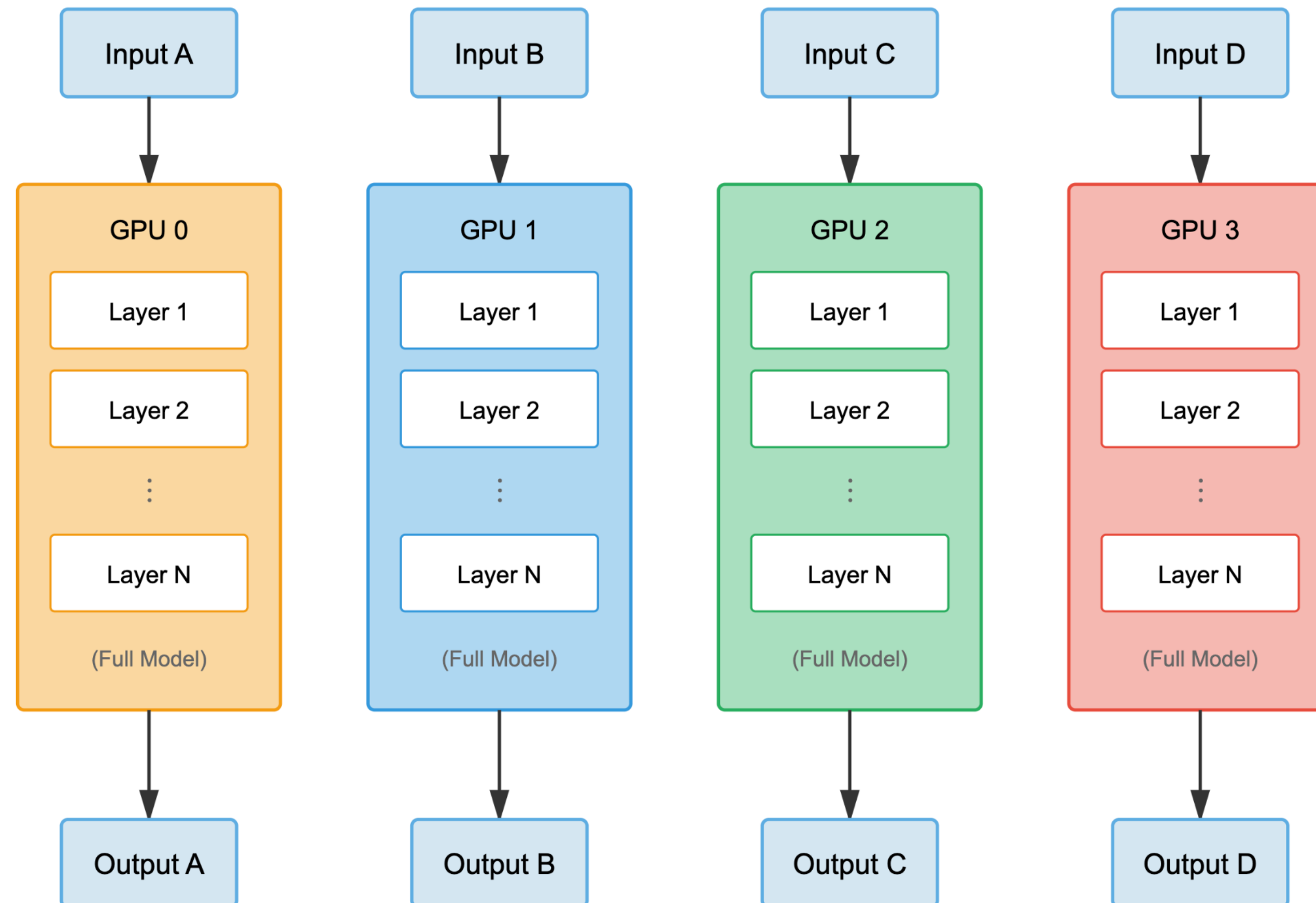
Efficient decoding algorithms

e.g. speculative decoding (beta)



