Qwen 3.5アーキテクチャにおけるllama.cpp環境下の投機的デコーディング最適化とハードウェアバックエンド別効率性に関する包括的調査

大規模言語モデルにおける推論のボトルネックと投機的デコーディングの基礎理論

大規模言語モデル（LLM）のローカル推論環境において、C/C++ベースで構築されたllama.cppは、その依存関係の少なさと多様なハードウェアバックエンドへの適応性から、業界標準の推論エンジンとして広く普及している 。しかし、自己回帰型言語モデルの推論プロセスにおけるトークン生成フェーズは、本質的に計算能力（FLOPs）ではなくメモリ帯域幅（Memory Bandwidth）によって律速されるという物理的制約を抱えている 。1つのトークンを生成するたびに、数ギガバイトから数十ギガバイトに及ぶモデル全体のパラメータをメモリから演算器へと転送する必要があり、最新のGPUを用いても計算ユニットの稼働率は著しく低い状態に留まる 。
この「メモリの壁（Memory Wall）」を突破するための中核技術が、投機的デコーディング（Speculative Decoding）である 。投機的デコーディングは、推論プロセスを「軽量な予測（Drafting）」と「並列検証（Verification）」の二段階に分離することで、メモリ帯域幅の浪費を防ぐアプローチを取る 。具体的には、ターゲットとなる大規模モデルよりも遥かに計算コストの低いドラフト機構（小規模モデルや統計的N-gramモデルなど）を用いて、未来の γ 個のトークンを連続して高速に予測する 。その後、ターゲットモデルは予測された γ 個のトークン系列と過去のコンテキストを1つのバッチとして受け取り、1回のフォワードパス（単一の重みロード）で全てのトークンの確率分布を並列に計算し、検証を行う 。
検証フェーズにおいて、ターゲットモデルの確率分布とドラフトモデルの予測が一致した最長のプレフィックスまでが受容（Accept）され、不一致が生じた時点で残りのトークンは破棄（Rollback）され、ターゲットモデル自身が算出した正しいトークンに置き換えられる 。プロンプト処理（Prefill）におけるバッチ計算は、シーケンシャルな生成よりもハードウェアの計算資源を遥かに効率的に活用できるため、ドラフトの受容率（Acceptance Rate）が一定水準を超えれば、出力品質を一切犠牲にすることなく、実効トークン生成率（Effective Token Rate: ETR）を1.5倍から3倍にまで引き上げることが理論上可能となる 。

Qwen 3.5アーキテクチャの特異性とllama.cppにおける互換性の変遷

Qwen 3.5ファミリ（およびその後継であるQwen 3.6）は、単なるDense（密）なTransformerモデルではなく、推論効率を高めるためにGated DeltaNetと呼ばれるハイブリッドアテンション機構や、スパースなMixture of Experts（MoE）構造を統合した極めて複雑なアーキテクチャを採用している 。この構造的進化は、モデル単体の推論効率を飛躍的に向上させた一方で、llama.cppにおける投機的デコーディングの実装において深刻な互換性問題を引き起こした 。
最も顕著な問題は、ドラフトトークンが検証フェーズで拒否（Reject）された際に発生するコンテキスト

の部分的削除（Partial Sequence Removal）に関する制約であった 。投機的デコーディングのサイクルを維持するためには、拒否されたトークンに紐づくKVキャッシュや再帰的（Recurrent）なメモリ状態をターゲットモデルの空間から正確にロールバック（巻き戻し）する必要がある 。しかし、Qwen 3.5が採用するGated DeltaNetやRNN的な状態空間モデル（SSM）の初期実装においては、メモリ状態の部分的な破棄がサポートされておらず、推論エンジンは common_speculative_is_compat: the target context does not support partial sequence removal という致命的なエラーを吐き出し、投機的デコーディングの実行を完全にブロックしていた 。
この非互換性問題は、開発コミュニティにおける集中的なエンジニアリングによって段階的に解消された。特に、PR #19493（Speculative checkpointing）やPR #22400（GDNモデル向けの部分的seq_rmサポート）のメインブランチへの統合により、ハイブリッドモデルやMoEモデルにおいても安全にKVキャッシュの巻き戻しが可能となり、Qwen 3.5ファミリに対する投機的デコーディングへの道が開かれた 。

