DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving (OSDI 2024)

Akira Kawata

NOTE:

- 今日のスライドは NotebookLM をフル活用して作りました。
  - https://notebooklm.google.com/notebook/f6107573-1fd6-4e2d-99a2-0bf540ab9882
- 「オペレーティングシステム特論　輪講」 で調べた人がいたっぽい
  - A survey of DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving

1 Introduction - Prefill と Decode

- LLMの応答は 2 段階
  - プロンプトを処理して最初のトークンを生成するプレフィル
    - ここで重要なのは TTFT (Time To First Token)
  - それに続くトークンを逐次生成するデコーディング
    - ここで重要なのは TPOT (Time Per Output Token)

1 Introduction - 現状の課題

- 既存のLLMサービングシステムは、同じGPU上で両方のフェーズを行っているが問題がある
- プリフィル処理はデコーディング処理よりも時間がかかるため、これらを同じバッチに入れると、デコーディング処理がプリフィルの完了を待たされることになり、TPOTが大幅に悪化する
- プリフィルとデコーディングは計算特性や遅延要件が異なるが、同じGPUに配置されているため、それぞれに最適なリソース割り当てや並列化戦略（モデル並列化など）を個別に適用することができない
    - PP とか?

1 Introduction - 分離による解決

- これらの問題を解決するために、プリフィルとデコーディングを異なるGPUに割り当てる「分離（Disaggregation）」を提案している
- 各フェーズを独立したGPUで実行することで、相互の干渉をなくす
- 各フェーズの遅延要件（TTFTとTPOT）に合わせて、リソース割り当てと並列化戦略を個別に最適化する
- 物理的なクラスタの帯域幅を考慮して配置を決定し、分離によって発生する中間状態（KVキャッシュ）の転送によるオーバーヘッドを最小限に抑える

1 Introduction - 主な成果と貢献

- DistServeは既存の最先端システムと比較して、遅延要件を守りつつ、最大7.4倍多くのリクエストを処理できる、あるいは12.6倍厳しいSLO（サービスレベル目標）に対応できることが示された
- DistServeの目標は、各GPUでSLOを満たしながら処理できる最大リクエストレート（グッドプット）を最大化することにあり、これがクエリあたりのコスト低減に直結する
  - Goodput という単語は他の論文でも見る

2.1 LLM Inference

- プリフィルの特性: 新しいシーケンス（プロンプト）の全トークンを同時に処理するため、計算量が多く、一般に演算ネック
- デコーディングの特性: 前のステップで生成された1トークンのみを処理するため、計算量に対してメモリアクセスが多く、**メモリ帯域ネック
- KVキャッシュ: 再計算を避けるため、中間状態であるKVキャッシュをGPUメモリに保存する。既存システムでは、重みやこのキャッシュを共有するため、両フェーズを同じGPUに配置（コロケーション）するのが一般的である

2.2 LLM Serving Optimization

- バッチ処理: 「継続的バッチ処理（continuous batching）」により、新規リクエストのプリフィルと進行中のデコーディングを混ぜて処理し、スループットを最大化する
- モデル並列化
  - 演算内並列（Intra-operator parallelism）: 行列演算などを分割して実行時間を短縮し、主にTTFTを改善するが、通信オーバーヘッドが大きい => TP
  - 演算間並列（Inter-operator parallelism）: 層ごとにステージを分割（パイプライン化）し、スループットを線形に拡張する => PP

