

Strata: 12 GB の GPU で 1,250 億パラメータの MoE モデルを動かす推論エンジン

Claude Opus 5.5 (Anthropic) で作成

GitHub リポジトリ [Niko1221/Strata](#) の文書 (commit 477ac13、2026 年 10 月 16 日) に基づく解説・2026 年 10 月 18 日

要旨

Strata は総パラメータ 1,250 億の MoE モデル Qwen3.8-Flash-Next を、12 GB 程度のグラフィックカード 1 枚と 64 GB 前後の RAM を持つ普通の PC で動かす推論エンジンである。最小の量子化ファイルでも 66 GB あり、VRAM に収まらない。Strata はこの一つのモデルの形に合わせてエンジンを書き、毎トークン必ず読む小さな部分とよく使う expert を VRAM に、すべての expert を RAM に、巨大な n-gram embedding table を SSD に置く。VRAM がない expert は CPU がその場で計算し、一部は PCIe で GPU へ運ぶ。さらにモデル自身の MTP 層で数トークン先を推測し、48 層を 1 回通すだけでまとめて検証する。RTX 5070・Q2_0 で、エンジン 0.1.36 は 4K コンテキストの生成 94 tokens/s、32K プロンプトの読み込み 2,650 tokens/s を記録した [1] [2]。

1 はじめに

大規模言語モデルは 1 トークンを作るたびに、そのトークンが使う重みをメモリから読み出す。一般的な PC で最も速いメモリは VRAM (数百 GB/s) で、次がシステム RAM (数十 GB/s)、最も遅いのが SSD である。したがって生成速度は計算能力より先に「1 トークンあたり何バイトを、どのメモリから読むか」でほぼ決まる [1]。

Strata が対象とする Qwen3.8-Flash-Next は総パラメータ 1,250 億の **Mixture-of-Experts (MoE)** モデルで、1 トークンあたりに使うパラメータは 60 億にとどまる。それでも品質を保てる最小の量子化ファイルは 66 GB あり、12 GB のカードの 5.5 倍になる [1]。通常の解決策は llama.cpp のような汎用エンジンで収まらない部分を CPU とシステム RAM で動かすことだが、この方法は動きはするものの、マシンの大部分がほとんどの時間遊んでいる状態になる [1]。

Strata の答えは特化である。この一つのモデルの正確な形と、この種の PC の構成に合わせてエンジンを書き、GPU・CPU・RAM・PCIe リンクのそれぞれが得意な仕事を同時にこなすようにする [1]。図 1 はその分担を示す。毎トークン必ず使う中核とよく使われる expert はグラフィックカードに、24,576 個の expert すべては RAM に、29 GB の参照テーブルは SSD に置く。カードがない expert が要るときは、CPU が RAM 上で同時に計算する [1] [2]。

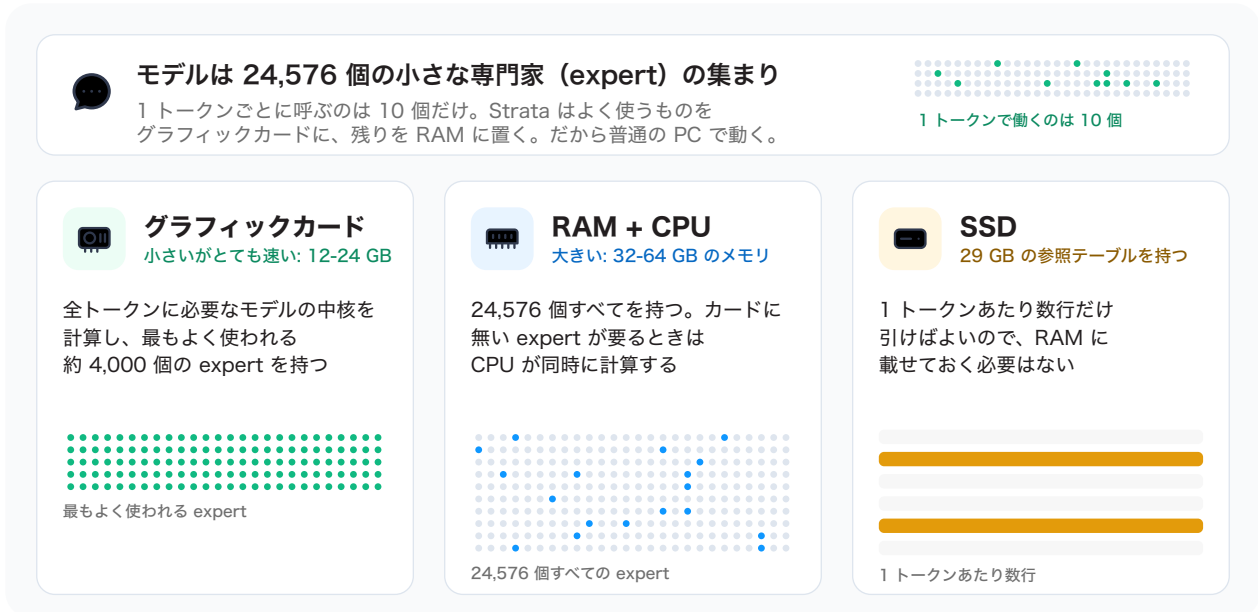


図 1 Strata が PC 全体に仕事を分ける様子。よく使う expert をグラフィックカードに、24,576 個すべてを RAM に、大きな参照テーブルを SSD に置き、カードにない expert は CPU が同時に計算する

これに、モデル自身が持つ **MTP 層** (multi-token prediction) による推測と、VRAM に置く expert を会話に合わせて入れ替える仕組みが加わる。PC の各部品は互いを待つのではなく同時に働く。GPU は全トークン共通の重みと VRAM 内の expert を処理し、CPU は RAM 内の expert を処理し、PCIe リンクはその合間に expert の一部を GPU へ運び、SSD はテーブルの行をそれを使う層より一歩先に読む [1]。

主な結果を先に述べる。RTX 5070 (12 GB) ・ Ryzen 5 7600 ・ 64 GB DDR5 の PC で、2 ビットの Q2_0 ファイルを使うと、エンジン 0.1.36 で 4K コンテキストの生成が 94 tokens/s、32K トークンのプロンプトの読み込みが 2,650 tokens/s だった [2]。投機的デコーディングは生成を 1.6~1.8 倍速くし、出力は通常の greedy decoding とトークン単位で一致する (i-quant のファイルには例外がある。4 節) [1] [3, §Guess, then check]。

本論文は GitHub の Niko1221/Strata (commit 477ac13、2026-10-16) のドキュメントと、既存の英語論文 (以下、論文) に基づく。Strata は版 (0.1.x) ごとに機能と既定が変わるので、数値には版を添える [1] [4, §Speed]。2 節で対象モデルを 1 トークンあたりのバイト数の観点から説明し、3 節でメモリ tier と 1 層の中の分担を、4 節で推測と検証によるデコードを、5 節で prefill と長いコンテキストを、6 節で会話状態と同時リクエストを扱う。7 節は複数 GPU と他のハードウェア、8 節は評価、9 節は限界、10 節は関連研究と結論である。

2 対象モデル

Qwen3.8-Flash-Next の設計者は、アーキテクチャ上の選択の一つひとつを「1 トークンあたりに動くバイト数」で見積もって決めており、それがモデルの形に表れている [1]。Strata の設計はこの形をそのまま前提にしているので、まず構造を見ておく。表 1 に諸元を、図 2 に全体の構成と 1 層の処理順を示す。

表 1 Qwen3.8-Flash-Next の諸元

項目	値
総パラメータ / 1 トークンあたりに使うパラメータ	1,250 億 / 60 億
層	48 層 (mixer は GDN 36 層 + QSA 12 層)
QSA	KV ヘッド 2 本、参照する位置は高々約 2,048
routed expert	各層 512 個 (計 24,576 個)、router が上位 10 個を選ぶ (約 2%)。ほかに shared expert

項目	値
n-gram embedding table	510 億パラメータ、IQ4_NL で 28.8 GB、直前 3 トークンの id で 1 トークン 16 行を引く
MTP 層	1 層。GGUF には含まれず、BF16 チェックポイントから取得 (5 GB)、VRAM 上で 0.8 GB
学習したコンテキスト長	262,144 トークン

[1] [5] [4, §How it works] [4, §Speed]

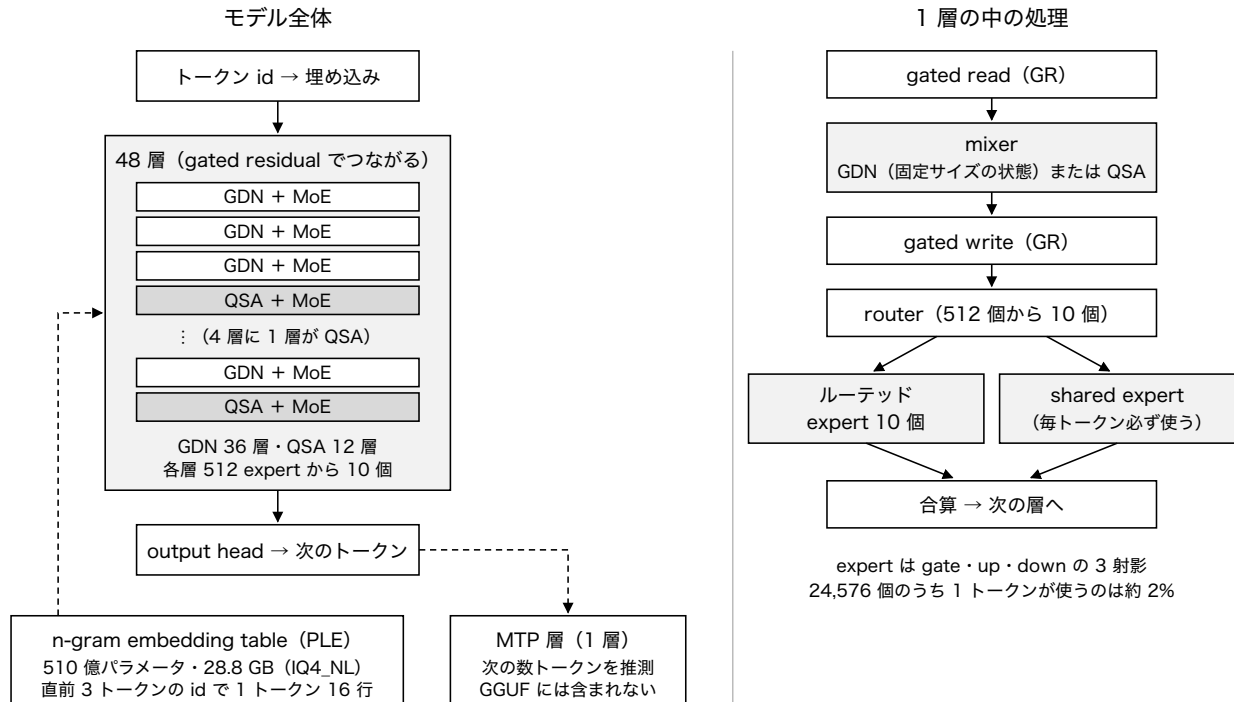


図2 Qwen3.8-Flash-Next の構成の模式図。左はモデル全体、右は 1 層の中の処理の順序。QSA 層の並び方は「4 層に 1 層」という比率だけを示した模式である

