NetApp による KV キャッシュオフロードNetApp

NetApp artificial intelligence solutions

NetApp
August 20, 2026

目次

NetApp による KV キャッシュ オフロードNetApp

企業向けAI投資の第一波は、モデルのトレーニングとファインチューニングによる能力構築に重点が置かれており、大規模な運用には焦点が当てられていなかった。組織が実験から本番環境へと移行するにつれて、重心はリアルタイム推論へとシフトします。すなわち、ユーザーが継続的にやり取りする検索アシスタント、チャットボット、コパイロット、およびエージェントワークフローです。このような環境では、GPUの使用率が急速に上昇するのは、すべてのリクエストで完全なトレーニングパスが実行されるからではなく、フォローアップの質問、会話の新たなターン、および同時接続ユーザーが増えるたびに推論スタックに負荷がかかるためです。

はじめに

すぐに一つのパターンが見えてくる。GPUは驚くほど多くの反復作業を行っており、既に処理したトークンに対してアテンションを再計算しているのだ。その繰り返しは、チューニングの問題だけではなく、メモリアーキテクチャの問題でもある。業界の対応策は、KV キャッシュを中心とした階層化ストレージ アーキテクチャに基づくメモリ戦略である。

KVキャッシュとは何ですか？

簡単に言うと、KV（キーバリュー）キャッシングとは、大規模言語モデル（LLM）の推論を最適化するために使用される技術であり、以前に計算された値をKVキャッシュに保存することで、生成される新しいトークンごとにこれらの値を再計算する必要がなくなります。これを行わない場合、再計算が必要になります。

注：より詳細な技術概要については、"Hugging Face KV キャッシングの概要"を参照してください。

KV キャッシュ オフロードとは何ですか？

ほとんどの推論エンジンは、デフォルトでKVキャッシュをGPUメモリに格納します。需要が増加するまでは、それで十分です。KVキャッシュのサイズは、バッチサイズ（同時に処理されるプロンプトの数）とシーケンス長（各会話または生成ストリームの長さ）に比例してスケールします。コンテキストウィンドウが拡大し、複数ターンのチャットが一般的になり、推論サービスを利用するユーザーやエージェントが増えるにつれて、KVキャッシュの要件は利用可能なGPUメモリをすぐに超えてしまう可能性があります。そうなると、有用なキャッシュエントリが削除されることが多く、エンジンは既に処理したトークンに対するアテンションを再計算する必要が生じます。

KVキャッシュのオフロードは、以前に処理されたプロンプトのKVキャッシュを、GPUやアクセラレータのメモリ以外の外部ターゲット（システムRAM、ローカルディスク、共有ストレージなど）に移動することで、この制約に対処します。オフロードされたエントリは、容量が許す限り保持され、GPUメモリがいっぱいになったときに破棄されるのではなく、類似のプレフィックスまたは後続のリクエストが到着したときに再度参照されます。事実上、これらのオフロードターゲットはGPUメモリの階層型スワップ領域として機能します。VRAMよりも低速ですが、容量が大きく耐久性も高いため、繰り返しプリフィル処理を行うことなく、システムが維持できる会話コンテキストと同時ワークロードの量を拡張できます。

共有ストレージ階層の重要性

KVキャッシュのオフロードは、単一の推論エンジンがGPUメモリの容量を超えてしまった場合に既に効果を発揮します。キャッシュブロックをCPUのRAMに移動することで、1台のサーバーの処理能力を拡張できま

す。これは有用ですが、本番環境における問題の一部しか解決しません。実際の推論プラットフォームは、単一のサービングエンジンインスタンスとして動作することはほとんどありません。これらはロードバランサーの背後で動作し、水平方向に拡張可能で、多数の同時ユーザーとエージェントに対応します。その世界では、共有ストレージこそがKVキャッシュをローカルな最適化からプラットフォーム機能へと変えるものです。