Qwen環境下における投機的生成手法の効率性評価

llama.cppは、単純なドラフトモデルから高度な自己投機メカニズムまで、多様な投機的デコーディングの手法を提供している 。Qwen 3.5/3.6アーキテクチャに対して「最も効率の良い生成方法」を特定するためには、各手法の理論的背景と実際のハードウェア上でのベンチマークを定量的に比較分析する必要がある。

古典的ドラフトモデルの破綻と**MoE**特有の**Net Negative**現象

最も一般的な投機的デコーディングの手法は、ターゲットモデルと同一のボキャブラリを持つ小規模な独立モデル（例: Qwen3.5-0.8B）をドラフトモデルとして並行稼働させるアプローチ（--model-draft）である 。Dense（密）モデル間ではこの手法が一定の成果を上げるが、Qwen 3.6 35B-A3B（Active 3B）のようなMoE（Mixture of Experts）アーキテクチャに対して適用した場合、「推論速度が向上するどころか劇的に低下する」というNet Negative（純損失）現象が確認されている 。
NVIDIA RTX 3090（24GB VRAM）単体を用いた広範なベンチマーク調査によると、Qwen 3.6 35B-A3Bに対してQwen 3.5 0.8Bをドラフトモデルとして設定した場合、投機を行わないベースラインの推論速度が135.7～139.9 tok/sであったのに対し、ドラフトモデルを有効化した構成では55～65 tok/sへと約50%～60%もの劇的な速度低下が引き起こされた 。驚くべきことに、この速度低下はドラフトの受容率が100%（完全に予測が的中している状態）であっても発生している 。
この崩壊の根本原因は、MoE特有のルーティングメカニズムとメモリアクセスパターンの不一致にある。MoEモデルは、各トークンに対して全Expertパラメータのうちごく一部（例: 3B相当）のみをアクティブにすることでメモリ帯域幅を節約する 。しかし、ドラフトモデルによって予測された K 個のトークンをバッチとして検証する際、バッチ内の各トークンはそれぞれ異なるExpertにルーティングされる可能性が高い。結果として、1回のフォワードパスでロードしなければならないExpertの総数が急増し、コンシューマ向けGPUのVRAMメモリ帯域幅を完全に飽和させてしまうのである 。つまり、並列検証による計算効率の向上分を、散発的なExpertウェイトのフェッチにかかるメモリアクセスレイテンシが完全に相殺・凌駕してしまうという構造的欠陥が露呈した形となる 。

自己投機（**N-gram**機構）によるオーバーヘッドの回避

古典的ドラフトモデルにおける追加のVRAM消費と同期オーバーヘッドを回避するため、llama.[span_13](start_span)[span_13](end_span)cppにはモデルの外部に依存しない「自己投機（Self-Speculative Decoding）」の手法が実装されている 。これらは、過去の生成履歴からパターンを検索し、一致するシーケンスをドラフトとして提案するアプローチである。

主な自己投機パラメータと構成は以下の通りである。

自己投機手法 (--spec-type)	動作メカニズムと特徴	パラメータの例
ngram-mod	共有ハッシュプールを用いたベーシックなローリングN-gramハッシュ。VRAMをほとんど消費せず、計算オーバーヘッドが極めて低い 。	--spec-ngram-mod-n-match 24, --spec-draft-n-min 48, --spec-draft-n[span_20](start_span)[span_20](end_span)-max 64
ngram-simple	トークン履歴から現在のN-gramに一致する過去のパターンを検索し、それに続くM個のトークンをドラフトとして抽出する 。	--spec-ngram-simple-size-n 12, --spec-ngram-simple-size-m 48
n[span_21](start_span)[span_21](end_span)gram-map-k	N-gramキーを用いたパターンマッチング 。	--spec-ngram-map-k-size-n 12, --spec-ngram-map-k-size-m 48