各層はトークン間で情報を混ぜる **mixer** と、トークンごとに独立に働く MoE の二段からなる。図2の右側のとおり、1 層の中では gated read、mixer、gated write、router、shared expert と選ばれた expert の計算、合算の順に進む [1]。層と層の間の残差の経路は複数のストリームを持つ hyper-connections で、Strata の資料では **gated residual** (GR) と呼ぶ [1]。

mixer の 4 分の 3 にあたる 36 層は Gated DeltaNet (GDN) である。GDN は再帰型の mixer で、過去の全トークンのキーと値を蓄える代わりに固定サイズの状態を 1 トークンごとに更新する。状態の大きさはコンテキスト長に依存しないので、デコードの仕事は 1 トークンあたり一定である [1]。残る 12 層は Qwen Sparse Attention (QSA) で、KV ヘッドを 2 本しか持たず、コンテキストがどれほど長くても選び出した高々約 2,048 か所だけに注意を向ける [1]。参照する位置の数に上限があるので注意の計算量はコンテキスト長に比例して増えず、KV キャッシュも小さい。それでも 262K トークンでは無視できず、12 の注意層の KV キャッシュ、インデクサのキー、大きくなったプロンプト用バッファが合わせて約 4 GB を占める [1]。

MoE は各層 512 個の **routed expert** から **router** が 10 個を選ぶ。48 層で 24,576 個のうち 1 トークンが触れるのは約 2% にすぎない [1]。これとは別に、毎トークン必ず使う **shared expert** がある [1]。router の選択は均等ではなく、一部の expert は他よりはるかに頻繁に選ばれる。よく使うものを VRAM に置けば限られた VRAM でも多くの選択を GPU で処理できるので、この偏りは Strata にとって決定的に重要である [1]。

層の外側には二つの大きな部品がある。一つは 510 億パラメータの **n-gram embedding table** で、直前 3 トークンの id で引かれるため、どの行が要るかはその行を使う層に到達する前に分かる [1]。もう一つ

は次の数トークンを安価に推測する 1 層の MTP 層で、Strata はこれを投機的デコーディングの draft に使う。GGUF ファイルには含まれないので、Strata は BF16 チェックポイントからこの層のテンソルだけを取得し、expert を 2 ビットにし、output head を部分集合の語彙に絞った形で VRAM に置く [1]。

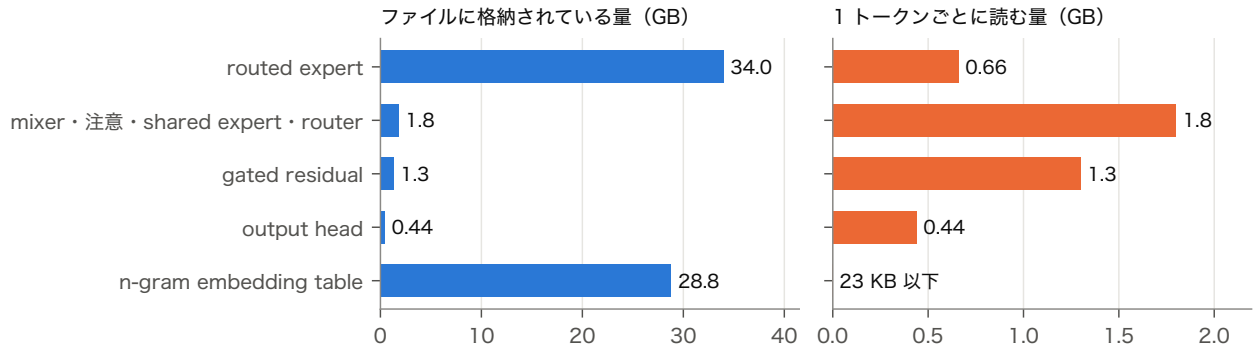


図 3 Q2_0 ファイルで、モデルの各部分がファイルに格納されている量 (左) と 1 トークンごとに読まれる量 (右)。論文の表 1 による

図 3 は Q2_0 ファイルについて、各部分の格納量と 1 トークンあたりの読み出し量を並べたものである。論文はこの勘定を中心に Strata の設計を組み立てたと述べている [1]。読み取ってほしい点は三つある。第一に、mixer・注意・shared expert・router (1.8 GB)、gated residual (1.3 GB)、output head (0.44 GB) は合わせて約 3.5 GB と小さいが、毎トークン全部が読まれる。最も帯域の広い VRAM に置くのが合理的で、12 GB のカードにも収まる [1]。第二に、routed expert は格納量 34.0 GB と大きいですが、1 トークンが読むのはその 2% の 0.66 GB だけである。論文はこの 0.66 GB が設計の中心に置かれた量だと述べる [1]。第三に、n-gram テーブルはファイルの 4 割を占めるが 1 トークンあたり 23 KB 以下しか読めないため、SSD に置いたままでよい [1]。

0.66 GB が中心になるのは、どこで計算するかでその読み出し速度が一桁変わるからである。論文のテストマシンでは VRAM の帯域が 672 GB/s、RAM が約 41~52 GB/s、PCIe 4.0 x16 が約 26 GB/s だった [1]。VRAM にある expert は GPU が速く読めるが、VRAM の外にあるものは RAM の帯域で CPU が読むか、PCIe 越しに GPU へ運ぶしかない。VRAM に置ける expert が 1 個増えるごとに CPU が計算するものが 1 個減る。「GPU の速さより VRAM の容量が効く」「VRAM が 1 GB 増えるごとに expert が約 700 個多く入る」とドキュメントが繰り返すのはこのためである [1] [3, §Who does what]。残る問題は VRAM に入りきらない expert の読み出しを毎トークンどう処理するかであり、その答えが次節のシステム設計である。

3 システム設計

3.1 三つのメモリ tier

Strata は PC のメモリを VRAM、システム RAM、SSD の三つの **tier** として扱い、前節のバイト勘定に従って中身を割り振る (図 4) [1]。

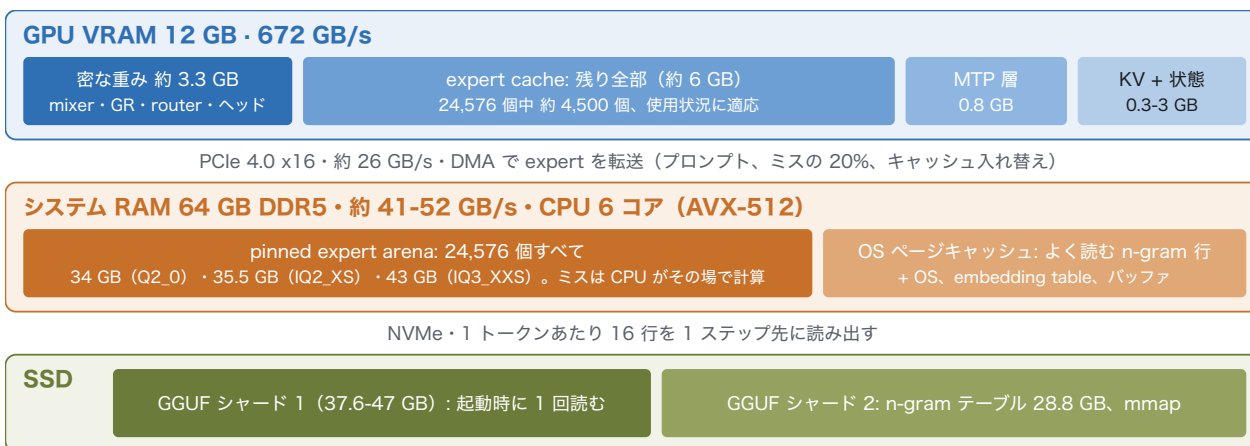


図 4 三つのメモリ tier と、Strata がそれぞれに置くもの (RTX 5070 12 GB、64 GB DDR5、PCIe 4.0 x16、NVMe SSD。帯域はテストマシンでの実測)

VRAM には毎トークン必要なものをすべて置く。mixer、gated residual、router、shared expert、output head、MTP 層、KV キャッシュと再帰状態である。12 GB のカードでは密な重みが約 3.3 GB、MTP 層が 0.8 GB、KV と状態が 0.3~3 GB で、残りの約 6 GB がすべて **expert cache** になり、24,576 個のうち約 4,500 個が入る [1]。VRAM の expert が 1 個増えるごとに CPU の仕事が 1 個減るので、空いた VRAM はすべてキャッシュに回すのが得になる [1] [3, §Who does what]。

システム RAM には 24,576 個の expert を、VRAM にあるものも含めてすべて、一つの **pinned memory** (page-locked) の **arena** に置く。大きさは Q2_0 で 34 GB、IQ2_XS で 35.5 GB、IQ3_XXS で 43 GB である [1]。pinned にする理由は二つある。CPU がどの expert でもその場で計算できること、そして GPU がどの expert でも PCIe 越しに DMA で取り込めることである。全部を pinned しておけば、VRAM にない expert を GPU に運ぶか CPU で計算するかを層ごとに自由に選べる [1]。Windows では 34~43 GB を一つの範囲としてページロックしようとするとう失敗するので、層ごとの範囲に分けて pinned する [1]。

SSD には 28.8 GB の n-gram テーブルを置いたままにし、各トークンが必要とする 16 行を、それを使う層より一歩先に読み出す [1]。このテーブルを RAM にも VRAM にも置かずに済むことが、66 GB 以上のモデルを 64 GB の PC で動かす前提になっている [1] [6, §Will it fit?]

