LLM SERVING AND ALIGNMENT

Team 6
Fengyu Gao, Shunqiang Feng, Wei Shen, Zihan Zhao

SGLang: Efficient Execution of Structured Language Model Programs

Zihan Zhao (rxy6cc)

vLLM

43.3k ★

```python
# In this script, we demonstrate how to pass input to the chat method:
conversation = [
    {
        "role": "system",
        "content": "You are a helpful assistant"
    },
    {
        "role": "user",
        "content": "Hello"
    },
    {
        "role": "assistant",
        "content": "Hello! How can I assist you today?"
    },
    {
        "role": "user",
        "content":
        "Write an essay about the importance of higher education.",
    },
]
outputs = llm.chat(conversation, sampling_params, use_tqdm=False)
print_outputs(outputs)
```

Format the inputs **by hand**

Run the LLM with the inputs

Just like how you regularly code

SGLang

12.7k ★

```python
import sglang as sgl


@sgl.function
def multi_turn_question(s, question_1, question_2):
    s += sgl.user(question_1)
    s += sgl.assistant(sgl.gen("answer_1", max_tokens=256))
    s += sgl.user(question_2)
    s += sgl.assistant(sgl.gen("answer_2", max_tokens=256))


def single():
    state = multi_turn_question.run(
        question_1="What is the capital of the United States?",
        question_2="List two local attractions.",
    )

    for m in state.messages():
        print(m["role"], ":", m["content"])

    print("\n-- answer_1 --\n", state["answer_1"])
```

Define the interaction flow

Run the LLM directly from the flow!

A much better programming paradigm

vLLM vs. SGLang

- vLLM
  - A framework focusing on system efficiency

- SGLang
  - A framework focusing on system efficiency and programming efficiency

Motivations

- Lack of efficient systems
  - *vLLM was not built yet*
  - Redundant computations
  - Redundant memory usage
- Lack of programming efficiency
  - Convoluted code to start a server, expose an API, and run LLMs
    - LLM applications !== Any other applications (e.g. webapp)

Frontend Language

- Challenges
  - String manipulations
    - Prompt construction (e.g. roles, message, attachments, etc.)
  - Multimodality supports
    - Multimodal token placement (e.g. audio tokens, video tokens, image tokens, etc.)
  - Output parsing
    - Yes-or-no selection
    - Code extraction

Frontend Language

- Language primitives
  - Roles
    - `system / user / assistant`
  - Multimodal files
    - `image / video`
  - Control flow
    - `fork / select / gen`

```
dimensions = ["Clarity", "Originality", "Evidence"]
@function
def multi_dimensional_judge(s, path, essay):
  s += system("Evaluate an essay about an image.")
  s += user(image(path) + "Essay:" + essay)
  s += assistant("Sure!")
  # Return directly if it is not related
  s += user("Is the essay related to the image?")
  s += assistant(select("related", choices=["yes", "no"]))
  if s["related"] == "no": return
  # Judge multiple dimensions in parallel
  forks = s.fork(len(dimensions))
  for f, dim in zip(forks, dimensions):
    f += user("Evaluate based on the following dimension:" +
      dim + ". End your judgment with the word 'END'")
    f += assistant("Judgment:" + gen("judgment", stop="END"))
  # Merge the judgments
  judgment = "\n".join(f["judgment"] for f in forks)
  # Generate a summary and a grade. Return in the JSON format.
  s += user("Provide the judgment, summary, and a letter grade")
  s += assistant(judgment + "In summary," + gen("summary", stop=".")
                 + "The grade of it is" + gen("grade"))
  schema = r'\{"summary": "[\w\d\s]+\.", "grade": "[ABCD][+-]?"\}'
  s += user("Return in the JSON format.")
  s += assistant(gen("output", regex=schema))
state = multi_dimensional_judge.run(...)
print(state["output"])
```

Handle chat template and multi-modal inputs

Select an option with the highest probability

Fetch result; Use Python control flow

Runtime optimization: KV Cache Reuse (Sec. 3)

Multiple generation calls run in parallel

Fetch generation results

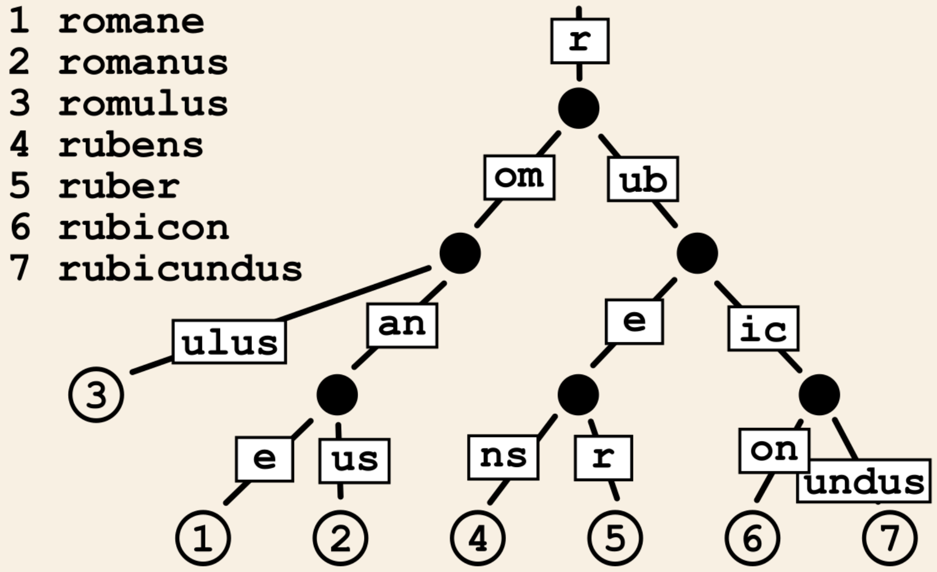
Runtime optimization: API speculative execution (Sec. 5)

Runtime optimization: fast constrained decoding (Sec. 4)

Run an SGLang program

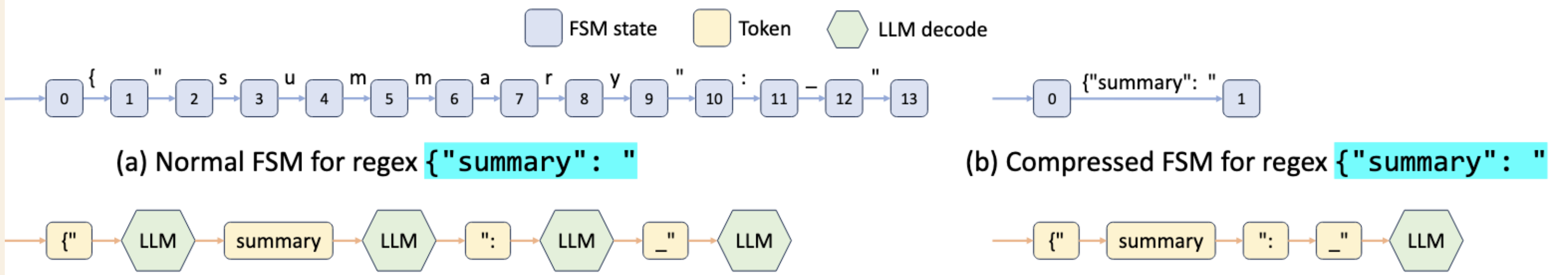
KV Cache Reuse

- RadixAttention
  - Prefix sharing
    - Radix tree + LRU cache
  - Prompt deduplication
    - Substring repetition discoveries



