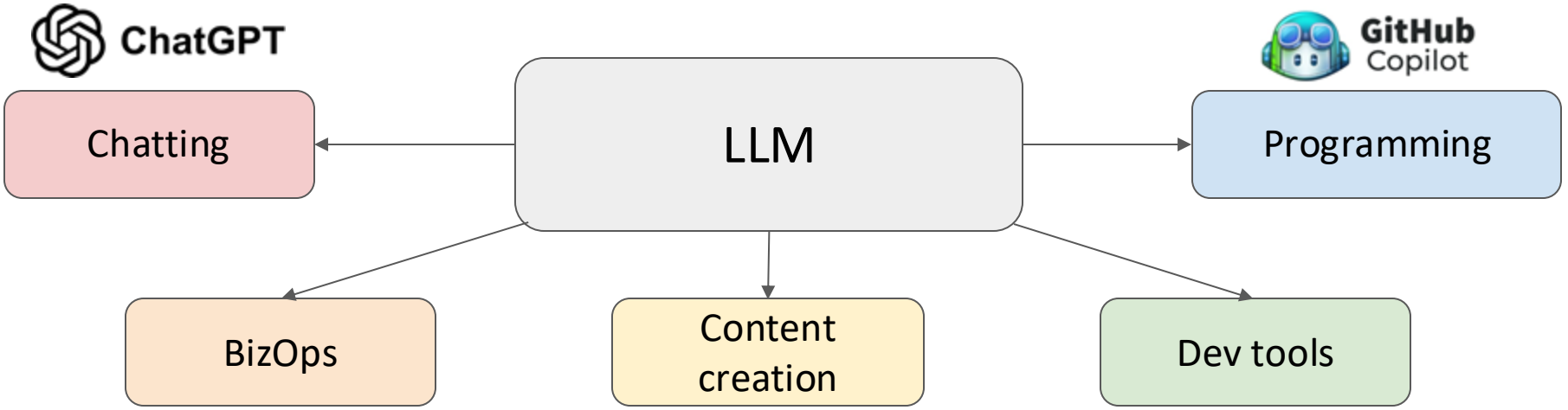
Disaggregating prefill and decode for goodput-optimized LLM serving

Speaker: Hao Zhang (UCSD)

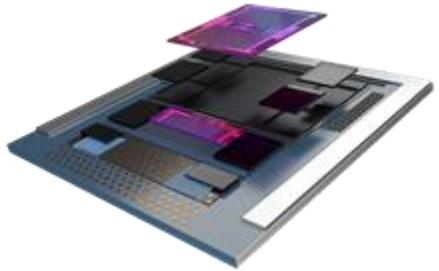
Agenda

- LLM Serving
- Continuous Batching
- Disaggregated prefill and decode
- DistServe/Dynamo: Key Design and Implementations
- Systems and Models based on Disaggregation

The era of Large Language Models (LLMs)



Served by GPUs

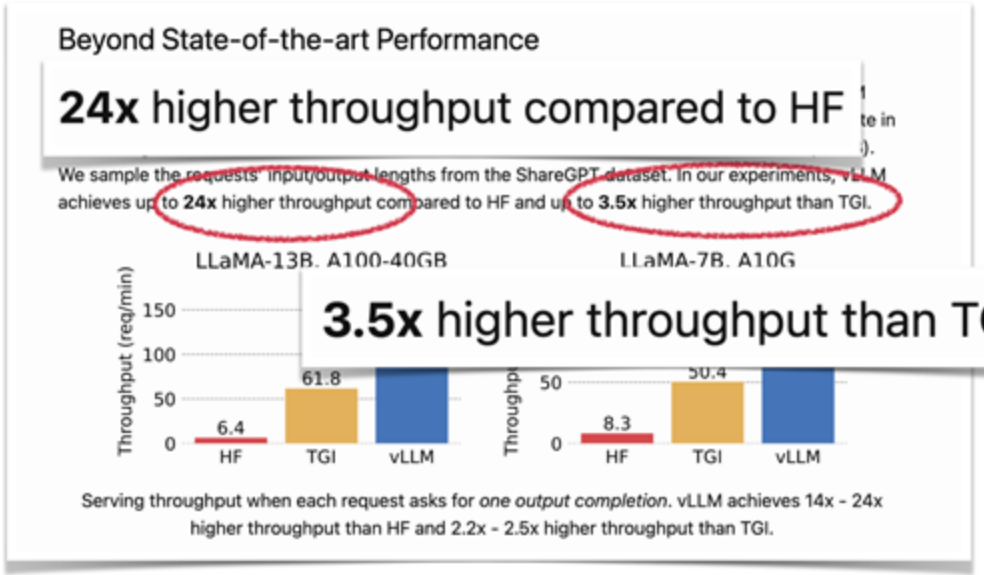


LLM System Today Optimize **Throughput**

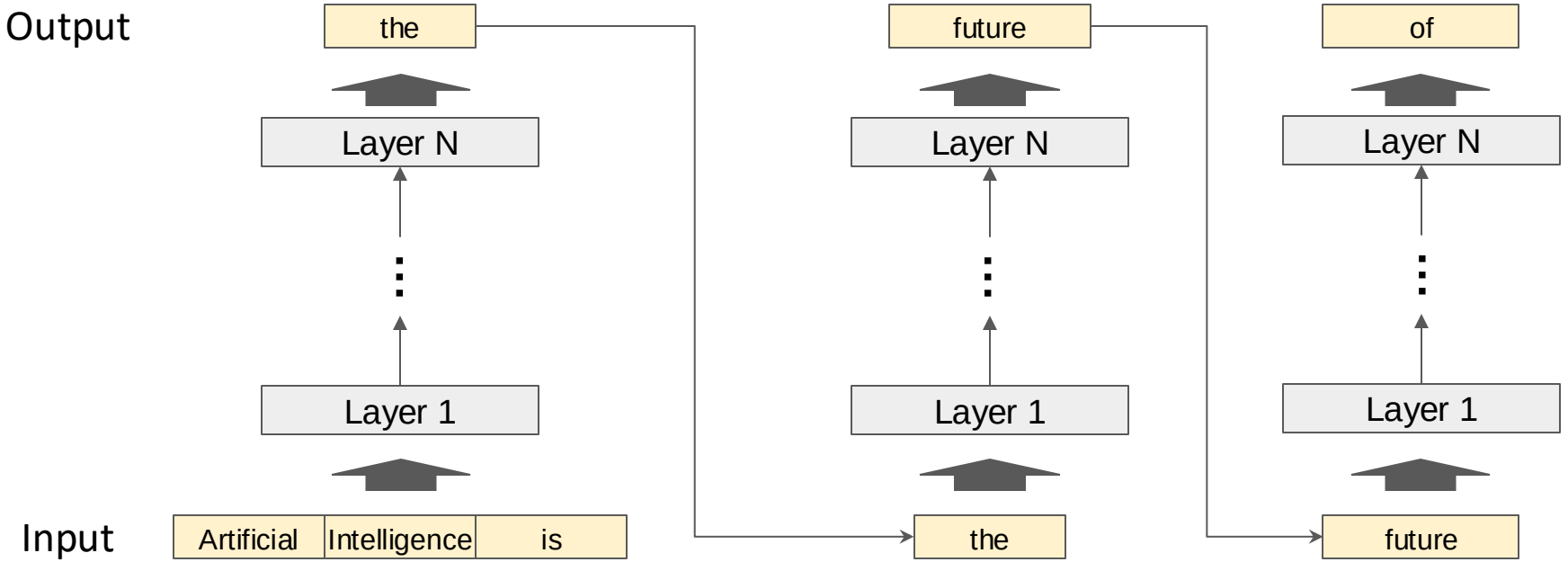
vLLM

DeepSpeed MII

NVIDIA



Inference process of LLMs



Repeat until the sequence

- Reaches its pre-defined maximum length (e.g., 2048 tokens)
- Generates certain tokens (e.g., “<|*end of sequence*|>”)

Generative LLM Inference: Autoregressive Decoding

- Pre-filling phase (0-th iteration):
  - Process **all** input tokens at once
- Decoding phase (all other iterations):
  - Process **a single** token generated from previous iteration

- Key-value cache:
  - Save attention keys and values for the following iterations to avoid recomputation
  - Woosuk Kwon has covered in the previous lecture ☺

Serving vs. Inference

large b

Serving: many requests, online traffic,

emphasize cost-per-query.

$b \rightarrow 1$

Inference: fewer request, low, offline traffic,

Emphsize latency

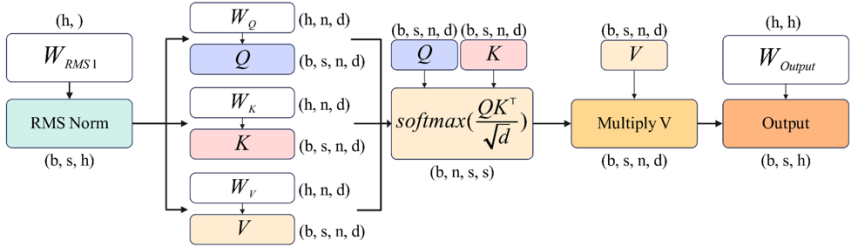
One of the Key Problem in LLM Serving

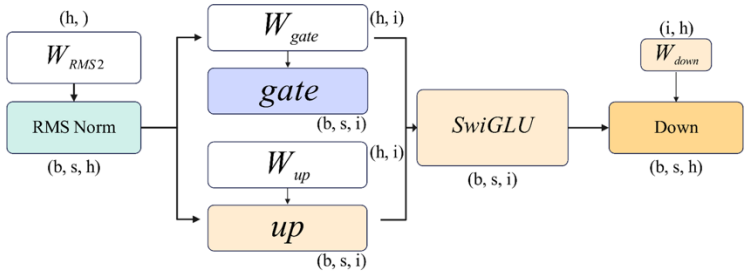
large b

Challenge: How to efficiently serve many users requests

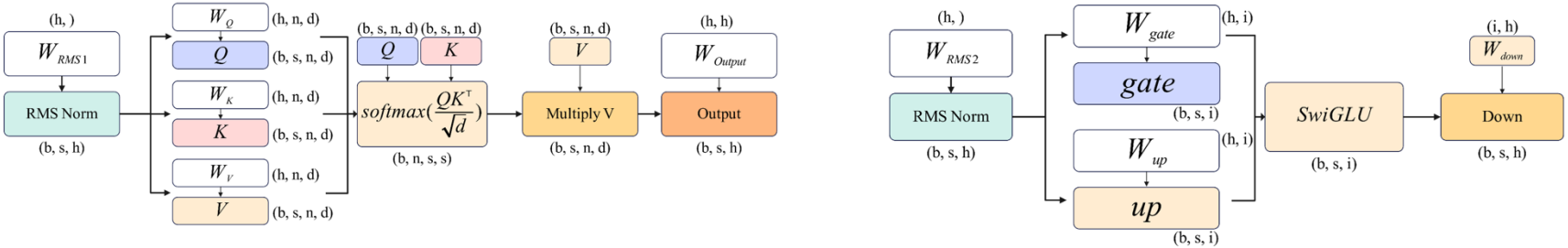
While minimizing the $ cost
(= max throughput)
(= min #GPU used)
(= max GPU utilization)

Review: A Typical LLM's Architecture





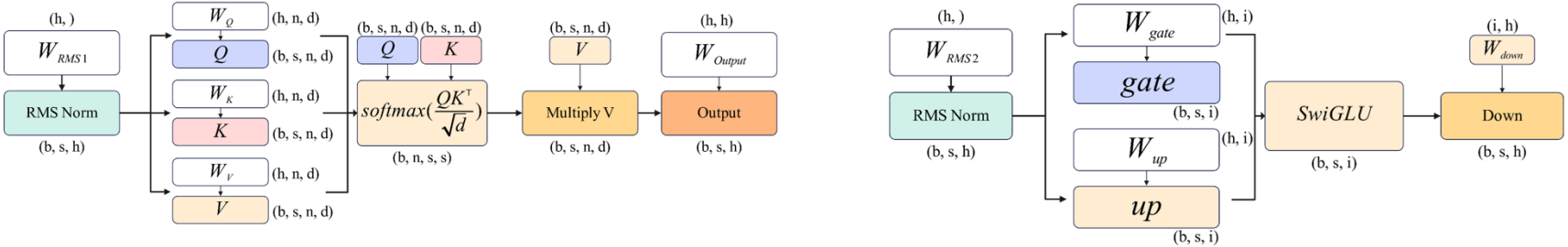
Review: LLM Inference Compute Characteristics



- Compute:
  - Prefill: attention and large GEMM (mostly same with training)
  - Decode: s = 1, GEMM degenerates to GEMV
- Memory
  - New: KV cache
- Communication
  - mostly same with training

Q: how batch size b changes the picture?