- 共有ストレージにより、サービスインスタンス間での再利用が可能になります 共有ストレージの主な利点の1つは、複数のインスタンスやノード間で同じファイルやオブジェクトにアクセスできることです。これは特に、ロードバランサーの背後に2つ以上のサービングエンジンインスタンスを大規模に配置し、それぞれがKVキャッシュブロックを同じ共有バケットまたはNASボリュームにオフロードするように構成した場合に当てはまります。例えば、あるインスタンスがプロンプトプレフィックスを処理して結果として得られるキャッシュブロックをオフロードすると、別のインスタンスは、同じプリフィル処理を再計算する代わりに、キャッシュ ヒット時にそれらのブロックをロードできます。これは重要な点です。なぜなら、本番環境のトラフィックは分散しており、ユーザーの最初の質問はインスタンスAに届き、その後の質問や同様のシステムプロンプトを持つ別のユーザーはインスタンスBに届く可能性があるからです。共有ストレージがない場合、各インスタンスは独自のプライベートキャッシュ環境を維持します。
- **CPU**メモリだけではマルチインスタンス展開には不十分です CPU層は高速ですが、各サービスエンジンプロセスにローカルです。ピアツーピアチャネルを実装しない限り、あるインスタンスが別のインスタンスのローカルCPU RAMにオフロードしたKVキャッシュエントリにアクセスすることはできません。ピアツーピアチャネルを実装すると、レイテンシが増加し、運用上の複雑さも増します。共有ストレージは、この障壁を取り除きます。CPU RAMは各ノード上の高速ステージング層として依然として価値がありますが、共有S3またはNASは、複数のエンジンが読み書きできる共通のメモリプレーンを提供します。もはやGPUを単独でスケーリングするのではなく、共有キャッシュインフラストラクチャを使用してスケーリングします。
- **3**層設計戦略の実現 + 望ましいアーキテクチャは以下を組み合わせたものです：
  - **GPU**メモリ — 処理中のリクエスト用のアクティブキャッシュ。
  - **CPU**メモリ — プロンプトがキューに入ると、高速にステージングされます。
  - 共有ストレージ — 耐久性があり、大容量で、共有可能なバッキングストア。

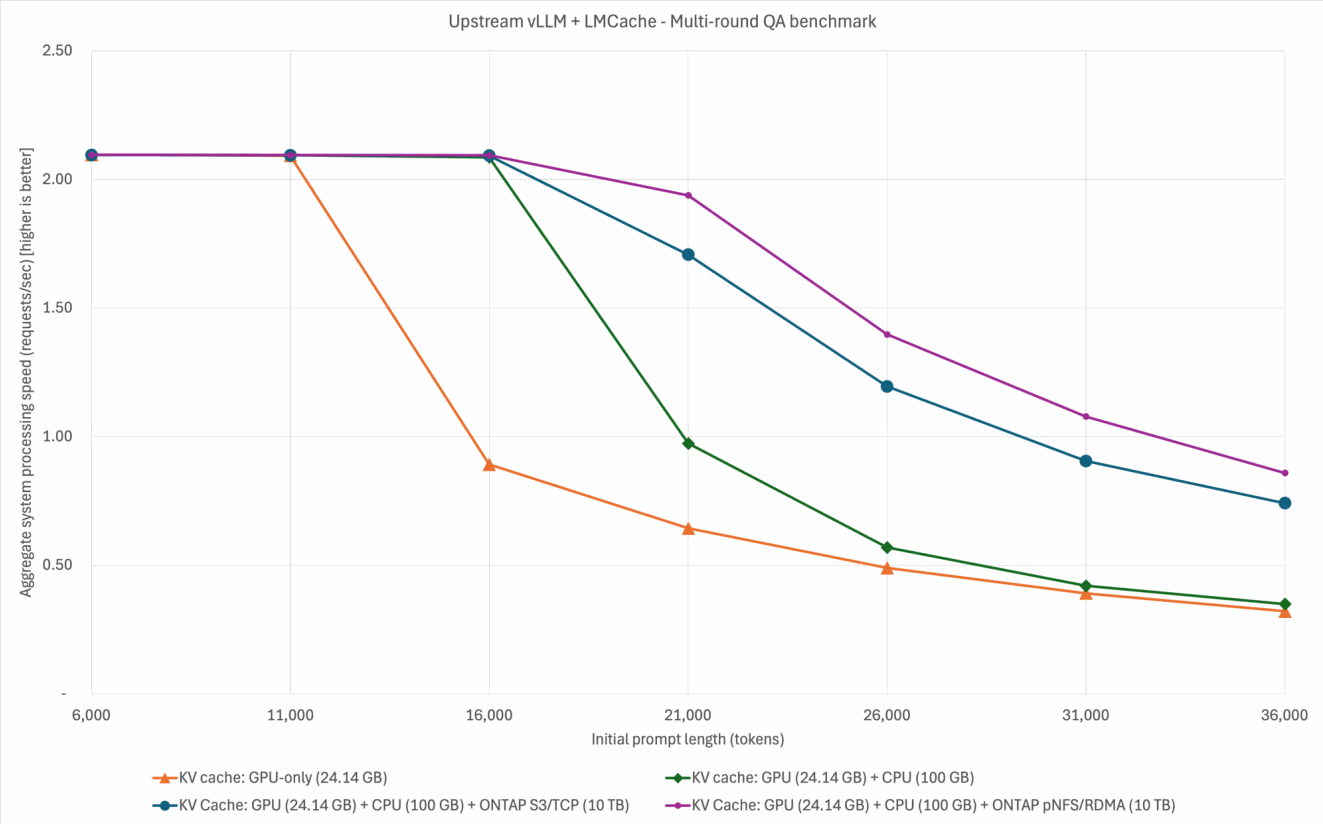
  このアーキテクチャでは、新しいリクエストが到着すると、KV キャッシュ ブロックを共有層から CPU メモリにステージングできるため、GPU はアクティブな処理に集中しつつ、ストレージ上に永続的なキャッシュを保持することができます。