Qwenアーキテクチャに対して --spec-type ngram-mod を適用した検証では、古典的ドラフトモデルに見られたような壊滅的な速度低下は防げるものの、一般的なチャットプロンプト（高エントロピー環境）においてはヒット率が低く、ベースラインから約4%の速度低下に留まるケースが多い 。しかし、JSONフォーマットの生成やソースコードの修正など、反復的で予測可能性の高い（低エントロピーな）タスクにおいては状況が一変する。特定のコード編集タスクにおいて --spec-typ[span_14](start_span)[span_14](end_span)e ngram-mod を適切に設定（例: --spec-ngram-mod-n-match 24 --draft-min 48 --draft-max 64）することで、ベースラインを上回る実効速度の向上が報告されている 。

ここで重要な洞察は、「ドラフト数を少なく設定するよりも、非常にアグレッシブに大きなドラフトウィンドウ（例: 48〜64）を設定する方が、検証にかかるKVキャッシュ管理のオーバーヘッドを償却しやすく、結果的にパフォーマンスが安定する」という事実である 。自己投機は、MoEにおけるNet Negativeを回避する安全なフォールバックとしての地位を確立している。

Multi-Token Prediction (MTP) の圧倒的優位性

古典的ドラフトモデルのメモリ飽和問題と、N-gramの低いヒット率という双方向の課題を根本的に解決する技術として、2026年5月にllama.cppのメインブランチにマージされた「Multi-Token Prediction (MTP)」が現在の最適解である 。

MTPは、モデルアーキテクチャ自体に複数の未来予測ヘッドを埋め込み、単一のモデル内で効率的に未来のトークンを自己予測するパラダイムである 。MTP対応のGGUFモデル（例: Qwen3.6-27B-MTP-GGUF）を使用し、llama.cppにおいて --spec-type draft-mtp フラグ（旧 --spec-type mtp）を指定することで有効化される 。

MTP最大の利点は、ターゲットモデルとドラフトモデルが完全に融合しているため、MoEのルーティング状態やKVキャッシュをシームレスに共有できる点にある 。これにより、独立したドラフトモデルを用いた際に発生した「Expertの不均一な飽和」や「デバイス間でのコンテキストスイッチングによるオーバーヘッド」が完全に排除される。

ベンチマーク結果はMTPの優位性を明確に示している。Qwen 3.6 35B-A3B MTPモデルをRTX 3090（12GB VRAM環境を模倣）で動作させた場合、ベースラインと比較してコンテキスト長5Kで約1.8倍、80Kで3.3倍、128Kの超長文コンテキストにおいては最大4.5倍もの実効速度向上（80 tok/sec以上）が記録されている 。また、Strix Haloプラットフォームにおける検証でも、Qwen 3.6 27B（Dense）モデルにおいてトークン生成速度が 7.63 tok/s から 16.15 tok/s へと+111%の加速を達成している 。したがって、Qwen環境下において最も効率の良い生成方法は、議論の余地なく「MTPモデルを用いた投機的デコーディング」であると結論付けられる。

ハードウェアバックエンド別の特性と最適化戦略

投機的デコーディング（特にMTPや自己投機）の効率は、それを実行するハードウェアバックエンド（CUDA、ROCm、Vulkan、CPU）のアーキテクチャ特性に強く依存する。各環境におけるボトルネックの所在と、llama.cppのパラメータチューニングによる具体的な最適化手法を以下に詳述する。

CUDAバックエンド（NVIDIA GPU）の最適化

CUDAバックエンドは、llama.cppのGPUオフロードにおいて最も成熟し、広範な最適化が施された実行環境である 。NVIDIAのAmpere世代（RTX 3090等）やAda Lovelace世代（RTX 4090等）における強大なCUDAコアの演算能力は、MTPによるバッチ並列検証フェーズで最大限に活用される。
しかし、Qwen 3.5/3.6クラスの大規模モデル（27B～35B）を単一のコンシューマ向けGPUに収める場合、VRAM容量の制約が最大の障壁となる。MTP予測層や長大なコンテキストのKVキャッシュがVRAMから溢れ、システムRAM（CPU）へのフォールバック（ページフォールト）が発生した瞬間、PCIeバスのレイテンシによって投機的デコーディングの恩恵は完全に消滅する 。したがって、CUDA環境における至上命題は「徹底したVRAMメモリ帯域の保護と容量管理」となる。
最適化のための具体的な構成は以下の通りである。

