LLMのオフロード推論におけるコンテキスト長に応じた重み・KVキャッシュの配置最適化

杉浦　直人†　　松谷　宏紀†

† 慶應義塾大学　〒223–8522 神奈川県横浜市港北区日吉 3–14–1
E-mail: †ko1234@keio.jp, ††matutani@arc.ics.keio.ac.jp

あらまし　モデルの重みとキーバリュー（KV）キャッシュをホストメモリにオフロードすることで、メモリ容量が限られた単一 GPU 上でも大規模言語モデル（LLM）を実行できる。デコード中に KV キャッシュが増加すると、GPU メモリ上で必要な容量や、KV キャッシュの転送および attention 計算のコストが変化する。しかし、既存のオフロード手法の多くは、計算の割り当てとモデルの重みおよび KV キャッシュの配置を静的に決定するため、これらの変化に適応できず、推論性能が低下する。本論文では、コンテキスト長を考慮したオフロード推論手法 Norn を提案する。Norn は、オフラインプロファイリングに基づき、サブレイヤ単位で計算の割り当てとデータの配置を同時に最適化し、複数の実行プランとその切り替え点を事前に決定する。デコード中には、コンテキスト長の増加に応じてプランを切り替え、計算とデータ転送をオーバーラップさせる。また、プリフィルでは、operator-major chunked prefill を用いて重みの繰り返し転送を回避し、チャンクサイズを自動的に選択する。2 種類の単一 GPU 環境で Qwen3-14B と Qwen3-32B を用いた実験の結果、Norn は Pipelined Sharding に対して End-to-End レイテンシで最大 16.8 倍、入出力長を事前に既知とするその oracle 版に対して最大 4.2 倍の高速化を達成した。

キーワード　大規模言語モデル、推論、オフロード、スケジューリング

1. はじめに

LLM の活用は、短い対話から、複雑な問題の論理的思考や大量の文書を扱う長文処理へと多様化している。 このような用途の拡大に伴い、LLM をクラウドだけでなく、PC やスマートフォンなどのローカル環境で実行する動きも広がっている。個人情報や機密情報の外部送信に伴うセキュリティ上の懸念も、ローカル推論への需要を高めている [1]。端末内で動作する言語モデルを用いた文章作成や要約などの AI 機能が実用化され [2], [3]、ゲーム内 AI など、応答性が求められる用途への応用も進められている [4]。高スループットを重視するデータセンター環境とは対照的に、ローカル環境では小さなバッチを低レイテンシで処理することが重視される。

しかし、パラメータ数の多い LLM の実行には大量の GPU メモリが必要であり、コンシューマ向け GPU の限られたメモリ容量が大きな障壁となる。推論では、モデルの重みに加え、過去のトークン情報を保持する KV キャッシュもメモリを消費する。重みのメモリ使用量が一定であるのに対し、KV キャッシュはコンテキスト長に比例して増加する。例えば、Qwen3-32B では、FP16 の場合、128K トークンの KV キャッシュだけで約 32 GiB に達し、コンシューマ向け GPU のメモリ容量に匹敵する。

必要な KV キャッシュ容量はリクエストの入出力長によって大きく異なる。特に、長い出力を生成する場合、デコード中に KV キャッシュが増加し、最終的な出力長も事前には分からない。そのため、推論開始時に必要な GPU メモリ量を適切に見積もることは容易ではない。また、異なるメモリ要求を持つリクエストを混在して処理する場合、リクエスト毎に必要な容量を設定する必要がある。したがって、ローカル LLM 推論では、限られた GPU メモリで多様な入出力長のリクエストに対応しながら、低レイテンシな推論を実現することが重要となる。

GPU メモリ容量の制約を緩和する有力な手段が、ホストメモリを活用するオフロード推論である。オフロード推論では、重みや KV キャッシュの一部をホストメモリに保持し、必要に応じて GPU へ転送することで、GPU メモリ容量を超える LLM の実行を可能にする。一方で、CPU と GPU の間のデータ転送は新たなオーバーヘッドとなるため、単にデータをオフロードするだけでは十分な性能を得られない。そこで、CPU と GPU への計算の割り当てや、重み・KV キャッシュの配置を最適化し、計算とデータ転送をオーバーラップさせることでレイテンシを削減する手法が提案されている [4]〜[6]。

しかし、既存のオフロード手法では、計算の割り当てと配置を、生成に伴うコンテキスト長の増加に応じて変更することは想定されていない。例えば、Pipelined Sharding [4] は処理する新規トークン数に応じて実行プランを選択するが、バッチサイズ 1 のデコード中に増加するコンテキスト長を考慮した配置の切り替えは行わない。このような静的配置には、サポートするコンテキスト長と推論性能の間にトレードオフが存在する。長いコンテキストに備えて KV キャッシュ用の GPU メモリを確

ⓒ2026 Information Processing Society of Japan

保すると、その分だけ GPU に常駐できる重みが減少し、実際には、短いリクエストを処理する場合、推論速度が低下する。反対に、短いコンテキストに合わせて多くの重みと KV キャッシュを GPU に常駐させると、生成の進行に伴う KV キャッシュの増加によって GPU メモリが不足し、同じ配置では推論を継続できなくなる場合がある。Pipelined Sharding を用い、Qwen3-14B で測定したところ、入出力長を固定したままサポートする最大コンテキスト長を 2K から 128K に増やすと、1 トークンあたりのデコード時間が 18.7 倍に増加することを確認した。

さらに、コンテキスト長の増加はメモリ要求だけでなく、オフロードの実行効率そのものを変化させる。KV キャッシュの転送時間と attention の計算時間はコンテキスト長に応じて増加する一方、重みの転送時間や Feed-Forward Network（FFN）の計算時間は一定である。その結果、計算と転送の相対的なコストが変化し、最適な計算割り当てや配置も推論中に変わりうる。したがって、あるコンテキスト長で効率的な配置が、その後のデコードにおいても効率的であるとは限らない。

そこで、本研究では、コンテキスト長の増加に応じて実行プランを動的に切り替えるオフロード推論手法 Norn を提案する。Norn は、オフラインプロファイリング結果を用いて、attention と FFN のサブレイヤ単位で計算の割り当てと重み・KV キャッシュの配置を最適化する。コンテキスト長に応じた性能変化とプラン切り替えに伴うデータ移動のコストを考慮し、複数の実行プランとその切り替え点を事前に決定する。デコード時には、Norn はプランを切り替え、GPU メモリの割り当てを調整し、計算とデータ転送をオーバーラップさせる。プリフィルでは、operator-major chunked prefill を用いて、アクティベーションのメモリ使用量を抑えながら重みの繰り返し転送を削減する。また、プリフィルのレイテンシとデコードへの遷移コストを考慮して、チャンクサイズを自動的に選択する。2 種類のデスクトップ環境で Qwen3-14B と Qwen3-32B を評価した結果、Norn は Pipelined Sharding に対して推論レイテンシを最大 16.8 倍高速化し、入出力長を既知とする oracle 版に対しても最大 4.2 倍の高速化を達成した。