- 推論の経済性、速度だけではない 生産の観点から見ると、共有ストレージの重要性はレイテンシだけではありません。これは、メモリ、並行処理、およびコストの制御に関するものです。vLLM のようなサービスエンジンは、1 つのノード内で KV キャッシュを効率的に管理します。LMCache のような KV キャッシュオフロードフレームワークは、KV キャッシュを CPU メモリとストレージ間で移動可能な、ポータブルで共有可能な資産へと変換します。NetApp ONTAP や StorageGRID などの共有ストレージプラットフォームは、複数のインスタンス間での共有を実用的にする耐久性のある階層を提供します。LMCache が共有ストレージに接続されている場合、複数の vLLM インスタンスが同じオフロードされた KV キャッシュエントリにアクセスできるため、同時実行数の増加に伴いスループットが向上し、冗長な計算が削減されます。これにより、チャットボット、検索アシスタント、エージェント、および複数ターンのスレッド、共有システムプロンプト、または繰り返し発生するコンテキストパターンを持つあらゆるワークロードに直接関連する、繰り返し行われるプリフィルによる GPU サイクルの無駄を削減できます。
- 推論パフォーマンスと**GPU**投資を向上 + このベンチマークは、会話の長さと同時負荷の増加に伴う推論パフォーマンスを比較します。KV キャッシュが GPU メモリのみに保持されている場合、利用可能な余裕はすぐに枯渇し、スループットが低下します（オレンジ色の線）。3 層構造（アクティブな処理には GPU、高速なステージングには CPU、耐久性のある共有可能なキャッシュには共有ストレージ）を採用することで、このプラットフォームはより多くのセッションを維持し、同様の急激なパフォーマンス低下を回避します。実際には、階層化によって繰り返し行われるプリフィル計算を削減することで、同じ GPU からより多くの処理を引き出すことができます。

◦ 簡単に言うと：

- **GPU tier** — 進行中のアクティブな処理を担当します。
- **CPU** 層 — 最近使用したキャッシュをサービス エンジンの近くに保持します。
- 共有ストレージ層 — サーバー間でキャッシュを保持し、新しいセッションのためにGPU/CPUメモリを解放します。