パラメータ / 手法	CUDA環境における目的と効果
KVキャッシュの量子化	大規模コンテキストとMTPの併用には必須。-ctk q8_0 -ctv q8_0 を指定してVRAM消費を半減させるか、さらに高度なTurboQuant WHT-rotated フォーマット（-ctk turbo3 -ctv turbo3）を使用し、同等品質で3～6倍のコンテキスト長を確保する 。
Flash Attentionの有効化	-fa on は不可欠である。これにより、アテンション行列の計算がオンチップのSRAM内で完結し、HBM（VRAM）へのグローバルメモリアクセスが劇的に削減されるため、検証フェーズのバッチ処理が高速化される 。
MTPドラフト数の最適化	--spec-draft-n-max の値は 2 または 3 が最適解である 。4以上に設定すると、予測が外れる確率が高まり、受容率が83%から50%へと急落するため、無駄なフォワードパスに計算資源を浪費することになる 。
バッチサイズとスロット制御	検証フェーズのスループットを最大化するため、論理バッチサイズ（-b）を2048等の大きな値に取り、物理マイクロバッチサイズ（-ub）を512～1024の範囲でGPUのVRAMに合わせて調整する 。また、コンテキストスイッチを防ぐためパラレルスロットは1（-np 1）に固定する 。

さらに、VRAMを極限までチューニングする場合、-fitt <MB> オプションを用いてGPU内に残す空きメモリを明示的に指定（例: -fitt 1536）することで、MTPドラフトモデルのワークスペースや動的なKVキャッシュ展開に必要な領域を正確に確保し、OOM（Out of Memory）クラッシュを予防することが可能である 。

ROCm / HIPバックエンド（AMD GPU）の課題と最適化

ROCm（Radeon Open Compute）バックエンドは、Radeon RX 7900 XTX（RDNA3）等のAMD製

GPUにおいて、CUDAに匹敵する計算スループットを提供する可能性を秘めている 。しかし、ソフトウェアスタックの成熟度やHIPを通じたカーネル移植の制約により、投機的デコーディングを適用する際には特有の性能劣化メカニズムに直面する。
最大の問題は、「大規模バッチサイズにおけるFlash Attentionカーネルの著しいパフォーマンス低下」である 。llama.cppのHIPポートにおいて、バッチサイズが大きくなるとCUDA由来のFlashAttentionカーネルが極めて非効率に動作し、結果としてパフォーマンスの劣る小規模バッチ用カーネルにフォールバックする挙動が仕様として知られている 。具体的には、バッチサイズ16でのプロンプト処理速度が 678 tok/s から 375 tok/s へと半減する事象が報告されている 。また、ROCmのバージョンアップデートに伴い、Prefill（プロンプト処理）全体のパフォーマンスが約5%〜9%低下する現象も観測されている 。投機的デコーディングの検証フェーズは本質的に「バッチ処理」であるため、このROCm特有のFlash Attentionの弱点は、MTPやドラフトモデルの効率を直接的に削ぐ結果となる。
ROCm環境における最適化戦略は、このバッチ処理のボトルネックをいかに回避するかに集約される。

1. コンパイルレベルでの最適化: llama.cppをソースからビルドする際、AMDのアーキテクチャに特化したフラグを確実に付与する必要がある。具体的には -DGGML_HIP=ON, -DAMDGPU_TARGETS=gfx1100 (7900 XTXの場合), -DGGML_HIP_ROCWMMA_FATTN=ON, -DGGML_HIPBLAS=ON などを組み合わせ、ハードウェアの行列演算ユニット（WMMA）を最大限に活用するバイナリを生成する 。
2. マイクロバッチサイズの縮小: Flash Attentionのボトルネックを緩和するため、マイクロバッチサイズ（-ub）をCUDA環境よりも小さく設定する（例: -ub 128 や -ub 256）ことで、HIPカーネルの効率低下領域を回避するアプローチが有効である 。
3. 専用フォークの採用: メインブランチのROCmサポートが抱える制約を回避するため、Radeon RX 6000/7000シリーズ向けに独自に最適化されたフォーク版（例: llama.cpp-1-bit-turbo）を利用することが推奨される。これらのフォーク版には、RotorQuant KV圧縮カーネルや、EAGLE3 Speculative Decoding、Ghost-draft投機などの高度なアルゴリズムがHIPネイティブで実装されており、AMD GPUにおける推論スループットを大幅に引き上げる可能性がある 。