Accelerated Constrained Decoding

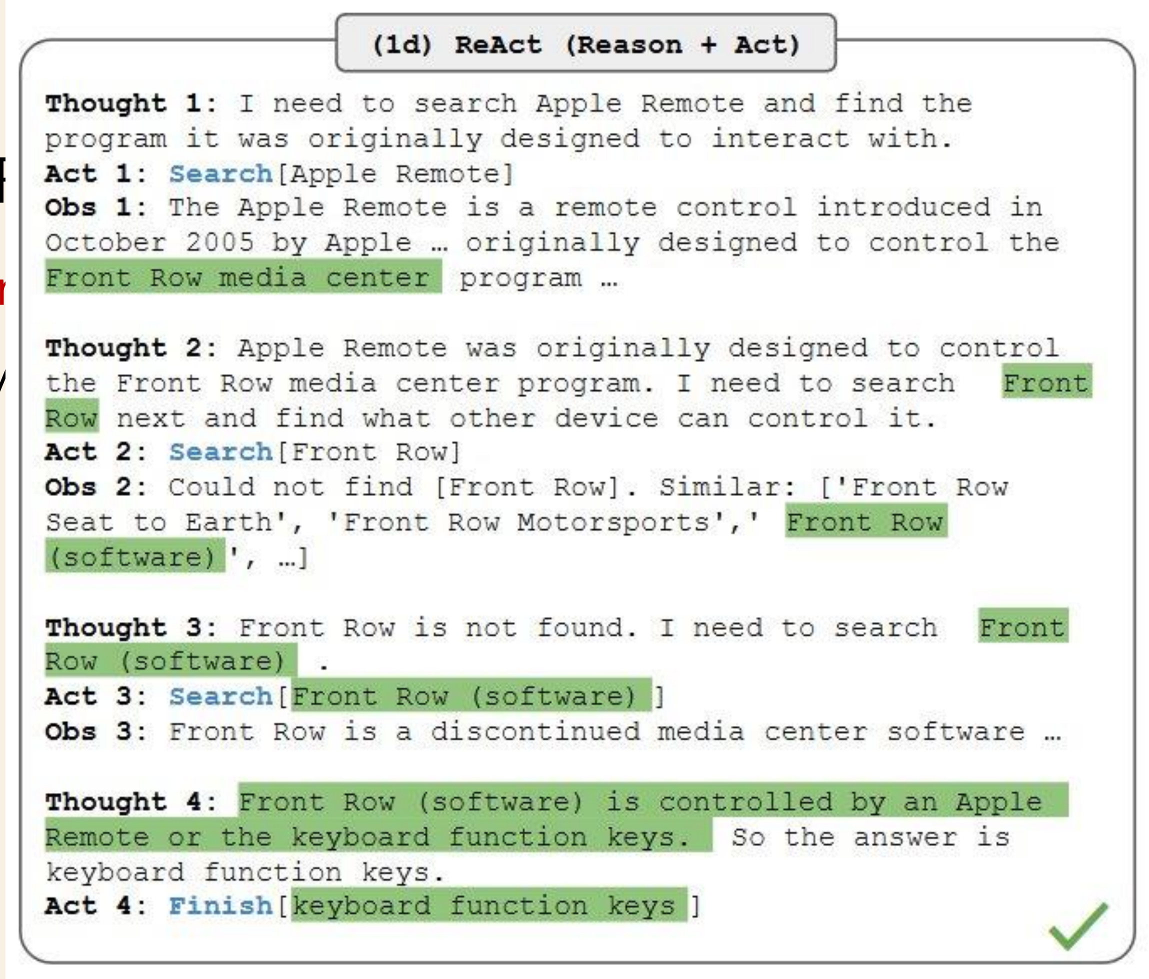
- Compressed finite state machines (CFSM)
  - FSM determines whether generated tokens met the constraints
    - *It's token-by-token*
  - CFSM supports multi-token processing



(a) Normal FSM for regex `{"summary": "`
(b) Compressed FSM for regex `{"summary": "`
(c) Decoding process with normal FSM
(d) Decoding process with compressed FSM

API Speculative Execution

- For black-box AP
  - Intuition: The fi eric
  - Imagine an LLM



Evaluation

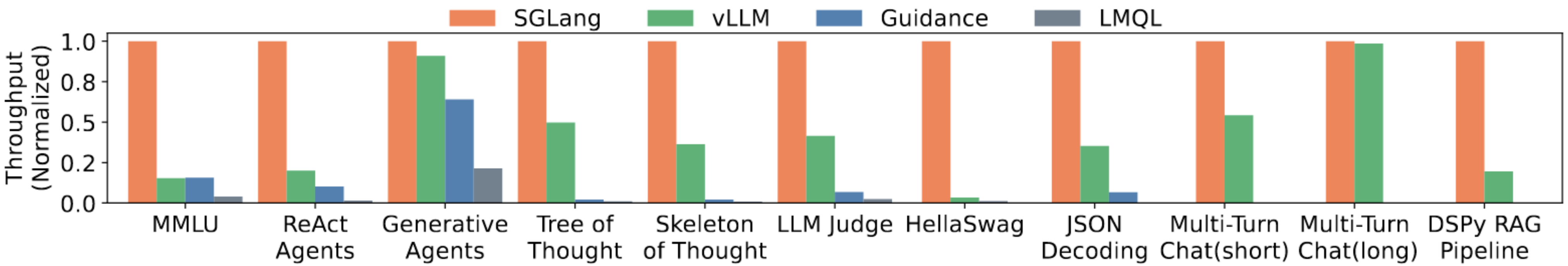


Figure 5: Normalized throughput on Llama-7B models. Higher is better.

Takeaways

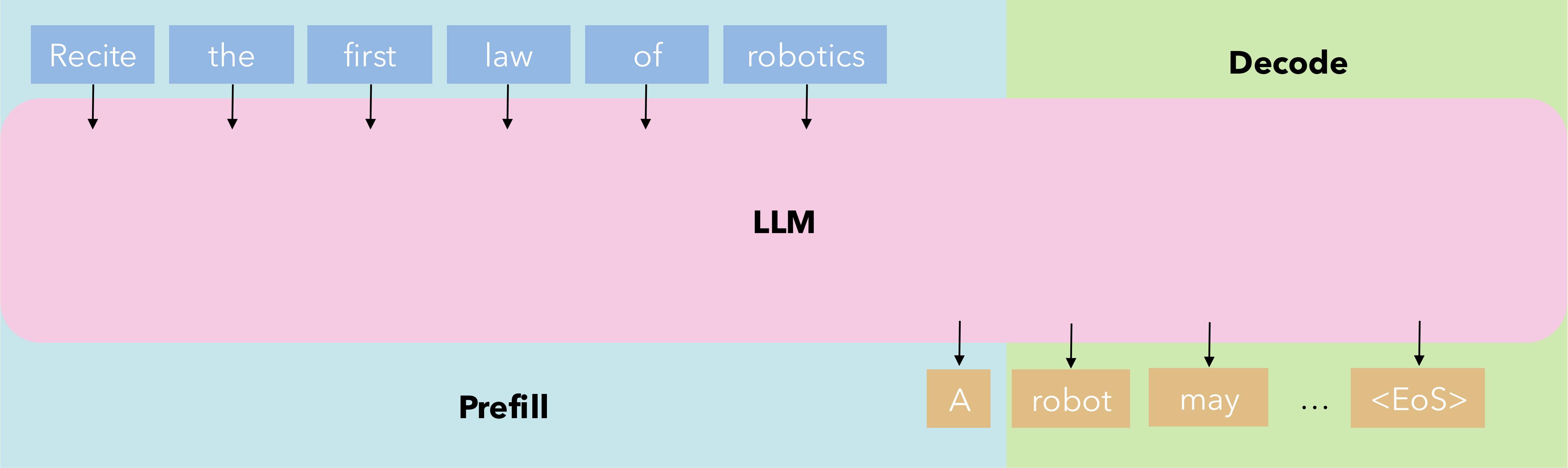
- A developer-friendly frontend "language"
- An efficient backend runtime
  - RadixAttention to increase KV cache reusability
  - Compressed FSM to accelerate constrained decoding
  - API speculative execution to reduce E2E latency

Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve:

An efficient LLM inference scheduler that significantly improves throughput while maintaining low latency.