図1 - マルチラウンド推論ベンチマーク



ベンチマークによると、CPUメモリに加えて共有ストレージを追加してもパフォーマンスは低下せず、スループットの低下が現れるまでの動作可能範囲が拡大します。階層化によって推論用のメモリ層が効果的に追加されるため、セッション数、コンテキスト長、または同時実行数が増加するたびにGPUを追加する必要性が軽減されます。

- エンタープライズ向け：標準プロトコル、独自のクライアントなし + 共有ストレージは重要ですが、すべての共有ストレージが同じように作られているわけではありません。NetAppは、業界標準のプロトコルとツールを使用したKVキャッシュオフロードをサポートします。

  ◦ **StorageGRID S3**

  ◦ **ONTAP NAS** 向け **NFS / pNFS**

カスタムの独自推論クライアントは不要です。運用チームは、他のデータサービスで既に運用しているストレージプロトコルをそのまま使用でき、使い慣れたデータ保護、セキュリティ、マルチクラウドモビリティを基盤として活用できます。

KV キャッシュ オフロードの導入

共有ストレージはKVキャッシュの容量を拡張し、複数のサービスインスタンス間での再利用を可能にします。その層を本番環境で使用するには、推論エンジンとKVキャッシュオフロードフレームワークを設定する必要があります。以下のセクションでは、一般的なツールオプションを使用してこのスタックを実装する方法について説明します。

vLLM

vLLM は人気のあるハイパフォーマンスなオープンソース LLM 推論エンジンです。このソリューションでは、エンジンがユーザーからのリクエストを受信し、レスポンスを生成し、推論中に GPU メモリ内の KV キャッシュを管理します。vLLM はプラグイン可能で、使用するオフロードフレームワークを選択できます。以下のセクションでは、一般的なオフロードフレームワークを使用して vLLM ベースのサービングスタックを実装する方法について説明します。

LMCache（インプロセスモード）

LMCacheは、人気の高いオープンソースのKVキャッシュオフロードフレームワークです。このセクションでは、LMCacheを使用してKVキャッシュをオフロードするvLLMベースのサービングスタックを実装する方法について説明します。この実装では、LMCache は vLLM と連携して KV キャッシュを GPU メモリから CPU RAM、ローカルディスク、共有ストレージ （StorageGRID S3 または ONTAP NAS マウントなど）などのより大きな階層に移動します。

デフォルト設定では、vLLMはKVキャッシュをGPUメモリのみに保持します。システム負荷が増加すると、それがボトルネックとなります。このソリューションでは、vLLMはLMCacheとペアになっており、vLLMがモデルを提供し、LMCacheはCPUと共有NetAppストレージ間でキャッシュをオフロードして再利用します。LMCacheはコネクタを介してvLLMに接続します。起動時にvLLMの—kv-transfer-configフラグを使用して有効にすると、vLLMとLMCacheはキャッシュをストレージに保存し、キャッシュ ヒット時にそれを再ロードします。

インプロセスモードは、vLLMでLMCacheを実行する本来の方法です。インプロセスモードを使用する場合、LMCacheはvLLMプロセス内で実行されます。LMCacheの後のバージョンではマルチプロセスモードが追加され、これは現在、より優れた機能サポートとパフォーマンスを実現するための推奨手法となっています。このセクションでは、現在のLMCacheドキュメントで非推奨とされているインプロセスモードについて説明します。新規導入の場合は、マルチプロセスモードの使用を検討してください。

このデプロイメントでは、以下の操作を行います。

- vLLMとLMCacheをインストールする
- LMCacheコネクタを有効にしてvLLMを起動する
- vLLMルーターの背後に、2つのvLLMインスタンス（それぞれ別のGPU上で動作）を配置し、両方のインスタンスが同じ共有ストレージバックエンドを使用する3層構成（GPU + CPU + 共有ストレージ）をデプロイします。

前提条件

- Linux GPU サーバー（NVIDIA ドライバー（CUDA を含む）および少なくとも 1 つの GPU（完全な 2 インスタンス＋ルーター構成の場合は 2 つの GPU または複数のサーバー））
- Pythonとuvがインストールされています
- Docker と NVIDIA Container Toolkit がインストールされていること（手順 4 の vLLM ルーターに必要）