要約すると、本論文の貢献は以下のとおりである。

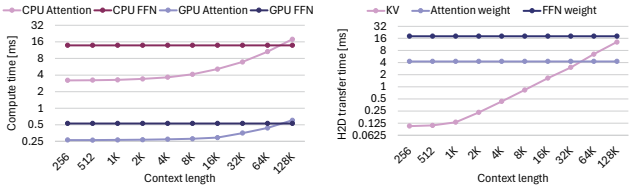
- 静的配置では、将来の KV キャッシュ容量を見込んだ GPU メモリ確保が、短いコンテキストでのデコード性能を低下させることを示した。
- 計算割り当てと配置を最適化した複数の実行プランと切り替え点を事前に決定する動的プランニング手法を提案した。
- 長い入力でのメモリ使用量を抑えつつ、オフロードした重みの再転送を回避するプリフィル戦略を提案し、初期デコードへの移行コストを考慮したチャンク長の自動選択を導入した。
- 2 種類の GPU 環境と Qwen3-14B/32B を用いて評価し、多様な設定で既存手法より短い推論レイテンシを達成した。

2. 背　　景

2.1 LLM のオフロード推論のスケジューリング

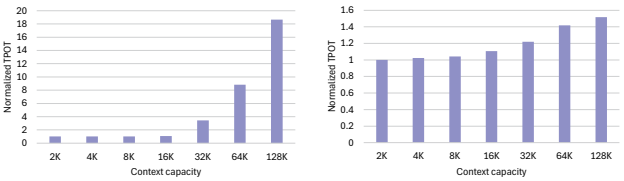
LLM のオフロード推論では、重みと KV キャッシュを GPU メモリまたはホストメモリに配置し、モデルの計算を CPU と GPU に割り当てることができる。GPU で計算する場合、重みを GPU メモリに常駐させる resident 実行と、重みをホストメモリに保持し、実行時に GPU へ転送する streaming 実行がある。また、attention が参照する KV キャッシュについても、GPU への常駐と実行時の転送を選択できる。

計算中に後続の GPU 計算で必要となる重みや KV キャッシュを転送することで、レイテンシを削減できる。既存手法では、このオーバーラップを考慮して、CPU・GPU への計算割り当てと配置を最適化する [4]〜[6]。例えば、Layer-wise Scheduling [6] は、各層を CPU、GPU resident、GPU streaming に割り当て、CPU 計算中に後続層の重みを転送することで、オーバーラップを最大化する。Pipelined Sharding [4] は、attention と FFN を単位として、attention と KV キャッシュの GPU 常駐を優先しながら、CPU・GPU への計算割り当てと配置を決定する。CPU 計算と GPU streaming を併用するプランでは、CPU 計算中に 1 つ後のサブレイヤの重みを先行転送し、計算と転送をオーバーラップさせる。



(a) 計算時間　　(b) 転送時間

図 1: Qwen3-32B のサブレイヤあたりの計算時間と転送時間



(a) Qwen3-14B　　(b) Qwen3-32B

図 2: 固定コンテキスト容量がデコード性能に与える影響。TPOT は各モデルごとに 2K 設定で正規化されている。

2.2 計算・転送コストの変化

コンテキスト長の増加に伴う計算・転送コストの変化は、一様ではない。図 1 に、Qwen3-32B のサブレイヤあたりの計算時間と転送時間を示す。コンテキスト長が 256 から 128K へ増加すると、CPU での attention 計算時間は 5.6 倍、GPU では 2.3 倍に増加する。Attention は過去のすべての KV キャッシュを参照するため、コンテキスト長の増加に伴って計算量が増加する。また、KV キャッシュの転送量はコンテキスト長に比例して増加し、転送時間は 123 倍と大幅に増加する。一方、CPU および GPU での FFN の計算時間は一定であり、サブレイヤの重み転送時間もコンテキスト長に依存しない。Qwen3-14B においても同様の傾向が確認された。このように、計算と転送の相対的なコストが変化するため、効率的な計算割り当てや配置もコンテキスト長によって変わりうる。

ⓒ2026 Information Processing Society of Japan

2.3 静的配置の限界

既存手法の多くは、CPU・GPU への計算割り当てと重み・KV キャッシュの配置をデコード中に変更しない。KV キャッシュはコンテキスト長に比例して増加するため、静的配置では将来の KV キャッシュ容量を見込んで GPU メモリを配分する必要がある。例えば、Pipelined Sharding や llama.cpp [7] では、サポートする最大コンテキスト長を指定する。また、Hugging Face Accelerate [8] や Layer-wise Scheduling では最大コンテキスト長を指定しないものの、KV キャッシュの増加を見込んで GPU に配置する重みの量を制限する必要がある。

このような静的配置は、単一リクエストのデコード中と、コンテキスト長の異なるリクエスト間の双方で性能上の問題を招く。第一に、論理的思考のような長い出力を生成するリクエストでは、生成初期のデコード性能が低下する。静的配置では、生成終盤の KV キャッシュ容量を見込んで GPU メモリを確保するため、生成初期には KV キャッシュが小さいにもかかわらず、GPU に常駐できる重みが制限される。その結果、短いコンテキスト長に適した配置と比較して、CPU 計算や重み転送による性能低下が生じる。さらに、2.2 節で示した計算・転送コストの変化により、単一の配置がデコード全体を通して効率的とは限らない。第二に、コンテキスト長の異なるリクエストが混在する場合にも性能が低下する。長いリクエストに対応できるように GPU メモリを配分すると、短いリクエストでも重みのオフロードが増加し、推論性能が低下する。反対に、短いコンテキスト長に合わせると、長いリクエストでは KV キャッシュの増加によって GPU メモリが不足し、実行できなくなる。リクエストごとに最大コンテキスト長を設定することも考えられるが、実際の出力長は事前に不明であり、ユーザーが各リクエストに適切な設定を手動で行うことは現実的ではない。

この影響を定量的に確認するため、Pipelined Sharding を用いて、入力長を 1024、出力長を 32 に固定し、サポートする最大コンテキスト長のみを 2K から 128K まで変更した。図 2 に結果を示す。Qwen3-14B では、16K 設定までは time per output token (TPOT) の変化が小さいものの、32K 設定では 2K 設定の約 3.4 倍、128K 設定では約 18.7 倍に増加した。これは、2K 設定ではすべての重みと KV キャッシュを GPU に配置できるのに対し、最大コンテキスト長を大きくすると、KV キャッシュ用のメモリを確保するために重みのオフロードが増加するからである。Qwen3-32B でも、128K 設定では 2K 設定と比較して TPOT が 51.8% 増加した。これらの結果は、実際に処理するコンテキスト長が同一であっても、将来の KV キャッシュ容量を見込んだ静的配置によってデコード性能が低下することを示している。特に、リクエストごとに最大コンテキスト長を適切に設定できたとしても、生成初期から終盤まで同一の配置を使用する限り、デコード中の性能変化には適応できない。したがって、コンテキスト長の増加に応じて、重みと KV キャッシュの配置を動的に変更する必要がある。

2.4 動的配置の課題

動的配置では、実行プランの生成と切り替え点の決定を一体として扱う必要がある。各プランは次の切り替えまで増加する KV キャッシュを収容しなければならず、そのために確保する GPU メモリ容量が重みの配置と推論性能を制約する。したがって、どの配置を選択するかは、その配置をどのコンテキスト長まで利用するかに依存する。実際のリクエストでは入出力長が多様であり、特に出力長は推論開始時には未知であるため、この決定をより一層困難にする。