Shunqiang Feng (mpp7ez)

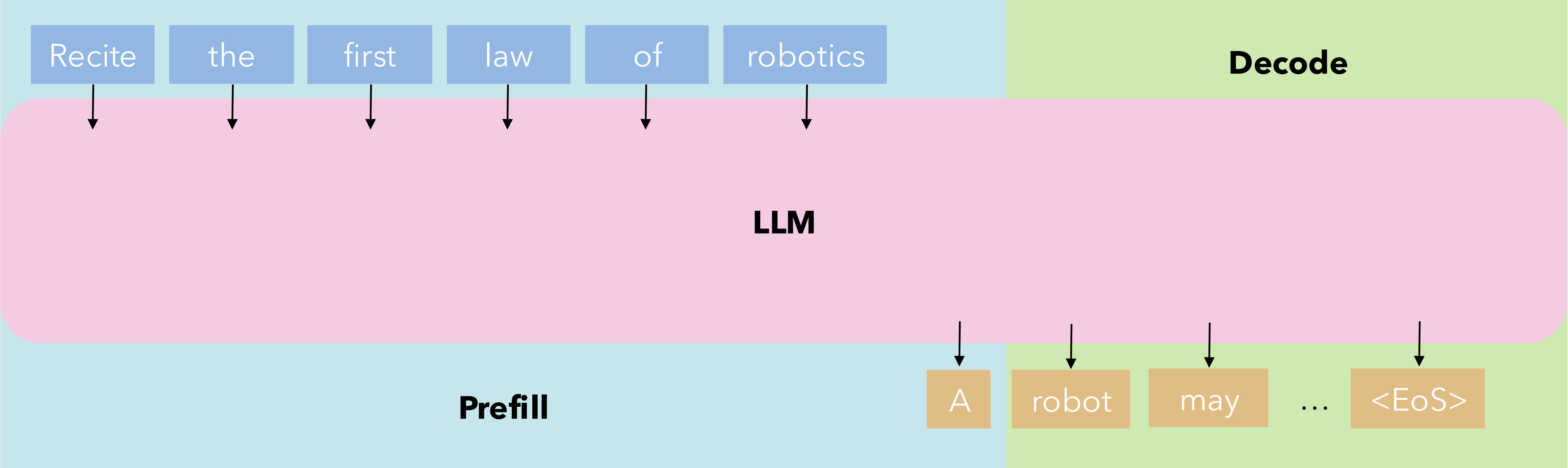
1. Background
(a) Prefill & Decode



Each LLM serving request goes through two phases:

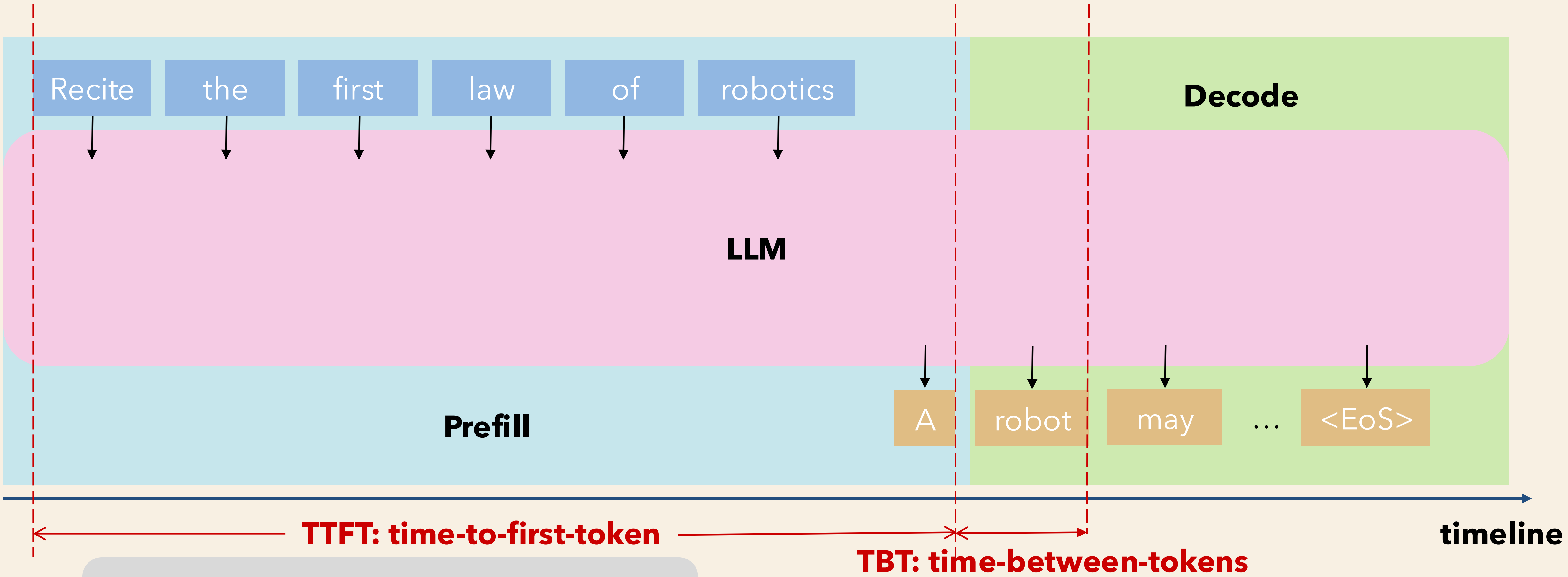
1. Prefill
   To process the entire input prompt and produces the first output token
2. Decode
   To generate the rest of output tokens, one-at-a-time.

1. Background
(a) Prefill & Decode



Stage	**Resource Demand**	**Processing Style**	**benefit from batching?**
Prefill	Computation-Intensive	Processes all input tokens in parallel	No
Decode	Memory-Intensive	Processes one token at a time	Yes

1. Background
(b) LLM Service Metrics: Latency



the latency of generating the first output token from the moment a request arrives in the system

the interval between the generation of consecutive output tokens of a request, and affects the overall perceived fluidity of the response.

1. Background
(b) LLM Service Metrics: Throughput

Capacity
- Definition:
  the maximum request load (queries-per-second) a system can sustain while meeting certain latency targets.

Higher capacity is desirable because it reduces the cost of serving.