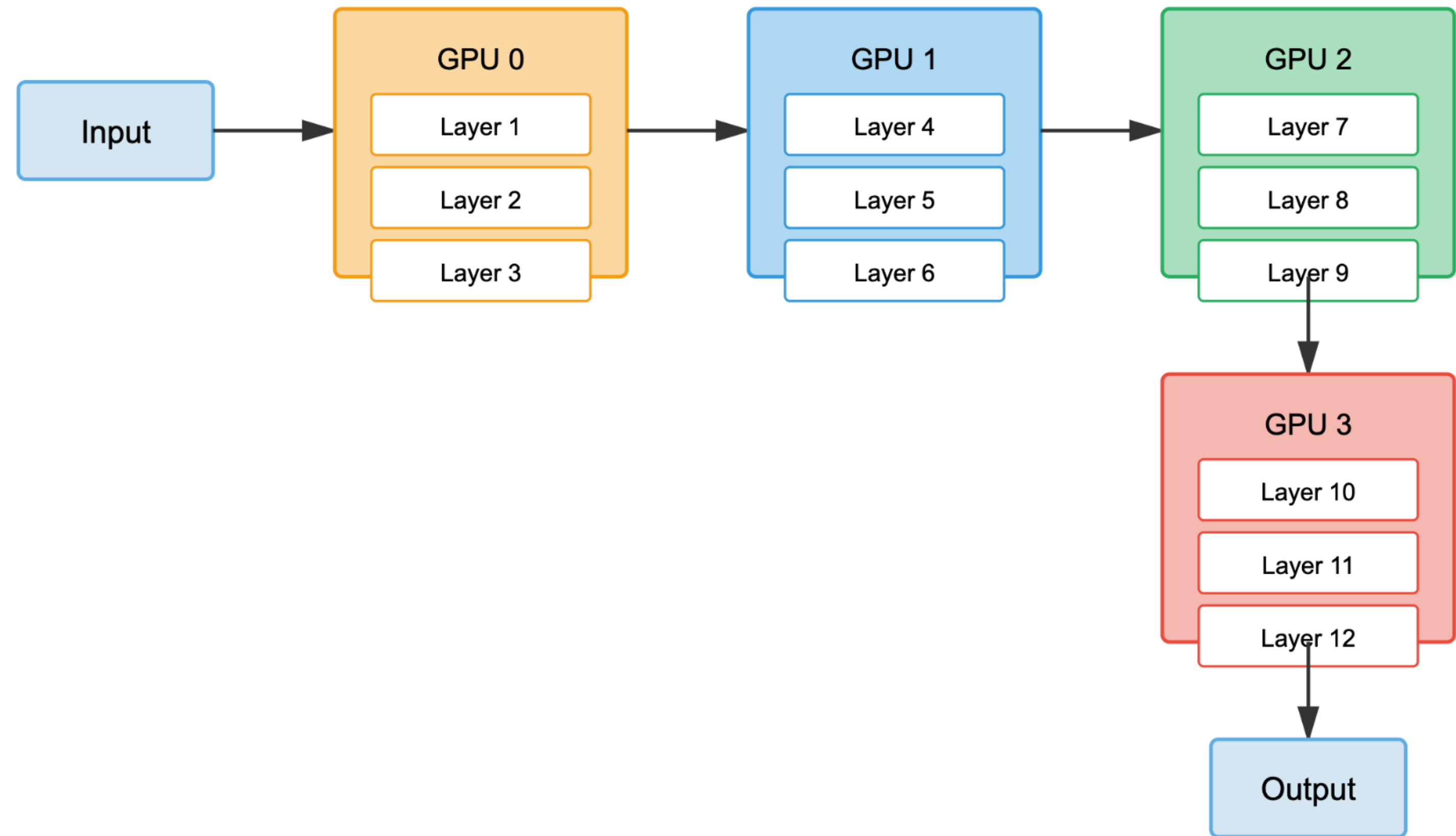
Multi-GPU parallelism

data parallelism



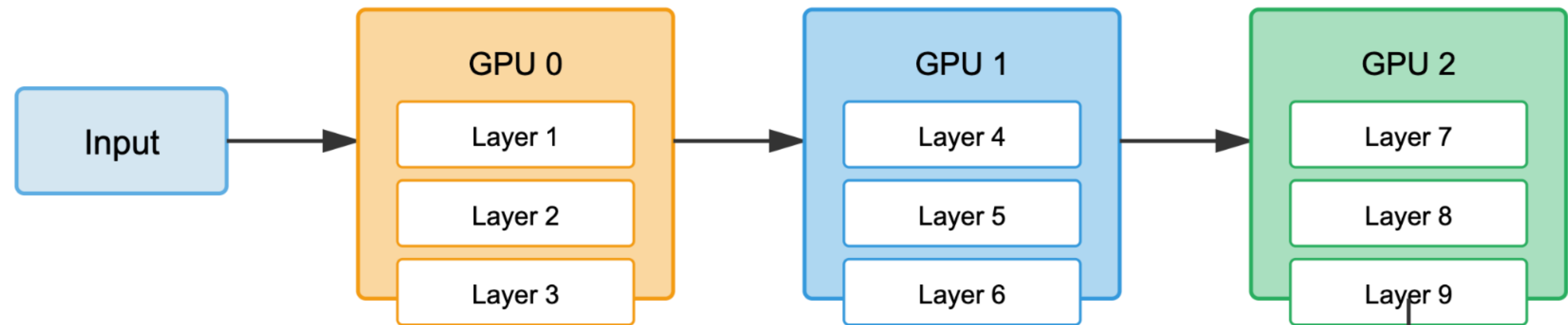
