A survey of DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving

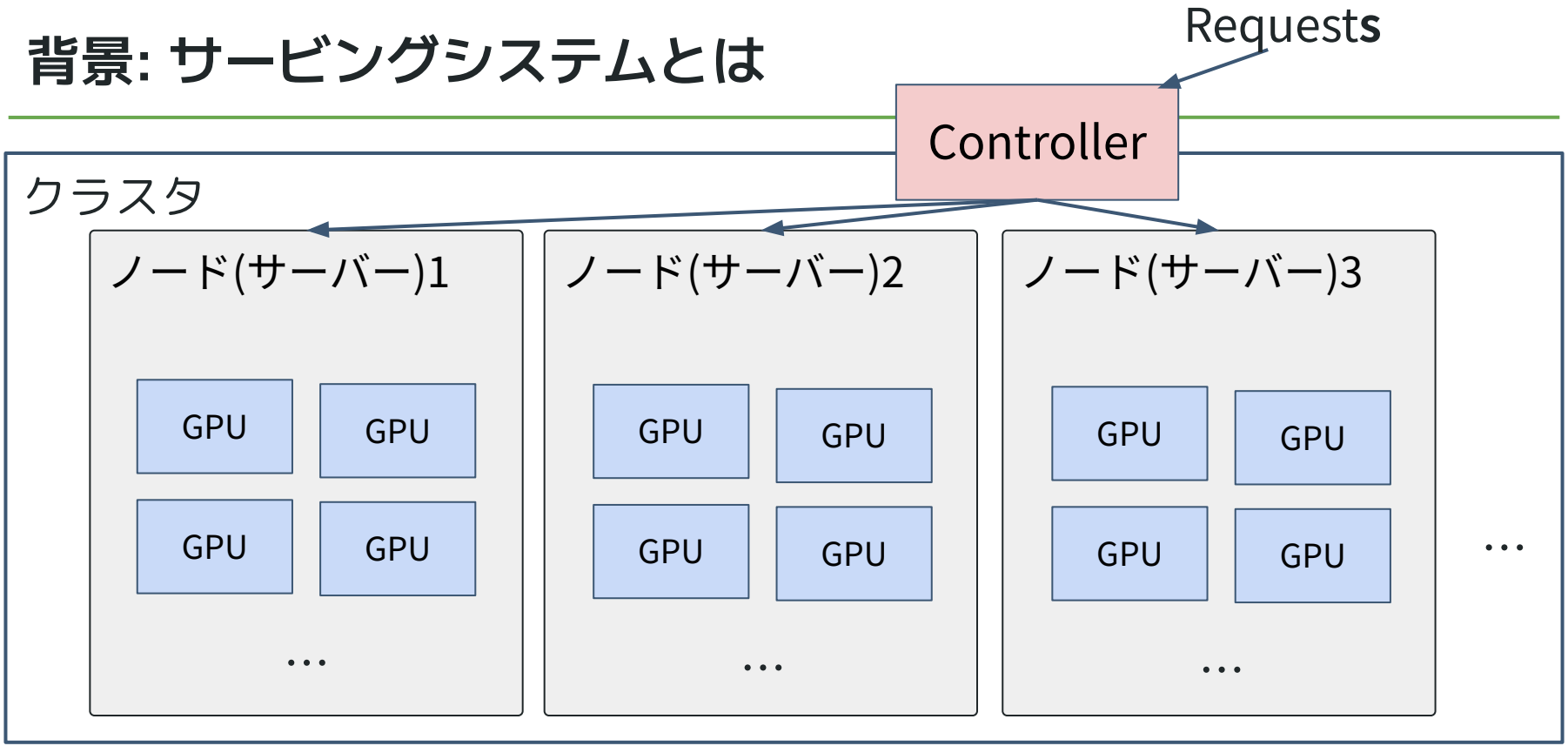
[Zhong et al. OSDI2024]