1. Background
(c) Current LLM Schedulers

Current LLM Schedulers

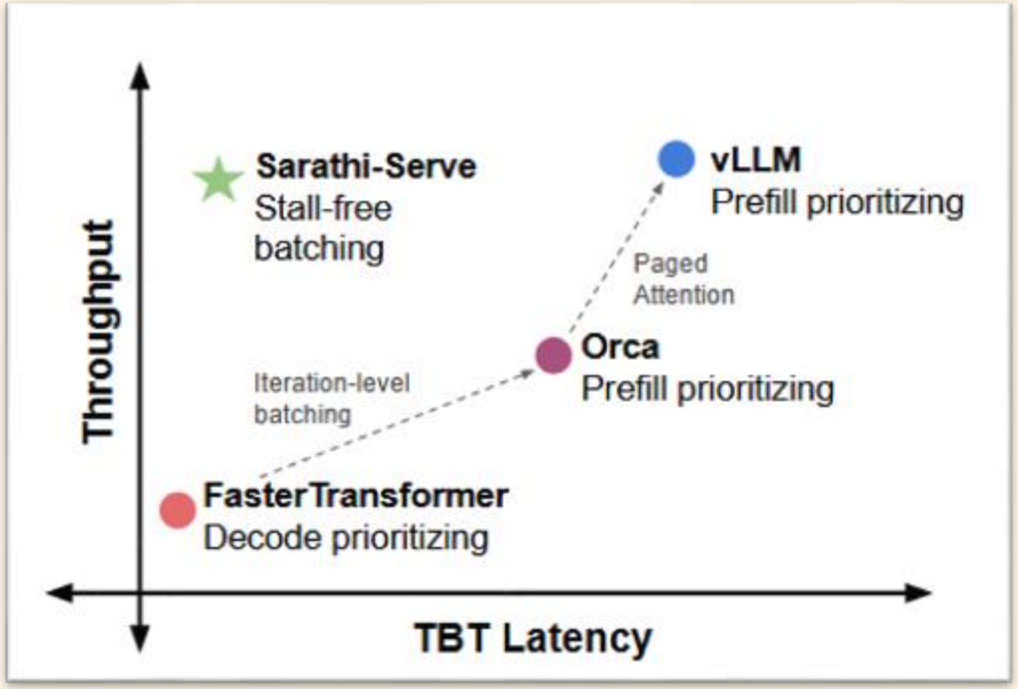
Decode-Prioritizing
Waits for all decodes to finish before new prefills.
(e.g. FasterTransformer)

Low Latency
(No interference with ongoing decodes)
Low Throughput
(Batch size shrinks as requests finish early)

Prefill-Prioritizing
Eagerly schedules prefills when GPU memory is available
(e.g., vLLM, Orca)

High Throughput
(Larger batch size for decodes)
High Latency
(Generation stalls pause ongoing decodes)

1. Background
(c) Current LLM Schedulers

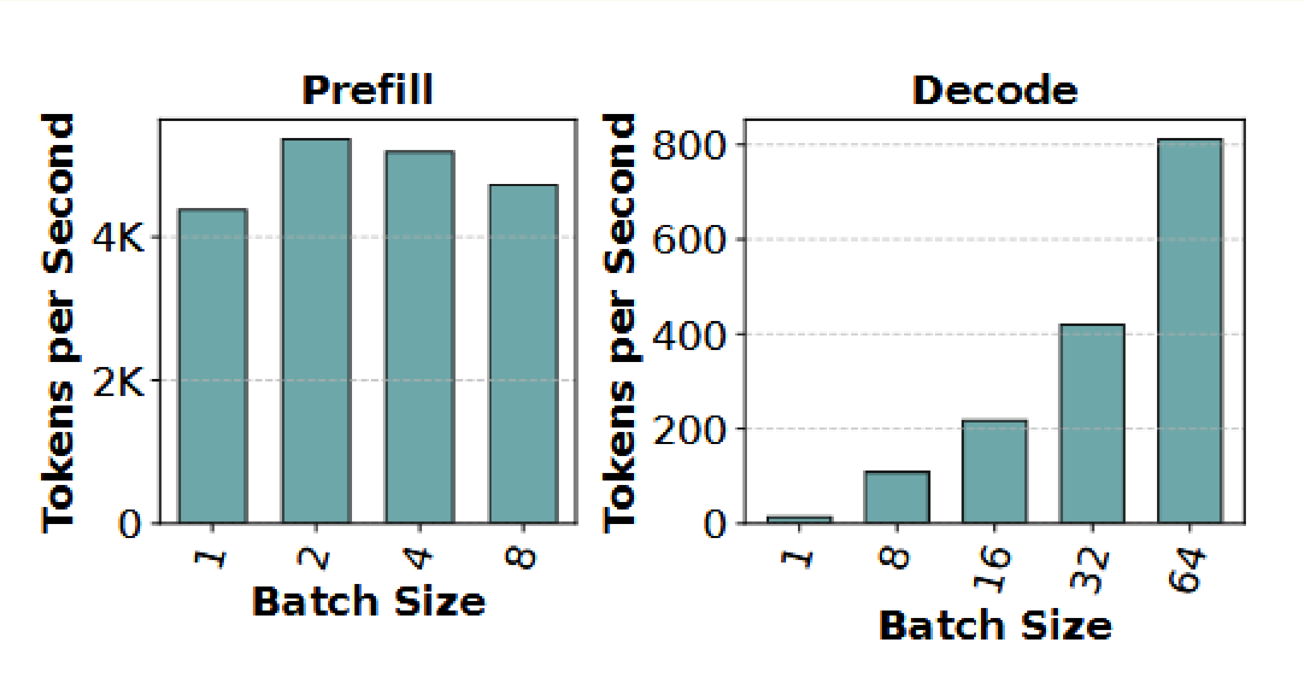


Tradeoff between throughput and latency in current schedulers

1. Background
(d) Another Challenge for Prefill-Prioritizing

- Background
  - Pipeline-parallelism (PP) used for cross-node LLM inference
- Issue: Pipeline bubbles waste GPU cycles
  - Caused by varying runtimes of prefill and decode micro-batches
- Impact: Degrades system throughput