Vulkanバックエンドのメモリ構造依存性とシリアライズ問題

Vulkanバックエンドは、OSやGPUベンダーに依存しないクロスプラットフォームなアクセラレーションを提供する 。理論上は広範なハードウェアで動作するが、投機的デコーディングを実行する際のパフォーマンスは、対象のデバイスがUMA（Unified Memory Architecture: 統合メモリ構造）であるか、ディスクリートGPU（独立したVRAMを持つグラフィックスカード）であるかによって、天と地ほどの差が生じる 。
特に、Mesa RADVドライバを使用するAMD Radeon 780M（RDNA3 iGPU）などのUMAデバイス上で、ターゲットモデルと古典的ドラフトモデルの2つを同時にロードして投機的デコーディングを実行した場合、ドラフトモデルの単一トークン生成にかかるレイテンシが約**100,000**倍に膨張し、生成速度が実質的にゼロになるという致命的なバグ（Issue #23126）が特定されている 。
この壊滅的なスローダウンの根本原因は、「Vulkanコンピュートキューの直列化（Serialization）」と「キャッシュのフラッシング」にある 。 現在のVulkan実装と一部のオープンソースドライバの組み合わせでは、GPUに対して複数のコンピュートキューを並列に発行・処理することができず、ジョブが完全に直列化される 。つまり、ターゲットモデルの巨大な検証計算グラフが実行されている間、ドラフトモデルの軽量な予測グラフはGPU上で実行を開始できず完全にブロックされる。さらに、2つの異なるモデル間でコンピュートコンテキストが切り替わるたびに、GPUのL2キャッシュが強制的にフラッシュされ、パイプラインの計算状態が完全に無効化される 。これにUMA特有のステージングバッファのオーバーヘッドが重なることで、投機的デコーディングが本来意図している「並列パイプライン化」の

利点が完全に破壊されてしまうのである 。
この問題に対するVulkan環境での最適化・回避策は以下の通りである。

1. ディスクリート**GPU**の活用: 独立したVRAMとより高度なハードウェアスケジューラを持つディスクリートVulkan環境（例: AMD Radeon AI PRO R9700）では、この種の極端なコンテキストスイッチペナルティは発生しにくい。実際のベンチマークでは、Qwen3-30B-A3Bモデルに対してVulkanバックエンドとFlash Attentionを組み合わせることで、183 tok/s という理論帯域幅の86%に達する極めて高いデコード速度が記録されており、Vulkanバックエンド自体に根本的な欠陥があるわけではないことが証明されている 。
2. 古典的ドラフトモデルの完全排除: UMA環境におけるキュー直列化の根本原因は「2つの異なるモデル（ターゲットとドラフト）を切り替える」ことにある。したがって、UMAデバイスでは古典的ドラフトモデル（--model-draft）の使用を完全に禁止すべきである。
3. **MTP**または自己投機への一元化: ターゲットモデルの内部で予測処理を完結させ、コンパイルされる計算グラフを1つに統合できる **MTP**（**--spec-type draft-mtp**） や、軽量な 自己投機（**--spec-type ngram-mod**） を採用することが、Vulkan環境における唯一の現実的な解となる。これにより、コンテキストスイッチに伴うL2キャッシュのフラッシュが防止され、安定した投機的推論が可能となる。
4. **CPU**フォールバックの禁止: ターゲットモデルをVulkanで、ドラフトモデルをCPUで処理する（-ngld 0）というワークアラウンドは、PCIe/メモリバスを介したデバイス間の同期バリアオーバーヘッドを引き起こし、推論速度が 1.8 tok/s にまで低下するため、適用してはならない 。