3.2 1 層の中の分担

デコードの処理は 48 層を順に通る。Strata の性能を決めるのは、1 層の中で GPU · CPU · PCIe の三つの資源をいかに重ねて動かすかである。図 5 はその流れを示す。

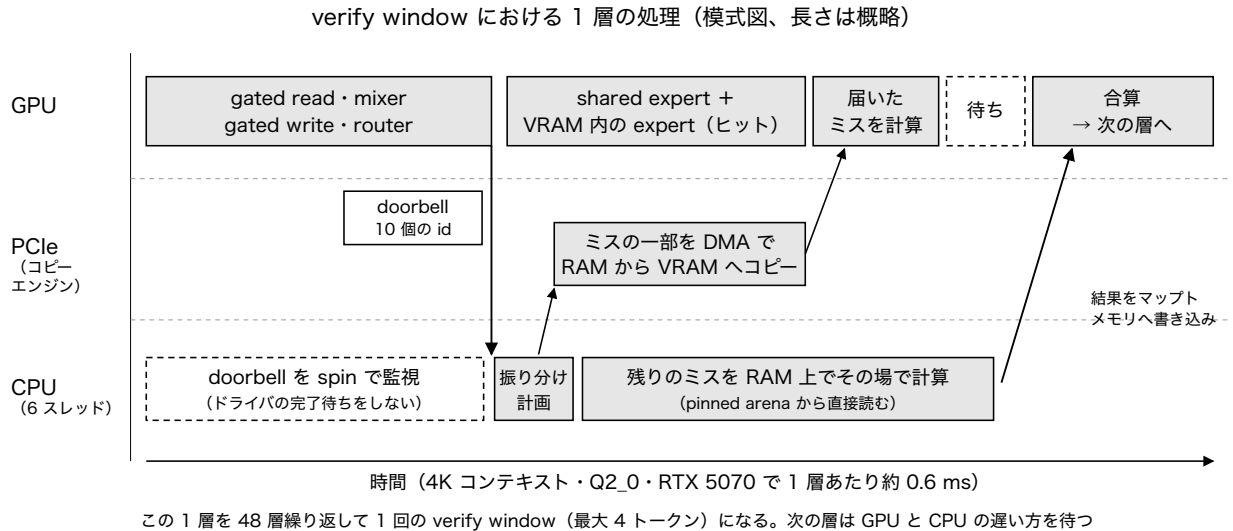
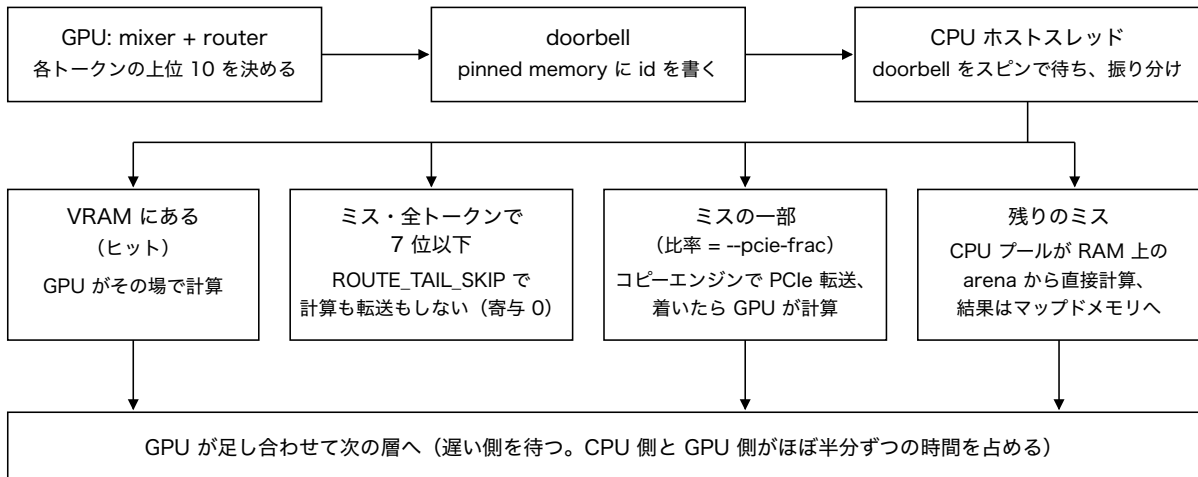


図 5 verify window における 1 層の処理のタイムライン (模式図、長さは概略)。GPU が VRAM 内の expert を計算している間に、CPU が VRAM がない expert を RAM 上で計算し、PCIe のコピーエンジンがミスの一部を GPU に運ぶ。次の層は遅い方待つ

各層で GPU はまず gated read、mixer、gated write、router を実行する。router が選んだ 10 個の expert の id は、pinned のホストメモリ上の小さなブロックに書き込まれる。これを **doorbell** と呼ぶ [1]。CPU はドライバの完了待ちではなくこの doorbell を spin で監視しており、id が書かれるとすぐに expert を三つに振り分ける。VRAM のキャッシュにあるもの (**ヒット**) は GPU に任せる。キャッシュにないもの (**ミス**) のうち一定の割合は GPU のコピーエンジンが PCIe 越しに運び、残りのミスは RAM のすぐそばにある CPU のコアが計算する [1]。CPU は結果を GPU から見える pinned memory に書き戻し、GPU がすべてを足し合わせて次の層に進む。GPU によるキャッシュ済み expert の計算と CPU によるミスの計算は同時に走り、次の層はその遅い方待つ。4K コンテキストでは 1 層がおよそ 0.6 ms で、verify window (4 節) ごとにこれを 48 層繰り返す [1]。

doorbell を使うのは、1 層 0.6 ms 程度の処理を毎ウィンドウ 48 回繰り返すので、同期的なドライバ呼び出しで待つたびにに入る遅れが無視できないからである。逆方向も同じで、48 層のパス全体はウィンドウの大きさごとに記録された CUDA グラフとして実行され、ホストが同期的なドライバ呼び出しで待つことはない [1]。

この構造が速い理由は、expert の計算を置き場所の近くで行う点にある。VRAM にある expert は 672 GB/s で GPU が読み、RAM にある expert は RAM の帯域で CPU が読む。RAM の expert を毎回約 26 GB/s の PCIe で GPU に運ぶより、CPU がその場で計算した方が多くの場合に速い。ただし CPU の計算能力にも RAM の帯域にも限りがあるので、コピーエンジンという三つ目の資源でミスの一部を GPU に回し、両者の負荷を釣り合わせる [1]。

verify window の 1 層で、選ばれた expert がどこで計算されるか (CUDA・1 GPU の既定)

-pcie-frac を明示しない CUDA・1 GPU では、0.1.42 から PCIe 比率を CPU プールの実測速度 R と PCIe 速度 L から $L / (L + 1.85 R)$ で決め直す。tail skip は「ウィンドウ内のどのトークンにとっても下位」のときだけ働き、残した expert の重みは再正規化しない。

図 6 verify window の 1 層で、選ばれた expert がどこで計算されるか (CUDA・GPU 1 枚の既定)。ヒットは GPU、ミスの一部は PCIe 転送のあと GPU、残りのミスは CPU が計算する。ウィンドウ内のどのトークンにとっても下位のミスを飛ばす経路は 4 節で述べる

図 6 はミスの振り分けを詳しく示す。どれだけを PCIe に回すべきかは、CPU が何に律速されているかで変わる。論文の時点では、i-quant のファイル (IQ2_XS, IQ3_XXS) でミスの 55% を PCIe に回していた。i-quant では CPU がコードブックの復号という演算に律速されて RAM の帯域が余るからで、DMA コピーにしたことで 4K の IQ2_XS は 70 から 78 tokens/s、IQ3_XXS は 56 から 65 tokens/s になった [1]。一方 Q2_0 では CPU のカーネルがすでに RAM の帯域を使い切っており、DMA は CPU から帯域を奪うだけになるので、20% にとどめていた [1]。論文はこのコピーエンジンの利用を「重ね合わせのうち報われたもの」と呼ぶ。最初の実装では GPU 上のコピーカーネルが RAM から直接読んでおり、この読み出しが GPU のクリティカルパスに乗るため、ミスの 20% を超えて取り込むと遅くなった [1]。

この方式と割合は後の版で変わっている。エンジン 0.1.14 では、IQ のパックで DMA コピー中にホストがドライバの中で待ち続ける停止が見つかり、expert は GPU のカーネルでコピーされるようになった [4, §Troubleshooting]。0.1.42 以降、CUDA・GPU 1 枚で割合を指定しない場合は、CPU プールがミス 1 個にかかる実時間から求めた速度 R と PCIe の速度 L を使い、割合を $L / (L + 1.85 R)$ で決め直す。CPU が速いほど PCIe の割合を下げ、リンクが速いほど上げる向きである。係数 1.85 は 3 台のマシンへの当てはめで、最適値そのものではない [4, §The PCIe share of the missed experts is...]。

逆に報われなかった重ね方もある。各ウィンドウを二つのトークングループに分け、一方の CPU expert を計算している間に GPU がもう一方を処理する方式は、4K で 91.8 から 85.8 tokens/s へと約 7% 遅くなった。密な重みを 2 回読むことになり、グループごとにトークン間で共有される expert が減るからである [1]。重ねる価値があるのは別々の資源を同時に使う場合であって、同じ資源を二度使う場合ではない。

3.3 会話に追従する expert cache

Strata は起動時に、ほかのプロンプトで記録したプロファイルに従って、最もよく使われる expert で VRAM のキャッシュを埋める。そして 4 ラウンドごとに、現在の会話が繰り返す求める expert を最大 96 個、最も使われていないスロットに入れ替える。入れ替えは GPU が draft している間に行う [1]。論文の測定では、プロファイルで埋めた約 4,500 個のキャッシュのヒット率 (VRAM から供給された選択の割合) は学習に使っていないプロンプトで約 0.50 で、会話に合わせた入れ替えを加えると 12 GB のカード・4K で 0.72 になった [1]。入れ替えのために誰かを待たせることもない。古い expert はすぐに追い出し、新しい

expert はコピーが届いてから受け入れるように変えたところ、4K の Q2_0 は 91.7 から 94.4 tokens/s になった [1]。

キャッシュの大きさはスロット数ではなくバイト数で決める。IQ3_XXS の expert は層によって 1.3~2.3 MB と大きさが違い、各スロットを最大の expert に合わせるのをやめて層ごとの大きさに合わせたところ、入る expert は 2,673 個から 3,556 個に増え、1K のデコードが 13% 速くなった [1]。

3.4 低 RAM モード

ここまでは全 expert を RAM に置けることを前提にしていた。RAM が expert とシステム (expert + 約 10 GB) を同時に保持できない PC のために、エンジン 0.1.26 から低 RAM モードがあり、setup が選ぶ。expert を RAM にコピーする代わりにファイルからマップし、GPU が持たない expert は OS のファイルキャッシュに任せる方式と、GPU が持たない分だけを起動時に RAM にコピーする方式 (0.1.30) がある。大きな GPU ではほぼ通常で動くが、小さな GPU ではほとんどの expert が SSD から来るので大幅に遅くなる [4, \$Speed]。

4 デコード: 推測して検証する

4.1 MTP 層による投機的デコーディング

1 トークンごとに約 3.5 GB の密な重みを読むことが GPU の主なコストであり、このコストは 4 トークンをまとめて処理してもほとんど増えない。重みを 1 回読めば、その間に複数トークンの行列ベクトル積をまとめて計算できるからである [1]。そこで Strata は MTP 層でトークンを安価に推測し、全 48 層を 1 回通すだけで複数のトークンを検証する**投機的デコーディング**をデコードの基本単位にしている。図 7 はその考え方を、図 8 は 1 ステップの中身を示す。