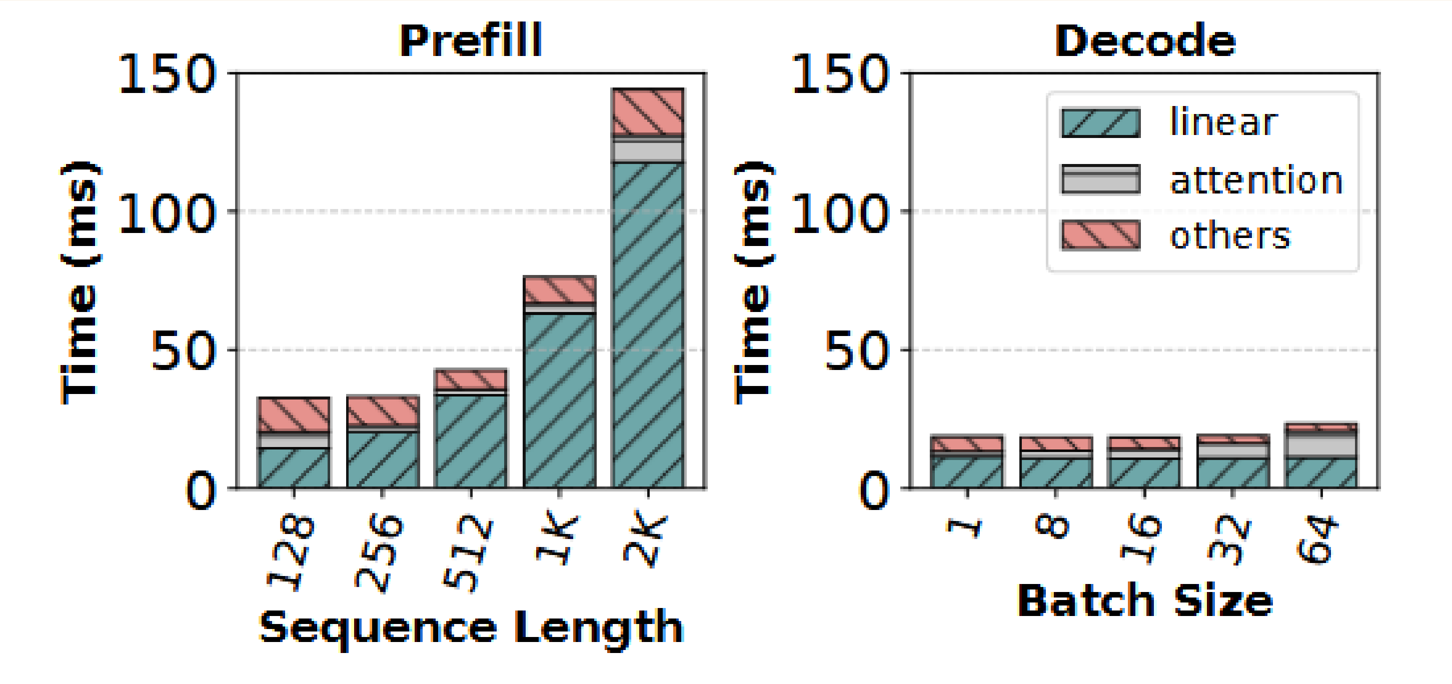
2. Motivation
(a) Cost Analysis of Prefill and Decode



Takeaway-1

Prefill and Decode phase demonstrate contrasting behaviors wherein <u>batching boosts decode phase throughput immensely but has little effect on prefill throughput</u>.

2. Motivation
(a) Cost Analysis of Prefill and Decode

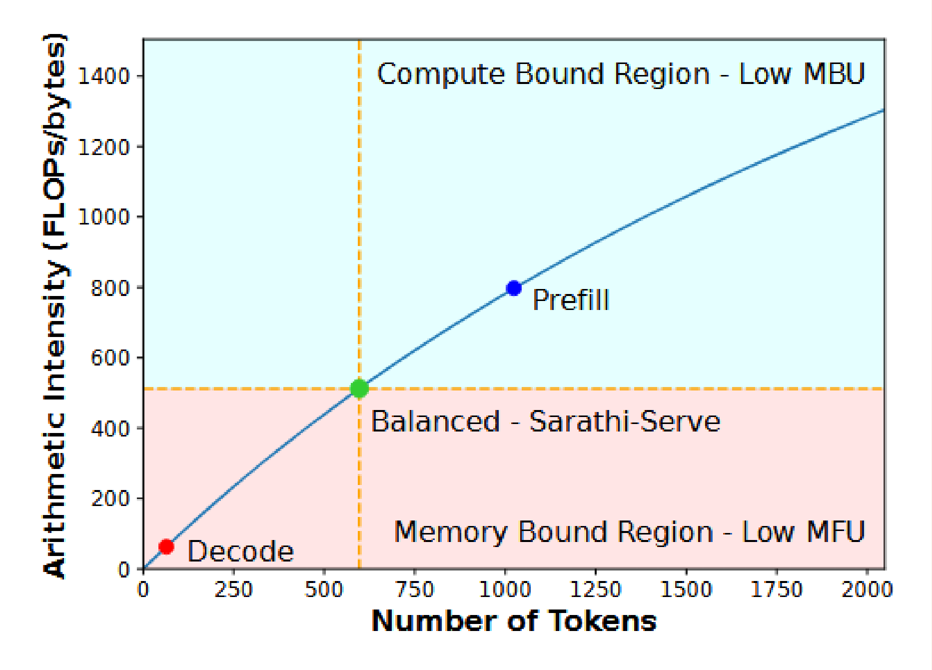


From the figure, we see that linear operators contribute to the majority of the runtime cost. Therefore, optimizing linear operators is important for improving LLM inference. (Let's focus on linear operators later)

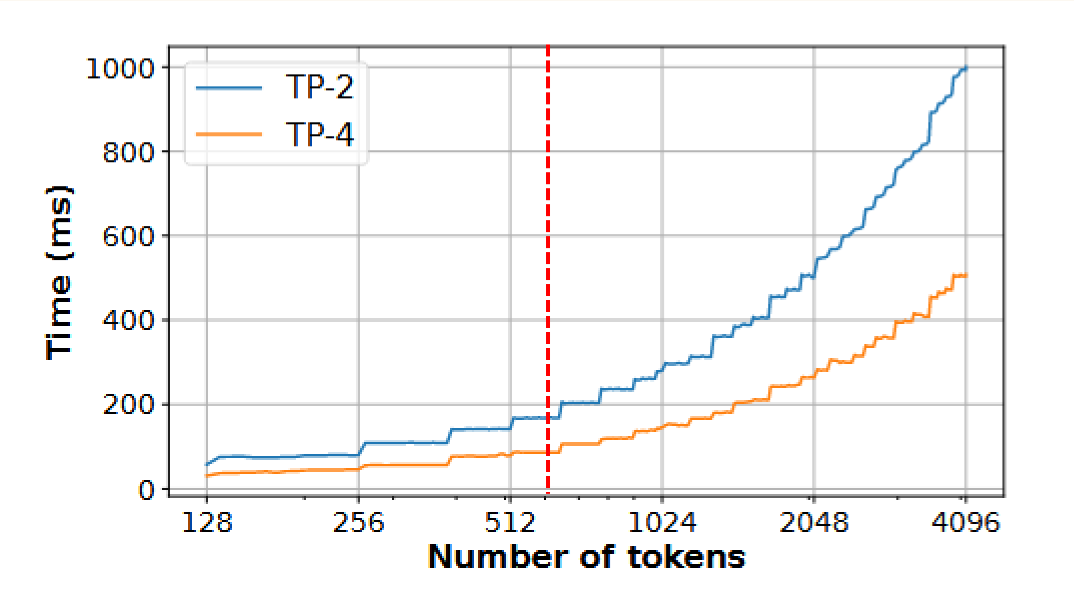
2. Motivation
(a) Cost Analysis of Prefill and Decode

For an operation, Total Runing Time = max(Tmath, Tmem)
If Tmem > Tmath: Operation is Memory-Bound
If Tmath > Tmem: Operation is Compute-Bound
if Tmath = Tmem, both compute and memory bandwidth utilization are maximized.

2. Motivation
(a) Cost Analysis of Prefill and Decode



Arithmetic intensity trend for LLaMA2-70B linear operations with different number of token running on four A100s.