オペレーティングシステム特論　輪講

2025/7/3

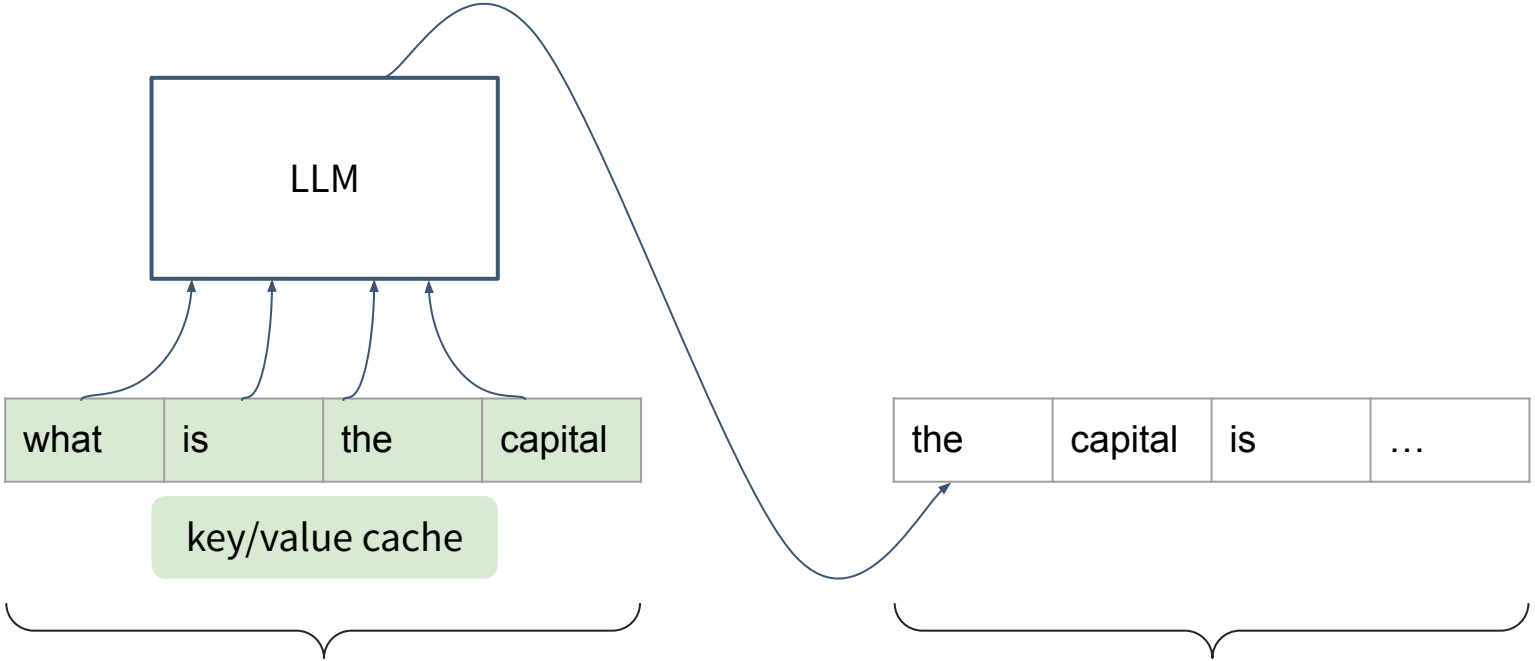
朱 光漪

背景: サービングシステムとは

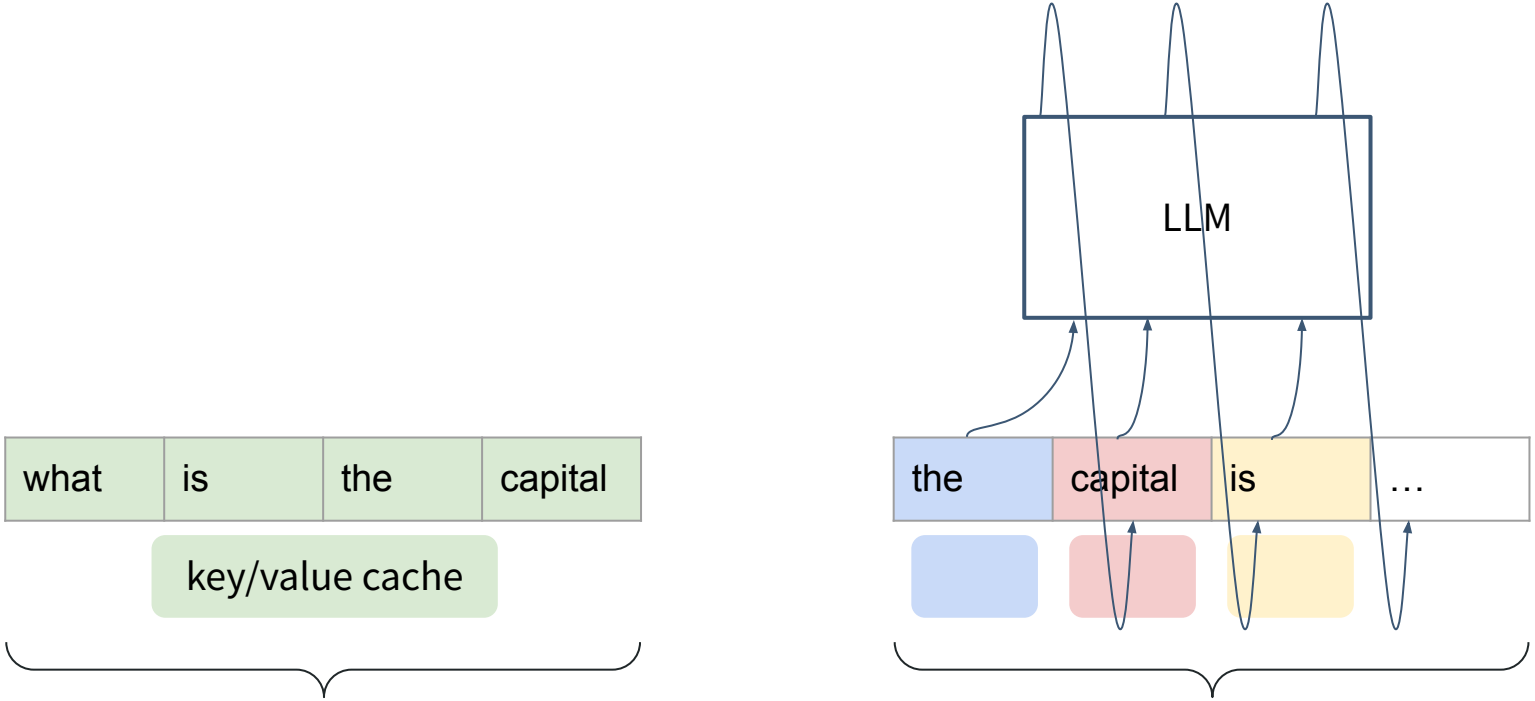


- リクエスト処理をどう計算資源に割り当てるかなどを制御する

背景: LLM 推論


LLM
what
is
the
capital
key/value cache
the
capital
is
…
Prefill phase
入力プロンプトを一括で処理
Decoding phase