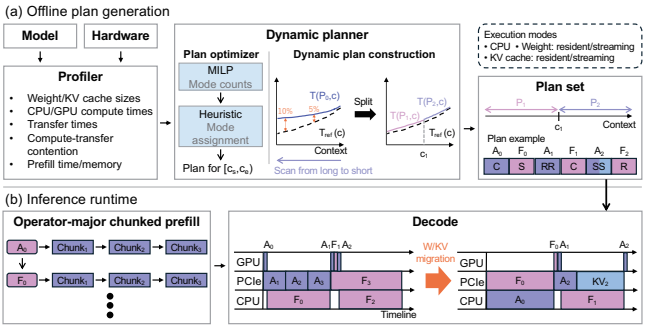


図 3: Norn の概要図

実行プランの生成には、組合せ的な配置探索の難しさがある。各 attention と FFN について、CPU・GPU への計算割り当てと重み・KV キャッシュの配置を決定する必要があり、候補数はサブレイヤ数に対して組合せ的に増大する。TwinPilots [5] は、CPU・GPU への演算割り当てを NP 完全な負荷分散問題として定式化している。このような探索をデコード中に繰り返すことは、大きな計算オーバーヘッドを伴う可能性がある。

切り替え点の決定には、GPU メモリ容量の制約だけでなく、コンテキスト長に伴う実行プラン間の性能変化を考慮する必要がある。あるプランがメモリ制約を満たしていても、コンテキスト長の増加によって別のプランの方が高速になる場合がある。一方、プランの切り替えでは重みや KV キャッシュの移動が伴い、頻繁な切り替えはマイグレーションコストを増加させる。したがって、各コンテキスト長で最速の配置を独立に選択するのではなく、各プランの実行性能と切り替えコストを考慮し、プラン集合とその適用範囲を共同で決定する必要がある。

3. 提案手法

3.1 Norn の概要

Norn は、コンテキスト長に応じて実行プランを切り替えるオフロード推論手法である。図 3 に示すように、Norn は、Profiler、Dynamic Planner、Inference Runtime の 3 つから構成される。Profiler は、計算時間、転送時間、および両者の競合特性を計測する。Dynamic Planner は、計測結果に基づいて、コンテキスト長に応じた複数の実行プランとその切り替え点をオフラインで決定する。Inference Runtime は、プリフィルでは GPU 計算を用い、長い入力には operator-major chunked prefill を適用する。デコードでは、各プランに従って計算と転送をオーバーラップさせ、コンテキスト長が切り替え点に達すると必要なデータを再配置する。これにより、出力長の予測や推論中の再最適化を必要とせず、動的配置を実現する。

3.2 Profiler

Profiler は、対象モデルと実行環境に応じた性能最適化を行う

ため、実機で計算・転送時間を計測する。また、モデルの構成情報から、各サブレイヤの重みサイズと 1 トークン当たりの KV キャッシュサイズを取得する。デコードでは、各サブレイヤの CPU と GPU の実行時間、重み、KV キャッシュ、activation の転送時間、計算と転送の競合特性、および GPU 演算発行時のホスト側オーバーヘッドを実機で計測する。コンテキスト長に依存する attention の計算時間と KV キャッシュの転送時間は、複数のコンテキスト長で計測し、測定点間を線形補間する。プリフィルでは、複数の入力長とチャンク長に対して、各サブレイヤの GPU 実行時間、およびピーク GPU メモリ使用量を計測する。

3.3 Dynamic Planner

Dynamic Planner は、Plan Optimizer による単一プランの最適化と、性能劣化および切り替えコストを考慮したプラン集合と切り替え点の決定からなる。

3.3.1 Plan Optimizer

Plan Optimizer は、コンテキスト範囲 $I = [c_s, c_e)$ に対して、推定総デコード時間の最小化を目的に、単一の実行プラン P を生成する。プラン P は、各層の attention と FFN の実行モード、および重みと KV キャッシュの転送に用いるバッファ容量 B から構成される。FFN には、CPU 実行、GPU resident 実行、GPU streaming 実行の 3 種類を設ける。Attention には、CPU 実行に加えて、重みと KV キャッシュそれぞれについて resident または streaming 実行を独立に選択することで、4 つの GPU 実行モードを設ける。

Plan Optimizer は、各モードの採用数と層間の割り当て順序を分離して最適化する。L 層のモデルでは、実行モードの組み合わせは最大 15^L 通りとなる。さらに、計算とデータ転送のオーバーラップは、モードの割り当て順序や転送の開始・完了時刻などの時間的制約に依存するため、厳密な最適化問題として定式化すると計算コストが大きくなる。しかし、各モードに割り当てるサブレイヤ数が決まれば、GPU メモリ使用量と計算・転送の総時間を推定できる。そこで、Plan Optimizer では、まず混合整数線形計画法（MILP）を用いて各モードの採用数を決定し、その後、シミュレータに基づくヒューリスティック探索によって割り当て順序を決定する。これにより、実行順序に依存する計算とデータ転送のオーバーラップを考慮しながら、MILP の整数変数の数を削減する。

MILP では、Profiler の計測結果に基づくデコード時間の下界を最小化する。コンテキスト範囲内の全デコードステップを個別に評価すると計算コストが大きいため、範囲を等間隔に S 個の区間に分割し、各区間の中点 c_i を代表点として用いる。そして、w_i を各代表点が表すデコードステップ数として重み付けする。各代表点 c_i における TPOT の下界を τ_i とし、次の目的関数を最小化する。

$$\min \sum_{i=1}^{S} w_i \tau_i \tag{1}$$

各サブレイヤ種別について、モードの採用数の合計が層数 L に一致する制約を課す。サブレイヤ種別の集合を $\mathcal{Q} = \{\text{Attention}, \text{FFN}\}$、種別 $q \in \mathcal{Q}$ で選択可能なモードの集合を $\mathcal{M}_q$ とする。整数変数 $x_{q,m} \in \{0, \ldots, L\}$ を、種別 q でモード m を採用するサブレイヤ数とすると、制約は次式で表される。

$$\sum_{m \in \mathcal{M}_q} x_{q,m} = L \tag{2}$$

GPU メモリ制約では、コンテキスト範囲の終端における重みと KV キャッシュの常駐容量を用い、デコード中の KV キャッシュの増加を考慮する。モード m のサブレイヤがコンテキスト長 c_e で必要とする GPU 常駐容量を $R_{q,m}(c_e)$、固定 GPU 使用量を M_{base}、GPU メモリ容量を M_{GPU} とし、次の制約を課す。

$$M_{\text{base}} + B + \sum_{q \in \mathcal{Q}} \sum_{m \in \mathcal{M}_q} x_{q,m} R_{q,m}(c_e) \le M_{\text{GPU}} \tag{3}$$

また、転送対象がバッファ B に収まらないモードは除外し、必要に応じてホストメモリの容量制約も課す。

各代表点 c_i において、モードの採用数 $\mathbf{x}$ に基づき、計算、host-to-device（H2D）転送、device-to-host（D2H）転送それぞれの単独実行時間を合計した値を $C_i(\mathbf{x})$、$H_i(\mathbf{x})$、$D_i(\mathbf{x})$ とする。TPOT の基本的な下界制約として、次式を課す。

$$\tau_i \ge C_i(\mathbf{x}), \quad \tau_i \ge H_i(\mathbf{x}), \quad \tau_i \ge D_i(\mathbf{x}) \tag{4}$$