- モデル（例：Hugging Face）をダウンロードし、ストレージエンドポイントに到達するためのネットワークアクセス

StorageGRID S3バックエンド

- KVキャッシュ用に作成されたS3バケット
- バケットの読み取り/書き込み認証情報
- GPUサーバーからS3エンドポイントへのネットワーク接続
- 仮想ホスト型S3リクエストが有効

ONTAP pNFS/NFS バックエンド

- ONTAP ボリュームが GPU サーバーにエクスポートされました
- vLLMを実行する*各*サーバー上にマウントパス（例：/mnt/kvcache）が作成されます。ハイパフォーマンス pNFS over RDMA (RoCE) のマウントコマンドの例：

```
mount -o
vers=4.1,proto=rdma,trunkdiscovery,write=eager,rsize=262144,wsize=262144
my_storage.my_company.net:/kvcache /mnt/kvcache
```

[[1-install-vllm-and-lmcache]]
=== 1.vLLM と LMCache をインストールする

仮想環境を作成し、vLLM と LMCache をインストールします。詳細については、LMCache "インストール手順書" を参照してください。お客様の CUDA バージョンに応じた正しいインストールパスに従うことが非常に重要です。

```
uv venv .venv
source .venv/bin/activate
uv pip install --upgrade lmcache vllm
```

この時点で、推論エンジン（vLLM）とキャッシュ レイヤー（LMCache）が揃っています。

[[2-configure-lmcache]]
=== 2.LMCacheの設定

vLLMはそれ自体ではKVキャッシュのオフロードを行いません。LMCacheは、vLLMのKV転送コネクタを介して有効にします。CPUと共有ストレージ層の設定を定義するYAMLファイル（lmcache-config.yaml）を作成します。LMCacheは、vLLM推論の起動シーケンスの一部として、このYAMLファイルを読み込みます。

サンプル スニペット：**StorageGRID S3** 共有ティア

```
local_cpu: True
max_local_cpu_size: 100
save_decode_cache: True
```

```
remote_url: "s3://<bucket>.<storagegrid_s3_endpoint_fqdn>"
remote_serde: "naive"
extra_config:
  s3_num_io_threads: 320
  s3_prefer_http2: False
  s3_enable_s3express: False
  save_chunk_meta: False
  disable_tls: <True/False>
  aws_access_key_id: "<storagegrid_s3_access_key>"
  aws_secret_access_key: "<storagegrid_s3_secret_key>"
  s3_region: "us-east-1"
```

注：バケット名、エンドポイント、認証情報、 `disable_tls`をお客様の環境に合わせて置き換えてください。

サンプル スニペット：**ONTAP pNFS/NAS** 共有層

```
local_cpu: true
max_local_cpu_size: 100
save_decode_cache: true
remote_storage_plugins: ["fs"]
extra_config:
  remote_storage_plugin.fs.base_path: /mnt/kvcache
  save_chunk_meta: false
```

ステップ3の前に、サイトのNFS/pNFS手順を使用してONTAPエクスポートをマウントしてください。

[[3-run-two-vllm-instances-shared-cache-across-servers]]
=== 3.vLLMインスタンスを2つ実行する（サーバー間でキャッシュを共有する）

両方のインスタンスで共通の環境変数を設定します：

```
export LMCACHE_CONFIG_FILE=$(pwd)/lmcache-config.yaml
export PYTHONHASHSEED=0
export VLLM_API_KEY='<shared_api_key>'
```

インスタンス **1** （GPU 0、ポート 8001）：

```
export CUDA_VISIBLE_DEVICES=0
vllm serve Qwen/Qwen3-8B \
  --tensor-parallel-size 1 \
  --kv-transfer-config
'{"kv_connector":"LMCacheConnectorV1","kv_role":"kv_both"}' \
  --port 8001
```