2.3 Problems and Opportunities

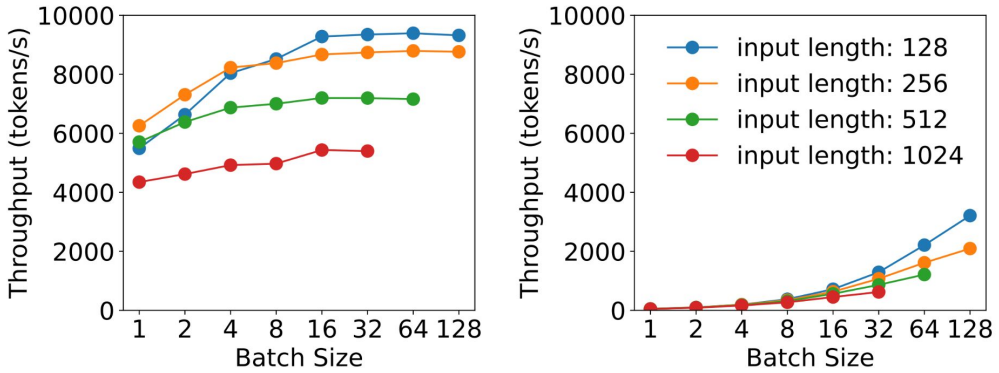
- プリフィルとデコーディングの干渉
  - プリフィルはデコーディングより実行時間がはるかに長いため、同じバッチに入れるとデコーディングが待たされ、TPOTが大幅に増加する
  - 逆に、デコーディングを混ぜることでプリフィルの完了も遅れ、TTFTが悪化する
- リソースと並列化戦略の結合
  - 両フェーズを同じGPUで動かすため、リソース割り当てや並列化の設定を共有せざるを得ない
  - 演算ネックのプリフィルには演算内並列を強めたいが、メモリ帯域ネックのデコーディングには別の構成が適しているなど、最適な戦略が異なるにもかかわらず個別最適化ができない
    - コメント: prefill => TP / decode => PP が良いはず

3.1 Analysis for Prefill Instance

- プリフィルは一般に演算集約型である。入力長がある閾値を超えるとGPUが演算ネックに達し、それ以上にバッチサイズを増やしても処理効率は向上せず、むしろリクエスト全体の遅延を増大させる。そのため、バッチサイズは小さく抑えるのが適切である
- 低いリクエスト到着率では、実行時間を短縮できる演算内並列（Intra-op parallelism）が有利である。一方で到着率が高くなると、待ち行列の遅延を抑制できる演算間並列（Inter-op parallelism）の方が平均TTFTの面で優位になる
  - コメント: ここちょっとわかってない
- TTFTの遅延目標（SLO）が厳しいほど、実行時間を直接短縮できる演算内並列がより有効な選択肢となる

3.1 Analysis for Prefill Instance

- Fig 3 (a): バッチサイズを増やしても計算が飽和してスループットが上がらない様子



(a) Prefill phase (b) Decoding phase

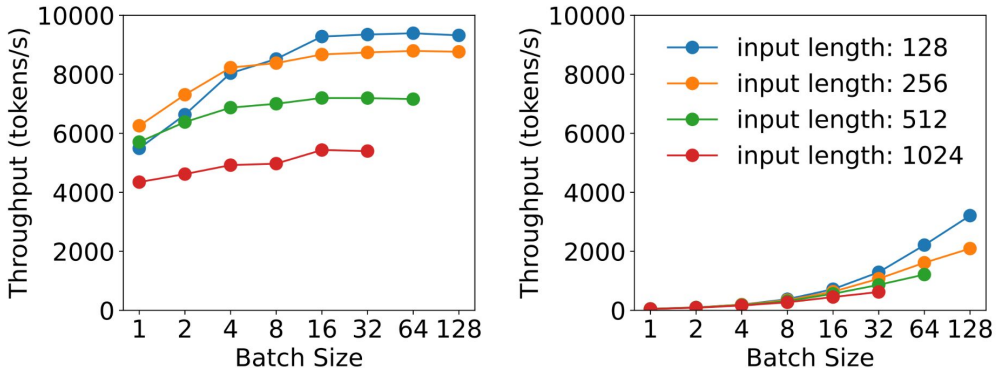
Figure 3: Throughput for two phases with different batch sizes and input lengths when serving an LLM with 13B parameters.