CPUバックエンドにおける**NUMA/**スレッド競合の制御

llama.cppは、もともとMacBookや汎用PCのCPU上での実行を主眼に開発された背景を持ち、AVX/AVX2/AVX512/AMX（x86）やNEON（ARM）といったベクトル命令セットに高度に最適化されている 。BLAS（OpenBLAS, BLIS）やZenDNNなどを介して計算を加速するCPUバックエンドは、GPUのように広帯域のVRAMを持たないため、メインメモリの帯域幅（DDR4/DDR5）が極端なボトルネックとなる 。このため、複数トークンを一度のメモリアクセスで処理する投機的デコーディングは、CPU推論においてこそ最も劇的な効果（理論上スループットを数倍に引き上げる効果）を発揮するはずの技術である 。
しかし、CPU環境特有の問題として「スレッドの競合（Oversubscription）」と「NUMA（Non-Uniform Memory Access）ノードを跨ぐレイテンシ」が存在し、これらを適切に制御しなければ投機的デコーディングの恩恵は得られない。llama.cppはデフォルトでシステム上の全ての論理コアを検出しようとするが、生成フェーズにおいてすべてのスレッドを割り当てると、OSレベルでのコンテキストスイッチングやキャッシュのスラッシングが発生し、パフォーマンスが著しく低下する。例えば、32スレッドを割り当てるよりも8スレッド（物理コアのみ）に制限する方が高いパフォーマンスを発揮するケースが多々報告されている 。
投機的デコーディングを有効にした場合、「ターゲットの検証（バッチ行列積）」「ターゲットの自己回帰生成」「ドラフトの検証」「ドラフトの生成」という特性の異なる4つの計算プロファイルがCPU上で同時に発生するため、スレッド管理の複雑さは頂点に達する。
これを最適化するため、llama.cppは投機的デコーディング専用のきめ細かなスレッド制御パラメータを提供しており、CPU環境ではこれらを厳密に設定する必要がある。

実行フェーズ	制御フラグ	CPU環境における最適化の指針と理論
ターゲットモデル生成	-t, --threads N	自己回帰トークン生成用。メモリアクセスが支配的であるため、物理コア数（Hyper-Threadingを無効化した数）やメモリチャネル数

実行フェーズ	制御フラグ	CPU環境における最適化の指針と理論
		に合わせて少なく設定し（例: 4〜8）、キャッシュ効率を最大化する。
ターゲットモデル検証	-tb, --threads-batch N	プロンプト処理およびMTP/ドラフトの検証（並列バッチ計算）用。行列積（GEMM）が支配的となり計算律速に近づくため、生成時よりも多くのスレッド（全ての物理コア等）を割り当て、並列計算能力を最大限に活用する 。
ドラフトモデル生成	-td, --threads-draft N	ドラフトモデルによる自己回帰生成用。非常に軽量な処理であるため、メインの推論リソースを阻害しないよう、少数のスレッド（例: 2〜4）に限定して割り当てる。
ドラフトモデル検証	-tbd, --threads-batch-draft N	ドラフトモデルのバッチ処理用。通常は -td と同等かやや多めに設定する 。
CPUアフィニティ制御	-C, --cpu-mask M -Cd, --cpu-mask-draft M	スレッドを特定のCPUコアにピン留め（Affinity Pinning）するための16進数マスク 。AMDの複数CCD（チップレット）を持つCPUや、Intelのbig.LITTLEアーキテクチャにおいては、コア間の通信レイテンシやL3キャッシュの分断を防ぐため、特定のノードに処理をバインドすることが極めて重要である 。

CPU環境においても、VRAM（システムメモリ）の占有量とメモリフェッチのオーバーヘッドを削減するため、古典的なドラフトモデルをロードすることは推奨されない。QwenのMTPモデルを利用し、-t と -tb のスレッド数を上記の理論に基づいて最適化することが、CPU推論における投機的デコーディングの最善のアプローチとなる。

結論と最適化ガイドラインの統合

本稿の調査結果に基づき、Qwen 3.5/3.6アーキテクチャにおいて、llama.cpp環境下でドラフトモデルを用いた投機的予測を行う際の「最も効率の良い生成方法」を以下に結論付ける。