計算と転送の競合を考慮し、両者のオーバーラップが実行時間に与える影響を近似する。演算種別 a の計算時間と転送種別 b の転送時間の合計を、それぞれ $C_{i,a}(\mathbf{x})$、$H_{i,b}(\mathbf{x})$ とする。これらは競合がない場合の実行時間を、各モードの採用数に応じて集計したものである。両者の重複実行時間を $z_{i,a,b} \ge 0$、計測結果から推定した競合時の相対処理速度を $r^C_{i,a,b}$、$r^H_{i,a,b}$ とする。ここで、相対処理速度は、各種別の単独実行時の処理速度をそれぞれ 1 として正規化した値である。計算と転送を同時実行した際の、逐次実行に対する単位時間当たりの短縮量 $\gamma_{i,a,b} = r^C_{i,a,b} + r^H_{i,a,b} - 1$ を用いて次の制約を追加する。

$$\sum_b r^C_{i,a,b} z_{i,a,b} \le C_{i,a}(\mathbf{x}), \tag{5}$$

$$\sum_a r^H_{i,a,b} z_{i,a,b} \le H_{i,b}(\mathbf{x}), \tag{6}$$

$$\tau_i \ge C_i(\mathbf{x}) + H_i(\mathbf{x}) - \sum_{a,b} \gamma_{i,a,b} z_{i,a,b} \tag{7}$$

重複実行時間 $z_{i,a,b}$ の間に進む計算と転送の処理量は、それぞれ $r^C_{i,a,b} z_{i,a,b}$、$r^H_{i,a,b} z_{i,a,b}$ で表される。最初の 2 つの制約は、これらの処理量の合計が、各種別の処理全体を超えないように制限する。最後の制約は、重複実行による時間短縮量を考慮し、TPOT の下界を定める。

以上の目的関数と制約に基づき、MILP により各モードの採用数を決定する。各層のモード選択を直接変数化する場合、$8L$ 個の二値変数が必要となる。これに対し、本手法では採用数を表す 8 個の整数変数に削減できる。各サブレイヤ種別の採用数の合計は L となるため、採用数の組合せ数は $O(L^6)$ 通りとなり、指数的な配置探索空間を縮約できる。

一方、MILP では各層への割り当て順序を決定しないため、次にシミュレータを用いて各層への割り当てを決定する。配置割

り当てでは、プラン P の総実行時間を $\widehat{T}(P,I)=\sum_{i=1}^{S} w_i T(P,c_i)$ と近似し、これを評価関数として用いる。ここで、$T(P,c_i)$ はシミュレータが推定する TPOT であり、サブレイヤ間の依存関係と計算・転送の実行スケジュールを考慮する。

まず、attention と FFN それぞれについて、各モードの採用数を重みとする平滑化重み付きラウンドロビンを用い、各モードを層全体に分散させた初期配置を生成する。具体的には、各モードの目標採用比率に対して累積割り当て数が最も不足しているモードを優先して割り当てる。次に、各サブレイヤのモード列を個別に循環シフトし、両者の相対的な配置順序を調整する。一方のモード列を固定して他方を循環シフトした配置をシミュレータで評価し、$\widehat{T}(P,I)$ が最小となる配置を選ぶ。この操作を各サブレイヤについて交互に繰り返す。

続いて、同種の異なる 2 層に割り当てられた実行モードを交換する局所探索を行う。探索では $\widehat{T}(P,I)$ を比較し、局所最適解への停滞を避けるため、悪化する交換も一定確率で受理する。探索中の最良プランを保持し、最終的に採用する。交換ではモードの採用数が変わらないため、MILP で決定した GPU 常駐容量を維持できる。

転送用バッファ容量 B についても複数の候補を探索する。B を大きくすると事前転送の余地が増える一方、GPU に常駐できる重みや KV キャッシュの容量が減少する。そこで、転送対象のサイズから得られる容量の境界とシミュレーションで検出したバッファ不足を用いて候補を生成し、各候補で MILP と配置順序の最適化を実行する。最終的に、$\widehat{T}(P,I)$ が最小となる実行可能なプランを選択する。

3.3.2 Dynamic Plan Construction

Dynamic Planner の目的は、各プランの適用区間における総デコード時間と、プラン切り替えに伴うデータ移動時間の和を最小化することである。Dynamic Planner では、コンテキスト範囲を再帰的に分割した後、切り替えコストを考慮して境界を調整し、不要な区間を統合する。区間分割には、参照レイテンシ曲線に対する各プランの性能劣化を用いる。コンテキスト長の増加に伴い、attention の計算時間や KV キャッシュの転送時間も増加するため、TPOT の増加だけではプラン変更による改善余地を判断できない。そこで、各コンテキスト長で実行可能な参照プランの最小 TPOT を基準とし、性能劣化が大きい領域を区間分割の候補とする。

まず、プランの性能劣化を評価する基準として、参照レイテンシ曲線を構築する。対象コンテキスト範囲から複数の参照点を選択し、各参照点に対して Plan Optimizer を用いてポイントプランを生成する。ポイントプランは、その参照点における TPOT を最小化するプランである。参照プラン集合を $\mathcal{P}_{\mathrm{ref}}$ とすると、コンテキスト長 c における参照レイテンシは $T_{\mathrm{ref}}(c)=\min_{P\in\mathcal{P}_{\mathrm{ref}}} T(P,c)$ で表される。プラン P の参照曲線に対する性能劣化率を、$\delta(P,c)=\frac{T(P,c)-T_{\mathrm{ref}}(c)}{T_{\mathrm{ref}}(c)}$ と定義する。

Dynamic Planner は、コンテキスト長の大きい側から小さい側へ向かって区間を分割する。これは、KV キャッシュのサイズがコンテキスト長に対して単調増加するためである。あるプランが区間の終端で GPU メモリ制約を満たせば、それより短いコンテキスト長でも実行可能である。一方、短いコンテキスト長から順にプランを決定する場合、そのプランをどこまで使用するかを考慮して KV キャッシュ用のメモリを予約する必要があり、予約容量が不足すると強制的なプラン切り替えが発生する。予約容量によって、プランの性能自体も変わるため、適切な容量を決定するのは容易ではない。これに対し、長いコンテキスト長から区間を決定すれば、既知の区間終端に基づいてメモリ容量を確保できるため、KV キャッシュの増加による予定外の切り替えを回避できる。

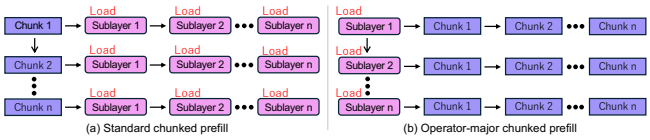


図 4: Chunked prefill の実行方式の比較

Dynamic Planner は、各プランの適用区間において、参照曲線からの性能劣化率が 10% を超える場合、その区間を再帰的に分割する。区間分割では、各プランの適用区間を長いコンテキスト長側から短い側へ走査し、参照曲線からの性能劣化率が 5% を超える最初の点を特定する。その後、さらに短い側へ走査し、10% を超える最初の点を特定する。10% は分割の必要性を判定する閾値、5% は分割点を決定する閾値として用いる。この分割規則に基づき、まず対象コンテキストの全範囲 $I=[1,c_{\max})$ をカバーする初期プランを生成する。このとき、$c_{\max}$ が分割点の探索開始点となることを防ぐため、$c_{\max}$ における性能劣化率を 5% 以内とする条件を設ける。この条件を満たさない場合、最適化対象を全範囲 I から後半の区間 $I'=[(1+c_{\max})/2,c_{\max})$ に変更し、プランを再生成する。再生成したプランは、メモリ使用量の単調増加性により、元の全範囲 I でも実行可能である。続いて、初期プランに対して分割規則を適用する。分割が必要な場合には、暫定的な切り替え点で区間を分割し、分割後の各区間に対して Plan Optimizer を用いてプランを再生成する。この処理を、分割可能な性能劣化がなくなるまで繰り返す。