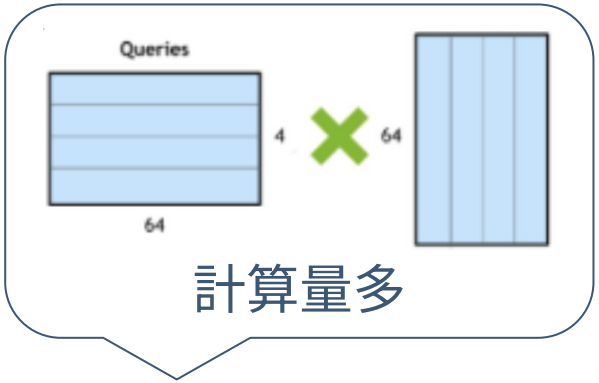
背景: LLM 推論

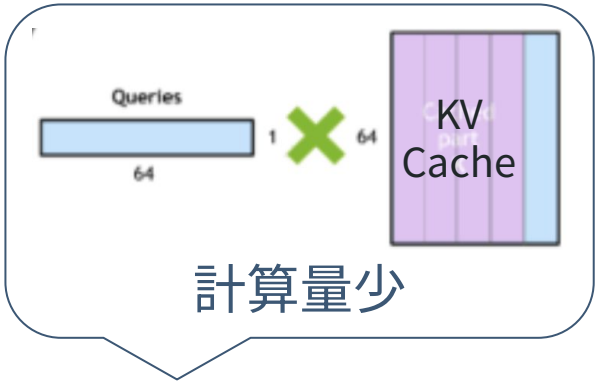


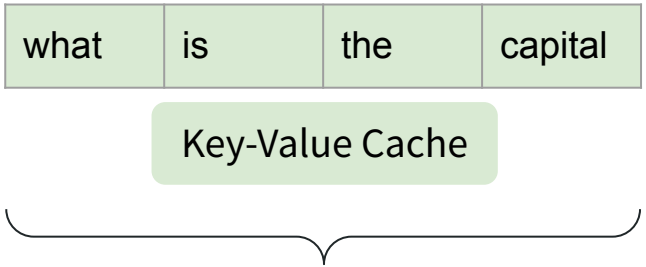
Prefill phase
入力プロンプトを一括で処理

Decoding phase
1トークンずつ逐次的に生成

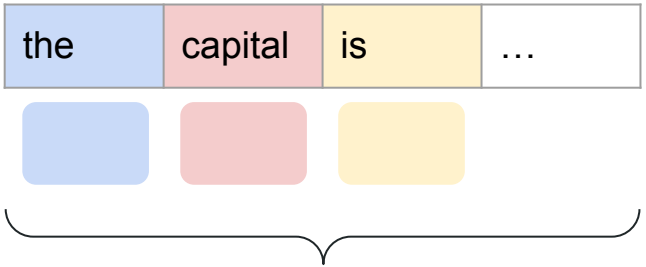
背景: LLM 推論







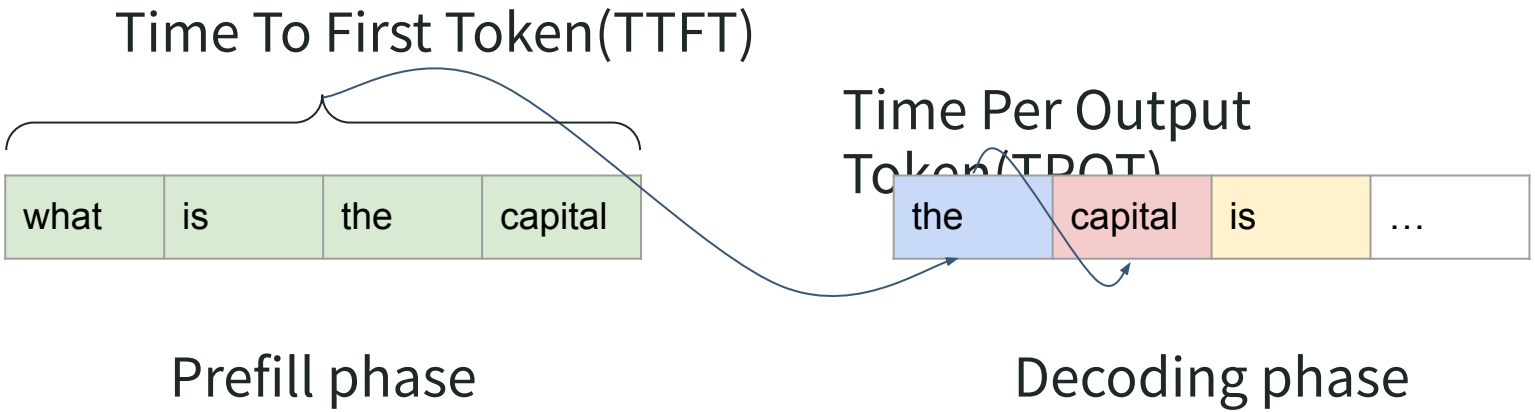
Prefill phase
入力プロンプトを一括で処理



Decoding phase
1トークンずつ逐次的に生成

図は[2]より

背景: LLM 推論のレイテンシ



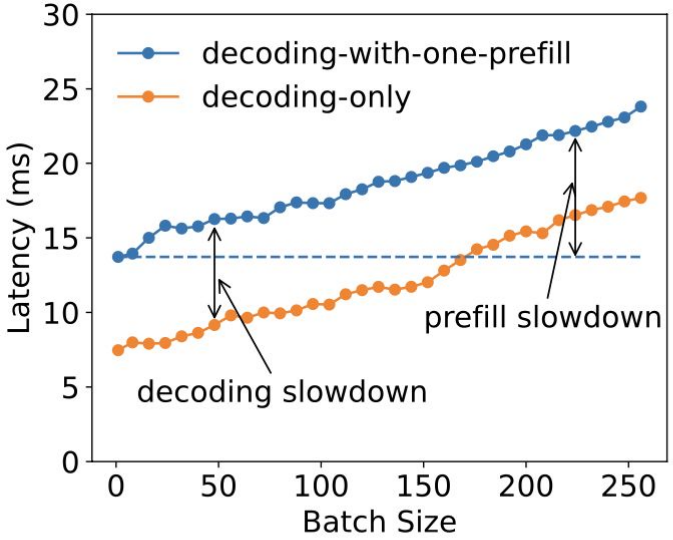
Latency = TTFT + TPOT * #tokens

LLM向けの分散推論サービングシステムの課題

- 既存手法：PrefillとDecodingフェーズをまとめてバッチ処理
  - LLMの重みとKVキャッシュは共通しているため

😕 PrefillとDecodingの相互干渉：
PrefillフェーズではGPU効率が高いが、
DecodingフェーズではGPU効率が低い
ため、互いにレイテンシに影響する(右図)

😕 PrefillとDecodingのレイテンシ要求と