Review: LLM Inference Compute Characteristics



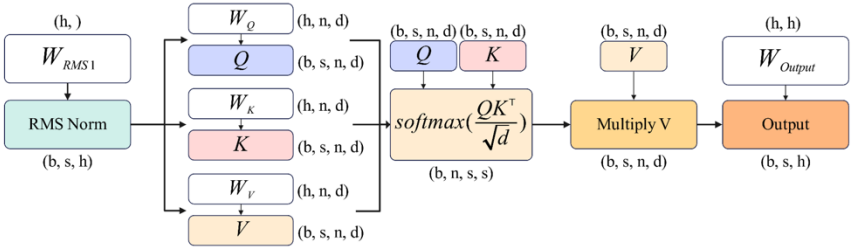
- Compute:
  - Prefill: attention and large GEMM (mostly same with training)
  - Decode: s = 1, GEMM degenerates to GEMV
- Memory
  - New: KV cache
- Communication
  - mostly same with training

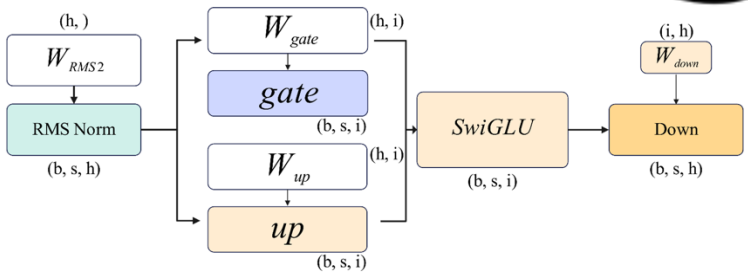
Our focus:

how batch size b changes the picture?

Potential Bottleneck in Serving

large b



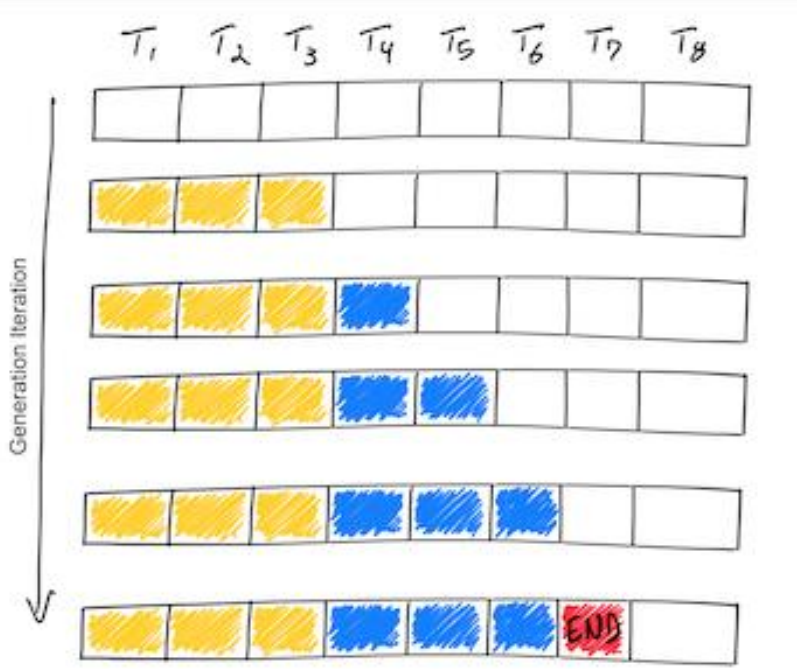


- Compute:
  - Prefill:
    - Different prompts have different length: how to batch?
  - Decode
    - Different prompts have different, unknown #generated tokens
- Memory
  - New: KV cache size grows with b
  - Solution: paged attention

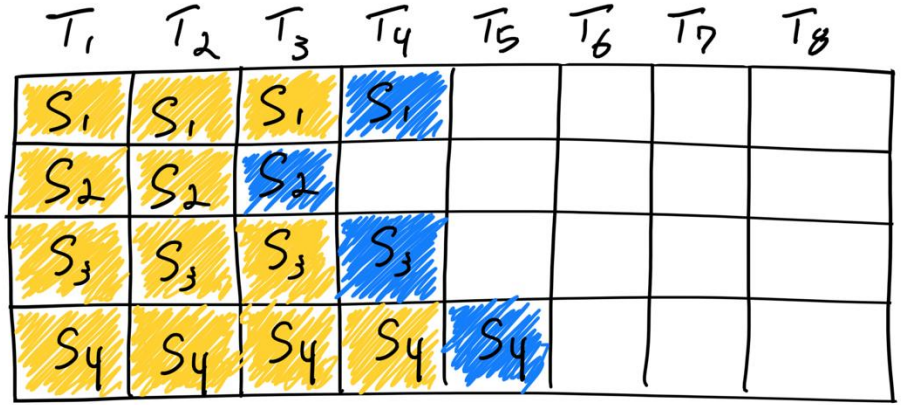
Agenda

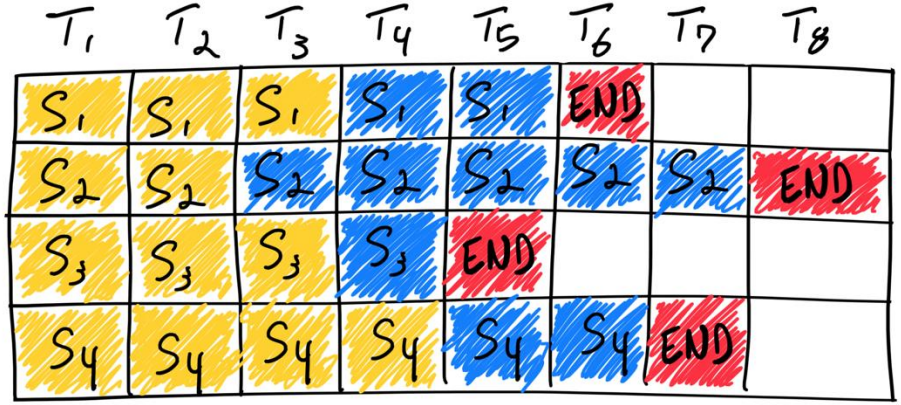
- LLM Serving
- Continuous Batching
- Disaggregated prefill and decode
- DistServe/Dynamo: Key Design and Implementations
- Systems and Models based on Disaggregation

LLM Decoding Timeline



Batching Requests to Improve GPU Performance

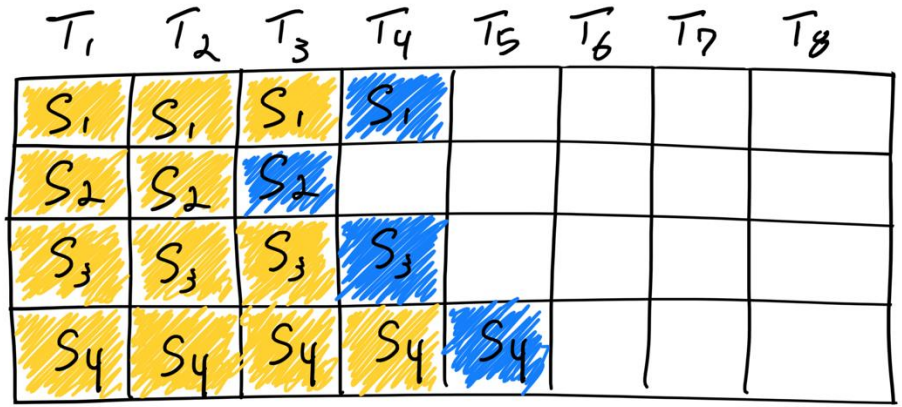


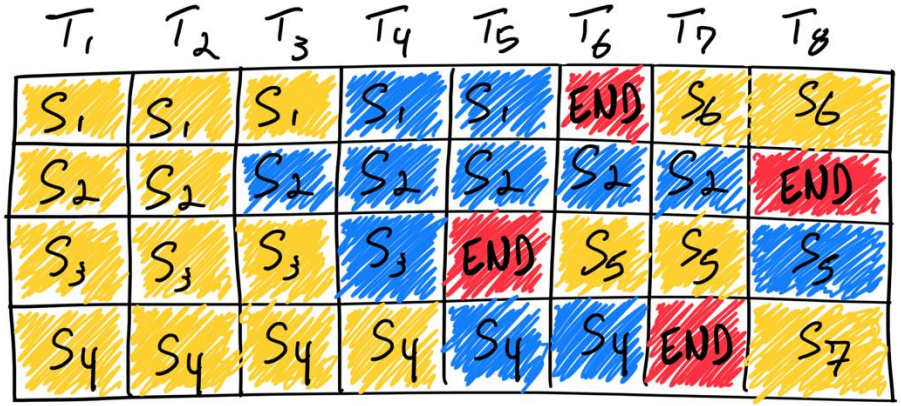


Issues with static batching:

- Requests may complete at different iterations
- Idle GPU cycles
- New requests cannot start immediately

Continuous Batching



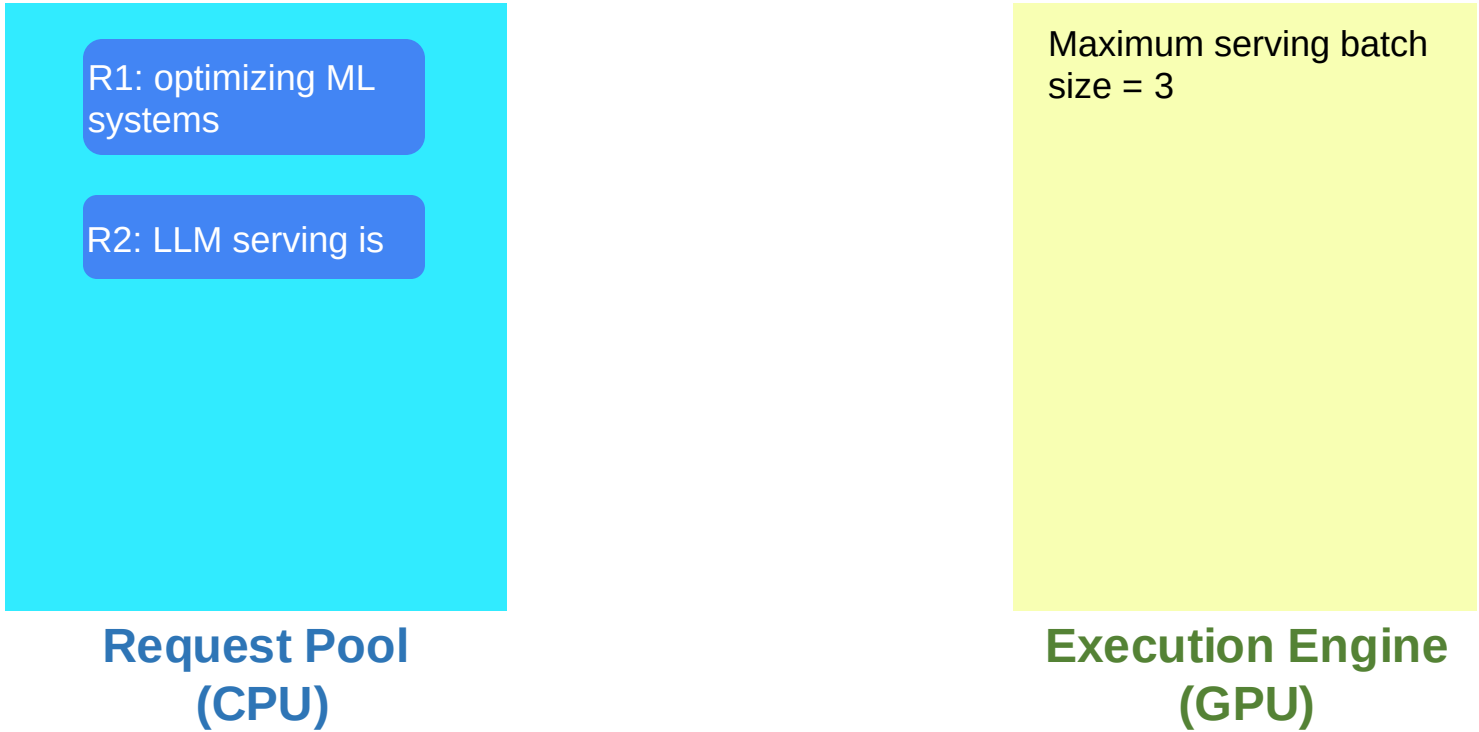


Benefits:

- Higher GPU utilization
- New requests can start immediately

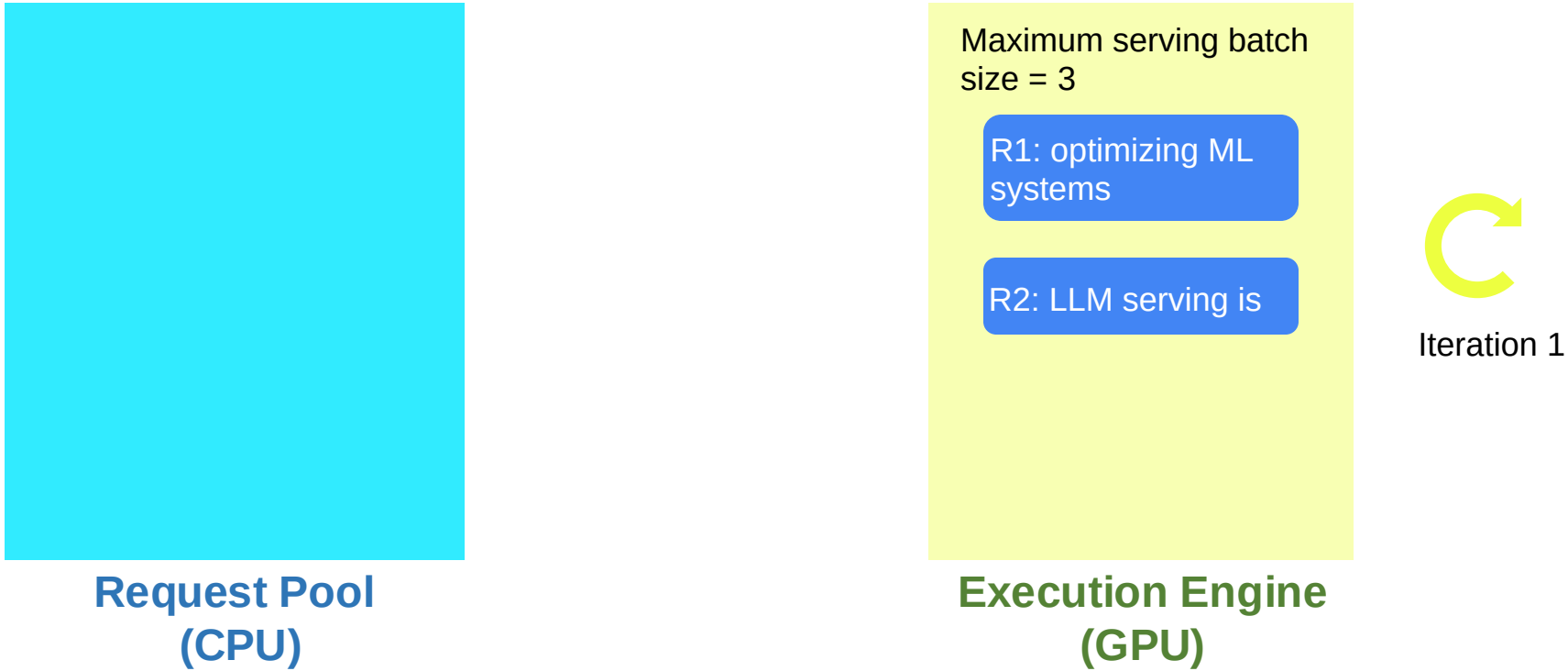
Continuous Batching Step-by-Step

- Receives two new requests R1 and R2



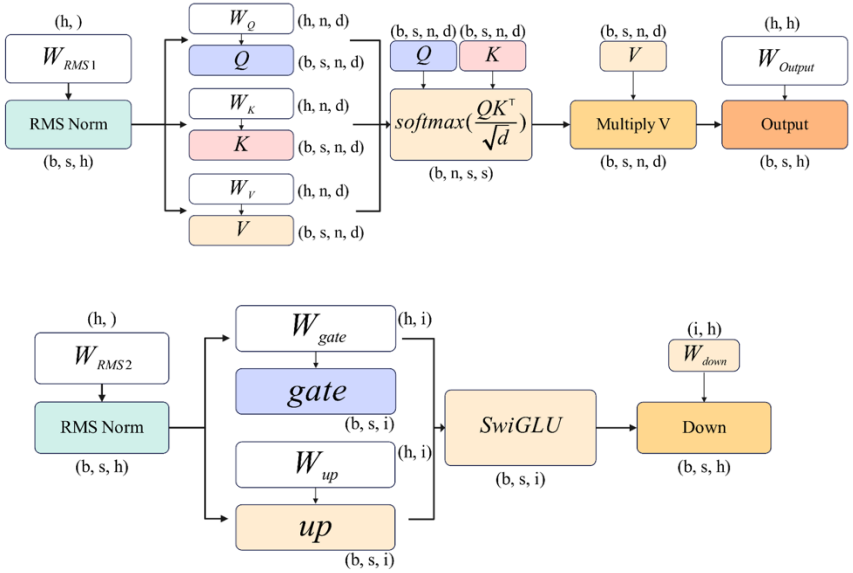
Continuous Batching Step-by-Step

- Iteration 1: decode R1 and R2



Continuous Batching Step-by-Step

- Iteration 1: decode R1 and R2



Q: How to batch these?

Maximum serving batch size = 3

R1: optimizing ML systems

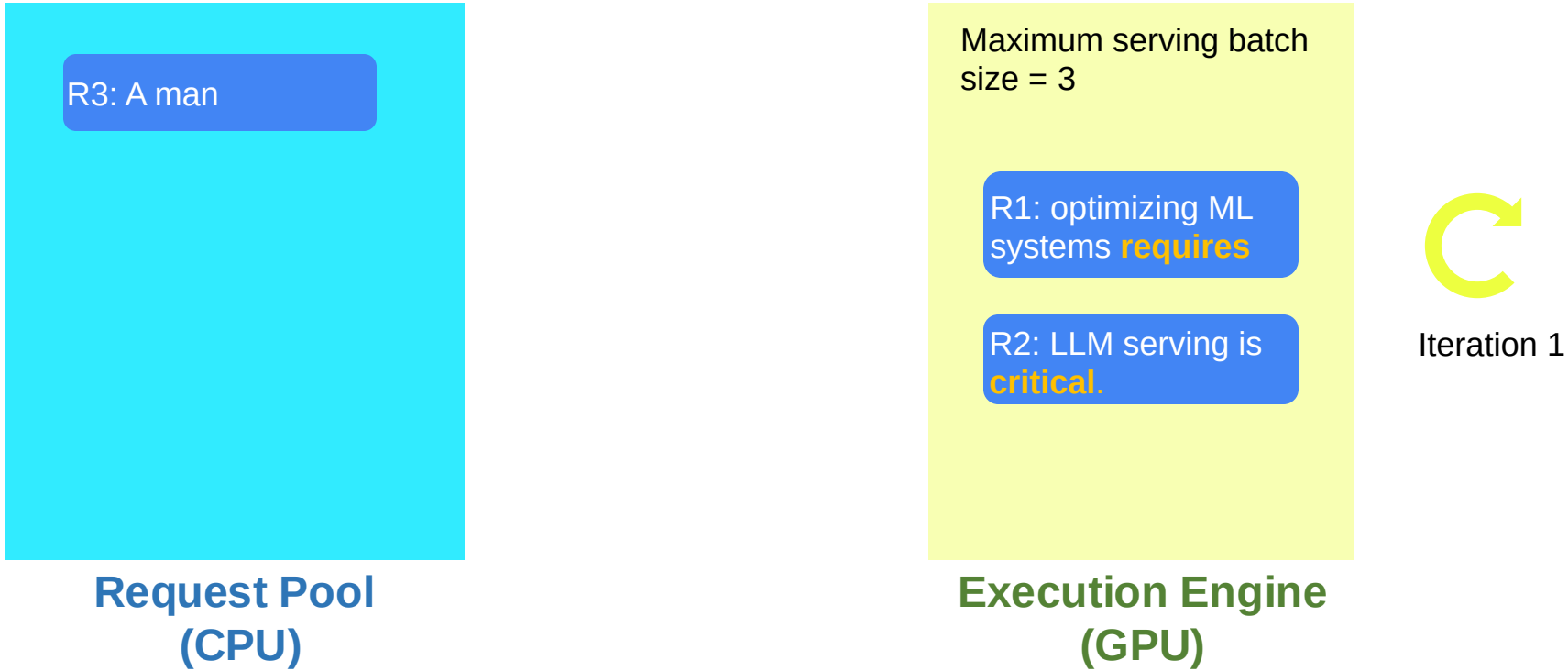
R2: LLM serving is

Iteration 1

Execution Engine (GPU)

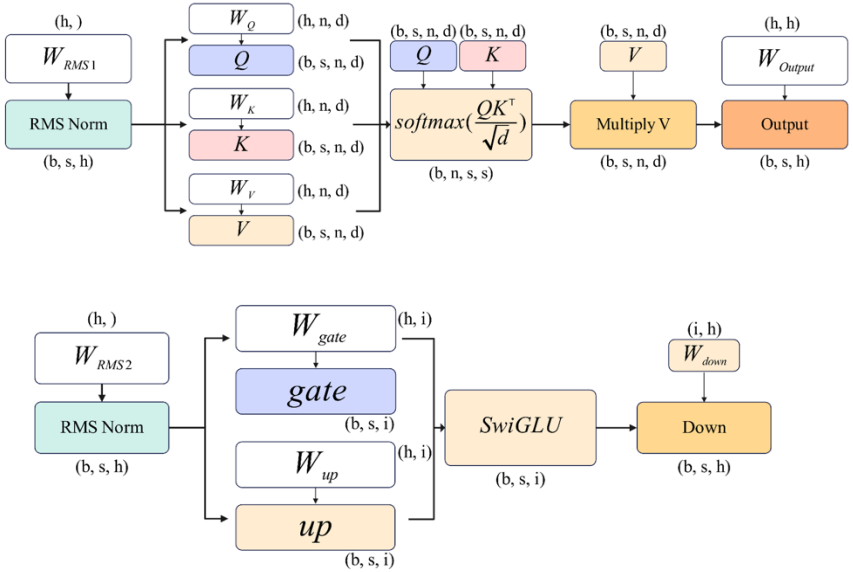
Continuous Batching Step-by-Step

- Receive a new request R3; finish decoding R1 and R2



Continuous Batching Step-by-Step

- Receive a new request R3; finish decoding R1 and R2



Q: How to batch these?

Maximum serving batch size = 3

R1: optimizing ML systems **requires**

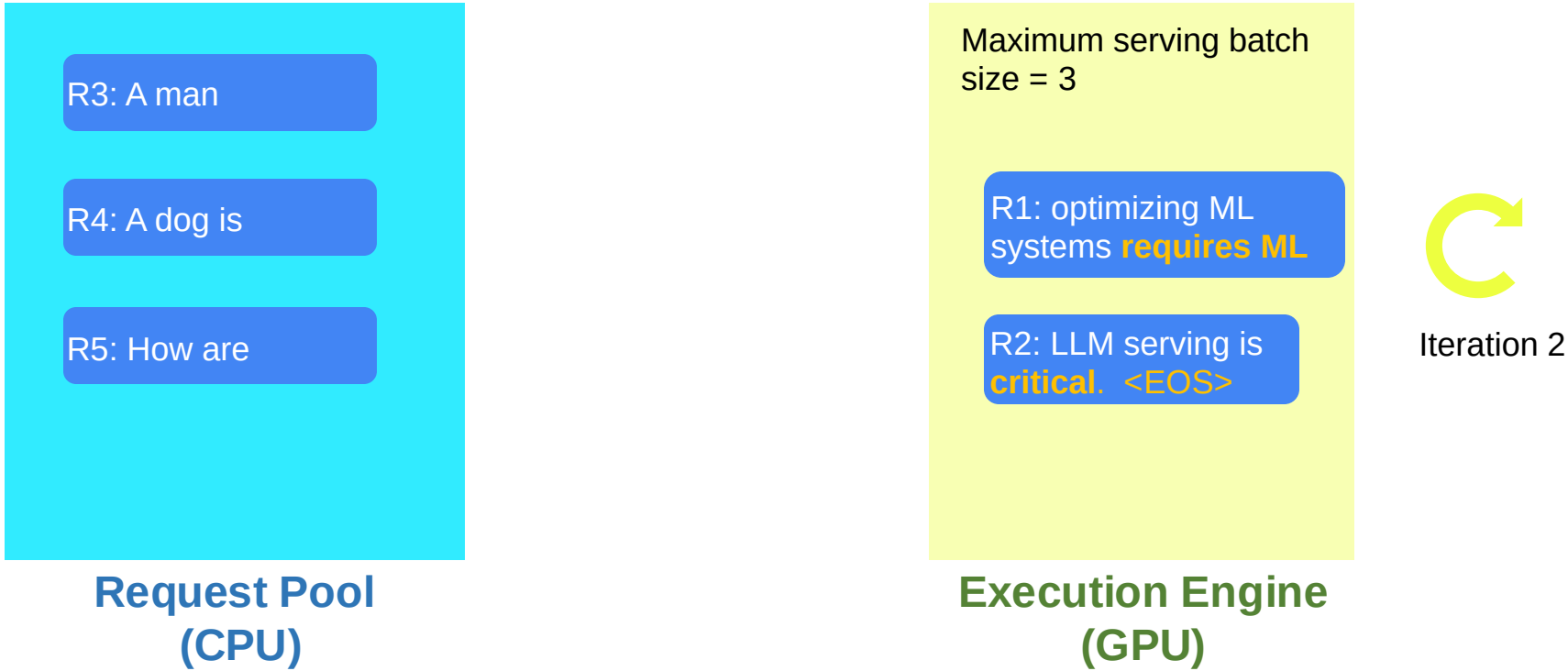
R2: LLM serving is **critical.**

Iteration 1

Execution Engine (GPU)