3.2 Analysis for Decoding Instance

- 単一のデコーディング処理はメモリ帯域ネックであるため、GPUの利用率を高めてグッドプットを最大化するにはバッチ処理が不可欠である
  - => 従来システムはメモリ帯域を使い切れていなかったらしい
- 既存の混在型システムでは、プリフィルとのリソース競合によりデコーディングのバッチサイズ拡大が困難であったが、分離構成では複数のプリフィルインスタンスからの出力を集約することで、TPOTを犠牲にすることなくバッチサイズを拡大できる
- バッチサイズが大きくなるにつれて計算特性はプリフィルに近づく。TPOT要件が厳しい場合は演算内並列 (TP) が必要だが、スループットを線形に拡張するには演算間並列 (PP) が適している

3.2 Analysis for Decoding Instance

- Fig 3 (b): decode はバッチサイズを上げたほうが良い



(a) Prefill phase (b) Decoding phase

Figure 3: Throughput for two phases with different batch sizes and input lengths when serving an LLM with 13B parameters.

3.3 Practical Problems

- プロンプトの長さが不均一な場合、演算間並列を用いるプリフィルインスタンスにおいてパイプラインの空き時間（バブル）が生じ、効率が低下する
- フェーズの分離に伴いGPU間でKVキャッシュを転送するコストが発生する。しかし、NVLINKなどの広帯域ネットワークを備えた近現代のGPUクラスタにおいては、適切な配置（Placement）アルゴリズムを用いることで、このオーバーヘッドは無視できるレベルに抑えられる
- ワークロードのパターン、物理的な配置の制約、SLO要件、並列化戦略などが複雑に絡み合うため、これらを自動的にナビゲートして最適な構成を見出す手法が必要となる

4 Method

- DistServeは、与えられたモデル、ワークロード、遅延要件（SLO）に基づき、「並列化戦略」「インスタンス数」「物理的な配置」を最適化し、GPUあたりのグッドプットを最大化することを目的としている

4.1 Placement for High Node-Affinity Cluster

- 前提条件: InfiniBandなどの高速なノード間ネットワークを備え、KVキャッシュ転送のオーバーヘッドが無視できる環境を想定している
- 総当たりで PP / TP の数を探している

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 $best\_plm$

4.2 Placement for Low Node-Affinity Cluster

- 前提条件: ノード内のNVLINKは高速だが、ノード間の帯域幅が限られている環境を想定している
- 同じステージを担当するプリフィルとデコーディングのインスタンス・セグメントを同一ノード内に配置するよう制約が入った全探索

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*

4.3 Online scheduling

- 基本ポリシー: 中央コントローラがリクエストを管理し、プリフィルは最短キューのインスタンスへ、デコーディングは最も負荷の低いインスタンスへ割り当てるFCFS（先着順）方式を採用している
- パイプライン・バブルの抑制: プロンプト長が不均一な場合、GPUを飽和させる閾値を目安にリクエストをバッチ化し、各ステージの実行時間を均衡させることでパイプラインの空き時間を最小化する
- バースト性への対策（プル方式）: プリフィル完了後にKVキャッシュを即座に「プッシュ」転送するのではなく、デコーディング側が必要な時に取得する**「プル方式」**を採用している。これにより、プリフィルのメモリをバッファとして活用し、急激な負荷変動によるメモリ不足を防ぐ
- 再プランニング（Replanning）: ワークロードのパターン（平均入力長や到着率など）の変化を監視し、大きな変動を検知した場合は自動的に配置アルゴリズムを再実行して構成を更新する

5 Implementation

- 実装に https://docs.ray.io/en/latest/ray-overview/getting-started.html を使っているらしい
  - Python から使える MPI みたいなものかな?

6 Evaluation

6.1 Experiments Setup

- テストベッド: 32枚のNVIDIA A100-80GB GPU（4ノード）を使用し、ノード内はNVLINK、ノード間は25Gbpsの帯域で接続されている。
- モデルとデータセット: モデルは**OPTシリーズ（13B, 66B, 175B）**を使用し、Chatbot（ShareGPT）、コード補完（HumanEval）、要約（LongBench）の3つのアプリケーションを想定したデータセットで評価を行っている。
- 比較対象（Baselines）：連続バッチ処理とPagedAttentionを用いるvLLMと、チャンク分割プリフィルをサポートするDeepSpeed-MIIを比較対象としている。
- 主要指標: 90%以上のリクエストがSLOを遵守できる最大レートである**「グッドプット（Goodput）」**を主な指標としている