最適な並列化戦略が異なる



(a) Input length = 128

提案手法: DistServe: Prefill と Decoding フェーズの分離

- インスタンス：モデルの重み全体を持つ推論を実行する単位
  - 複数GPUで1インスタンスを構成することもある
- DistServe: PrefillとDecodingフェーズを異なるインスタンスに配置
  - 相互干渉の解消
  - それぞれのフェーズに合ったGPUの割り当てと並列化戦略

✅ レイテンシの短縮

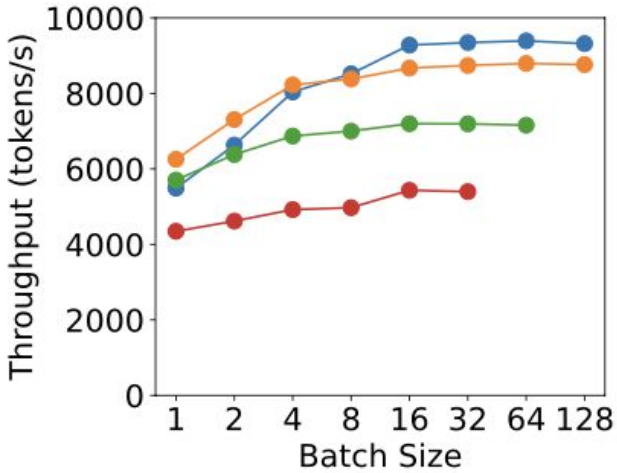
✅ GPUあたりのgoodput(要件を満たすthroughput)の向上

2種類の並列化手法

- Inter-operator Parallelism
  - パイプライン処理
  - スループット向上に有効
- Intra-operator Parallelism
  - 1つの（重い）演算を複数GPUに分割して並列実行
  - 通信コストが大きいがレイテンシ短縮に有効

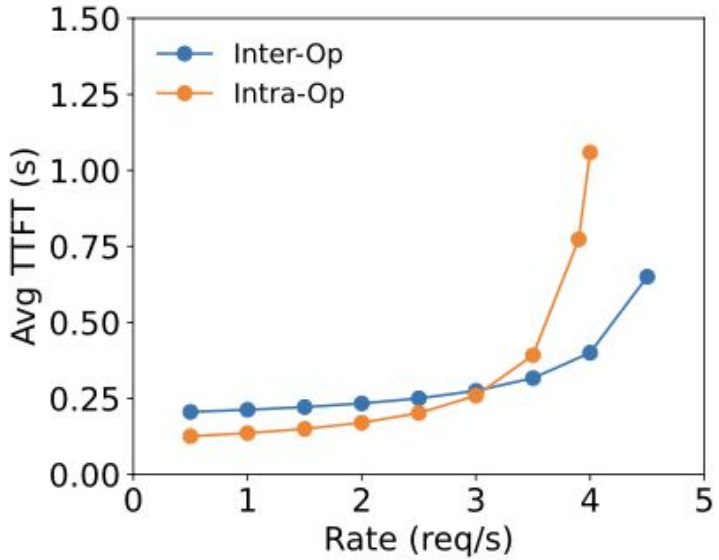
設計: Prefill インスタンス内の最適化戦略

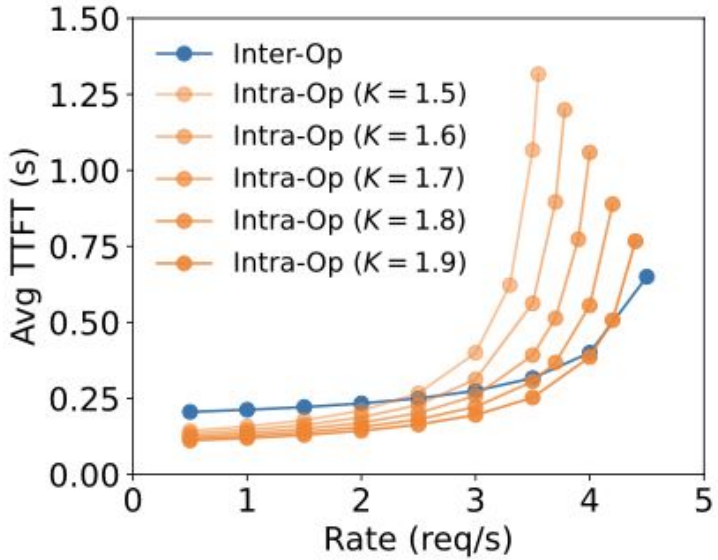
- 入力プロンプトを一括で処理するため、計算量が多い
- 一定のバッチ数を超えるとスループットが増えない→入力トークン数に応じてバッチ化するか判断