Multi-GPU parallelism

pipeline parallelism

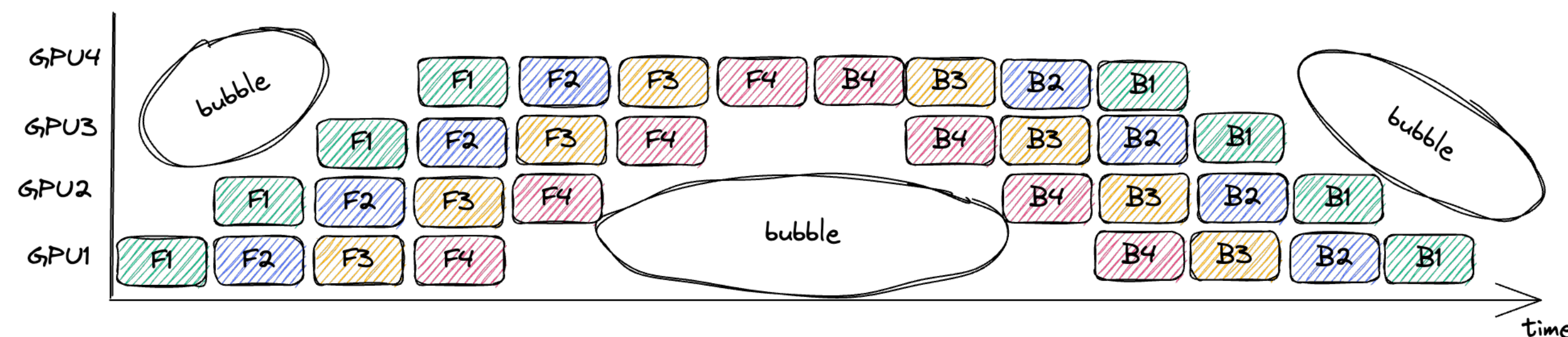
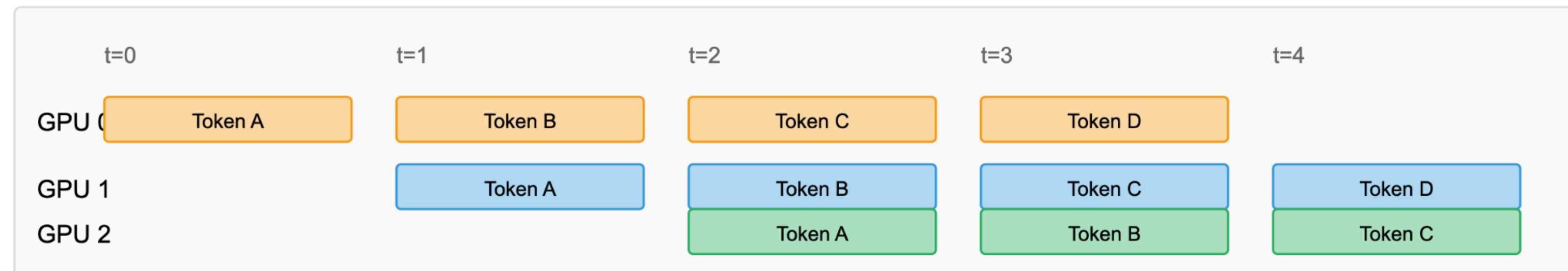