再帰的な区間分割によって得られた切り替え点は、性能劣化率に基づく暫定的な境界であり、必ずしも総実行時間を最小化する位置とは限らない。そこで、短いコンテキスト長側から順に隣接プランの TPOT を比較し、両プランの性能が交差する位置、またはメモリ容量による実行可能範囲の境界へ切り替え点を調整する。境界を変更する場合には、新しい適用区間に対してプランを再生成し、切り替えに伴うデータ移動時間を含めた総実行時間が短縮される場合に変更を採用する。

さらに、区間分割によるデコード時間の短縮が、プラン切り替えに伴うデータ移動コストに見合わない場合がある。そこで、隣接する 2 区間を統合した範囲に対して新たなプランを生成し、統合前後の総実行時間を比較する。この際、各区間のデコード時間に加え、切り替え時に発生する重みと KV キャッシュの転送時間も考慮する。統合によって総実行時間が短縮される場合には、その切り替え点を削除する。最終的なプラン集合には、各プランの適用コンテキスト範囲、各サブレイヤの実行モード、転送用バッファの容量、およびプラン切り替え時の

ⓒ2026 Information Processing Society of Japan

データ移動情報が含まれる。

3.4 Inference Runtime

Inference Runtime は、プリフィルとデコードで異なる実行方式を用いる。プリフィルでは、CPU 計算が PCIe 転送よりも遅い場合が多いため、全ての計算を GPU に割り当てる。後続サブレイヤの重み転送と GPU 計算をオーバーラップさせることで、レイテンシを削減する。Norn は、転送バッファサイズの自動選択と operator-major chunked prefill（OMCP）により、重み転送と activation のメモリ使用量を最適化する。

Norn は、GPU 計算と重み転送のオーバーラップに必要な転送バッファ容量を、Profiler の計測結果に基づいて決定する。attention と FFN の GPU 計算時間をそれぞれ C_A, C_F、重み転送時間を H_A, H_F とする。後続 1 サブレイヤを先行転送する場合の 1 層当たりの定常実行時間は $T_1 = \max(C_A, H_F) + \max(C_F, H_A)$ であり、計算と転送を完全にオーバーラップした場合の理論的下界は $T_{\mathrm{lb}} = \max(C_A + C_F, H_A + H_F)$ である。$T_1 = T_{\mathrm{lb}}$ であれば、後続 1 サブレイヤの先行転送で下界を達成できる。そこで、$T_1 \le T_{\mathrm{lb}}$ の場合は 1 サブレイヤ、そうでなければ 2 サブレイヤ先まで先行転送する。転送バッファ容量は、先行転送数と重みサイズから計算する。これにより、オーバーラップ効率を考慮しながら、先行転送に必要な最小容量を確保する。

長い入力では、activation のメモリ使用量を削減するため、入力をチャンクに分割する。図 4 に示すように、一般的な chunked prefill では、チャンクごとに全層を処理するため、オフロードした重みの再転送が発生する。そこで、Norn は OMCP を採用し、同じサブレイヤの全チャンクを処理してから次のサブレイヤへ進むことで、重みの再転送を回避する。Layered Prefill [9] は、MoE モデルのサービング環境を対象に、層グループ単位の処理によって chunked prefill に伴う GPU のオフチップメモリからのエキスパート重みの再読み込みを削減する手法である。一方、同論文では、Dense モデルに適用すると従来の chunked prefill より性能が低下すると報告している。これに対し、本論文では単一 GPU のオフロード環境において、Dense モデルでもサブレイヤ単位のチャンク処理による重み転送の削減が有効であることを示す。

通常の chunked prefill では、チャンク単位の hidden state のみを保持するのに対し、OMCP では次のサブレイヤへ渡すために入力系列全体の hidden state を保持する。入力長を N、チャンク長を M、hidden size を d、要素当たりのバイト数を b とすると、その増加量は約 $(N - M)db$ バイトとなる。例えば、Qwen3-32B、FP16、$N = 128K$、$M = 1K$ の場合、1.24 GiB 増加する。したがって、OMCP は、通常の chunked prefill より多くの activation メモリを必要とするものの、重みの再転送を回避できる。ただし、チャンク化しないプリフィルよりは、ピークメモリ使用量を大幅に減らすことができる。

チャンク長はプリフィルの実行時間と activation のメモリ使用量に影響する。チャンク長を小さくすると、activation のピークメモリ使用量が下がり、初期デコードプランで必要となる重みをプリフィル後も GPU 上に保持しやすくなる一方、計算と転送のオーバーラップ率が低下する場合がある。そこで、Norn は Profiler の計測結果とメモリ使用量の推定値を用い、プリフィルの実行時間と初期プランへの移行に必要な重みの転送時間の合計を評価する。GPU メモリ制約を満たす候補のうち、推定実行時間が最小となるチャンク長を選択する。プリフィル中に生成した KV キャッシュは初期デコードプランで指定されたデバイスに配置し、GPU メモリ容量が許す範囲で初期プランに必要な重みを保持する。これにより、プリフィルからデコードへの移行に伴うデータ転送を削減する。

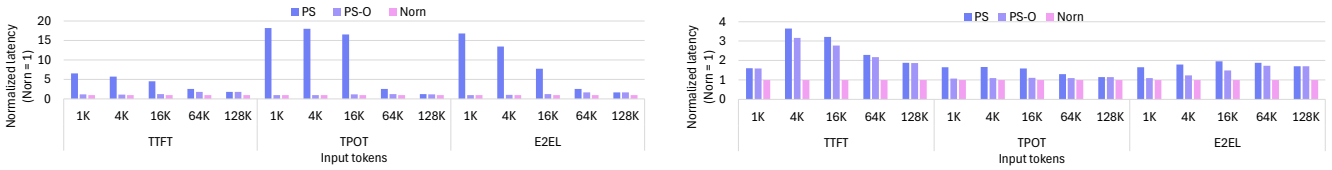
デコードでは、各サブレイヤをプランで指定された CPU または GPU 上で実行する。CPU 実行ではホストメモリ上のデータを使用し、GPU 実行では重みと KV キャッシュの配置に応じて GPU 常駐データまたはホストから転送したデータを使用する。GPU に常駐しないデータは後続サブレイヤで必要となる前に転送し、計算とオーバーラップさせる。コンテキスト長が切り替え点に達すると、トークン境界でプランを切り替える。切り替え前後の配置差分に基づいて、不要な GPU 常駐重みの解放、新たに必要な重みの転送、KV キャッシュの再配置、および転送バッファ容量の変更を行い、既存の GPU 常駐データやメモリ領域を可能な限り再利用する。

重みと KV キャッシュの H2D 転送には共通の CUDA stream を使用し、サブレイヤの実行順序に従って転送を発行する。先行する転送対象に対してバッファの空き容量が不足する場合は、後続の転送を先に発行せず、空き領域が確保されるまで待機する。また、CPU と GPU の間で実行デバイスが変わる場合には、activation を転送する。この転送には独立した CUDA stream を使用し、サブレイヤ間のデータ依存関係を満たすよう同期する。Streaming 実行では、重みと過去の KV キャッシュをホストメモリに保持するため、GPU で新たに生成した KV キャッシュのみを D2H 転送する。この転送は attention の計算完了後に専用の CUDA stream で非同期に実行し、後続の計算とオーバーラップさせる。