6.2 End-to-end Experiments

- SLO を守ったときのスループット (Goodput) が良い

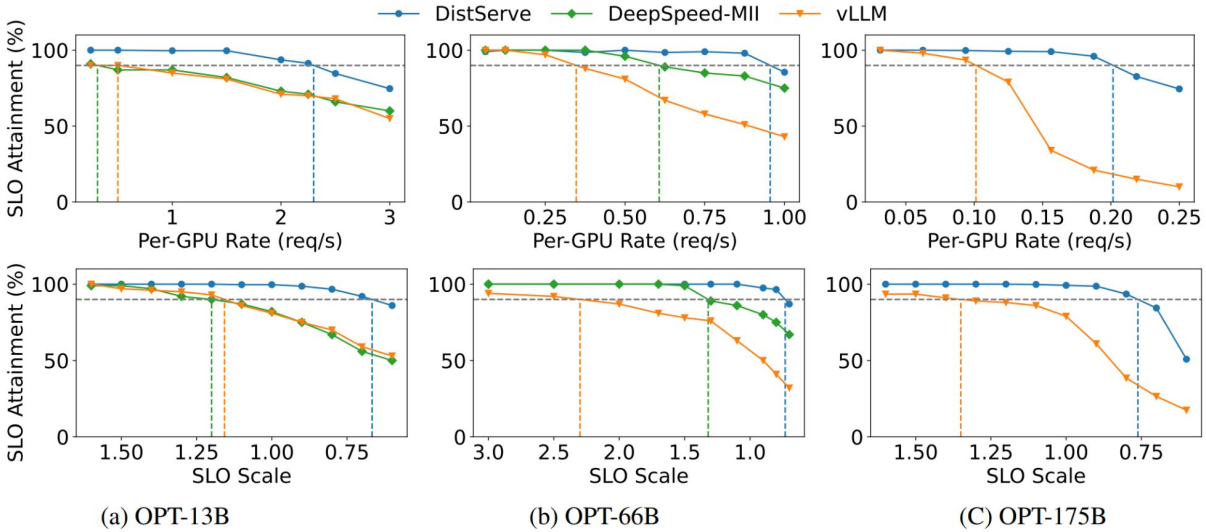


Figure 8: Chatbot application with OPT models on the ShareGPT dataset.

6.3 Latency Breakdown

- 分離によって発生する KV cache の転送時間は短い

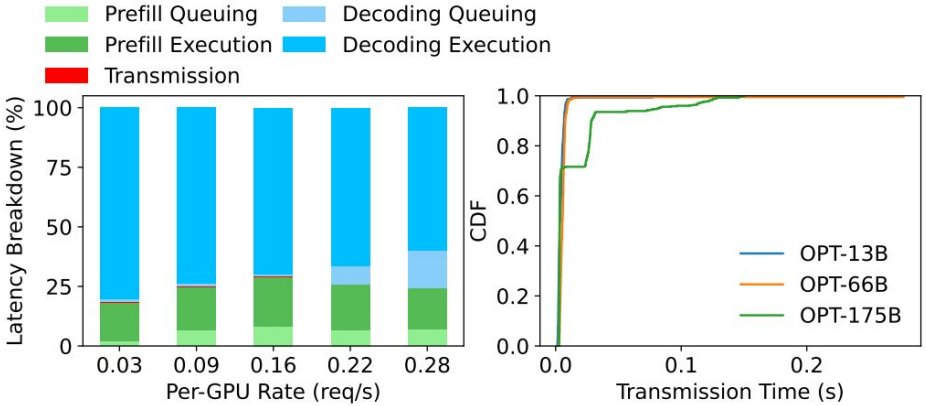


Figure 10: *Left:* Latency breakdown when serving OPT-175B on ShareGPT dataset with DistServe. *Right:* The CDF function of KV Cache transmission time for three OPT models.