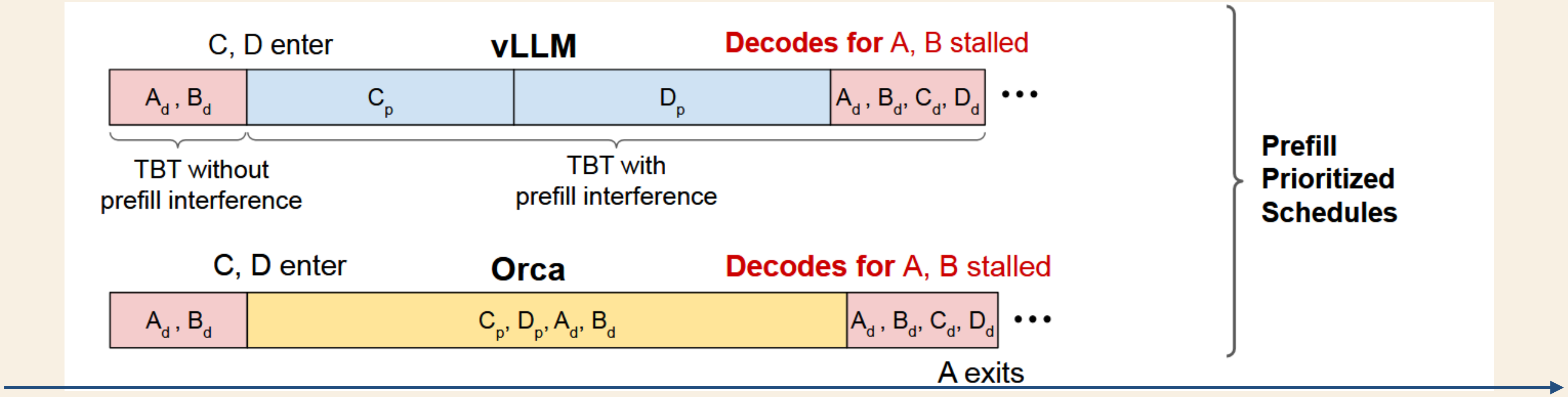
Linear layer execution time as function of number of tokens in a batch for LLaMA2-70B on A100(s) with different tensor parallel degrees.

Takeaway-2

Decode batches operate in memory-bound regime leaving compute underutilized. This implies that more tokens can be processed along with a decode batch without significantly increasing its latency.

2. Motivation
(b) Throughput-Latency Trade-off

A, B: existing requests in decode
C, D: new requests

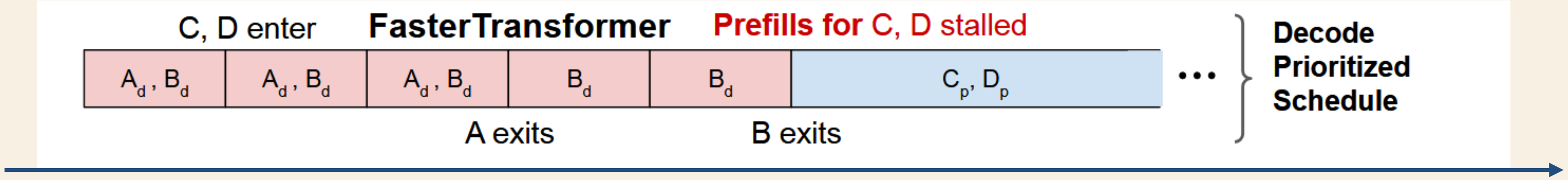


Processing Timeline

vLLM & Orca (Prefill-Prioritizing):
- Eagerly schedule prefills (C, D), pausing decodes (A, B)
- Result: Generation stalls (high TBT latency)

2. Motivation
(b) Throughput-Latency Trade-off

A, B: existing requests in decode
C, D: new requests



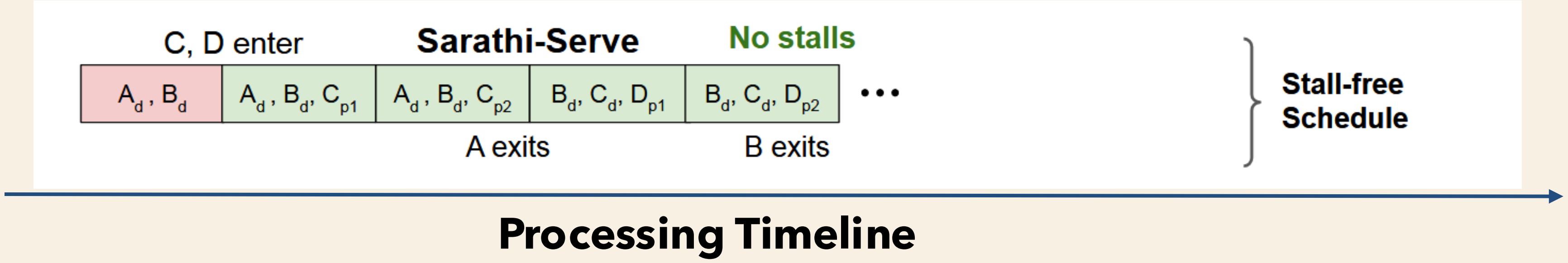
Processing Timeline

FasterTransformer (Decode-Prioritizing):

- Waits for decodes (A, B) to finish before prefills (C, D)
- Result: No stalls, but low throughput (small decode batch size)

2. Motivation
(b) Throughput-Latency Trade-off

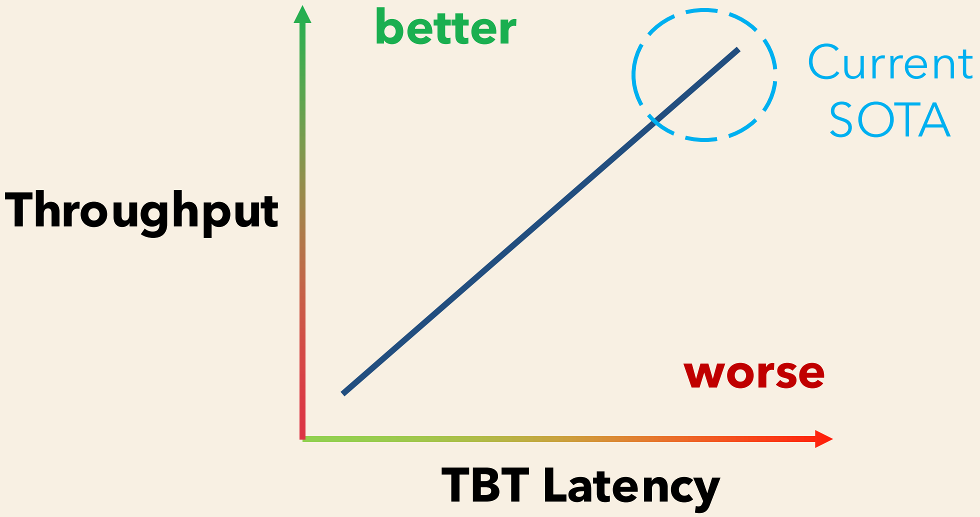
A, B: existing requests in decode
C, D: new requests