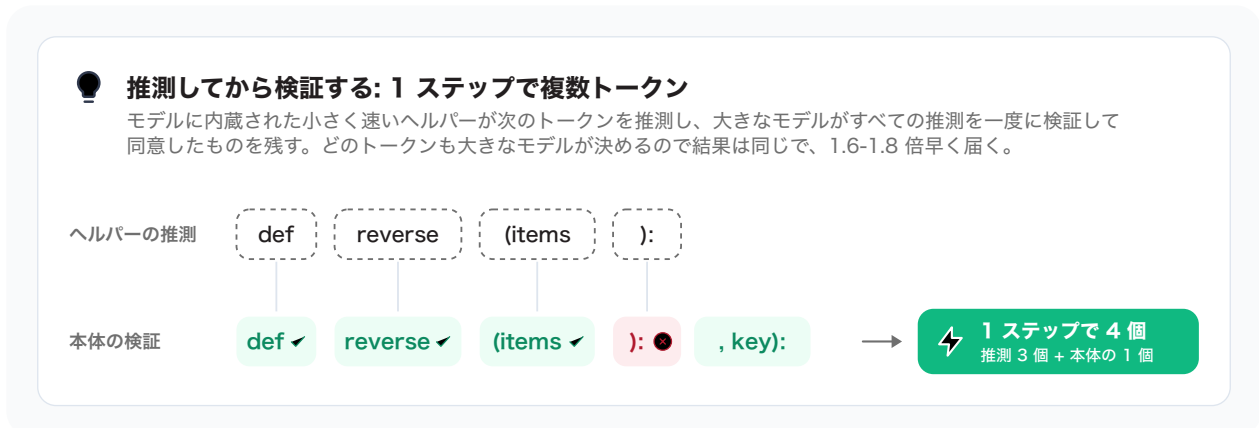
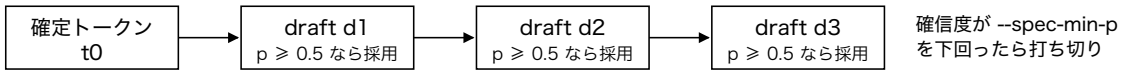
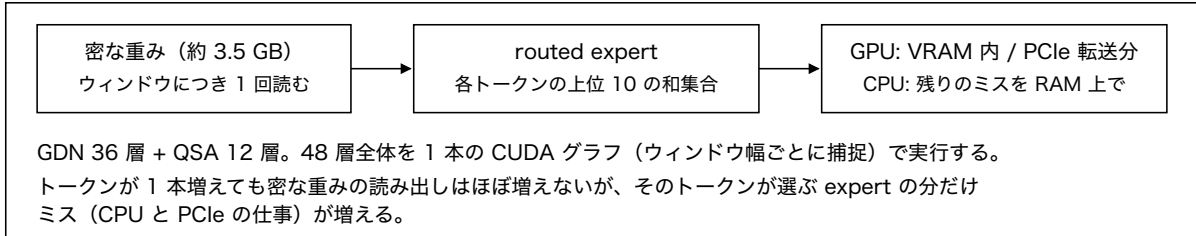
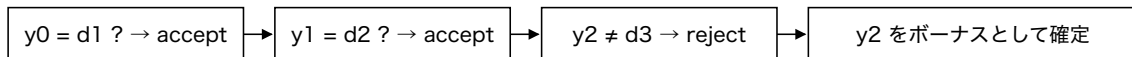


図 7 投機的デコーディングの考え方。モデルに内蔵された小さな MTP 層が次のトークンを推測し、本体のモデルがすべての推測を一度に検証して、同意したものと自分の 1 トークンを採用する

各ステップでは、最後に確定したトークンと最大 3 個の **draft** を一つの **verify window** として全 48 層に通す。draft がウィンドウに入るのは MTP 層の確信度が 50% 以上の間だけである。本体が同意した draft をすべて **accept** し、さらに本体自身の 1 トークンを加える [1]。1 回の検証で平均 2.4~3.2 トークンが確定し、draft なしより 1.6~1.8 倍速い [3, \$Guess, then check]。論文の測定では、draft なしのデコードが 4K で 47~57 tokens/s、verify window を使うと 82~92 tokens/s だった [1]。

(1) draft: MTP 層 (VRAM 上の 1 層) が次のトークンを順に推測する**(2) 検証: [t0, d1, d2, d3] を 1 つの verify window として 48 層に 1 回通す****(3) 照合: 各位置で本体モデルが出したトークン y と比べる**

この例では 1 回の検証で d1, d2, y2 の 3 トークンが確定する (accept した draft 数 + 1)。

reject した位置以降の状態 (KV と GDN の状態) は巻き戻し、次のステップは y2 を t0 として始める。

確定するトークンはすべて本体モデルの出力なので、greedy decoding では draft なしと同じトークン列になる。

図 8 MTP 層による draft と検証の 1 ステップ。draft を含む 4 トークンを 1 つの verify window として 48 層に通し、各位置で本体の出力と照合する。最初に食い違った位置では本体のトークンを選び、それ以降の状態を巻き戻す

出力が greedy decoding と一致するのは、確定するトークンがすべて本体自身の出力だからである。draft は「本体がそのトークンを出すか」を一度に確かめるための候補にすぎない。図 8 のように、各位置で本体が選んだトークンと draft を比べ、一致した位置までを accept し、最初に食い違った位置では本体のトークンを選ぶ。reject した位置以降の KV と GDN の状態は巻き戻す。論文はわざと誤った draft を与える試験と巻き戻しの試験でこの一致を確かめている [1]。ただし i-quant のモデルでは、CPU が expert を 1 トークンだけで使うときと複数トークンで共有するときで丸めがわずかに異なるカーネルを使うので、draft やキャッシュ状態が違えば温度 0 でも別の (同程度に良い) 答えになりうる [4, §Speed]。

効果が、全体を VRAM に載せるエンジンが報告している 2.5~3 倍に届かないのは、Strata の構成に固有の理由による。ウィンドウに入るトークンが 1 個増えるごとに、そのトークンが自分の expert を選ぶので、CPU が計算するミスも増える。密な重みの読み出しはトークン数に対してほぼ一定だが、CPU の expert の仕事はトークン数とともに増える [1]。投機の利得は密な重みを 1 回読むだけで複数のトークンを処理できる分と、ミスの増加分との差し引きで決まる。

回答が文脈を繰り返す場合 (コードの編集や引用) には、**prompt lookup** が文脈中の以前の出現から最大 5 トークンを draft する (エンジン 0.1.7)。測定した acceptance rate とコストから得になる場合にだけ使われ、コードの編集で 6~11% 速くなり、ほかの文章では変わらない。draft は MTP のものと同じく検証されるので出力は変わらない [4, §How it works]。得になる場合に限るのは、外れる draft でウィンドウを長くすると、確定トークンは増えないのにミスの仕事だけが増えるからである。

4.2 1 回の verify window の時間

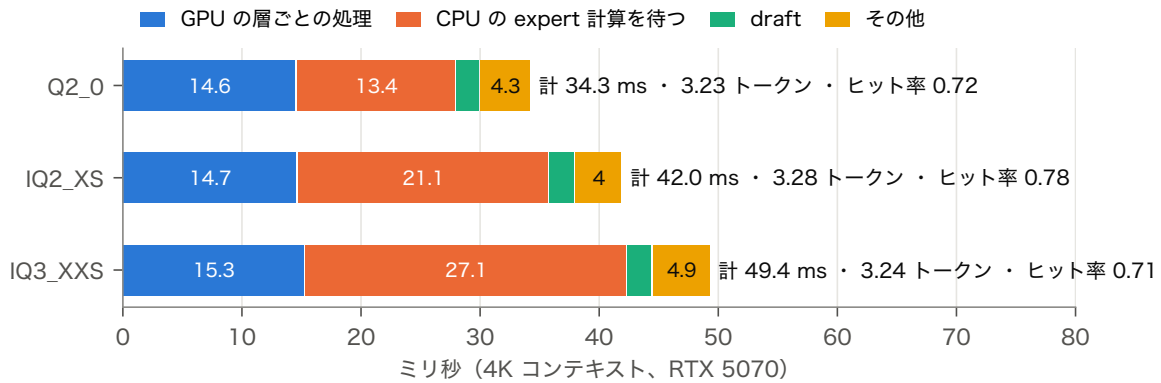


図 9 4K コンテキストでの verify window 1 回の時間の内訳 (RTX 5070、論文の表 5)。棒の横に、1 回の合計時間、確定したトークン数、VRAM のヒット率を記す

図 9 は論文が測った 1 回の verify window の内訳である。「GPU の層ごとの処理」は CPU が GPU の仕事 (ウィンドウにつき 1 回の重みの読み出し、注意、キャッシュ済み expert) を待った時間、「CPU の expert 計算を待つ」は GPU が CPU のミスの計算を待った時間である [1]。Q2_0 では 3.23 トークンを 34.3 ms で確定するので約 94 tokens/s になる。三つのファイルでウィンドウあたりのトークン数はほぼ同じなので、速度の差はほぼ CPU 側の時間の差から来ている。GPU 側は 14.6~15.3 ms とほとんど変わらない。GPU の仕事は主に約 3.5 GB の密な重みの読み出しだからである [1]。

論文はボトルネックを次のようにまとめている [1]。Q2_0 では、限界は CPU ではなくその背後の RAM である。expert のカーネルはすでに約 42 GB/s で重みを流しており、これは GPU も RAM から読んでいる状態で RAM が出せる帯域 (同時に 41 GB/s) に近い。i-quant では CPU 自体が限界である。コードブックの重みの復号にはテーブル参照と符号の処理が要り、6 コアで約 25 GB/s しか出ない。IQ3_XXS が 25% 大きいだけなのに遅いのはこのためである。RAM については帯域と容量の両方が効き、64 GB はぎりぎり足りる量である。GPU については速さより大きさが効く。GPU 自身の仕事 (ウィンドウあたり約 14 ms) は二番目の限界で、これはメモリ帯域で決まる。

まとめると、このマシンでは GPU と CPU がそれぞれウィンドウの約半分を必要とし、毎層で互いを待っている。エンジンは釣り合っているが、どれか一つの部品だけを強化しても速度は 2 倍にならず、片方だけを速くしても改善は高々 3 分の 1 程度にとどまる [1]。

4.3 tail skip: 順位の低いミスを計算しない

後の版では、この釣合いの CPU 側を直接削る既定が入った。数トークンの verify window で、ウィンドウの**すべての**トークンが上位 10 の 7 位以下 (重みの最も小さい側) に置いた VRAM 外の expert は、どのトークンにもわずかしき寄与しない。0.1.42 から CUDA ビルドでは、そのようなミスをコピーも計算もせず寄与を 0 にする (図 6)。ほかの expert の重みは再正規化しない。0.1.43 からは RDNA4 の HIP でも既定オンで、無効にするとこの変更のないビルドの答えがバイト単位で戻る [4, §Missed experts in a verify window]。

これは計算を省くので出力は厳密には変わる。teacher forcing の KL は 0.003~0.006 で、同じエンジンを 2 回動かしたときの揺れ (約 0.0007) より明確に大きく、Q2_0 と IQ3_XXS の差 (0.1006) より 1 桁以上小さい。速度は 1 枚構成のカード 3 台で 1.13~1.16 倍だったが、ほぼ全 expert を VRAM に持つ 2 GPU 構成では得られなかった。batch slot がある場合や、全 expert の 90% 以上が GPU にある場合は適用されない [4, §Missed experts in a verify window]。