設計: Prefill インスタンス内の最適化戦略

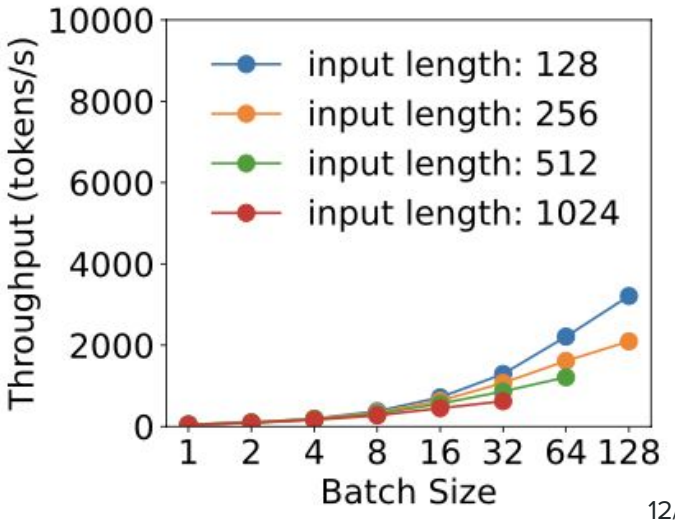
- 並列化戦略
  - 負荷が低い場合Intra-op、負荷が高い場合Inter-opが効果的