Proposed work Sarathi-Serve

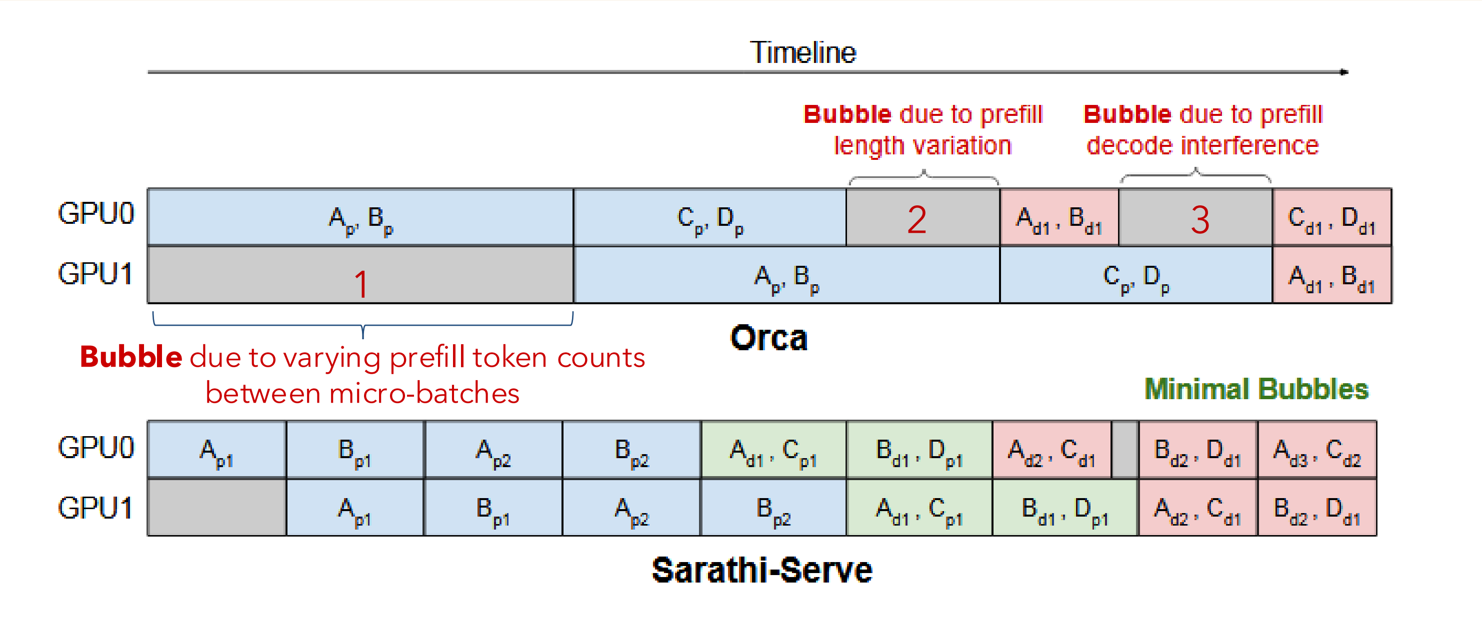
- Stall-free scheduling: No pauses, high throughput

2. Motivation
(b) Throughput-Latency Trade-off



Takeaway-3
The interleaving of prefills and decodes involves **a trade-off between throughput and latency** for current LLM inference schedulers. **State-of-the-art** systems today use prefill-prioritizing schedules that **trade TBT latency for high throughput.**

2. Motivation
(c) Pipeline Bubbles waste GPU Cycles



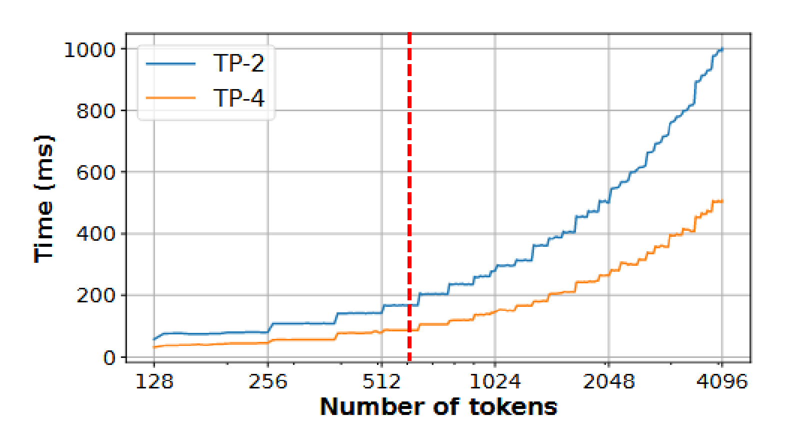
- A 2-way pipeline parallel iteration-level schedule in Orca across 4 requests (A,B,C,D) shows **the existence of pipeline bubbles due to non-uniform batch execution times.**

Takeaway-4

There can be a large variance in compute time of LLM iterations depending on composition of prefill- and decode- tokens in the batch. <u>This can lead to significant bubbles when using pipeline-parallelism.</u>

Sarathi-Serve is able to minimize these stalls by creating **uniform-compute** batches.

3. Method
(a) Chunked-prefills



Dataset	Prompt Tokens			Output Tokens		
	Median	P90	Std.	Median	P90	Std.
openchat_sharegpt4	1730	5696	2088	415	834	101
arxiv_summarization	7059	12985	3638	208	371	265

Insight

A prefill request with **modest sequence length** can effectively saturate GPU compute

Fact

Prompts are often very long, and long prefills with decode iterations increase TBT latency.

Solution: split large prefills to small chunks

Break large prefill requests into smaller units of compute which are still large enough to saturate GPU compute.

3. Method
(b) Stall-free batching

Prefill:
Computation-Bounded

Decode:
Memory-Bounded

Combine prefills and decodes to improve throughput while minimizing latency

Naïve Method in Orca & vLLM

- stall existing decodes to execute prefills

Sarathi-Serve: stall-free batching

- leverages the arithmetic intensity slack in decode iterations to execute prefills without delaying the execution of decode requests in the system.

3. Method
(b) Stall-free batching - Algorithm

```
Input: T_max, Application TBT SLO.
Initialize token_budget, τ ← compute_token_buget(T_max)
Initialize batch_num_tokens, n_t ← 0
Initialize current batch B ← ∅
while True do
  for R in B do
    if is_prefill_complete(R) then
      n_t ← n_t + 1
  for R in B do
    if not is_prefill_complete(R) then
      c ← get_next_chunk_size(R, τ, n_t)
      n_t ← n_t + c
  R_new ← get_next_request()
  while can_allocate_request(R_new) ∧ n_t < τ do
    c ← get_next_chunk_size(R_new, τ, n_t)
    if c > 0 then
      n_t ← n_t + c
      B ← R_new
    else
      break

  process_hybrid_batch(B)
  B ← filter_finished_requests(B)
  n_t ← 0
```

1. Calculates token budget based on user-specified SLO [Service Level Objective] (we will introduce later)

2. the batch is filled with ongoing decode tokens

3. include any partially completed prefill

4. Admit new requests within leftover token budget

3. Method
(c) Determining Token Budget

TBT SLO requirement

a smaller token budget is preferable:
- iterations with fewer prefill tokens have lower latency.