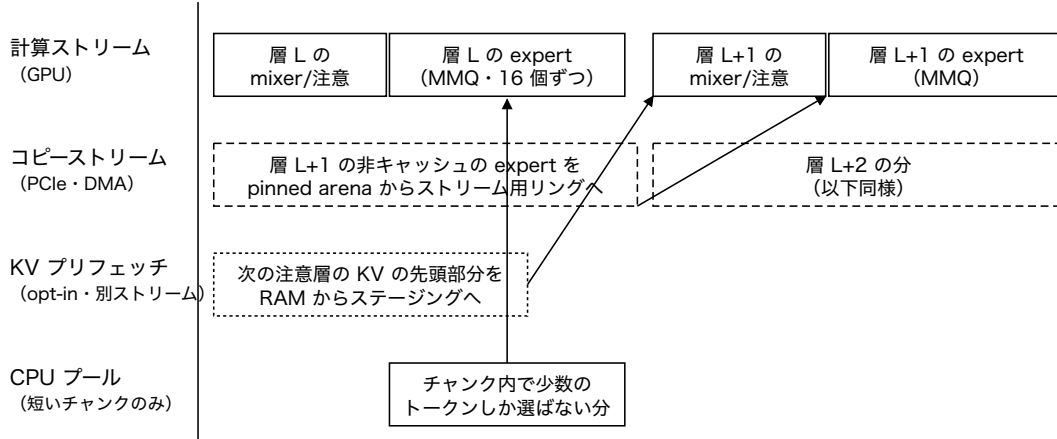
5 prefill と長いコンテキスト

5.1 転送と計算を重ねる

プロンプトの読み込み (prefill) では、デコードとは逆の結論になる。数千トークンを一度に流すので、どの expert も多数のトークンに使われ、計算すべき場所は GPU になる [1]。デコードではほとんどの expert が

1 トークンにしか使われないので、重みを PCIe で運ぶより置き場所の RAM で CPU が計算した方が速かった。prefill では 1 個の expert を数千トークンが共有するので、重みを一度 GPU に運べば GPU の演算能力で多数のトークンを処理でき、1 トークンあたりの運ぶコストは小さくなる。問題は VRAM にない expert を PCIe で運ぶ時間をどう計算と重ねるかに移る。

prefill の 1 チャンク (例: 8,192 トークン) を層ごとに処理するときの重なり



- ・バッファ (ストリーム用リング、チャンクの活性) は VRAM の expert cache のスロットを借り、プロンプト後に詰め直す。
- ・`--prefill auto` は、借りられる領域でリングを満たせる最大のチャンク (既定は最大 8,192、32,768 は opt-in) を選ぶ。
- ・CPU 分担 (STRATA_PREFILL_CPU_SHARE) は 1,024 未満のチャンクで既定オン (CUDA・1 GPU、0.1.41)。
- ・KV プリフェッチ (STRATA_KV_PREFETCH=1) は KV の一部が RAM にあるとき (KV ストリーミング) だけ意味がある。
- ・プロンプト末尾の数十トークン (`--short-read`) はバッチ経路ではなく verify window で読む。

図 10 prefill の 1 チャンクを層ごとに処理するときの重なり。GPU が層 L の計算をしている間に、コピーストリームが層 L+1 のキャッシュにない expert を pinned arena から運ぶ。短いチャンクでは CPU も一部を分担する

論文の時点では、プロンプトを 2,048 トークンのチャンクで処理し、各層でキャッシュにない expert を PCIe でコピーしながら、半精度に展開して cuBLAS で掛けていた。この経路は expert ごとのカーネル起動と半精度への展開に律速され、26 GB/s の PCIe のうち約 7 GB/s しか使っていなかった [1]。エンジン 0.1.13 でチャンクを最大 8,192 トークンにし、expert を展開せずに llama.cpp の量子化カーネルで掛け、次の層の expert を現在の層の注意の計算中に PCIe で運ぶようにした (図 10)。RTX 5070・32K トークンで、Q2_0 は 572 から 1,290 tokens/s、IQ3_S は 383 から 1,208 tokens/s になった [4, §Speed]。0.1.36 では Q2_0 のプロンプトの expert を融合した int8 テンソルコアのカーネル (RTX 30 以降) で計算するようになり、同じ PC の 32K は 2,170 から 2,653 tokens/s になった [4, §Speed]。

チャンク用のバッファは専用の VRAM を持たず、expert cache のスロットを借りてプロンプトのあとに戻す。チャンクが大きいほど速いが、借りるスロットが増える [1] [4, §How it works]。逆に短いチャンクでは、少数のトークンしか選ばない expert を CPU に計算させた方が速い場合があり、0.1.41 以降、CUDA・GPU 1 枚では 1,024 トークン未満のチャンクで CPU が expert の一部を分担する [4, §Short prompts]。QSA が参照する位置の数に上限があるので、プロンプトの速度はコンテキスト長に対してほぼ平らになる。論文の Q2_0 は 4K で 539、262K で 496 tokens/s だった [1]。

5.2 長いコンテキストのコストは VRAM

長いコンテキストのコストは主に計算ではなく VRAM である [1]。GDN 層の状態は固定サイズで、QSA の KV キャッシュもトークンあたりは小さいが、262K では 12 の注意層の KV キャッシュ、インデクサのキー、大きくなったプロンプト用バッファが約 4 GB を占める。自動で大きさを決めると GPU には約 4,500 個ではなく約 1,600 個の expert しか置けなくなり、draft の acceptance rate の低下も重なって、論文の Q2_0 のデコードは約 95 から約 56 tokens/s に下がった [1]。

エンジン 0.1.5 の **KV ストリーミング**はこの VRAM を取り返す。64K 以上のコンテキストで、setup が KV キャッシュを RAM に置き、注意が読む部分だけを VRAM に置く。注意が読む値はまったく同じで、変

わるのは KV の置き場所だけなので、出力は変わらない。その代わりコンテキスト 1 トークンあたり約 13.7 KB の RAM を使う (128K で 1.7 GB) [4, §Speed]。

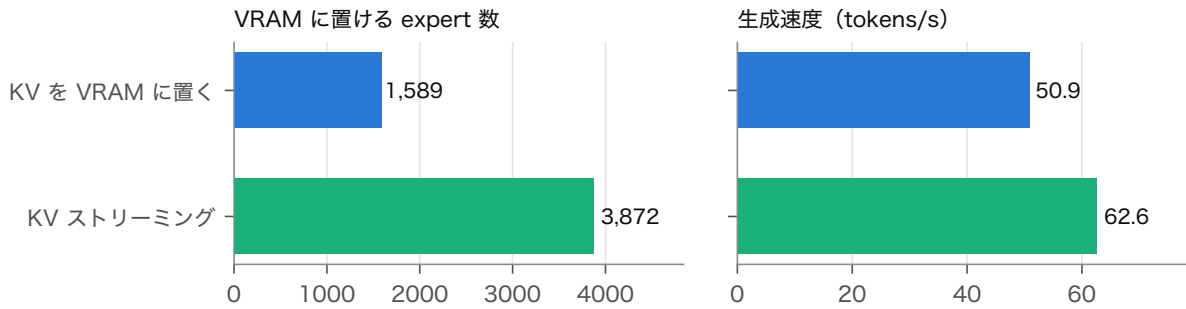


図 11 KV ストリーミングの効果 (RTX 5070、Q2_0、262K コンテキスト)。KV を RAM に移した分だけ VRAM に置ける expert が増え、生成速度が上がる

図 11 はその効果である。Q2_0・262K で、VRAM に置ける expert は 1,589 個から 3,872 個に増え、生成は 50.9 から 62.6 tokens/s になった。128K では約 6% の改善である [4, §Speed]。逆に KV を VRAM に多く置いてもほとんど得がないことは測定で示されている。Tesla P100・IQ2_XS で VRAM に置く KV を 4 倍以上に変えてもデコードはほぼ変わらず、expert のスロットを増やすとデコードが伸びた [7, §R3]。KV の形式には精度と VRAM のトレードオフがあり、setup は 4K を超えるコンテキストで 8 ビットの KV を使う (エンジン 0.1.26 の測定条件。論文の時点の本文では 8K 超) [4, §Speed] [1]。

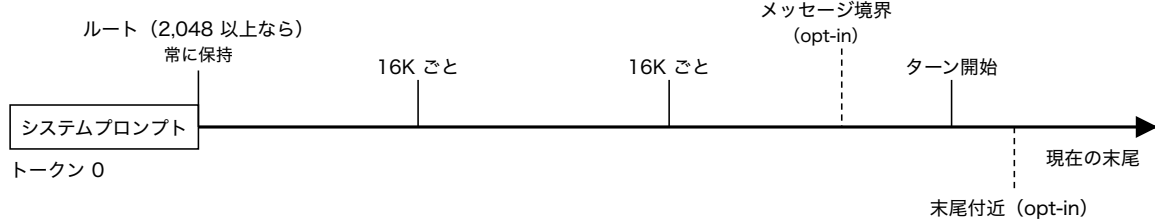
6 会話状態と同時リクエスト

6.1 会話キャッシュ

長いプロンプトの読み込みには数十秒から数分かかる。チャットやエージェントでは、各リクエストは直前までの会話をほぼそのまま含み、末尾に少しだけ新しい内容が付くので、毎回全体を読み直せばこの時間が毎ターンかかる。論文は執筆時点で「どのリクエストもプロンプト全体を処理し直す」と述べ、各ターンの終わりの状態を保存すれば長いチャットの 30 秒の読み直しが 1 秒未満になり、エージェントにとって最大の改善になると予想していた [1]。後の版の Strata はこれを実装しており、2 通目以降は新しい部分だけを読んで数秒で応答が始まる [2] [3, §Reading long texts]。

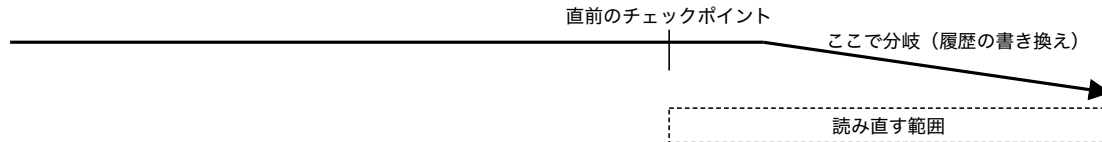
保存の単位はモデルの構造が決める。36 層の GDN はコンテキスト全体を固定サイズの状態に畳み込むので、ある位置までの状態は「その位置で写した状態」としてしか取り出せない。KV キャッシュのように後ろを切り詰めて任意の位置に戻すことはできない。そこで会話キャッシュはところどころの位置で GDN とインデクサの状態の写し (チェックポイント、1 個約 118 MiB) を取っておく。QSA の K/V はセッションに残るので、チェックポイントから再開した質問はそれより後ろだけを読めばよい [7, §What the engine already did]。

会話キャッシュのチェックポイントの位置（1 本の会話をトークン列として見る）



再開のしかた

新しいリクエストのトークン列（と画像）が、あるチェックポイントまで完全に一致すれば、そこから先だけを読む。



チェックポイント 1 つ = GDN の状態とインデクサの状態（約 118 MiB）。QSA 層の K/V はセッションに残るので、チェックポイントから再開すると、それより後ろの K/V だけを作り直せばよい。既定は最大 6 個を RAM に保持し、ルート以外は最近使われていない順に捨てる（`-prompt-cache-spread` で間隔を幾何級数的にできる）。`strata_prefix` で文書の終わりに置いたチェックポイントは pinned され、追い出されない。