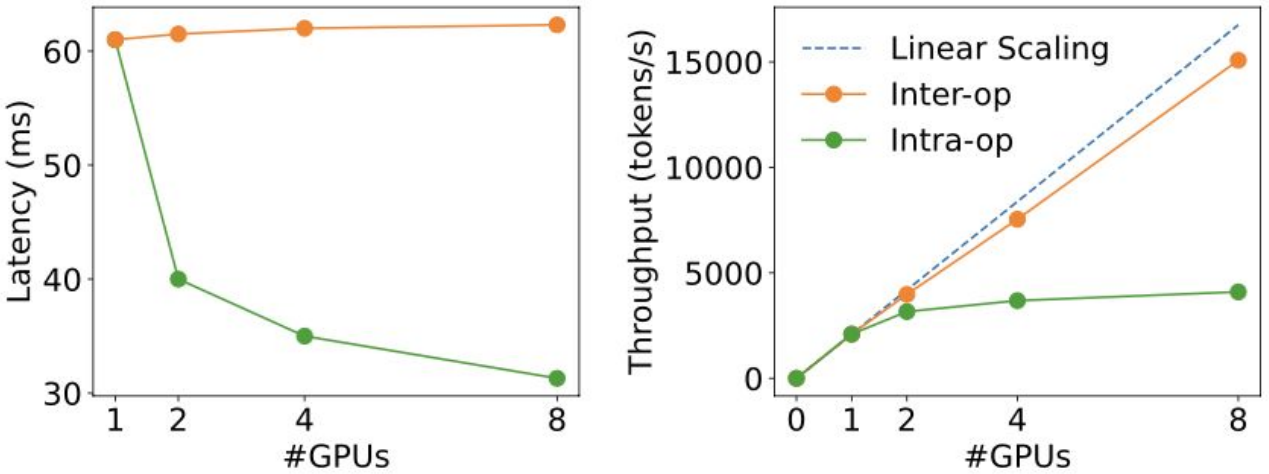
設計: Decoding インスタンス内の最適化戦略

- 1トークンずつ逐次的に生成
- 計算量が大きくないためバッチ数がスループットに直接反映する
  →帯域制約まで複数リクエストをバッチ処理するのが有効

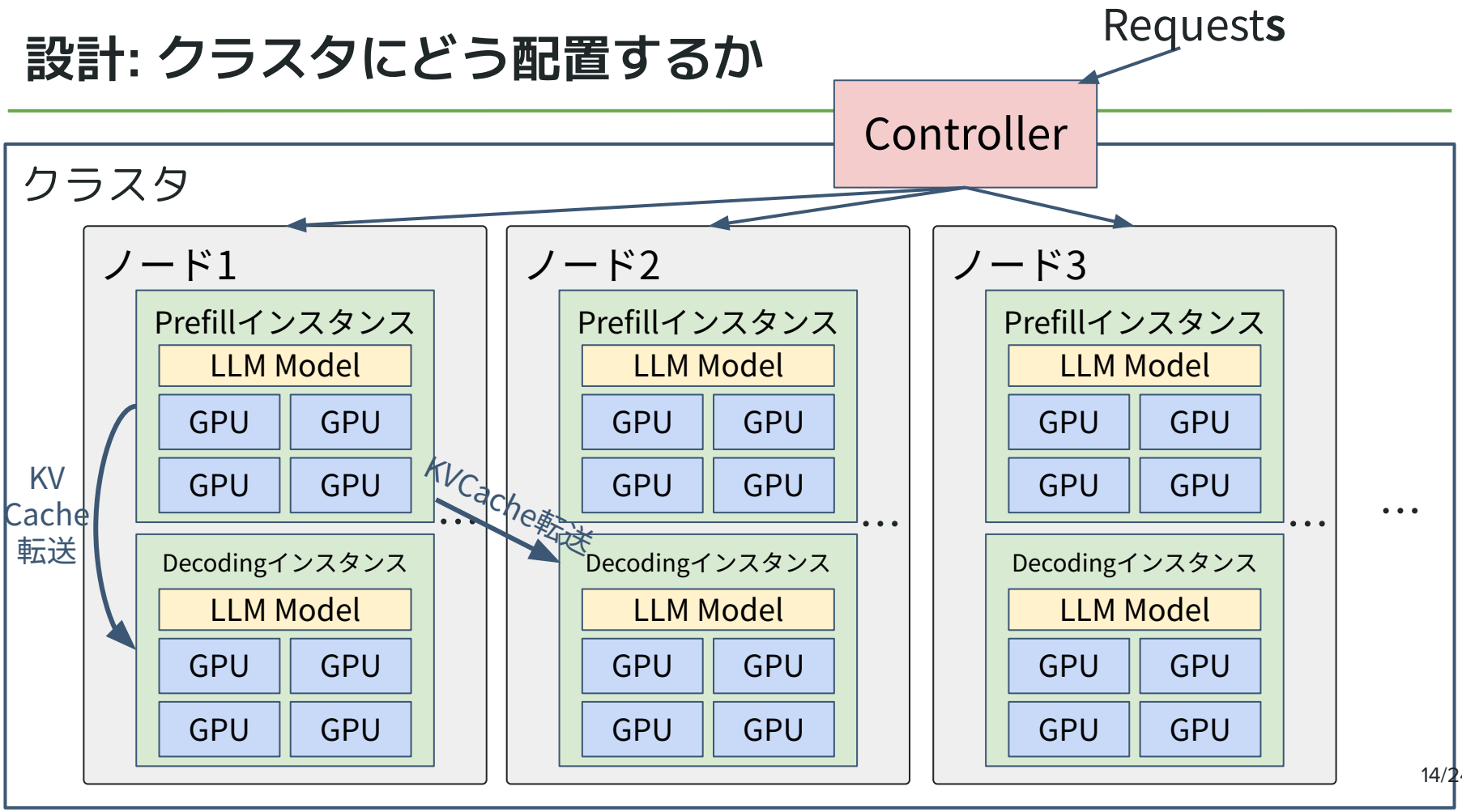


設計: Decoding インスタンス内の最適化戦略

- 並列化戦略
  - TPOT要件が厳しい場合はIntra-op並列を優先、それ以外はInter-op並列でスループットを最適化する



設計: クラスタにどう配置するか



設計: 広帯域ネットワーク付きクラスタ

- ノード間通信（KVキャッシュ転送）がほぼ無視できる

1. すべての並列構成（Inter-opの並列度、Intra-opのGPU数）を列挙
   a. それぞれに対してシミュレーターを使ってレイテンシ要件達成率を推定
   b. goodputが最大になる構成を選択
      現実のワークロードは到着タイミングや長さが不規則なので、単純な数式ではSLO達成率を計算できない→過去のトレースから分布を推定し、シミュレータで模倣
2. 得られた並列構成を必要なだけ複製し、ユーザのリクエスト量に対応（スケーリング）

設計: 広帯域でないネットワークのクラスタ

- 現実の多くのクラスタでは、ノード間の通信帯域が狭く、Prefill→DecodingのKVキャッシュ転送がボトルネックになる

1. LLMのレイヤーをステージに分割（Inter-Operator Parallelism）
2. 同じステージのPrefillとDecodingセグメントを同じノードに配置
   a. KVキャッシュ転送がノード内転送（NVLINK）となり高速
3. 各セグメントごとに可能なIntra-op並列構成を列挙、シミュレーションし、最適な構成を選択
4. 得られた並列構成を必要なだけ複製し、ユーザのリクエスト量に対応（スケーリング）

Algorithm 1 High Node-Affinity Placement Algorithm

Input: LLM G, #node limit per-instance N, #GPU per-node M, GPU memory capacity C, workload W, traffic rate R.
Output: the placement *best_plm*.

$config_p, config_d \leftarrow \emptyset, \emptyset$
for $intra\_op \in \{1, 2, ..., M\}$ **do**
  for $inter\_op \in \{1, 2, ..., \frac{N \times M}{intra\_op}\}$ **do**
    if $\frac{G.size}{inter\_op \times intra\_op} < C$ **then**
      $config \leftarrow (inter\_op, intra\_op)$
      $\hat{G} \leftarrow \text{parallel}(G, config)$
      $config.goodput \leftarrow \text{simu\_prefill}(\hat{G}, W)$
      if $\frac{config_p.goodput}{config_p.num\_gpus} < \frac{config.goodput}{config.num\_gpus}$ **then**
        $config_p \leftarrow config$
      $config.goodput \leftarrow \text{simu\_decode}(\hat{G}, W)$
      if $\frac{config_d.goodput}{config_d.num\_gpus} < \frac{config.goodput}{config.num\_gpus}$ **then**
        $config_d \leftarrow config$
$n, m \leftarrow \lceil \frac{R}{config_p.goodput} \rceil, \lceil \frac{R}{config_d.goodput} \rceil$
$best\_plm \leftarrow (n, config_p, m, config_d)$
return *best_plm*

Algorithm 2 Low Node-Affinity Placement Algorithm

Input: LLM G, #node limit per-instance N, #GPU per-node M, GPU memory capacity C, workload W, traffic rate R.
Output: the placement *best_plm*.