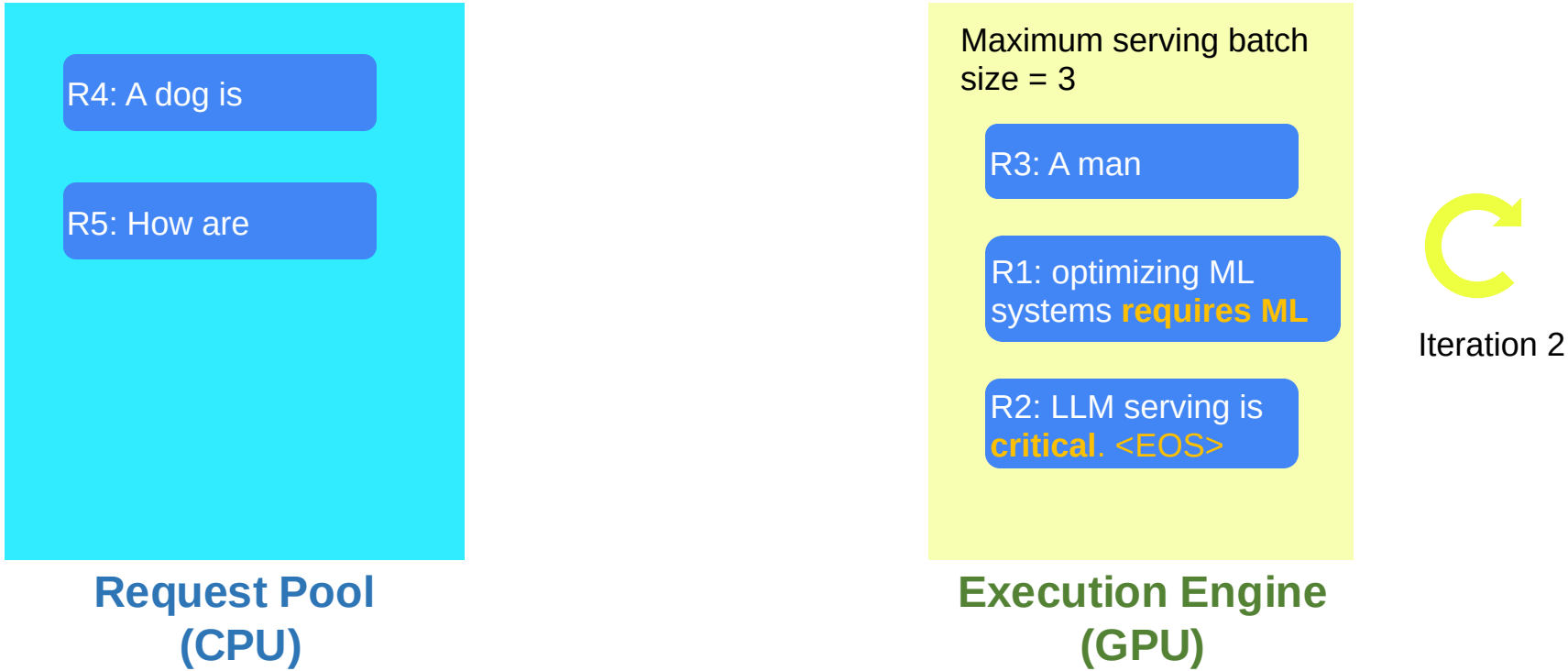
Traditional Batching

- Receive a new request R3; finish decoding R1 and R2



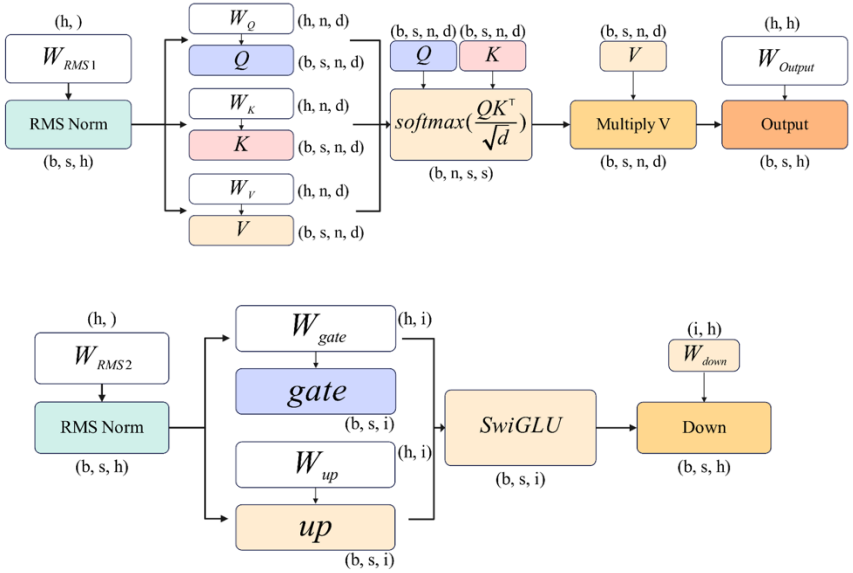
Continuous Batching

- Iteration 2: decode R1, R2, R3; receive R4, R5; R2 completes



Continuous Batching

- Iteration 2: decode R1, R2, R3; receive R4, R5; R2 completes



Q: How to batch these?

Maximum serving batch size = 3

R3: A man

R1: optimizing ML systems **requires ML**

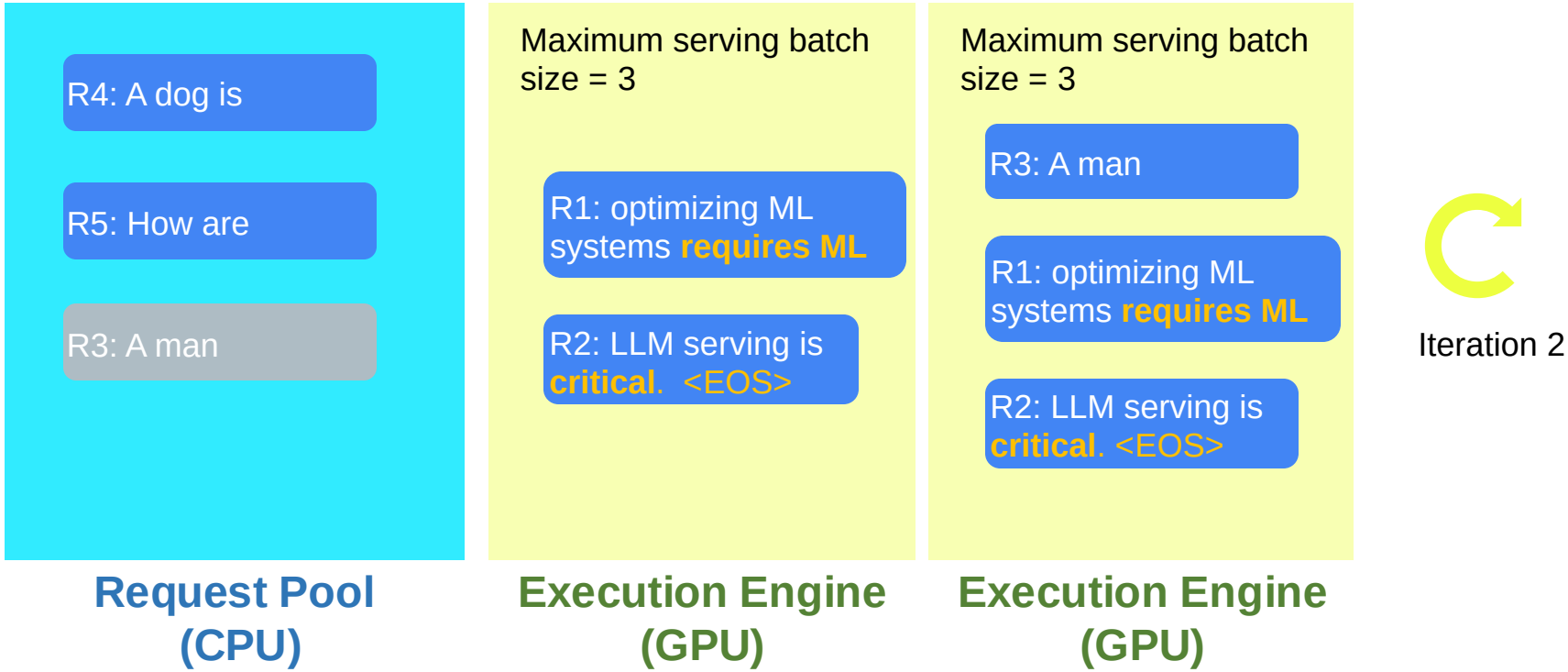
R2: LLM serving is **critical** <EOS>

Iteration 2

Execution Engine (GPU)