図 12 会話キャッシュのチェックポイントの位置と再開のしかた。新しいリクエストがあるチェックポイントまで完全に一致すれば、そこから先だけを読む

図 12 はチェックポイントの位置を示す。既定では RAM に最大 6 個を持ち、新しいアシスタントのターンの開始時と、プロンプトの 16K トークンごとに取り。チェックポイントは、プロンプトがそのトークン列と画像でちょうど始まる時にだけ使われる。最も古いもの、実際には同じクライアントのすべてのチャットが共有するシステムプロンプトの終わり（ルート）は常に保持し、残りは最近使われていない順に入れ替わる。このため同じ prefix を共有する新しいチャットは、トークン 0 からではなくその後ろから読み始められる。システムプロンプトが 2,048 トークン以上ならその終わりに必ずルートを作る（エンジン 0.1.20）。長いシステムプロンプトとツール一覧を持つエージェントのクライアントに向けた配慮である [4, §Using it]。

再開は完全一致の prefix に限るので、原理的には読み直した場合と同じ状態になる。ただし再開したリクエストは末尾だけをデコードの経路で読み、prefill のバッチ経路と verify window は丸めが違うので、これはリクエストをまたいで出力が変わる要因の一つになる。バイト単位で同じ繰り返しが必要な測定では会話キャッシュを切る [4, §Speed]。

6.2 batch slot

サーバは既定では一度に 1 つのリクエストしか処理しない。opt-in の **batch slot** を使うと、エンジンは複数の会話を開いたまま保持し、それらを同じ verify window に載せて一緒にデコードする。各ウィンドウには会話ごとに 1 トークンずつが載るので、密な重み、shared expert、output head、そして複数の会話が共通して選んだ routed expert は、ウィンドウにつき 1 回だけ読めば全会話の分に使える [8]。

出力が単独実行と一致するのは、バッチの各行の演算が 1 トークンのウィンドウの演算と同じだからである。行ごとに自分のスロットの状態（GDN の状態、QSA の K/V、インデクサ）を読み、同じカーネルで同じ重みを掛ける。したがって greedy decoding では、バッチ内の各会話は単独で実行したときと同じトークンを生成し、8 本の同時会話でトークンごとに確かめられている。ただしこの一致は CPU カーネルの選び方、PCIe の取り分、expert cache の入れ替えという丸めを左右する三つの要因を固定したときのものである。既定の設定では、出力はいくらかのトークンの後で単独実行から分かれていく [8, §Exactness]。

効果は expert がどこにあるかで正反対になる。RTX 5070 12 GB・Q2_0・32K で 4 つのリクエストを同時に送ると、4 スロットでは 4 つ目の最初のトークンが 11.2 秒ではなく 1.8 秒で出たが、合計の速度は一度に 1 つずつ処理するより 11% 遅かった。スロットのセッション（それぞれ 0.56 GiB）が expert cache から取られ、expert の大半が CPU で動くからである。このため setup は 12 GB のカードを一度に 1 つのままにする。expert の大半を VRAM に保持する構成や layer split（7 節）では合計の速度が伸び、16 GB

のカード 4 枚 (PCIe Gen3) の layer split・IQ3_S では、1 リクエストの 120 tokens/s に対し 8 同時で合計 360 tokens/s だった [8, §Measured]。

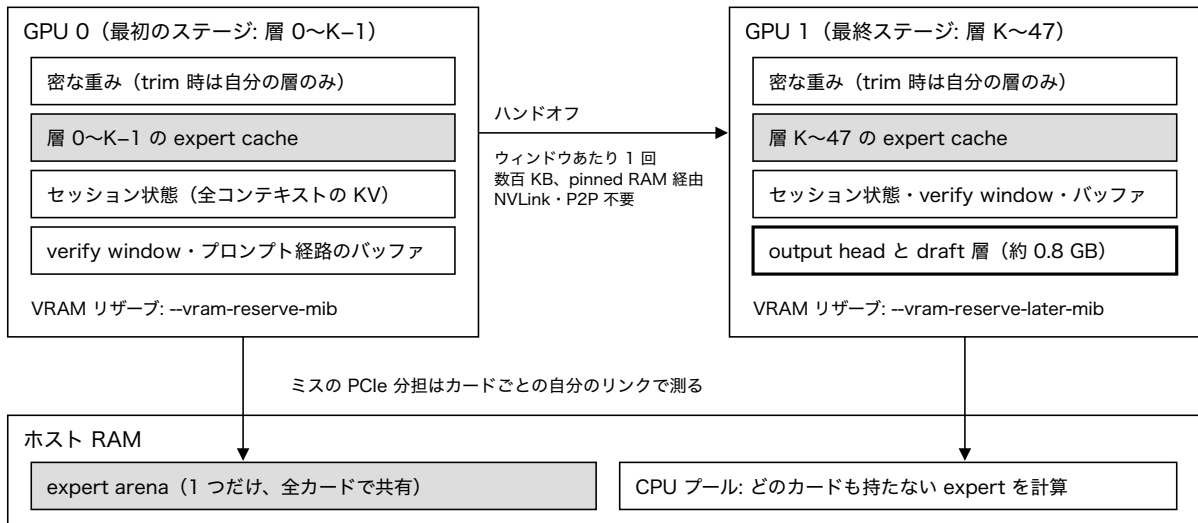
Strata のサーバは OpenAI 互換と Anthropic 互換の API と Web アプリを提供しており、アプリやコーディングエージェントからこれらの仕組みを使える [9] [2]。

7 複数 GPU と他のハードウェア

7.1 layer split

Strata は 1 台の PC に挿した複数の GPU で 1 つのモデルを動かせる。主な方法は **layer split** で、モデルの層を連続した範囲に分けて GPU ごとに 1 範囲ずつ受け持たせる。最初のカードが層 0 から K-1 を、次のカードが K 以降を受け持ち、最後のカードは output head と draft 層も持つ [10]。

layer split (2 枚、48 層を層 0~K-1 と K~47 に分ける) で各カードとホスト RAM が持つもの



プロンプトはチャンク単位でカードを順に流れ、後段がチャンク c を読む間に前段はチャンク c+1 を読む。

図 13 2 枚の layer split で各カードとホスト RAM が持つもの。各カードは自分の層の expert cache とセッション状態を持ち、expert の arena はホスト RAM に 1 つだけ置いて全カードで共有する

図 13 のように、各カードは自分の層についてだけ expert cache を持つので、2 枚のカードは 1 枚のおよそ 2 倍の expert を保持できる。速さの源はここにある。ホスト RAM の arena は 1 つだけで、どのカードも持たない expert は CPU が計算する [10] [10, §What each card holds]。これはパイプライン (レイヤー) 並列であって、テンソル並列ではない。トークンがあるカードから次へ渡るのは verify window あたり 1 回 (pinned RAM を経由する数百 KB) で、層ごとに通信が要るテンソル並列とは違う。そのため NVLink や P2P は要らず、x4 や x1 のスロットのカードでも動く [10]。

効果はプロンプトで最も大きい。各カードがチャンクの自分の層を読む間に、もう一方が次のチャンクを読むからである。RTX 5080 と RTX 3090 で Coder を動かした測定では、プロンプトが 18~20% 速くなり、デコードは速いカード単体と同等で、コードでは上回った [10, §Measured]。

分割の代償は、ステージを順に通るためにカードが交代で遊ぶことである。これを埋めるパイプライン化として、最後のカードがまだ検証している間に最初のカードに次のウィンドウを推測で始めさせる仕組みがある。推測が外れれば最初のカードの状態を戻して捨てるので、最後のカードは検証済みのウィンドウしか実行しない。AMD RDNA4 の 2 ステージ以上では既定オン、NVIDIA では opt-in である [10, §One conversation with both cards busy]。

7.2 バックエンド

ここまでの説明は NVIDIA の CUDA に沿っていたが、Strata にはほかに三つのバックエンドがある。表 4 にその位置づけをまとめる。

表 4 バックエンドの位置づけ

バックエンド	対象	位置づけ	主な違い
CUDA	NVIDIA GeForce RTX 20~50 シリーズ	既定。メンテナが測定	本論文の説明の基準。旧世代 (Volta、Pascal) は CUDA 12 で作る別のコミュニティビルド
HIP	AMD Radeon (RDNA2・RDNA3・RDNA4、別ビルドの gfx906)	CUDA と同じソースを AMD 向けにコンパイル。検証状態はアーキテクチャごとに異なる	整数内積を RDNA の dot 命令に、プロンプトの密な行列積を、任意で使える、世代とライブラリの版ごとに校正したテーブルで選ぶ
SYCL	Intel Arc (Battlemage、Alchemist)	experimental。コミュニティによる移植 (0.1.39)、Linux でソースからビルド	エンジンのツリーを oneAPI に移植したもの。別の CMake ターゲットで既定オフ
Metal	Apple Silicon の Mac (M1 以降、macOS 15 以降)	新しい (new)。実測は 3 台、ソースからビルド	カーネルだけを Metal に置き換える。統合メモリなので PCIe がなく、tier の考え方が変わる

[11] [12] [13] [14] [2]

表 4 で CUDA と HIP は同じエンジンである。HIP バックエンドは CUDA の実行時 API と BLAS 呼び出しを HIP と hipBLAS に写したもので、3 節から 6 節の設計はそのまま AMD でも働く。ただし後の版で既定になった最適化の多くは、バックエンドや GPU の世代ごとに既定かどうか異なり、バックエンド間で bit 単位で一致する答えを出すとは主張していない [11] [3, §Who does what]。

SYCL 移植は Strata 0.1.39 に experimental として入った。0.1.39 の時点ではメンテナは Intel GPU を持っておらず、Arc 実機での結果はすべてコミュニティから来ていた (0.1.40.2 以降はメンテナ側のテスト機での計測も加わった)。NVIDIA と AMD のエンジンには手が入っておらず、リリースに Intel 用のビルド済みエンジンはない [12] [12, §What has been run, and by whom]。

Mac の Metal バックエンドでは、tier の前提そのものが変わる。PC 版は、VRAM が小さくて速く、RAM が大きくて PCIe の向こう側にあるという非対称を前提にしていた。Apple Silicon では GPU がダイ上にあり CPU と同じ統合メモリを使うので、PCIe のリンクは存在しない。「VRAM に何を置くか」という問題は「統合メモリにモデル全体が載るか、載らないなら何を SSD から読むか」に置き換わり、メモリ配置は全体を常駐させるか、よく使う expert だけを統合メモリの arena に置いて残りを SSD から読むかの二通りになる [13] [13, §What you need]。