3.5 実装の詳細

Norn の Inference Runtime は、PyTorch と Hugging Face Transformers [10] を用いて実装した。Norn は特定の Dense モデルに依存しない。また、Norn 自体は量子化を伴わないロスレスな手法であり、量子化との併用も可能である。GPU 上の attention 計算には PyTorch の scaled dot-product attention を使用する。CPU 上の attention 計算には、NEO [11] の並列化方式を参考にした C++、ISPC、OpenMP による専用カーネルを使用する。このカーネルは、GQA の KV ヘッドを複製せず、KV キャッシュをコンテキスト方向に分割して attention を並列化する。ホストメモリ上の重みと KV キャッシュに pinned memory を使用している。各サブレイヤの重みは連続したメモリ領域に格納し、サブレイヤ単位でまとめて転送する。GPU 常駐 KV キャッシュは、最大コンテキスト長の容量を一括確保せず、プランの適用区間の終端とリクエストの生成上限に応じて容量を確保し、プラン切り替え時に必要に応じて拡張する。

ⓒ2026 Information Processing Society of Japan



(a) Qwen3-14B

(b) Qwen3-32B

図 5: デスクトップ 1 における入力長ごとの Norn を基準に正規化した TTFT、TPOT、E2EL。出力長を 32 トークンに固定している。

4. 評　　価

4.1 実験設定

Testbeds. 実験には 2 種類のデスクトップを使用した。デスクトップ 1 は、Intel Core Ultra 7 265K、128 GiB のメインメモリ、および 32 GiB の GPU メモリを搭載した NVIDIA RTX 5090 から構成される。CPU と GPU は PCIe 5.0 × 16 で接続されている。デスクトップ 2 は、Intel Core i9-10980XE、128 GiB のメインメモリ、および 24 GiB の GPU メモリを搭載した NVIDIA RTX 3090 Ti から構成される。この環境では、CPU と GPU は PCIe 3.0 × 16 で接続されている。全実験で CPU の計算には、8 スレッド割り当てた。

Models. 評価には Qwen3-14B と Qwen3-32B [12] を FP16 で使用した。異なるモデル規模を用いることで、重みおよび KV キャッシュによるメモリ要求が異なる条件で評価を行う。

Workloads. インタラクティブな環境を想定した多くの先行研究 [4], [13], [14] と同様に、バッチサイズ 1 で主要な評価を行う。再現性と公平性の観点から、greedy decoding で固定出力長まで生成する設定にした。ただし、本手法はモデル精度を維持し、実行される演算自体も変更しないため、入出力長に基づく性能評価のみを行った。入力には、入力長に基づいたダミーデータを与える。評価指標は、time to first token (TTFT)、TPOT、end-to-end latency (E2EL) である。

Baselines. Norn を最先端手法である Pipelined Sharding (PS) [4] と比較する。PS は、llama.cpp をベースに作られており、主要パラメータとして、`ctx-size` と `ubatch-size` がある。`ctx-size` は、サポートする最大コンテキスト長を決め、`ubatch-size` は、prefill と decode を含めて、一度の forward で計算する token 数を決める。バッチサイズ 1 では、chunked prefill のチャンクサイズに相当する。多様なワークロードで使用できるように、`ctx-size` を Qwen3 がサポートする最大コンテキスト長である 128K に設定する。また、入出力長が事前にわかる oracle として、PS-O を評価する。PS-O では、入出力長から、`ctx-size` を、リクエスト毎に設定する。チャンクサイズは、Pipelined Sharding の論文で基本的に使われている 1024 を使用する。

4.2 異なる入力長での推論性能比較

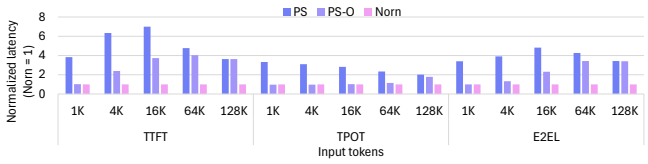
出力長を 32 に固定し、入力長を 1K から 128K まで変化させた結果を図 5 および図 6 に示す。Norn は、すべての条件で PS より短い E2EL を達成し、最大 16.8 倍の高速化を実現した。入出力長を既知とする PS-O に対しても、短い入力では概ね同等の性能を示し、最大 4.2 倍の高速化を達成した。TTFT では、Norn が PS と PS-O の双方をすべての条件で上回った。特に、デスクトップ 2 の Qwen3-32B では、入力長 16K において PS に対し 8.0 倍、PS-O に対し 7.1 倍の高速化を達成した。PS と PS-O は、chunked prefill において各チャンクを全層にわたって処理するため、オフロードした重みをチャンクごとに繰り返し転送する。直後のサブレイヤの重みを GPU に先行転送して計算と転送をオーバーラップさせるものの、転送コストを十分に隠蔽できない場合がある。一方、Norn は OMCP によって同じサブレイヤの全チャンクを連続して処理し、重みの再転送を回避する。さらに、計算時間と転送時間に応じて 1 または 2 サブレイヤ先まで重みを先読みし、計算と転送のオーバーラップを高める。これらにより、重み転送に伴う TTFT のオーバーヘッドを抑制した。

TPOT では、最大コンテキスト長を 128K に固定する PS の性能低下が、特に短い入力で顕著となった。デスクトップ 1 の Qwen3-14B では、入力長 1K における PS の TPOT が Norn の 18.2 倍となった。一方、PS-O は入出力長に応じて KV キャッシュの確保容量を調整するため、短い入力では Norn に近い性能を達成した。しかし、長い入力では PS-O の性能も低下し、デスクトップ 2 の Qwen3-32B では、128K において Norn の 3.1 倍の TPOT となった。PS-O に対する Norn の性能優位性には、配置と先行転送の違いが寄与していると考えられる。第一に、PS-O は attention と KV キャッシュの GPU 常駐を優先する固定的な規則に基づいて配置を決定するため、長いコンテキストでは CPU 計算への割り当てが増加する。入力長 128K では、デスクトップ 1 の Qwen3-14B において CPU で実行する FFN が PS-O の 32 層に対して Norn では 19 層、デスクトップ 2 の Qwen3-32B において CPU で実行する attention が PS-O の 32 層に対して Norn では 20 層であった。Norn は重みと KV キャッシュの配置、および CPU・GPU への計算割り当てを共同で最適化することで、CPU 計算を削減できたと考えられる。第二に、PS-O の先行転送は直後に実行されるサブレイヤに限定されるのに対し、Norn は複数の後続サブレイヤを先行転送できる。これにより、Norn では CPU・GPU 計算とデータ転送をより効率的にオーバーラップさせ、重みと KV キャッシュの streaming 実行を有効に活用できたと考えられる。

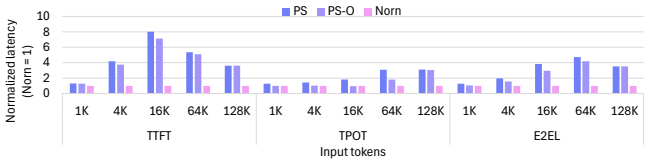
4.3 異なる出力長での推論性能

入力長を 1K と 16K に固定し、出力長を 1K、4K、16K に変化させた場合の平均 TPOT を図 7 および図 8 に示す。Norn はすべての条件で PS を上回り、最大 18.1 倍の高速化を達成した。PS は最大コンテキスト長 128K に合わせた静的配置を使用するため、特に短いコンテキスト長で性能が低下する。一方、PS-O に対しては 24 条件中 16 条件で Norn が上回り、特に出力長 16K