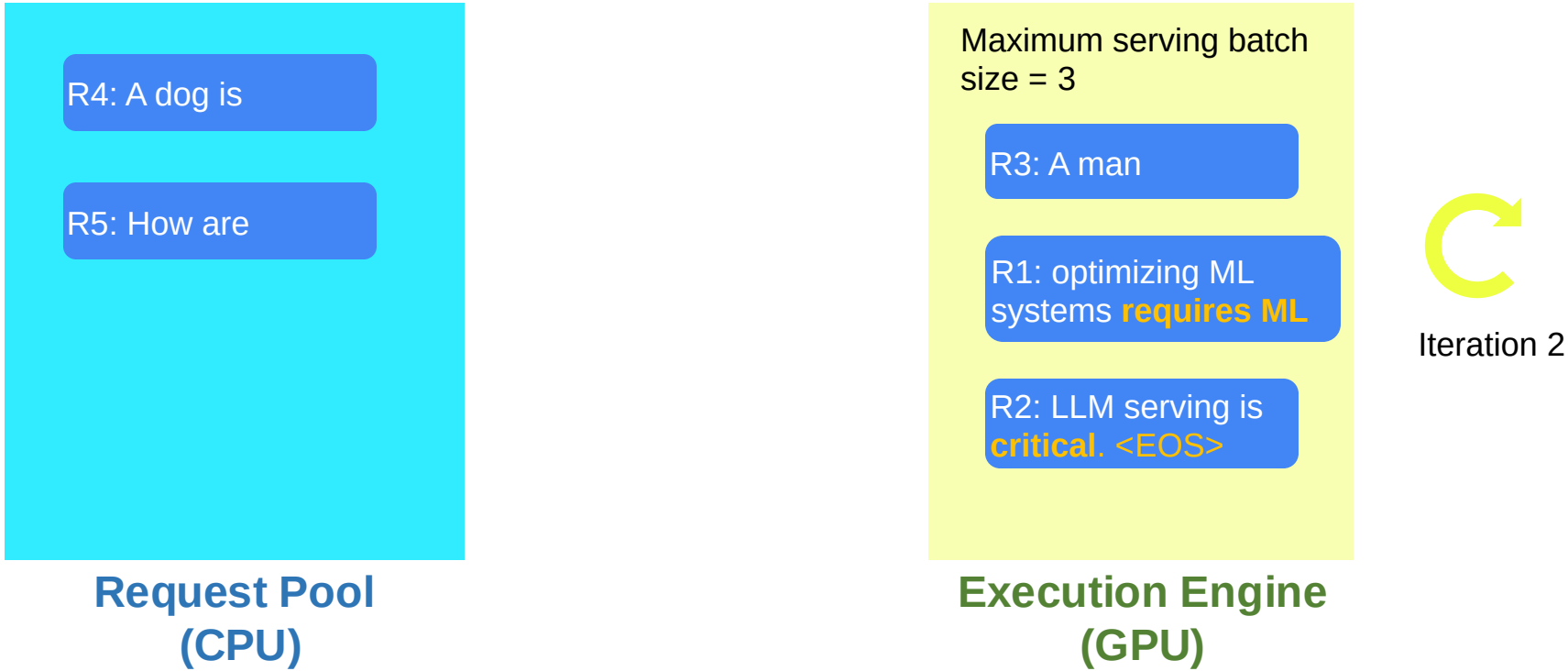
Traditional vs. Continuous Batching

- Iteration 2: decode R1, R2, R3; receive R4, R5; R2 completes



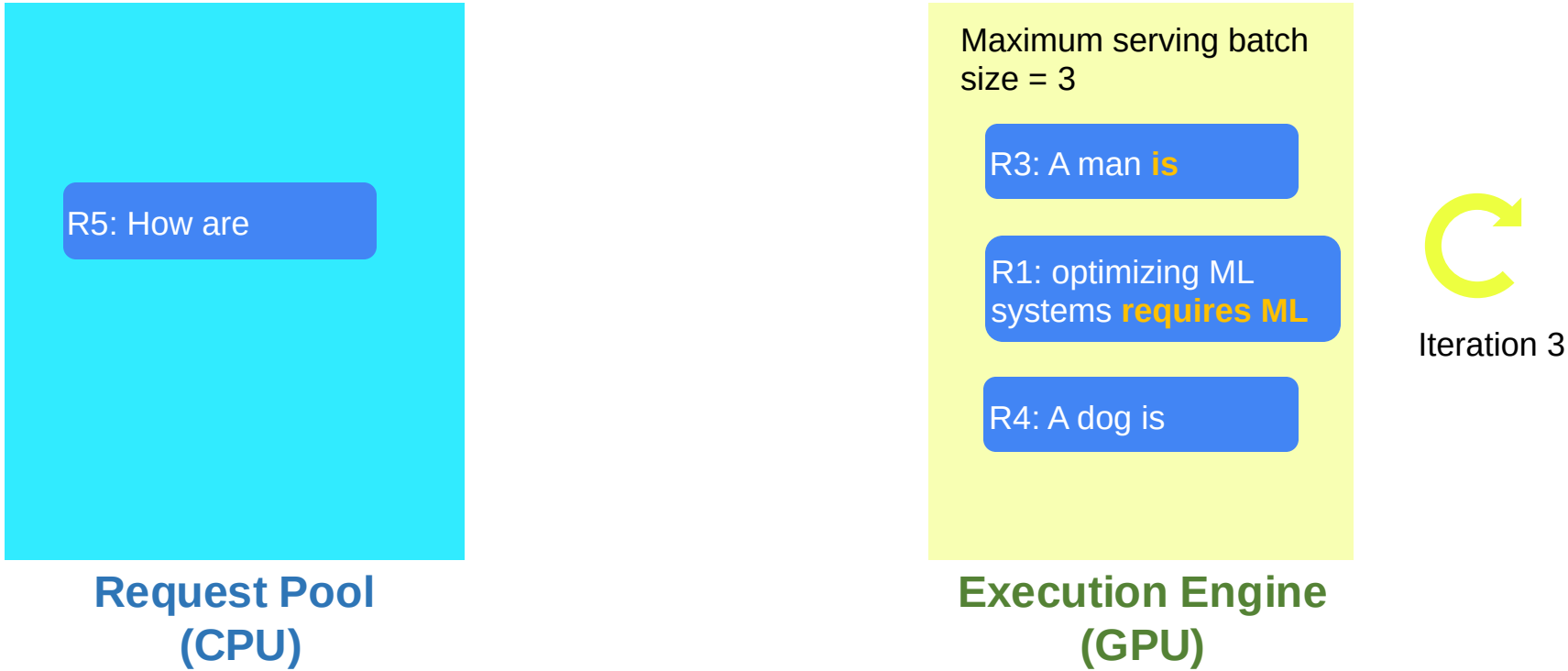
Continuous Batching

- Iteration 2: decode R1, R2, R3; receive R4, R5; R2 completes



Continuous Batching Step-by-Step

- Iteration 3: decode R1, R3, R4



Summary: Continuous Batching

- Handle early-finished and late-arrived requests more efficiently
- Improve GPU utilization
- Continuous Batching can improve the throughput by 10x compared to static batching

Agenda

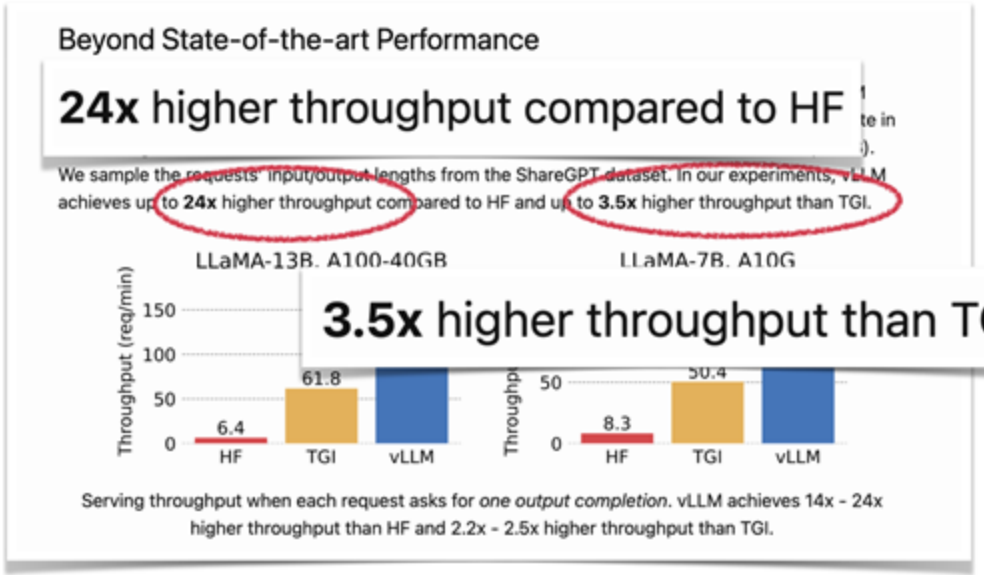
- LLM Serving
- Continuous Batching
- Disaggregated prefill and decode
- DistServe/Dynamo: Key Design and Implementations
- Systems and Models based on Disaggregation

LLM System Today Optimize **Throughput**

vLLM

DeepSpeed MII

NVIDIA



Motivation: Applications have Diverse SLO

- **TTFT**

  Time to first token

  Initial response time

- **TPOT**

  Time per output token

  Average time between two subsequent generated tokens

Chatbot

Fast initial response

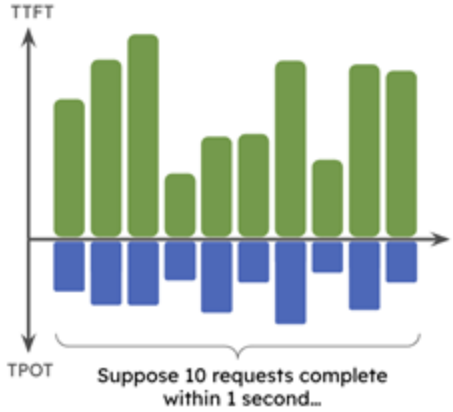
Human reading speed (P99 latency = 250ms)

Summarization

User can tolerate longer initial response

Data output generation (P99 latency = 35ms)

High Throughput ≠ High Goodput



Throughput = 10 rps

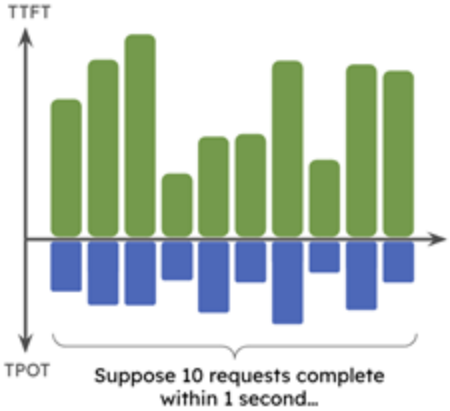
= completed request / time

High Throughput

System

...

High Throughput ≠ High Goodput

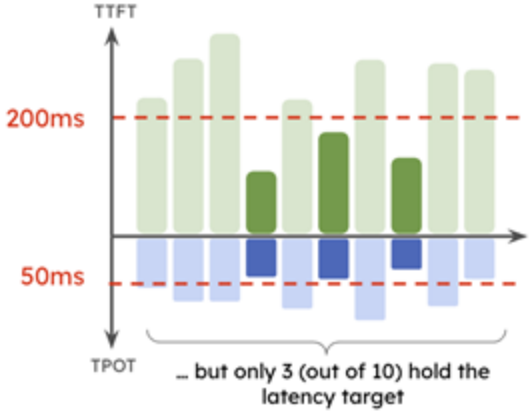


Throughput = 10 rps
= completed request / time

High Throughput
System

...

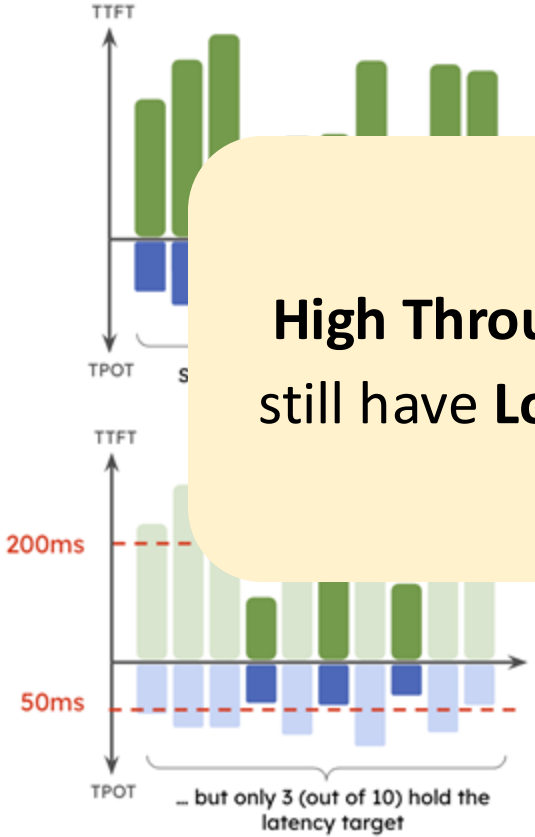
under SLO criteria



Goodput = 3 rps
= completed request **within SLO** / time

can have
Low Goodput!

High Throughput ≠ High Goodput



High Throughput

High Throughput can still have **Low Goodput**

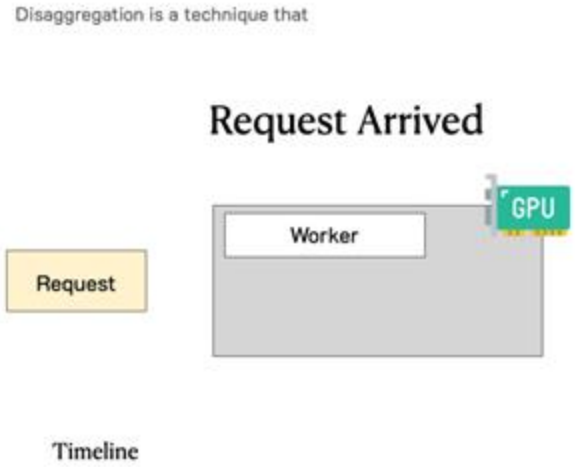
⇒ Poor UX 💔

Low Goodput!

Goodput = 3 rps

= completed request **within SLO** / time

Recall: Continuous Batching



Prefill and Decode have Distinct Characteristics

- **Prefill**

  Compute-bound

  One prefill saturates compute.