8 評価

8.1 測定の条件

公開されている主要な数値は 1 台の基準機で測られている。GPU は RTX 5070 (12 GB、PCIe 4.0 x16、ディスプレイ出力を兼ねる)、CPU は Ryzen 5 7600 (6 コア、AVX-512)、RAM は 64 GB の DDR5-5200 (EXPO 無効)、ストレージは NVMe SSD、OS は Windows である [1] [4, §Speed]。RAM を定格の速度で動かしていないこと、モニタを GPU につないでいることは、どちらも基準機の数値を控えめにする条件である [1]。

ワークロードは長さごとに 1 本のコードエージェント用プロンプト (長いソースファイルに対するコードレビュー課題) で、それぞれ 256 トークンを生成し、MTP による投機的デコーディングを有効にする。各数値はプロンプトを冷えた状態から読むエンジンの 1 回の完走である。プロンプト長の 1K~262K は名目で、実際は 1,017~259,943 トークンである。プロンプト速度は読み込んだトークン数を読み込み時間

で割ったもの、生成速度は生成したトークン数を生成時間で割ったもので、投機的デコーディングを含む [1] [4, §Speed]。

8.2 速度

表 2 は README が「普通のゲーミング PC」2 台の値として掲げる要約である。生成は 4K の短いチャット、プロンプトは 32K トークンの読み込みで、NVIDIA の Q2_0 はエンジン 0.1.36、それ以外は 0.1.26 の値である [2]。

表 2 2 台の PC での速度 (tokens/s. NVIDIA: RTX 5070 12 GB・Ryzen 5 7600・64 GB、AMD: RX 9070 XT 16 GB・Ryzen 9 3900X・47 GB。生成は 4K、プロンプトは 32K)

サイズ	RTX 5070 生成	RTX 5070 プロンプト	RX 9070 XT 生成	RX 9070 XT プロンプト
Q2_0	94	2,650	60	1,160
IQ2_XS	79	2,090	52	1,110
IQ3_XXS	62	1,750	—	—
IQ3_S	53	1,620	—	—
Coder	55	2,180	44	1,420

[2]

図 14 と図 15 は基準機での生成速度とプロンプト速度をコンテキスト長に対して描いたもので、エンジン 0.1.26 の測定である。262K の白抜きの点は再測定していない古い版の値 (生成は 0.1.14、プロンプトは 0.1.22 または 0.1.14) なので、ほかの点と同じ線の上で比べてはならない [4, §Speed]。

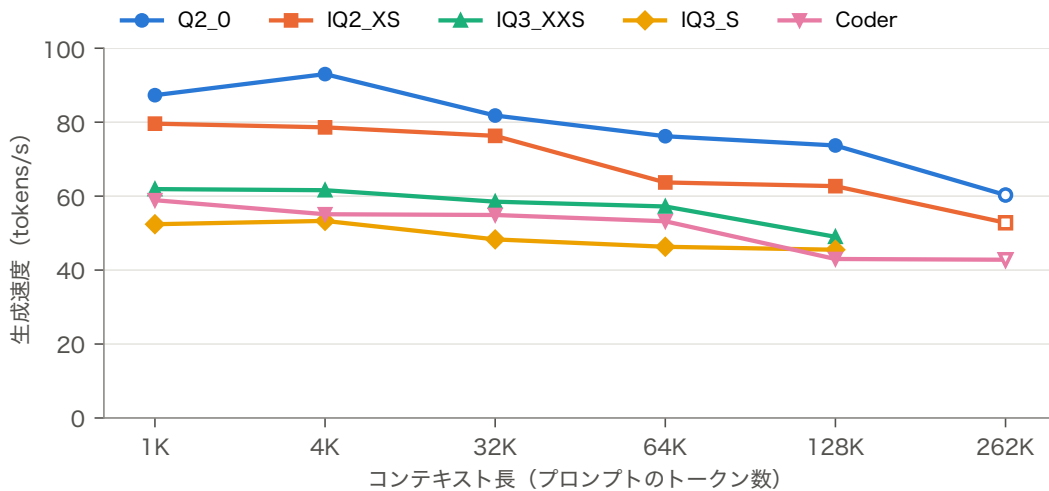


図 14 生成速度とコンテキスト長 (RTX 5070 12 GB、エンジン 0.1.26)。白抜きの点は古い版の値

図 14 で読み取ってほしいのは、生成速度がコンテキスト長に対してゆるやかにしか下がらないことである。Q2_0 は 4K の 93.0 tokens/s から 128K でも 73.7 tokens/s を保つ。下がる理由として論文は二つを挙げる。長いコンテキストの KV キャッシュが VRAM を取って expert cache が縮むことと、draft の acceptance rate が下がることである (5 節。0.1.26 では 64K 以上で KV ストリーミングが前者を和らげている) [1] [4, §Speed]。IQ2_XS と IQ3_XXS が Q2_0 より遅いのは、4 節で見たとおり CPU 側の expert 計算が長くなるからである [1]。

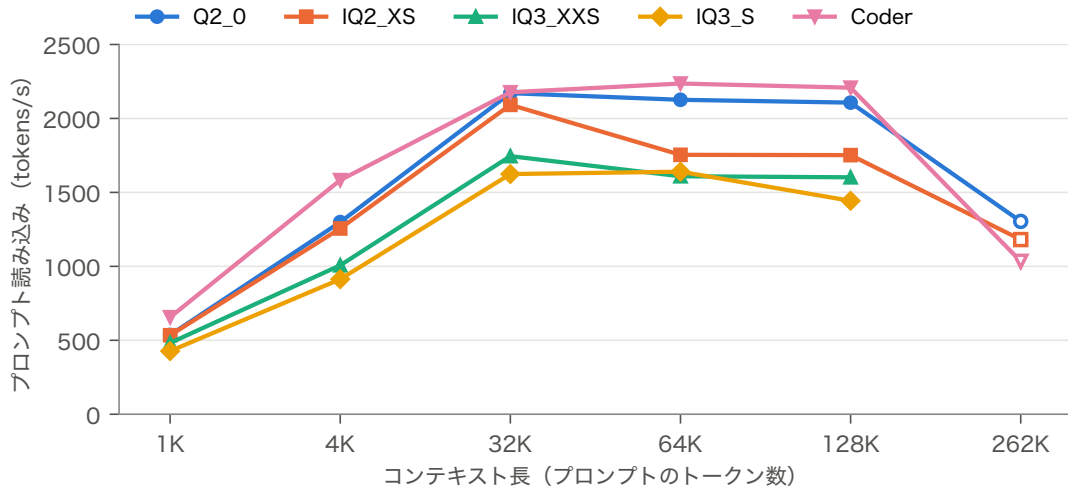


図 15 プロンプト読み込み速度とコンテキスト長 (RTX 5070 12 GB、エンジン 0.1.26)。白抜きの点は古い版の値

図 15 のプロンプト速度は 32K 前後で頭打ちになり、128K まで Q2_0 と Coder はほぼ平ら、ほかのファイルも 32K からの低下は 2 割以内である。長い側が平らなのは、QSA の参照する位置の数に上限があるからである [4, §Speed] [1]。0.1.26 の Q2_0 の 32K は 2,171 tokens/s で、表 2 の 2,650 tokens/s との差は 0.1.36 の融合 int8 カーネルによる [4, §Speed]。

8.3 品質

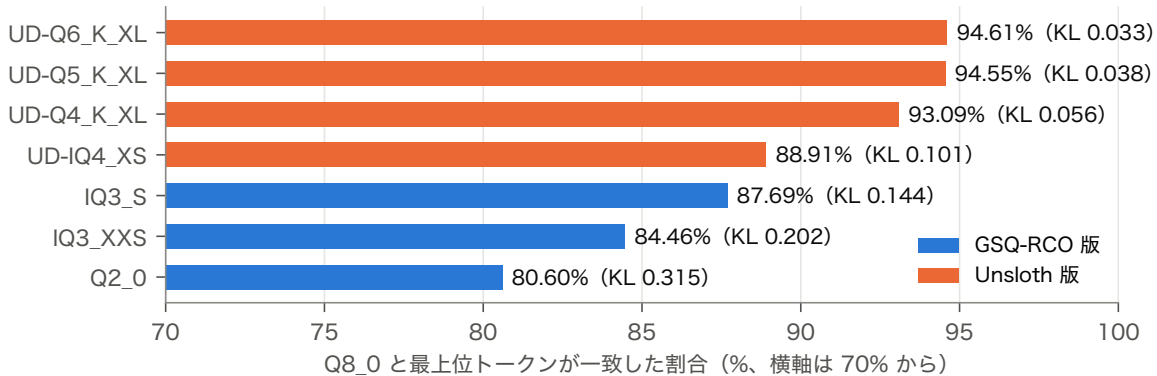


図 16 量子化ファイルごとの、Q8_0 と最上位トークンが一致した割合 (AMD の計測機、Q8_0 を基準にした評価基準)。括弧内は Q8_0 に対する KL

量子化ファイル同士の品質は Q8_0 を基準にした共通の**評価基準**で比べる。コード、ドキュメント、英語の文章、ドイツ語・フランス語・日本語・中国語、エージェントのやりとりからなる固定のテキスト集合 (5 × 1,000 トークン) を teacher forcing で読み、答えを揺らす要因 (expert の適応、PCIe の取り分、tail skip など) をすべて止める。指標は Q8_0 と最上位トークンが同じになる割合と、Q8_0 の上位 64 トークン上での KL である。Q8_0 を自分自身と比べると行がバイト単位で一致したので、この評価基準のノイズは 0 である [15, §Quality]。

図 16 はその結果で、AMD の機械 (EPYC 7452、R9700) で測られている。一致率は Q2_0 の 80.60% から、IQ3_XXS の 84.46%、IQ3_S の 87.69%、Unsloth の UD-IQ4_XS の 88.91% を経て、UD-Q6_K_XL の 94.61% まで上がる。UD-Q4_K_XL (93.09%) より上では差が小さくなり、UD-Q5_K_XL と UD-Q6_K_XL はほぼ同じである [15, §Quality]。ビット数を増やすと品質は上がるが、2 節で見たように 1 トークンあたりに読む expert の量も増え、より多くの RAM を必要とする。

8.4 他の GPU

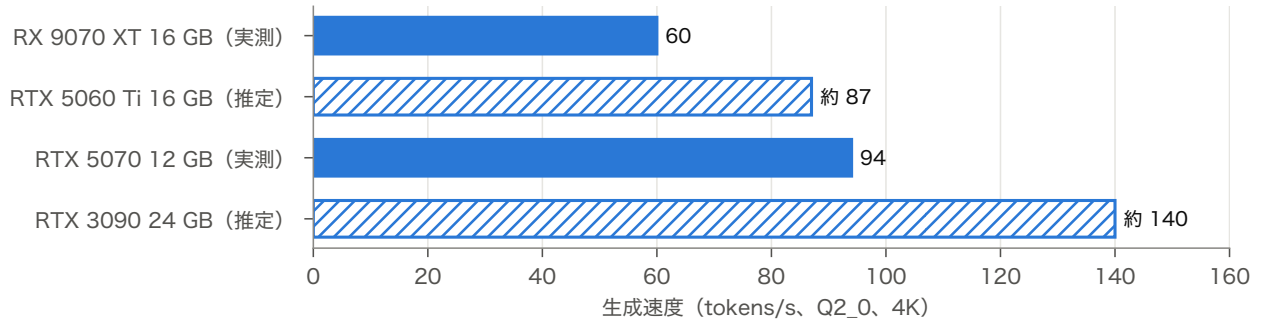


図 17 GPU 別の生成速度 (Q2_0, 4K)。RTX 5070 と RX 9070 XT は README の実測、RTX 5060 Ti と RTX 3090 は論文の推定。版と条件が揃っていない