$config^* \leftarrow \emptyset$
for $inter\_op \in \{1, 2, ..., N\}$ **do**
  $\mathcal{P} \leftarrow \text{get\_intra\_node\_configs}(G, M, C, inter\_op)$
  for $P_p \in \mathcal{P}$ **do**
    for $P_d \in \mathcal{P}$ **do**
      if $P_p.num\_gpus + P_d.num\_gpus \leq M$ **then**
        $config \leftarrow (inter\_op, P_p, P_d)$
        $\hat{G}_p, \hat{G}_d \leftarrow \text{parallel}(G, config)$
        $config.goodput \leftarrow \text{simulate}(\hat{G}_p, \hat{G}_d, W)$
        if $\frac{config.^*goodput}{config.^*num\_gpus} < \frac{config.goodput}{config.num\_gpus}$ **then**
          $config^* \leftarrow config$
$n \leftarrow \lceil \frac{R}{config.^*goodput} \rceil$
$best\_plm \leftarrow (n, config^*)$
return *best_plm*

実装

- 配置アルゴリズムモジュール
- RESTful API フロントエンド
- オーケストレーション層
  - リクエストのディスパッチ
  - KVキャッシュの転送
  - 出力結果の配送

Python

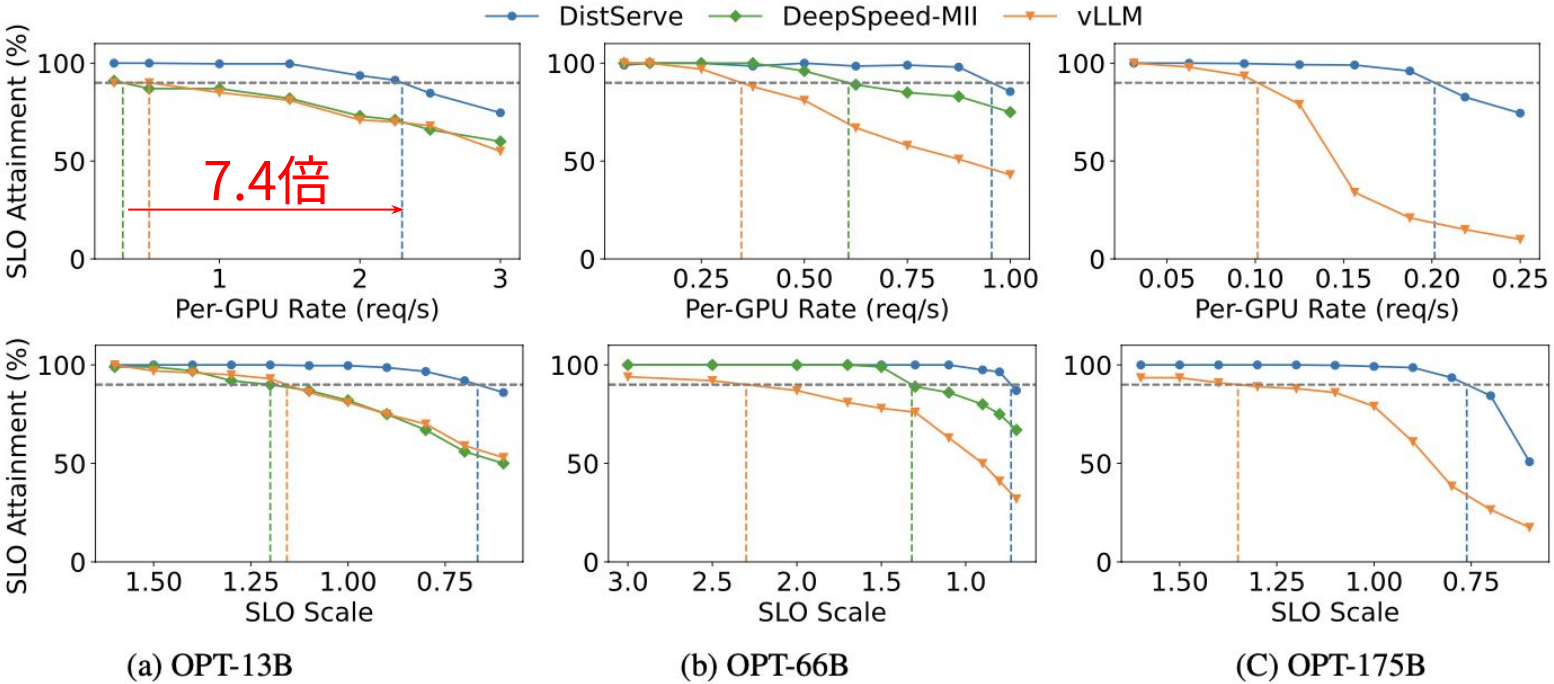
- 各インスタンスでの並列実行エンジン：C++/CUDA

評価

- 様々なLLMサイズとアプリにおいて、既存手法より良い結果となった
- アプリ：Chatbot、Code Completion、要約
- ベースライン
  - vLLM（代表的なLLM Servingシステム）：バッチ処理
  - DeepSpeed-MII：Prefillのチャンク化
- LLM：OPT（Open Pretrained Transformer）13B、66B、175B
- クラスタ：4ノード、32GPUs
  - 各ノードに8 NVIDIA SXM A100- 80GB GPUs
  - ノード間帯域幅：25Gbps