1. **MTP**（**Multi-Token Prediction**）の絶対的採用: Qwenが採用するMoEアーキテクチャの複雑なルーティングと、それに伴うメモリ帯域幅の飽和問題を回避するため、独立した古典的ドラフトモデル（--model-draft）の使用は排除すべきである。代わって、モデル内に組み込まれた予測ヘッドを利用する **MTP (--spec-type draft-mtp)** を採用することが、あらゆるハードウェアバックエンドにおいて最も効率的である。これにより、KVキャッシュやMoEルーティング状態がシームレスに共有され、ベースライン比で1.5倍から最大4.5倍（長文コンテキスト時）の実効スループット向上が達成される。

2. ハードウェア別のパラメータ最適化:
    - **CUDA**: VRAM帯域の保護のためKVキャッシュの量子化（-ctk q8_0 または turbo3）とFlash Attention（-fa on）を必須とし、MTPのドラフト数（--spec-draft-n-max）は受容率のバランスから「3」に最適化する。
    - **ROCm**: Flash Attentionの大規模バッチペナルティを回避するため、マイクロバッチサイズ（-ub）を絞り込み、HIPアーキテクチャに特化したコンパイルフラグを適用する。
    - **Vulkan**: UMAデバイスにおける致命的なキュー直列化バグを回避するため、ターゲットとドラフトを切り替える構成を厳禁とし、単一計算グラフで完結するMTPまたは自己投機（ngram-mod）に完全に一元化する。
    - **CPU**: メモリ帯域幅の制約を克服するため、生成フェーズ（-t, -td）は少数スレッドに、検証フェーズ（-tb, -tbd）は多スレッドに分離し、CPUマスク（-C）を用いて物理コアにピン留めする厳密なスレッド管理を適用する。

これらのアーキテクチャ特性に即した精緻なパラメータチューニングを適用することで、リソースの制約が厳しいローカル環境であっても、Qwenファミリの真の推論ポテンシャルを極限まで引き出すことが可能である。

引用文献

1. ggml-org/llama.cpp: LLM inference in C/C++ - GitHub, https://github.com/ggml-org/llama.cpp 2. Introduction - llama.cpp - Mintlify, https://mintlify.com/explore/ggml-org/llama.cpp 3. Qwen3.6-35B-A3B speculative decoding on RTX 3090 ... - GitHub, https://github.com/thc1006/qwen3.6-speculative-decoding-rtx3090 4. Speculative Decoding is AWESOME with Llama.cpp! : r/LocalLLaMA - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/1oq5msi/speculative_decoding_is_awesome_with_llamacpp/ 5. llama.cpp/docs/speculative.md at master · ggml-org/llama.cpp · GitHub, https://github.com/ggml-org/llama.cpp/blob/master/docs/speculative.md 6. Research: Speculative Decoding for Low-Latency CPU Inference in ..., https://github.com/ggml-org/llama.cpp/issues/21453 7. Speculative Decoding: A technique that makes LLMs faster without sacrificing quality, https://medium.com/@itssujeeth/speculative-decoding-a-technique-that-makes-llms-faster-without-sacrificing-quality-a2e712b52866 8. Qwen-3.6-27B, llamacpp, speculative decoding - appreciation post : r/LocalLLaMA - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/1stcer1/qwen3627b_llamacpp_speculative_decoding/ 9. Self-speculative decoding for Qwen3.5-35B-A3B in llama.cpp? : r/LocalLLaMA - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/1rh8o4b/selfspeculative_decoding_for_qwen3535ba3b_in/ 10. Qwen3.5 dense models should be compatible draft models for Qwen3.5 MoE speculative decoding · Issue #1597 · lmstudio-ai/lmstudio-bug-tracker - GitHub, https://github.com/lmstudio-ai/lmstudio-bug-tracker/issues/1597 11. How to run Qwen3.5-27B with speculative decoding with llama.cpp llama-server? - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/1sk89a5/how_to_run_qwen3527b_with_speculative_decoding/ 12. Feature Request: Speculative decoding on Qwen3.5 · Issue #20039 ..., https://github.com/ggml-org/llama.cpp/issues/20039 13. Speculative decoding with draft model: extreme slowdown on Vulkan (UMA iGPU) when both models on same device · Issue #23126 · ggml-org/llama.cpp - GitHub, https://github.com/ggml-org/llama.cpp/issues/23126 14. llama.cpp speculative checkpointing was merged : r/LocalLLaMA - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/1sprdm8/llamacpp_speculative_checkpointing_was_merged/ 15. Speculative Decoding | Unsloth Documentation,