図 17 は GPU 別の生成速度を並べたものだが、4 本の棒は同じ条件の比較ではない。RTX 5070 (94 tokens/s, 0.1.36) と RX 9070 XT (60 tokens/s, 0.1.26) は README の実測で、CPU と RAM も機械ごとに違う [2]。RTX 5060 Ti 16 GB (約 87) と RTX 3090 24 GB (約 140) は論文の推定であり、測定ではない。推定は RTX 5070 の時間の内訳から、GPU の部分をメモリ帯域の比で、CPU の部分を追加の VRAM に入る expert の増分で伸縮させたもので、測定されるまで $\pm 20\%$ として扱うよう明記されている [1] [4, §Other GPUs]。RTX 5060 Ti は帯域が小さい代わりに expert の置き場が広く、両者がほぼ打ち消し合う。RTX 3090 は帯域も VRAM も大きく、CPU の仕事の大部分がなくなる [1]。

8.5 正確さ

Strata の速度は出力を変えないことを前提にした速度である。本論文は原文の用法に合わせて二段階の一致を区別する。**bit 単位で一致**は同じ入力に対して出力のバイト列やロジットまで同じになることで、新しいスイッチを既定オフで足したときに既定の経路が前のリリースと変わらないことを確かめる基準である。**トークン列が一致**は生成されたトークンの列が同じになることで、投機やバッチの有無が答えを変えないことの基準である [9] [1]。表 3 に主な検証をまとめる。

表 3 正確さの検証

対象	比較	結果
投機的デコーディング	draft なしの greedy decoding	トークン列が一致 (誤った draft と巻き戻しを強制する試験を含む)
エンジン全体	llama.cpp の次トークン分布	平均 KL 0.022 (llama.cpp 自身の CPU ビルドと GPU ビルドの差は 0.058)
batch slot	単独実行	greedy でトークン列が一致 (8 本の同時会話、丸めを左右する要因を固定した場合)
KV ストリーミング	KV を VRAM に置く場合	注意が読む値が同じなので出力が同じ
tail skip (0.1.42 既定)	0.1.41 の挙動	KL 0.003~0.006 (同じエンジンを 2 回動かした揺れは約 0.0007)
融合 int8 プロンプトカーネル (0.1.36)	FP16 の参照	32K で KL 0.009 (従来のカーネルは 0.012)
UD-Q4_K_XL (実験的)	同じファイルの llama.cpp	同じトークンを選ぶ割合は短い greedy 回答で 97.5~99%、16K プロンプトの後で 90~91%

[1] [8, §Exactness] [4, §Speed] [4, §Missed experts in a verify window] [4, §Which model?]

論文は所見の一つとして「正確さは達成でき、こだわる価値がある」とまとめている [1]。ただし表 3 の一致には条件がある。既定の設定では、i-quant の CPU カーネルの丸め、サーバでの expert の入れ替えと会話キャッシュ、PCIe の取り分によって、温度 0 でも同じプロンプトの答えが実行ごとに変わりうる。原文は答えをバイト単位で揃えて繰り返すためのスイッチを記している (そのうち `--pcie-frac 0` はデコードが遅くなるので A/B 比較用) [4, §Speed]。また tail skip のように出力を変える既定には、KL とタスク評価の数値と、元の答えを bit 単位で戻す方法が添えられている [4, §Missed experts in a verify window]。

9 限界と今後

Strata の文書は測ったことと測っていないことを分けて書く方針を一貫して取っている。測定なしに主張しないことが文書の規則になっており、報告者にはテストしなかったことを書くよう求めている [9]。ここで挙げる限界の多くは現時点のものであり、版が上がれば変わりうる。表 5 に主なものをまとめる。

表 5 主な限界

分類	限界
設計	GPU と CPU がウィンドウの約半分ずつを要して互いを待つので、片側だけを速くしても改善は約 3 分の 1 が上限。投機的デコーディングの効果も 1.6~1.8 倍にとどまる
RAM の容量	通常モードでは全 expert を RAM に置くので、expert + 約 10 GB が必要。64 GB では IQ3_XXS と IQ3_S の 262K を測っていない。低 RAM モードは小さい GPU では大幅に遅い
OS とドライバ	Windows はページロックできる量に上限があり、WSL2 では pinned が約 1 GB に限られて KV ストリーミングが使えない
GPU と CPU	既製のエンジンは RTX 20~50 と AVX2 を持つ CPU が対象。旧世代の GPU と AVX2 のない CPU は実験的な経路
モデル	一つのモデルのためのエンジン。Coder はコード以外と英語以外の言語で弱く、Unsloth のファイルはサンプリングありのデコードと長い会話が未検証
機能の組み合わせ	多くの opt-in が特定の構成でだけ動く。たとえば batch slot は既定では MTP draft を運ばず、繰り返しペナルティを適用しない
未検証の領域	メンテナが持つ機械以外のプラットフォームの組み合わせ、3 枚以上の GPU、長時間の連続運転、答えの品質のベンチマーク（とくにサンプリングあり、長い会話、画像）

[1] [6, \$Will it fit?] [4, \$Speed] [4, \$Linux] [16, \$What you need] [16, \$Older CPUs] [6] [17, \$Scope and validation] [8, \$Limits] [18]

最も根本的な限界は設計そのものにある。VRAM に収まらない expert を CPU が同時に計算するからこそ 12 GB のカードで 1,250 億パラメータのモデルが動くのであり、その代わりにデコード速度は GPU の密な重みの読み出しと CPU の expert 計算でほぼ決まる。どちらの限界がどこで現れるかは、RAM の世代と量子化ファイルで変わる。DDR4 の機械では Q2_0 の CPU 側が約 2 倍の時間を要し、i-quant は損が小さい。いずれの場合も大きな GPU キャッシュが最良の補償になる [1]。

論文は今後の方向として、リクエスト間で会話の状態を保つこと、層の中ではなく層をまたいで重ねること（次の層の expert を現在の隠れ状態から予測して router より先にコピーを始める）、CPU での i-quant の演算を安くすること、速いプロンプト経路、温度サンプリング、無料でできるハードウェアの設定（RAM を定格の速度で動かす、モニタをマザーボードにつなぐ、Linux を使う）を挙げた [1]。このうち会話状態の保存（6 節）、プロンプト経路の高速化（5 節）、サンプリング時の draft の扱いは後の版で形を変えて実装されている。層をまたぐ予測については、後の版にあるのは主に、ファイルから読む低 RAM モードで予測した expert のページを先に要求する仕組み（計算する expert は変えない）と、予測の精度を測る計測専用の機能である。予測した expert を VRAM のスロットへ先にコピーする opt-in (STRATA_FS_SLOTS) も試されたが、測ったどの機械でもキャッシュが小さくなる損が上回った [4, \$Speed] [4, \$More opt-in switches measured for 0.1.41]。

未検証の領域は、メンテナが持たないハードウェアに集中している。メンテナは大きさの違う NVIDIA カード 2 枚の組、3~4 枚の NVIDIA GPU、Pascal 世代のカード、RAM 32 GB 以下の Linux ホスト、1 時間以上の連続運転での測定を求めている [18]。

10 関連研究と結論

MoE 層の一部を CPU で計算する手法は、Eliseev と Mazur、Fiddler、KTransformers が研究している。MoE-Infinity と Pre-gated MoE は expert のキャッシュと次の層の expert の予測を扱い、PowerInfer は一般向け GPU で活性化のよく使われる部分とそうでない部分の偏りを利用する [1]。一つのモデルのために一つのエンジンを書くという考え方は Splash、ninfer、HyperQwen に由来する [1] [3, \$Credits]。投機的デコーディングは Leviathan らに、マルチトークン予測ヘッドは Gloeckle らに、ルックアップテーブルによ

る CPU カーネルは T-MAC に、量子化 GPU 行列カーネルは Marlin に、ページ化された注意機構は vLLM に負っている。i-quant 形式とその内積、ggml の CPU バックエンドは llama.cpp のもので、Strata はその一部をそのまま使っている [1] [4, §Credits and licenses]。

Strata は一つのモデルの形に合わせてエンジンを書けば、VRAM に収まらない MoE モデルでも普通の PC の部品をすべて同時に働かせられることを示した。鍵は毎トークン読むバイト数で各部分の置き場所を決め、VRAM がない expert を置き場所の近くで CPU が計算し、MTP 層で推測して 1 回のパスで検証することにある。RTX 5070 12 GB・Q2_0 では 4K の生成が 94 tokens/s、32K プロンプトの読み込みが 2,650 tokens/s に達し (エンジン 0.1.36)、投機的デコーディングの出力は、i-quant の例外を除いて greedy decoding とトークン単位で一致する [2] [1]。一方でエンジンは釣り合っており、どれか一つの部品だけを強化しても速度は 2 倍にならない [1]。

参考文献

本書の一次資料はすべて GitHub リポジトリ Niko1221/Strata (commit 477ac13、2026-10-16) に含まれる文書である。番号は本文で最初に引用された順。§ に続く英語は原文の節見出しを示す。

- [1] Running a 125-Billion-Parameter Model on a Normal PC (Strata 英語論文) . docs/paper/Strata-Paper.pdf
- [2] Strata README. README.md
- [3] How does Strata work?. docs/HOW_IT_WORKS.md
- [4] Strata - the details. docs/DETAILS.md
- [5] QSA_ROWS: the decode window's QSA block scores with the queries in registers (STRATA_QSA_ROWS=1, opt-in). docs/QSAROWS.md
- [6] Which model? Sizes, versions and what fits. docs/MODELS.md
- [7] One long document, many questions (0.1.40.2, opt-in). docs/RESEARCH_RUNS.md
- [8] Several requests at once (batch slots). docs/BATCHING.md
- [9] AGENTS.md (開発者・エージェント向け案内) . AGENTS.md
- [10] Strata on two or more GPUs (layer split). docs/MULTI_GPU.md
- [11] AMD Radeon: the HIP backend (gfx1100, gfx1101, gfx1200, gfx1201, gfx1030). docs/AMD_HIP.md
- [12] Intel Arc (experimental). docs/INTEL_ARC.md
- [13] Strata on a Mac (Apple Silicon, Metal). docs/MAC.md
- [14] Older GPUs (experimental). docs/OLDER_GPUS.md
- [15] Unsloth's K-quants and Q8_0: formats, kernels and quality. docs/QUANTS.md
- [16] Installing Strata. docs/INSTALL.md
- [17] Unsloth UD-Q4_K_XL. docs/UNSLOTH_Q4.md
- [18] Test requests and hardware wanted. docs/TEST_REQUESTS.md