ⓒ2026 Information Processing Society of Japan

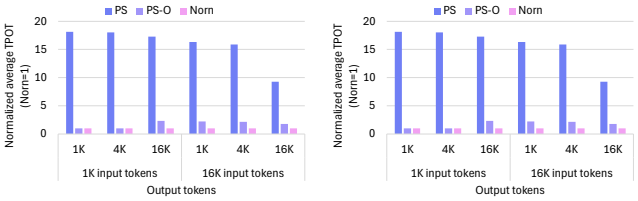


(a) Qwen3-14B



(b) Qwen3-32B

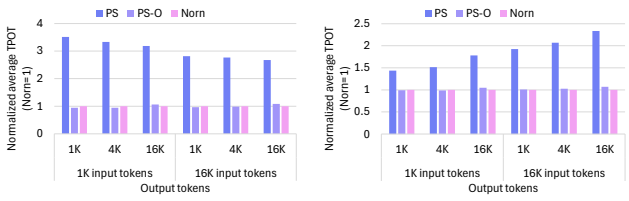
図 6: デスクトップ 2 における入力長ごとの Norn を基準に正規化した TTFT、TPOT、E2EL。出力長を 32 トークンに固定している。



(a) Qwen3-14B

(b) Qwen3-32B

図 7: デスクトップ 1 での正規化された平均 TPOT



(a) Qwen3-14B

(b) Qwen3-32B

図 8: デスクトップ 2 での正規化された平均 TPOT

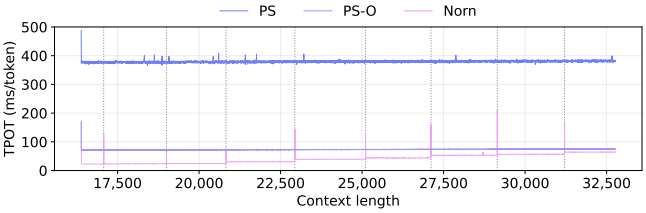


図 9: デスクトップ 1、Qwen3-14B、入力 16K、出力長 16K のコンテキストごとの TPOT

では全 8 条件で優位となった。Desktop1 の Qwen3-14B では最大 2.33 倍の高速化を達成した。ただし、短い出力では PS-O が優位な条件もあり、Norn の平均 TPOT は最大 5.8% 長かった。PS-O は入出力長を既知として最大コンテキスト長を設定できるのに対し、Norn は出力長を仮定せず、幅広いコンテキスト長を対象にプランを生成するため、個々のリクエストで常に最速となるとは限らない。図 9 に、Desktop1 の Qwen3-14B における入力長 16K、出力長 16K での TPOT の推移を示す。PS と PS-O がほぼ一定の TPOT で推移する一方、Norn はコンテキスト長に応じたプラン切り替えによって TPOT が段階的に変化する。プラン切り替え時には一時的な TPOT のピークが発生するものの、生成初期に高速なプランを使用することで、切り替えのオーバーヘッドを含めても平均 TPOT は PS に対して 9.3 倍、PS-O に対して 1.8 倍高速となった。これらの結果は、長い出力において、コンテキスト長に応じた動的配置がデコード性能の

表 1: 実行プラン数とプランニング時間

構成	最終プラン数	総時間 (s)
Desktop1 / Qwen3-14B	27	1211
Desktop1 / Qwen3-32B	2	245
Desktop2 / Qwen3-14B	21	2081
Desktop2 / Qwen3-32B	12	1643

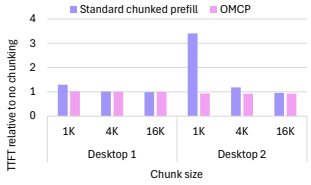


図 10: 入力長 64K での Qwen3-14B の正規化した TTFT

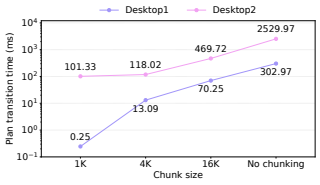


図 11: 入力長 64K での Qwen3-14B のプラン遷移時間

改善に有効であることを示している。

4.4 オフラインプランニングオーバーヘッド

表 1 に、最大 128K トークンを対象とした Dynamic Planner の実行プラン数と総生成時間を示す。プラン数は 2–27 個、総生成時間は 245–2081 秒であった。総生成時間は、最終プラン数、区間プランの生成回数と 1 回あたりの生成時間に依存する。Desktop1 では、Qwen3-14B と Qwen3-32B の区間プラン生成回数はそれぞれ 139 回と 8 回であり、総時間に占める割合は 64.1% と 27.7% であった。また、Qwen3-14B では Desktop2 の生成回数が 83 回と少ないものの、平均生成時間は Desktop1 の 5.6 秒に対して 13.1 秒と長く、総生成時間も増加した。MILP の求解には HiGHS [15] を使用した。Desktop1 では、Qwen3-14B と Qwen3-32B でそれぞれ 3248 回、1152 回の MILP を求解し、すべて最適解に到達した。平均求解時間はそれぞれ 84.5 ms、33.5 ms であり、総生成時間の 22.7%、15.7% を占めた。

4.5 Operator-major chunked prefill の有効性

デスクトップ 1 および 2 で Qwen3-14B を用い、入力長を 64K に固定し、チャンク長を 1K、4K、16K に変化させて TTFT とプラン遷移時間を測定した。図 10 に、チャンク化しない場合で正規化した TTFT を示す。チャンク長 1K では、デスクトップ 2 において通常の chunked prefill が 3.41 倍に悪化したのに対し、OMCP は 0.93 倍となり、通常方式に対して 3.66 倍高速であった。デスクトップ 1 でも、通常方式の 1.29 倍に対して OMCP は 1.02 倍にとどまった。これは、OMCP がチャンク間で重みを再利用し、再転送を回避するためであり、PCIe 帯域の限られるデスクトップ 2 で特に効果が顕著となった。一方、チャンク

長が大きくなると、通常方式でも計算時間の増加によって重み転送が隠蔽されるため、両方式の性能差は縮小した。図 11 に、Qwen3-14B におけるプリフィルから初期デコードプランへの遷移時間を示す。チャンク長 1K の OMCP では、チャンク化しない場合と比較して、デスクトップ 1 で 303 ms から 0.25 ms、デスクトップ 2 で 2530 ms から 101 ms に短縮された。また、チャンク長が小さいほど遷移時間が短縮された。これは、activation のピークメモリ使用量が減少し、初期デコードプランに必要な重みや KV キャッシュを GPU 上に保持しやすくなるためである。以上より、OMCP は小さいチャンク長でも TTFT の増加を抑えつつ、初期デコードへの遷移時間を短縮できる。

5. 関連研究

リソース制約のある環境において、CPU と GPU を協調利用することで LLM 推論を高速化する手法が提案されている。HeteGen [16] と TwinPilots [5] は、CPU 計算、GPU 計算、データ転送を並列化することでレイテンシを削減する。FlexInfer [17] と Pipelined Sharding [4] は、系列長やハードウェア条件に応じて、計算の割り当てや配置を決定する。Layer-wise Scheduling [6] は、各層を CPU、GPU resident、GPU streaming に配置し、CPU 計算の背後に重み転送を隠蔽する。これらの手法は、デコード中に増大する KV キャッシュに応じて重みと KV キャッシュの配置を切り替えることを主眼としていない。