Trade off

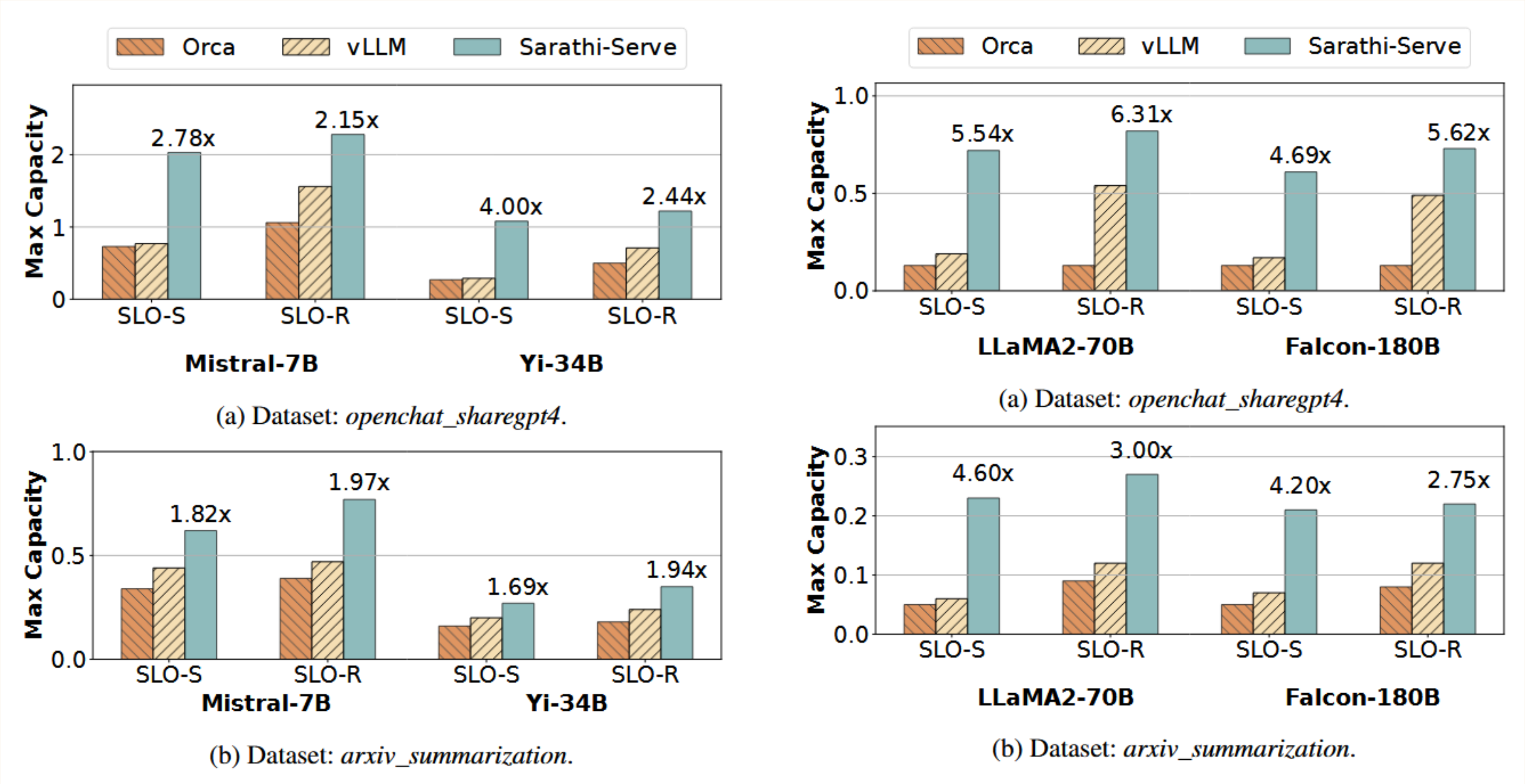
chunked-prefills overhead

smaller token budget causes:
1. lower GPU utilization
2. repeated KV-cache access in the attention operation

Solution

Profile batches to set token budget: Max tokens per batch before violating TBT SLO.

4. Evaluation
(a) Capacity

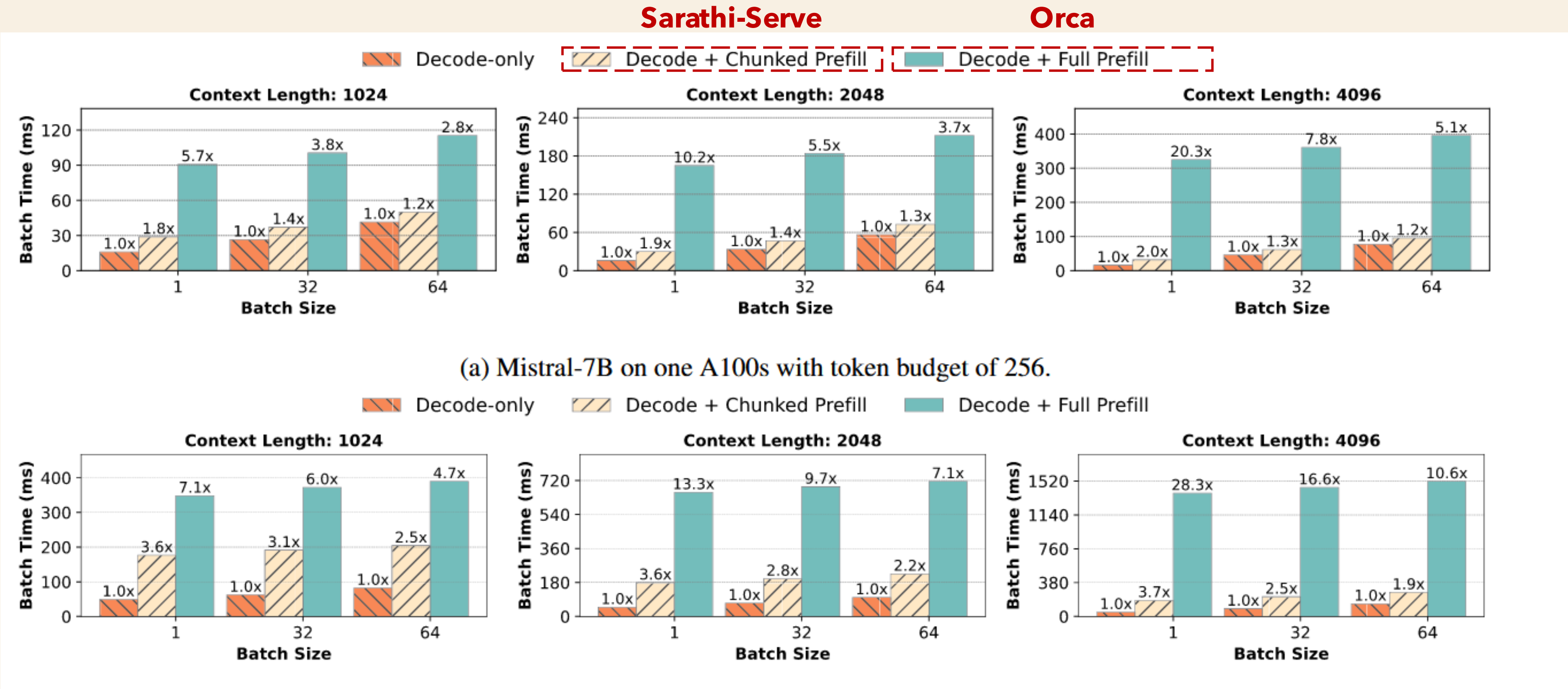


Capacity evaluation across different models and datasets

Note:
- SLO: Service Level Objective
- SLO-S: Strict SLO
- SLO-R: Relaxed SLO

Capacity
(in queries per second)
improves **up to 6x**

4. Evaluation
(b) Latency



(a) Mistral-7B on one A100s with token budget of 256.

(b) LLaMA2-70B on four A100s with token budget of 512.

Note:
- Decode + Full Prefill

the hybrid batching of Orca
- Decode + Chunked Prefill

Proposed work

Sarathi-Serve processes prefill tokens with much lower impact on the latency of decodes.

4. Evaluation
(c) Making Pipeline Parallel Viable

Sarathi-Serve optimizes pipeline parallelism by <u>creating hybrid batches with uniform computational demands</u>.

Key Benefits:

- Reduces pipeline bubbles, improving GPU utilization.
- <u>Enables efficient operation in multi-node deployments</u> over standard Ethernet connections.

Thank you!