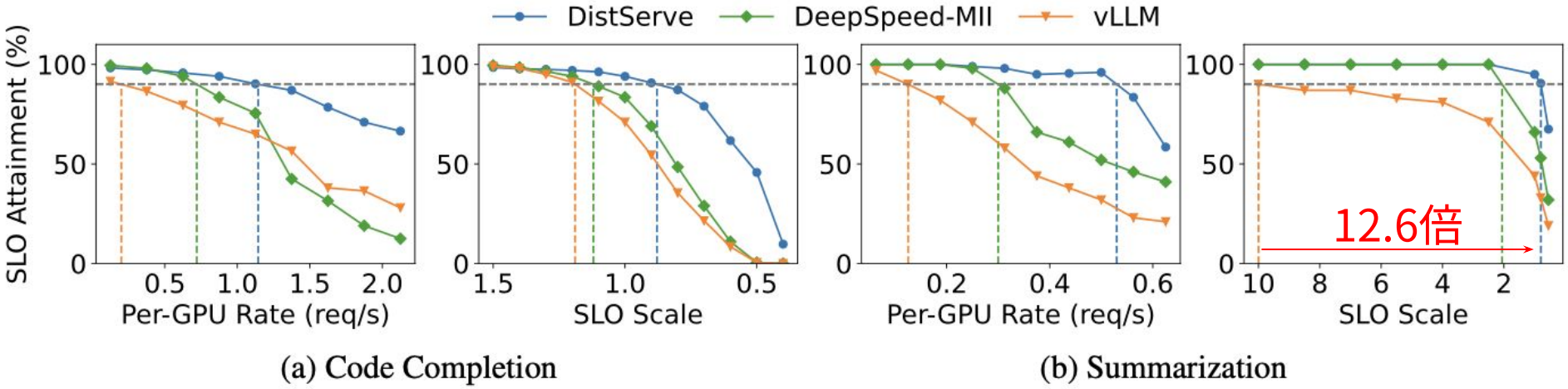
評価

- GPU あたりの goodput は vLLM に対し2.0倍~4.6倍
- SLO(Service Level Objective) Attainment: レイテンシ要件を満たすリクエスト割合



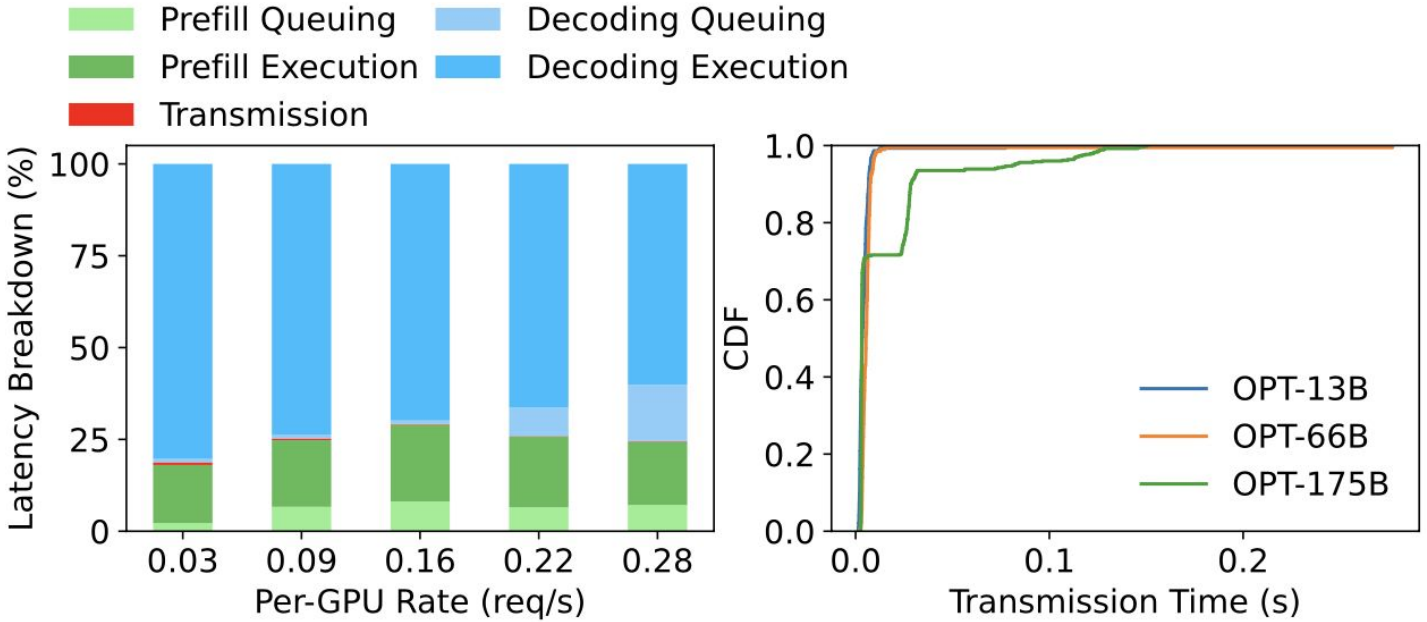
評価

- 要約で vLLM に対して12.6倍厳しい SLOを満たす



評価：Latency Breakdown

- KVキャッシュ転送が占める割合は0.1%以下
  - 同一ステージのPrefillとDecodingフェーズを同じノードに配置した効果



今後の展望

- Preemption の導入
  - 現在はFCFSスケジューリングのため、長いリクエストが短いリクエストをブロックすることがある
  - 既存研究に基づく preemption の導入により効率改善が期待される
- 耐障害性の強化
  - レプリケーションによる耐障害性がなく、prefillとdecodingの依存関係により、1つのインスタンスの障害による影響範囲が大きい

まとめ

- DisrServe: LLMサービングシステム
- 設計
  - Prefill と Decoding フェーズの処理を分離
  - 各フェーズに適した並列化戦略とGPU割り当て
  - ネットワーク帯域に応じたインスタンス配置
- 結果
  - 様々なアプリ、LLMサイズで goodput 向上（最大 7.4× ）
  - 大量のリクエスト、大規模LLMに特に有効

参考文献

[1] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, & Hao Zhang. (2024). DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving.

[2] https://developer.nvidia.com/ja-jp/blog/mastering-llm-techniques-inference-optimization/