PowerInfer [18] と LLM in a Flash [19] は、activation sparsity を利用し、活性化するニューロンを予測することで計算量やデータ転送量を削減する。しかし、これらの手法は活性化ニューロンを特定するための予測器を必要とし、十分な activation sparsity を持たないモデルでは効果が限定される。FlexGen [20] と TightLLM [21] は、オフラインのバッチ推論を対象とし、重み転送を複数のリクエスト間で償却することでスループットを最大化するが、レイテンシを犠牲にしている。NEO [11] と APEX [22] は、オンラインサービングを対象に、KV キャッシュと attention 計算の一部を CPU へオフロードし、CPU と GPU を並列に実行することでスループットを向上させるが、複数リクエストの処理を前提としている。HeadInfer [23] と InfiniGen [24] は、KV キャッシュのメモリ使用量と転送コストを削減する手法を提案しているが、重みと KV キャッシュの GPU メモリ配分を共同で最適化しない。

6. おわりに

本研究では、コンテキスト長に応じて実行プランを動的に切り替える LLM オフロード推論手法 Norn を提案した。Norn は、CPU・GPU への計算割り当てと重み・KV キャッシュの配置を最適化した複数の実行プランとその切り替え点を事前に決定し、推論中のコンテキスト長の増加に適応する。さらに、operator-major chunked prefill により、メモリ使用量を抑えつつ重みの再転送を回避する。2 種類のデスクトップ環境と Qwen3-14B/32B を用いた評価では、Pipelined Sharding に対して最大 16.8 倍、入出力長を既知とする oracle 設定に対しても最大 4.2 倍の推論高速化を達成した。

謝辞 本研究の一部は JST BOOST JPMJBS2409 の支援による。

文　献

[1] Bloomberg, “Generative ai 2025: Assessing opportunities and disruptions in an evolving trillion-dollar-plus market,” https://assets.bbhub.io/professional/sites/41/Generative-AI-Outlook.pdf, 2024. Accessed: 2026-10-23.

[2] E. Li, A.B.L. Larsen, C. Zhang, X. Zhou, J. Qin, D.A. Yap, N. Raghavan, X. Chang, M. Bowler, E. Yildiz, et al., “Apple intelligence foundation language models: Tech report 2025,” arXiv preprint arXiv:2507.13575, 2025.

[3] V. Pradeep, “Phi Silica, small but mighty on-device SLM,” https://blogs.windows.com/windowsexperience/2024/12/06/phi-silica-small-but-mighty-on-device-slm/, 2024. Accessed: 2026-10-23.

[4] A. Ukarande, D. Shekhar, M. Blackstein, and R. Rangan, “Efficient, vram-constrained xlm inference on clients,” Proceedings of Machine Learning and Systems, vol.8, pp.1884–1900, 2026.

[5] C. Yu, T. Wang, Z. Shao, L. Zhu, X. Zhou, and S. Jiang, “Twinpilots: A new computing paradigm for gpu-cpu parallel llm inference,” Proceedings of the 17th ACM international systems and storage conference, pp.91–103, 2024.

[6] N. Sugiura, Y. Tokusashi, and H. Matsutani, “Layer-wise cpu-gpu scheduling for llm inference on memory-limited consumer gpus,” European Conference on Parallel ProcessingSpringer, pp.301–315 2026.

[7] G. Gerganov, et al., “llama.cpp,” https://github.com/ggml-org/llama.cpp. Accessed: 2026-10-14.

[8] H. Face, “Hugging face accelerate,” https://github.com/huggingface/accelerate, 2025. Accessed: 2026-10-26.

[9] G. Lee, J. Kim, J. Park, Y. Lee, and J.H. Ahn, “From tokens to layers: Redefining stall-free scheduling for moe serving with layered prefill,” Proceedings of Machine Learning and Systems, vol.8, pp.398–421, 2026.

[10] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, et al., “Huggingface’s transformers: State-of-the-art natural language processing,” arXiv preprint arXiv:1910.03771, 2019.

[11] X. Jiang, Y. Zhou, S. Cao, I. Stoica, and M. Yu, “Neo: Saving gpu memory crisis with cpu offloading for online llm inference,” Proceedings of Machine Learning and Systems, vol.7, 2025.

[12] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv, et al., “Qwen3 technical report,” arXiv preprint arXiv:2505.09388, 2025.

[13] H.M. Chen, Z. Mo, G. Lu, S. Liang, L. Ma, W. Luk, and H. Fan, “Fasttts: Accelerating test-time scaling for edge llm reasoning,” Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, pp.732–748, 2026.

[14] H. Chen, W. Xie, B. Zhang, J. Tang, J. Wang, J. Dong, S. Chen, Z. Yuan, C. Lin, C. Qiu, et al., “Ktransformers: Unleashing the full potential of cpu/gpu hybrid inference for moe models,” Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, pp.1014–1029, 2025.

[15] Q. Huangfu and J.J. Hall, “Parallelizing the dual revised simplex method,” Mathematical programming computation, vol.10, no.1, pp.119–142, 2018.

[16] X. Zhao, B. Jia, H. Zhou, Z. Liu, S. Cheng, and Y. You, “Hetegen: Efficient heterogeneous parallel inference for large language models on resource-constrained devices,” Proceedings of Machine Learning and Systems, vol.6, pp.162–172, 2024.

[17] S. Na, G. Jeong, B.H. Ahn, A. Jezghani, J. Young, C.J. Hughes, T. Krishna, and H. Kim, “Flexinfer: Flexible llm inference with cpu computations,” Proceedings of Machine Learning and Systems, vol.7, 2025.

[18] Y. Song, Z. Mi, H. Xie, and H. Chen, “Powerinfer: Fast large language model serving with a consumer-grade gpu,” Proceedings of the

ⓒ2026 Information Processing Society of Japan

ACM SIGOPS 30th Symposium on Operating Systems Principles, pp.590–606, 2024.

[19] K. Alizadeh, S.I. Mirzadeh, D. Belenko, S. Khatamifard, M. Cho, C.C. Del Mundo, M. Rastegari, and M. Farajtabar, "Llm in a flash: Efficient large language model inference with limited memory," Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp.12562–12584, 2024.

[20] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang, "Flexgen: High-throughput generative inference of large language models with a single gpu," International Conference on Machine LearningPMLR, pp.31094–31116 2023.

[21] Y. Hu, X. Liu, G. Yang, L. Li, K. Zeng, Z. Zhao, S. Chen, L. Zhao, W. Li, and K. Li, "Tightllm: Maximizing throughput for llm inference via adaptive offloading policy," IEEE Transactions on Computers, vol.74, no.7, pp.2195–2209, 2025.

[22] J. Fan, Y. Zhang, X. Li, and D.S. Nikolopoulos, "Apex: Asynchronous parallel cpu-gpu execution for online llm inference on constrained gpus," 2026 IEEE International Parallel and Distributed Processing Symposium (IPDPS)IEEE, pp.1190–1203 2026.

[23] C. Luo, Z. Cai, H. Sun, J. Xiao, B. Yuan, W. Xiao, J. Hu, J. Zhao, B. Chen, and A. Anandkumar, "Headinfer: Memory-efficient llm inference by head-wise offloading," arXiv preprint arXiv:2502.12574, 2025.

[24] W. Lee, J. Lee, J. Seo, and J. Sim, "{InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management," 18th USENIX symposium on operating systems design and implementation (OSDI 24), pp.155–172, 2024.

ⓒ2026 Information Processing Society of Japan