https://unsloth.ai/docs/basics/inference-and-deployment/saving-to-gguf/speculative-decoding 16. Speculative decoding question, 665% speed increase : r/LocalLLaMA - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/1sq7grd/speculative_decoding_question_665_speed_increase/ 17. Qwen’s MTP test puts local AI back in startup math, https://startupfortune.com/qwens-mtp-test-puts-local-ai-back-in-startup-math/ 18. unsloth/Qwen3.6-27B-MTP-GGUF - Hugging Face, https://huggingface.co/unsloth/Qwen3.6-27B-MTP-GGUF 19. MTP support merged into llama.cpp : r/LocalLLaMA - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/1tes1wx/mtp_support_merged_into_llamacpp/ 20. llama.cpp just got faster: Qwen 27B & 35BA3B on 16GB VRAM (MTP Test) - YouTube, https://www.youtube.com/watch?v=ROGG36aEkhg 21. llama.cpp fork with TurboQuant WHT-rotated KV cache & weight compression + Gemma 4 MTP and Qwen 3.6 NextN speculative decoding (+30-50% throughput). · GitHub, https://github.com/AtomicBot-ai/atomic-llama-cpp-turboquant 22. 80 tok/sec and 128K context on 12GB VRAM with Qwen3.6 35B A3B and llama.cpp MTP : r/LocalLLaMA - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/1t82zxv/80_toksec_and_128k_context_on_12gb_vram_with/ 23. Strix Halo Llama.cpp MTP Benchmarks: 27B Gets Much Faster, 35B ..., https://www.reddit.com/r/LocalLLaMA/comments/1teypb8/strix_halo_llamacpp_mtp_benchmarks_27b_gets_much/ 24. llama.cpp: Fast Local LLM Inference, Hardware Choices & Tuning - Clarifai, https://www.clarifai.com/blog/ilama.cpp 25. Prima.cpp: Fast 30-70B LLM Inference on Heterogeneous and Low-Resource Home Clusters | OpenReview, https://openreview.net/forum?id=h0LjpOG1jq 26. ROCm on 7900 XTX significantly slower than Vulkan for llama.cpp (extensive testing, out of ideas) - Reddit, https://www.reddit.com/r/ROCm/comments/1s1vo37/rocm_on_7900_xtx_significantly_slower_than_vulkan/ 27. Just an FYI, this is writeup from August 2023 and a lot has changed (for the bet... | Hacker News, https://news.ycombinator.com/item?id=42500217 28. Flash Attention performs worse under ROCM · Issue #10439 · ggml-org/llama.cpp - GitHub, https://github.com/ggml-org/llama.cpp/issues/10439 29. Has prompt processing taken a massive hit in llama.cpp for ROCm recently? : r/LocalLLaMA, https://www.reddit.com/r/LocalLLaMA/comments/1s283xr/has_prompt_processing_taken_a_massive_hit_in/ 30. GitHub - carlosfundora/llama.cpp-1-bit-turbo: HIP/ROCm fork optimized for AMD RDNA2 (gfx1030) with PrismML Q1_0_G128 1-bit quant support, RotorQuant, TurboQuant, EAGLE3 and P-EAGLE speculative decoding, and full Wave32 kernel optimizations., https://github.com/carlosfundora/llama.cpp-1-bit-turbo 31. Issues · ggml-org/llama.cpp - GitHub, https://github.com/ggml-org/llama.cpp/issues 32. RTX 5090 (CUDA) vs Radeon AI PRO R9700 (Vulkan) — Qwen3.5-35B-A3B MoE Q4_K_XL llama-bench results #19890 - GitHub, https://github.com/ggml-org/llama.cpp/discussions/19890 33. llama.cpp guide - Running LLMs locally, on any hardware, from scratch :: - SteelPh0enix's Blog, https://blog.steelph0enix.dev/posts/llama-cpp-guide/ 34. llama.cpp CPU optimization : r/LocalLLaMA - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/190v426/llamacpp_cpu_optimization/