Multi-GPU parallelism

pipeline parallelism

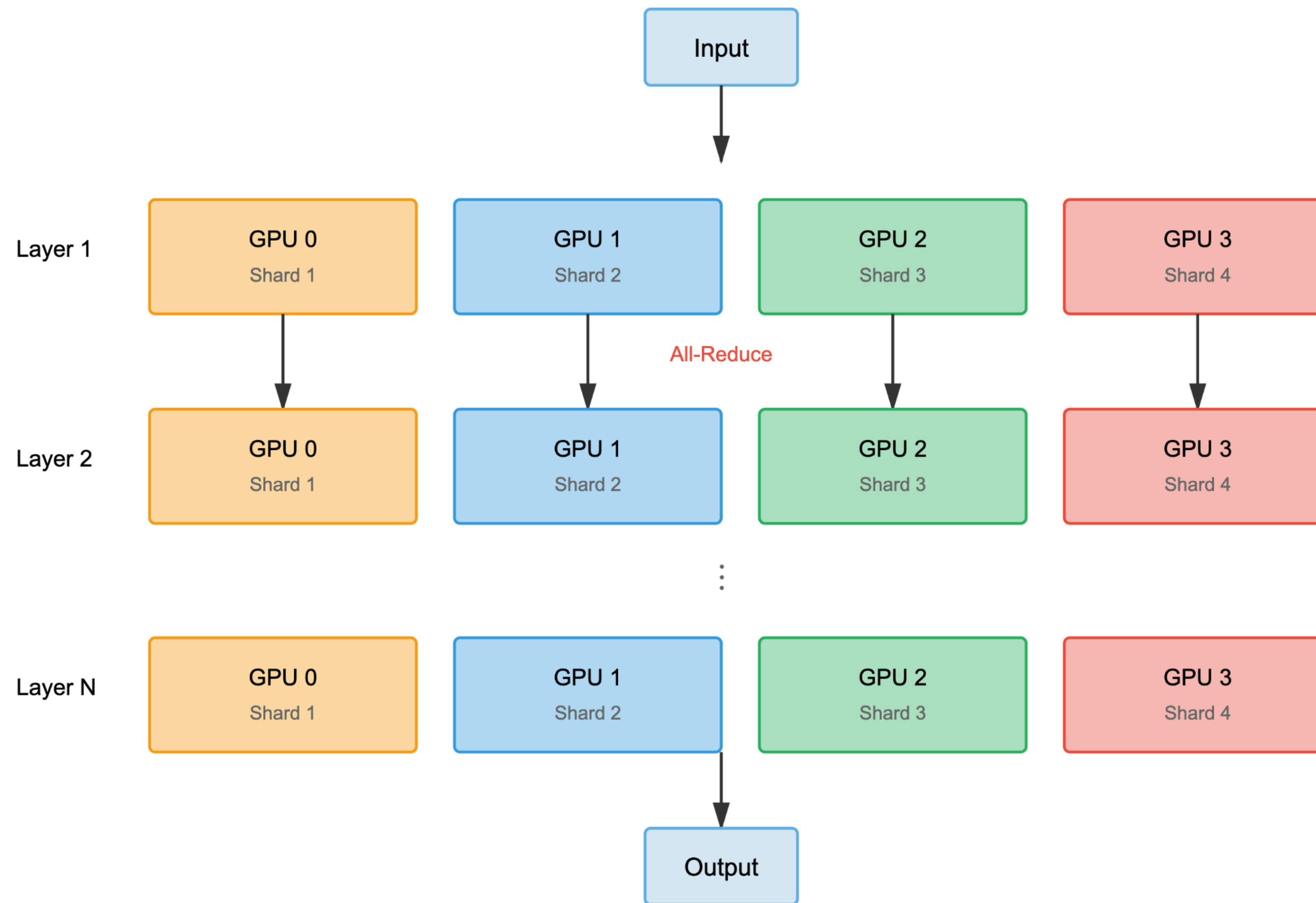


Pipeline Timeline

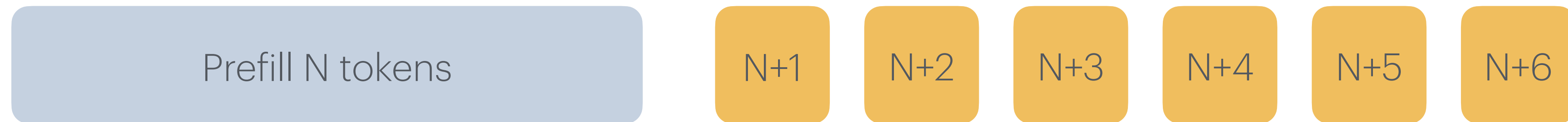


Multi-GPU parallelism

tensor parallelism



Metrics we care about



- When prefilling:
 - Measured by Time to first token (TTFT)
 - Cache hit rate
- When decoding
 - Runs one step per generated token.
 - Reads the growing KV cache every step.
 - Measured by inter token latency (ITL), the delay between each new token

Things to look out for

- `max_num_seqs` and `max_num_batched_tokens`:
 - balancing Prefill (TTFT) and Decode (ITL) latency
 - Larger batch means more concurrency, more throughput, and less queuing, but increases ITL.
- `tensor_parallel_size` and `pipeline_parallel_size`
 - Reducing the average model weight memory use per gpu, so as to have more KV cache

vLLM

Other optimizations

- Chunked Prefill (beta)
- Cross-gpu workers dispatch
- Fine-grained quantization support / CUDA kernel / etc

Last advice for developing LLM applications

- How would you describe your task?
 - Think about: simplicity, modularity, scale, efficiency
- How do you plan to measure whether an LLM works for your application?
 - Think about: offline metrics, online metrics, instrumentation, telemetry
- Now that you know how to measure “success,” does an existing LLM succeed at your problem?
 - Think about: prompt harness, latency, adversarial users
- If no, do you have the resources you need to build a custom LLM?
 - Think about: training data, infrastructure for deployment
- How do you expect to maintain your model in a changing world?
 - Think about: interface changes, market differences, tool deprecation, users

Q & A