- **Decode**

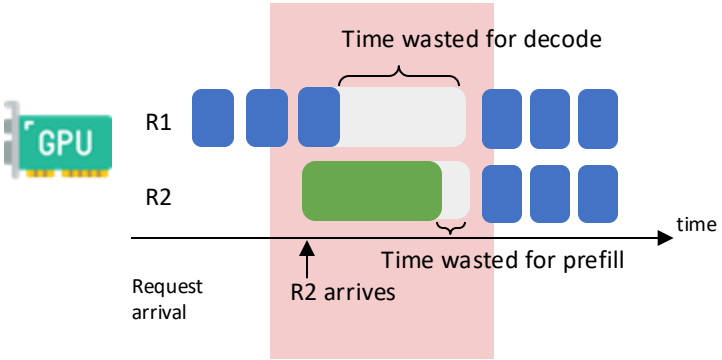
  Memory-bound

  Must batch a lot of requests together to saturate compute

Continuous Batching Cause Interference

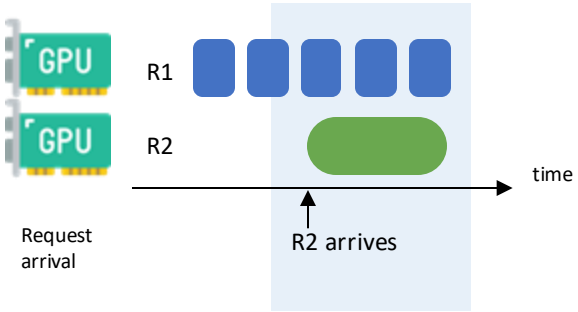
Continuous Batching

Batch R1 and R2 together in 1 GPU



Separate prefill / decode

R1 and R2 in separate GPUs

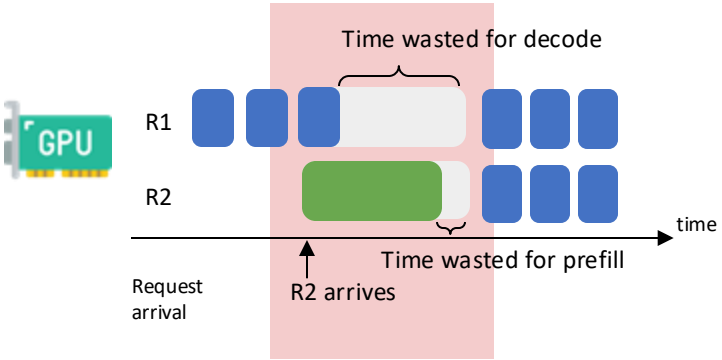


No Interference

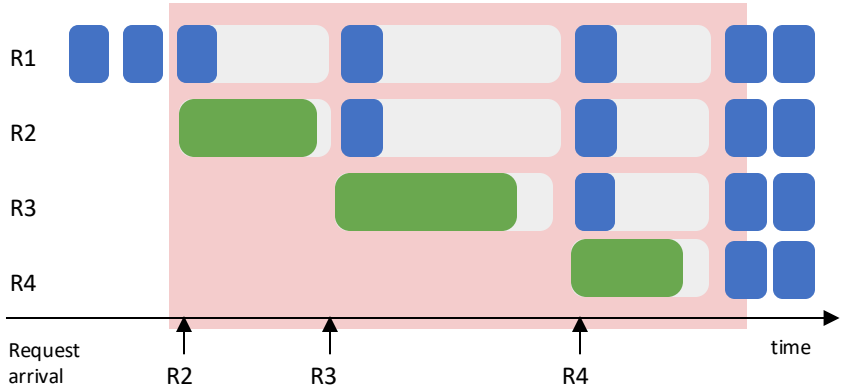
wasted time

Continuous Batching Cause Interference

Continuous Batching
Batch R1 and R2 together in 1 GPU

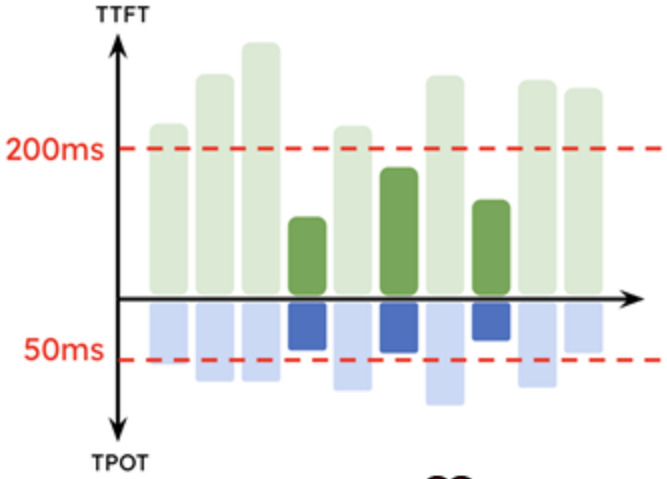


Continuous Batching
Batch R1~R4 together in 1 GPU



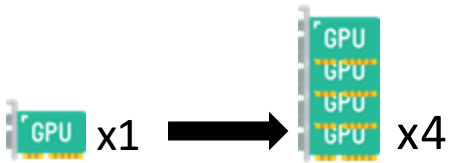
wasted time

Colocation → Overprovision Resource to meet SLO



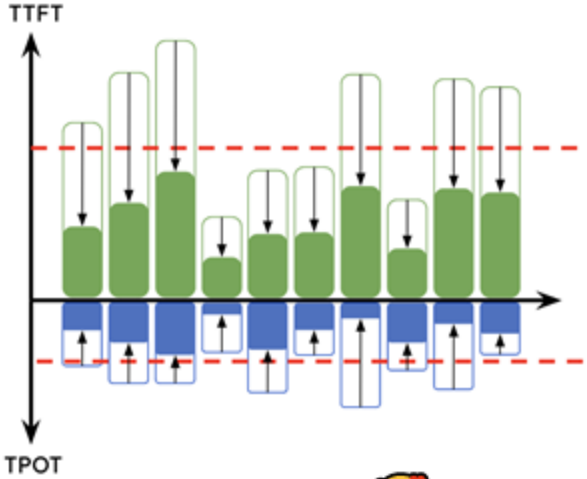
Poor UX 💔

lower cost 💰



add more GPU

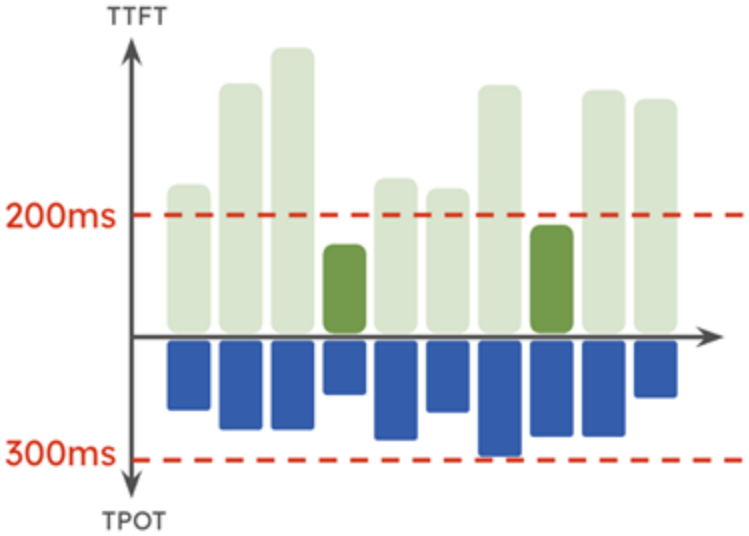
⚖️



Good UX 😈

Higher cost 💰💰💰💰

Colocation → Coupled Parallelism

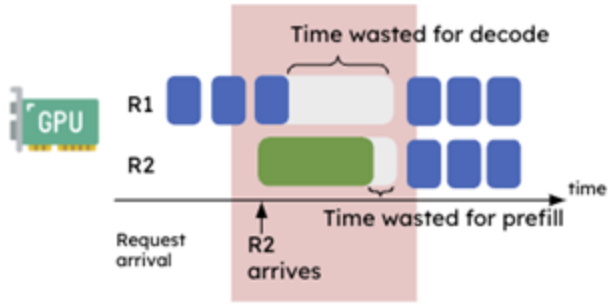


TTFT tight, TPOT loose

	PP	TP	DP
• Prefill	😐	❤️	❤️
• Decode	❤️	😐	😐

Prefill and Decode have different preferences

Summary: Problems caused by Colocation

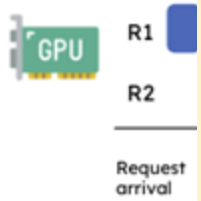


Continuous Batching Cause Interference

	PP	TP	DP
• Prefill	😐	❤️	❤️
• Decode	❤️	😐	😐

Coupled Parallelism Strategy

Summary: Problems caused by Colocation



Continuous Batching Cause Interference

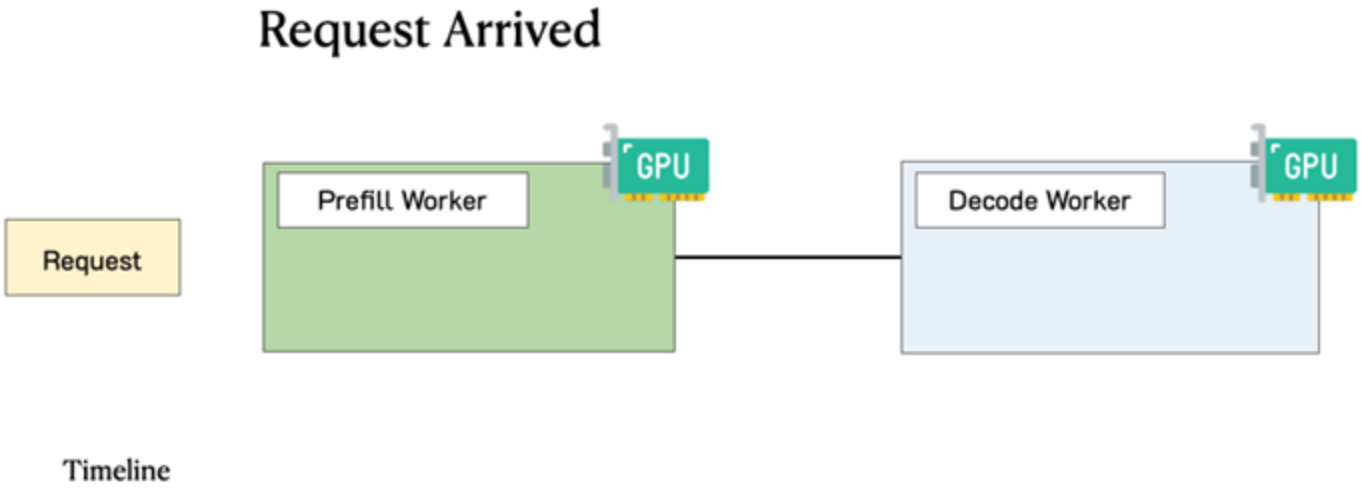
Is there a better way to achieve better **Goodput per GPU**?

TP	DP
❤️	❤️
😐	😐

Coupled Parallelism Strategy

Solution: Disaggregating Prefill and Decode

Disaggregation is a technique that



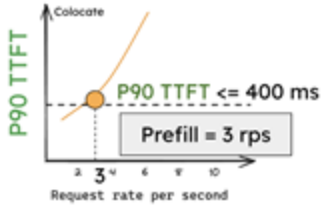
Opportunity: Disaggregating Prefill and Decoding

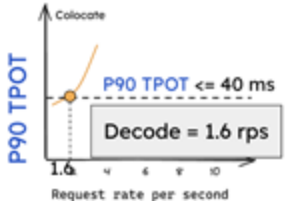
- Prefill-Decoding interference is immediately eliminated
- Naturally divide the SLO satisfaction problem into two optimizations:
  - Prefill instance optimizes for TTFT.
  - Decoding instance optimizes for TPOT.
  - Choose the most suitable parallelism and resource allocation for each phase.

Disaggregation achieves better goodput

Colocate

1 GPU for both Prefill and Decode





Max System goodput

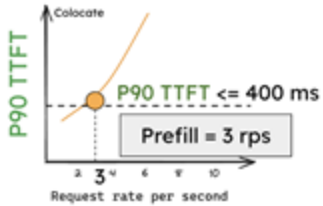
= Min(Prefill, Decode)

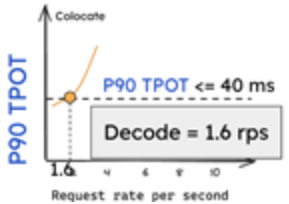
= 1.6 rps / GPU

Disaggregation achieves better goodput

Colocate

1 GPU for both Prefill and Decode





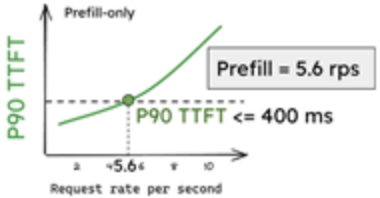
Max System goodput

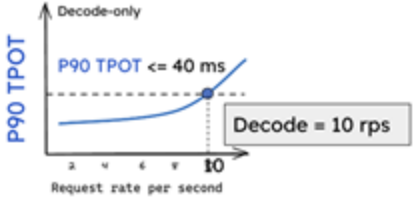
= Min(Prefill, Decode)

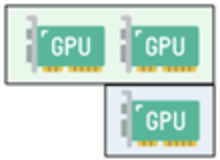
= 1.6 rps / GPU

Disaggregate (2P1D)

2 GPU for Prefill + 1 GPU for Decode







Disaggregate (2P1D) goodput

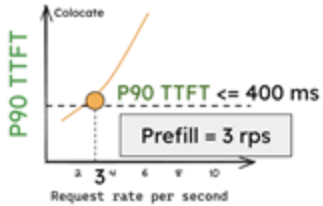
= Min (5.6 x **2**, 10) rps / 3 GPU

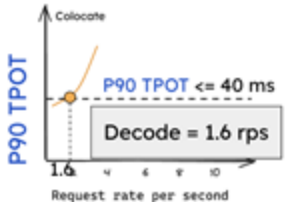
= 3.3 rps / GPU

Disaggregation achieves better goodput

Colocate

1 GPU for both Prefill and Decode





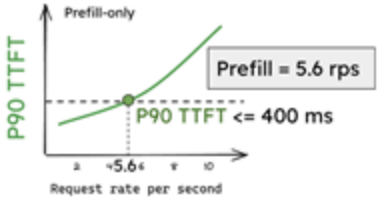
Max System goodput

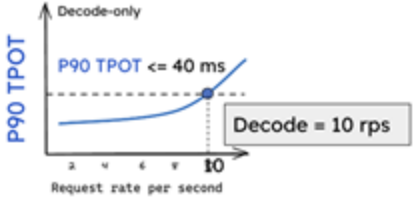
= Min(Prefill, Decode)

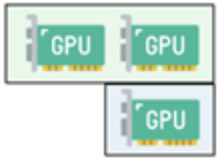
= 1.6 rps / GPU

Disaggregate (2P1D)

2 GPU for Prefill + 1 GPU for Decode







Disaggregate (2P1D) goodput

= Min (5.6 x **2**, 10) rps / 3 GPU

= 3.3 rps / GPU

Simple Disaggregation achieves **2x** goodput (per GPU)

Challenges of Disaggregation

- Communication overhead for KV-Cache transmission
- The optimization target — per-GPU goodput, is difficult to optimize:
  - The workload pattern
  - SLO requirements
  - Parallelism strategies
  - Resource allocation
  - Network bandwidth

Agenda

- LLM Serving
- Continuous Batching
- Disaggregated prefill and decode
- DistServe: Key Design and Implementations
- Systems and Models based on Disaggregation

Core Problems

XPYD

P1 - Placement: Solve X, Y given workload requirement that maximizes GPU goodput

P2 - Communication: Minimize the communication of KV Cache between XP and YD

DistServe Design Overview

Definition of Placement:

1. parallelism strategy for prefill/decoding instance
2. the number of each instance to deploy
3. how to place them onto the physical cluster

Featured algorithms:

1. Placement for High Node-Affinity Cluster
2. Placement for Low Node-Affinity Cluster
3. Online Scheduling Optimization

Placement for High Bandwidth Cluster

Assumption:

- Nodes are connected with high bandwidth network, e.g., InfiniBand.

Observation:

- We can optimize prefill and decoding instances separately.

Algorithm Sketch:

- Use simulation to measure the goodput for a specific parallelism config.
- Obtain the optimal parallelism config for each phase.
- Use replication to match the overall traffic.

Placement for Low Bandwidth Cluster

Assumption:

- GPUs inside one node are connected with NVLINK.

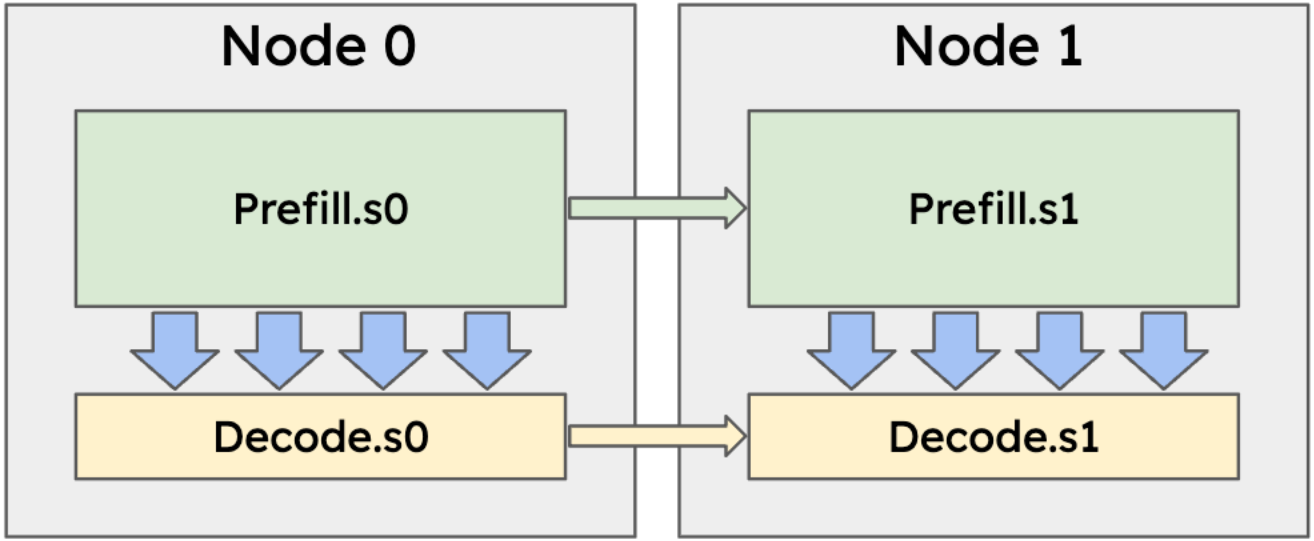
Observation:

- KV-Cache transmission only happens between the same layer

Algorithm Sketch:

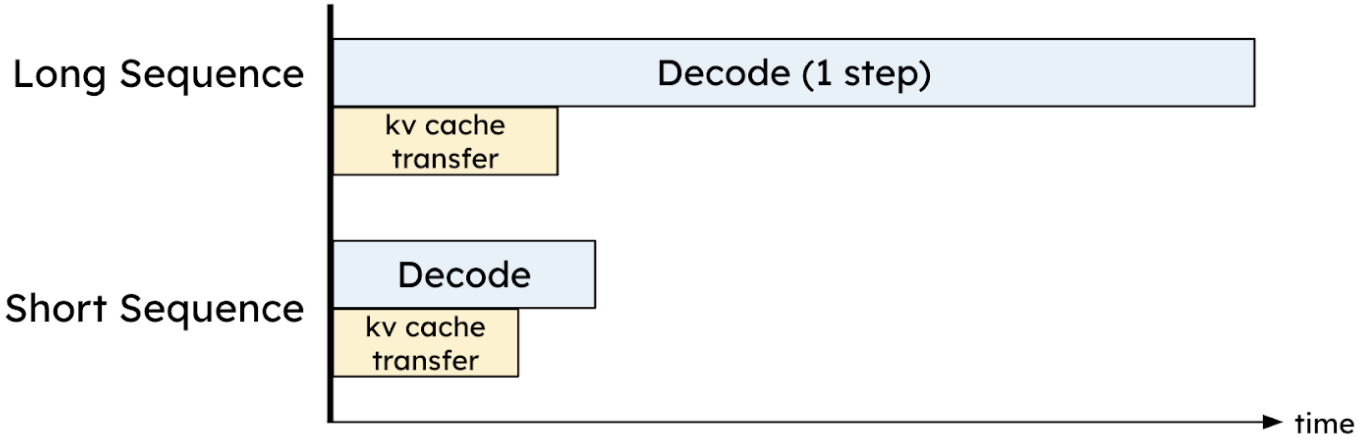
- Similar to the previous one but constraint the same stage of prefill/decoding instances to be on the same node.

Example Placement

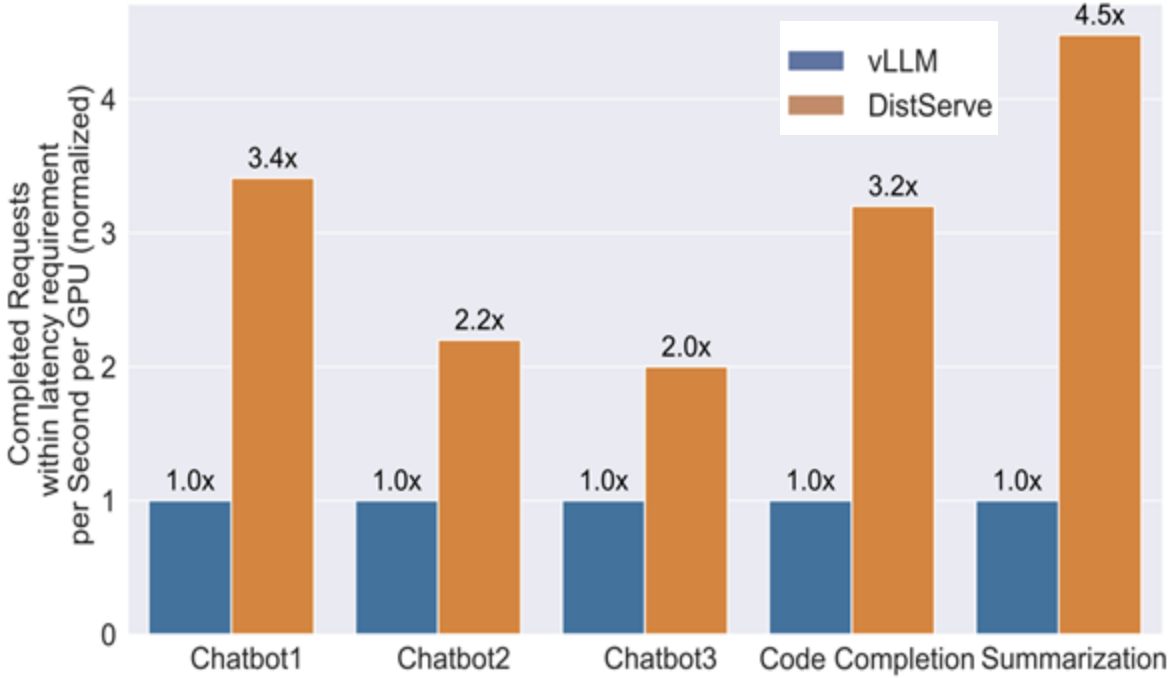


How Expensive is Communication of KV Caches?

- Assume: 175B Model, A100 GPUs
- If using PCIe, latency of KV Cache transfer < time of a decode step.
- NVLink + DistServe algorithm will do better



Evaluation



Achieves 2.0x - 4.48x compared to vanilla vLLM

- Chatbot: 2.0 - 3.4x
- Code Completion: 3.2x
- Summarization: 4.5x

DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving
Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, Hao Zhang

Continuous batching vs. disaggregation

- It seems we are going back and forth

- Actually no:
  - Continuous batching: improve GPU utilization hence throughput
  - Disaggregation: to address goodput -- throughput s.t. SLOs

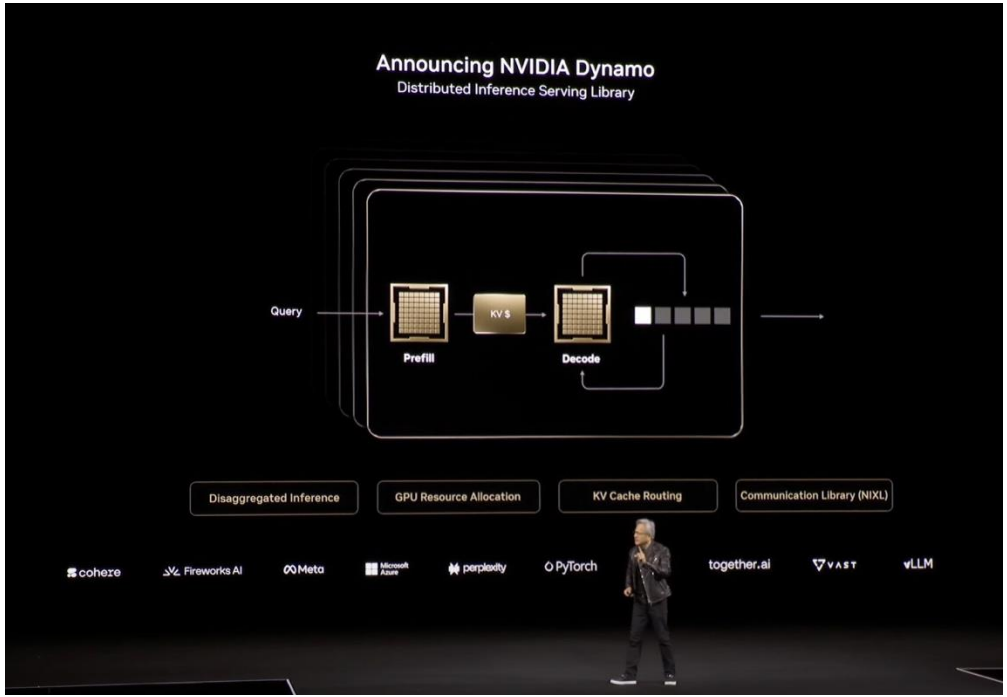
- Also, key insights of CB carries to disaggregation
  - Batch attentions and MLPs differently
  - Exit finished request and pick up new request asap

Disaggregation Fun History

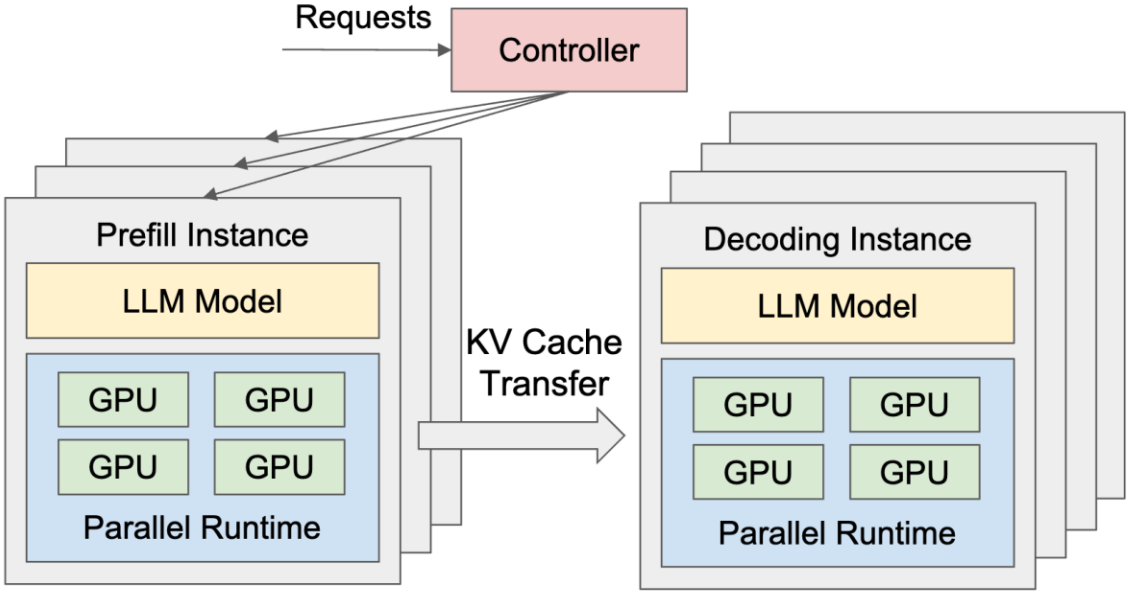
- 2023 end: Published and open sourced at UCSD (Hao's lab), with a concurrent work from Microsoft (not open source)
- 2024: OSS integration is slower compared to CB/paged attention as no significant gain was observed
- 2024: Yet, silently become the chosen architecture replacing continuous batching at large scale in large cooperates (e.g., Bytedance, Google)
- 2025: Deepseek-v3 uses prefill-decode disaggregation combined with different parallelisms for prefill and decoding instances.

Disaggregation Fun History

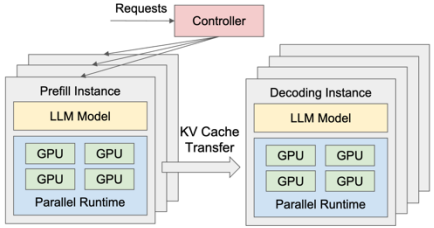
- 2025: Nvidia GTC Keynote



DistServe Architecture



Key Components (Delta from vLLM)




```python
class ContextStageScheduler(ABC):
    """
    ContextStageScheduler: The abstract class for a context scheduler.

    It should maintain all the requests in the current systems, and support two basic ops:
        - add_request: Add a newly arrived request into the waiting queue
        - get_next_batch_and_pop: Get the next batch for the context stage, and
          pop the requests in the batch from the waiting queue.

    This scheduler is much simpler than DecodingStageScheduler since one request
    will only be processed by one context stage.
    """
```

```python
class ContextStageLLMEngine(SingleStageLLMEngine):
    def _get_scheduler(self) -> ContextStageScheduler:
        return get_context_stage_scheduler(
            self.sched_config,
            self.parallel_config,
            self.block_manager
        )
```

```python
class DecodingStageScheduler(ABC):
    """The abstract class for a decoding stage scheduler.
    It should maintain all the requests in the current systems and their
    runtime statistics which are needed for scheduling. Before each iteration
    begins, the LLMEngine will call get_next_batch() method to get a
    BatchedRequets object for the next iteration. After each iteration ends,
    the LLMEngine will call the pop_finished_requests() method to get the
    finished requests in the current iteration.
    """
```

```python
class DecodingStageLLMEngine(SingleStageLLMEngine):
    def _get_scheduler(self) -> DecodingStageScheduler:
        return get_decoding_stage_scheduler(
            self.sched_config,
            self.parallel_config,
            self.block_manager,
            self._migrate_blocks
        )
```

Key Function: Decode Instances Migrate KV Caches

```
async def _migrate_blocks(
    self,
    migrating_req: MigratingRequest
) -> None:
    """
    Migrate one request from the context engine to the decoding engine

    This function will be called be the decoding stage scheduler

    This function performs the following steps:
    - Allocate blocks on the decoding engine's side
    - Transfer the blocks
    - Clear the blocks on the context engine's side
    """
```

On Optimizing the Communication of Model Parallelism. MLSYS'23

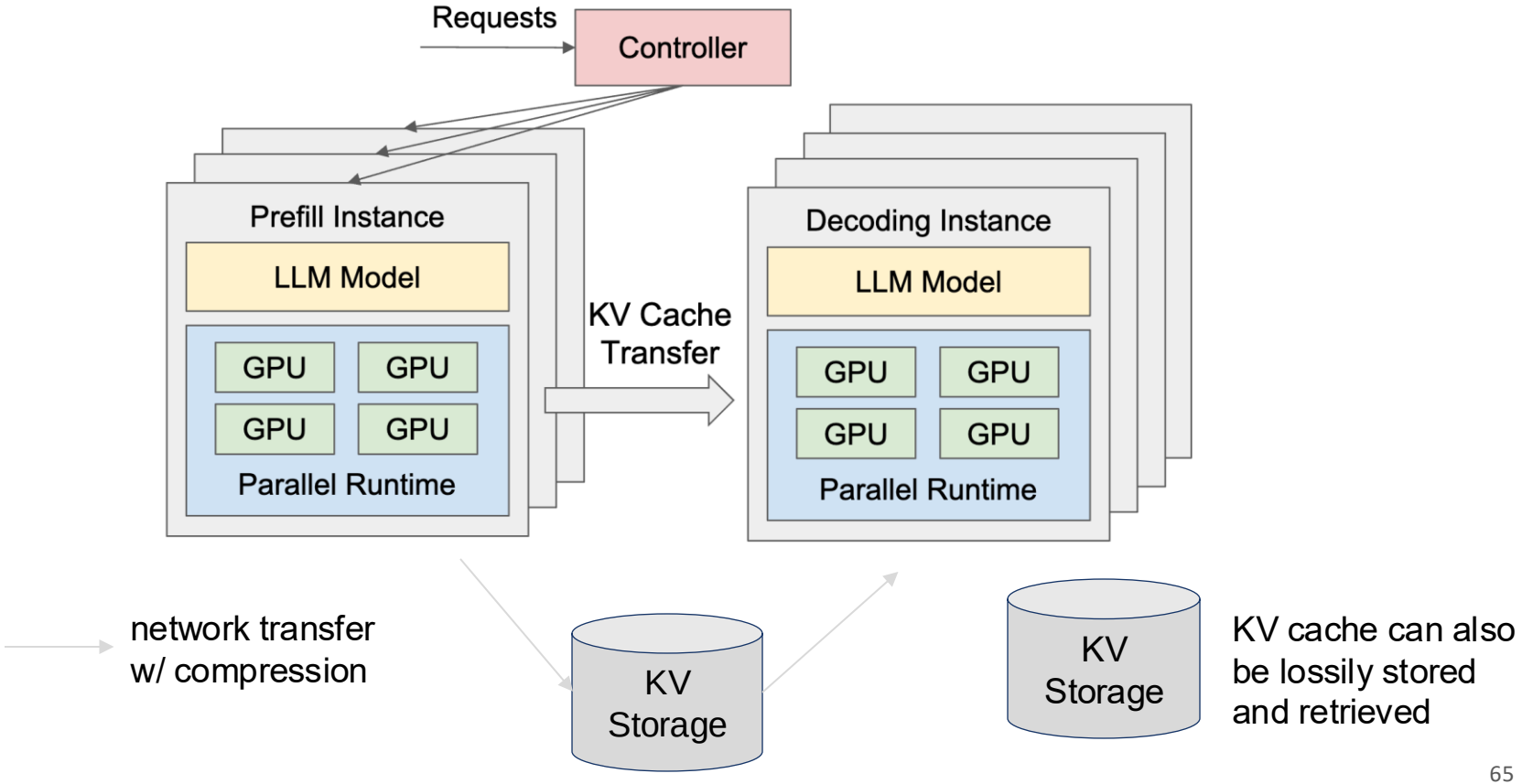
Agenda

- LLM Serving
- Continuous Batching
- Disaggregated prefill and decode
- DistServe/Dynamo: Key Design and Implementations
- Systems and Models based on Disaggregation

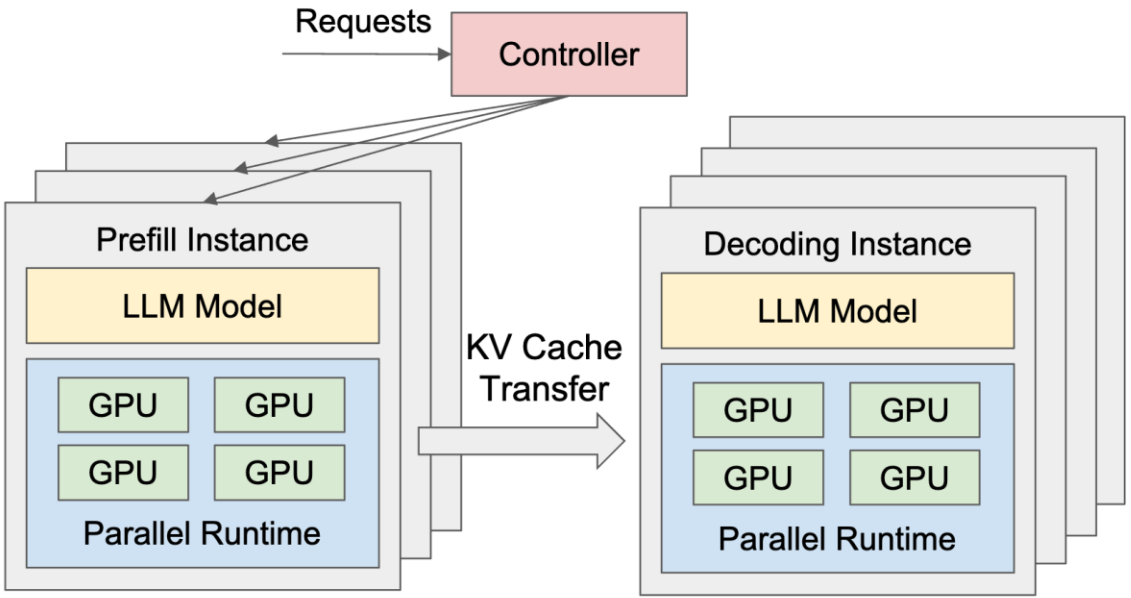
A Few New Models/Systems Build on DistServe

- LMCache/Cachegen
- DeepSeek-v3 Serving
- Nvidia Dynamo

LMCache/


Requests
Controller
Prefill Instance
LLM Model
GPU
GPU
GPU
GPU
Parallel Runtime
KV Cache Transfer
Decoding Instance
LLM Model
GPU
GPU
GPU
GPU
Parallel Runtime
network transfer w/ compression
KV Storage
KV Storage
KV cache can also be lossily stored and retrieved

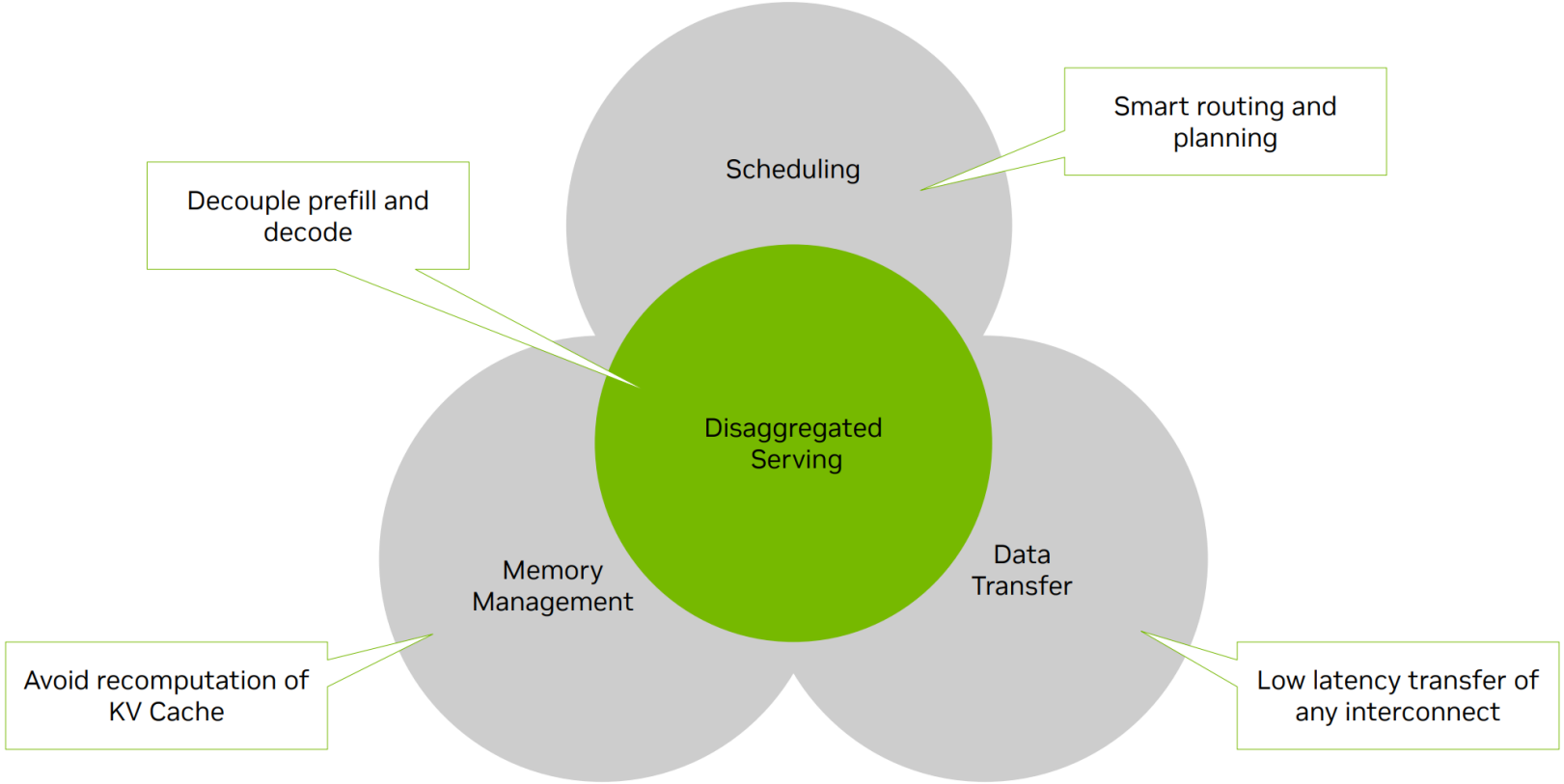
DeepSeek-V3: Disaggregation with Specialized Parallelisms



4 TP/SP + 8 DP
32EP in MoE
redundant experts

4 TP/SP + DP80
EP320

Nvidia Dynamo: Nvidia’s Implementation



Conclusion

- **From continuous batching to disaggregation**
- **From Throughput to Goodput**
- **Disaggregation** is effective to optimize **goodput**!
- **Disaggregation** achieves 2.0x - 4.48x