インスタンス**2**（GPU 1、ポート8002 — 別端末）：

```
export CUDA_VISIBLE_DEVICES=1
vllm serve Qwen/Qwen3-8B \
  --tensor-parallel-size 1 \
  --kv-transfer-config
'{"kv_connector":"LMCacheConnectorV1","kv_role":"kv_both"}' \
  --port 8002
```

同じ設定ファイルとハッシュシードを使用することで、両方のインスタンスがキャッシュブロックの識別情報を共有し、共有ストレージから互いのオフロードされたデータを読み取ることができます。両方のインスタンスで `VLLM_API_KEY`を設定してください。ルーターはこの値を `OPENAI_API_KEY`として渡すことで、ルーティングされたリクエストがインスタンスによって受け入れられるようになります。

注：単一の GPU インスタンスを使用して、階層化ストレージ アーキテクチャを実装することもできます。その場合は、手順 4 をスキップしてください。

[[4-deploy-the-vllm-router-single-entry-point]]
=== 4.vLLMルーター（単一エントリポイント）をデプロイする

両方のインスタンスが正常に動作したら、クライアントが OpenAI 互換の単一の URL を使用するようにルーターをデプロイします。

```
docker run -d --rm --name vllm-router \
  --network host \
  -e OPENAI_API_KEY="${VLLM_API_KEY}" \
  ghcr.io/vllm-project/production-stack/router:latest \
  --port 8000 \
  --service-discovery static \
  --static-backends "http://localhost:8001,http://localhost:8002" \
  --static-models "Qwen/Qwen3-8B,Qwen/Qwen3-8B" \
  --routing-logic roundrobin
```

トラフィックを http://localhost:8000/v1 に送信します。ルーターはリクエストを分散し、LMCache と共有ストレージにより、単一の GPU 内だけでなく、インスタンス間でキャッシュを再利用できます。

例えば、curl を使用してルーターにリクエストを送信するには、次のようにします（<shared_api_key> をそれぞれの API キーに置き換えてください）：

```
curl -s http://localhost:8000/v1/chat/completions \
  -H "Content-Type: application/json" \
  -H 'Authorization: Bearer <shared_api_key>' \
  -d '{
    "model": "Qwen/Qwen3-8B",
    "messages": [{"role": "user", "content": "What is 2+2?"}]
```

```
  }' | jq .
```

[[5-validate-that-tiering-is-working]]
=== 5.階層化が正しく機能していることを確認する

- 両方のvLLMプロセスは8001番ポートと8002番ポートで待機し、ルーターは8000番ポートで応答します。
- 長いプレフィックスを付けたリクエストをルーター経由で送信します。
- オブジェクトまたはファイルが共有ストレージ（S3バケットまたは ls -ltr /mnt/kvcache）に表示されていることを確認します。
- 同様のリクエストを再度送信してください。2回目のリクエストでは、ログにプリフィルまたはキャッシュ ヒットの動作がより速く表示されるはずです。
- ルーター経由で再度処理を行い、トラフィックが別のインスタンスに到達するようにします。

まとめ

階層化ストレージ アーキテクチャ（tiered KV cache architecture）を導入すると、KVキャッシュがGPUメモリからCPUメモリおよび共有NetAppストレージに移動するため、繰り返しのプリフィルでGPUサイクルを無駄にすることなく、ユーザー数やコンテキスト長に応じて推論を拡張できます。共有ストレージは、キャッシュを複数のサービング インスタンス間で再利用可能なインフラへと変え、既存のGPU投資からより多くの価値を引き出すのに役立ちます。

NetAppストレージは、このティアに最適です。なぜなら、S3およびNFS/pNFSを介した標準アクセス、複数のサービングエンジンインスタンスが同時に使用できる共有キャッシュ、そして耐久性、保護、ガバナンスのためにすでに利用しているエンタープライズデータサービスなど、本番環境の推論に必要な機能をすべて備えているからです。独自のクライアントは必要ありません。KVキャッシュオフロードフレームワークは、使い慣れたプロトコルを使用して StorageGRID S3 または ONTAP NAS マウントに接続します。

NetAppは、推論の実行方法やチームのデータ管理方法を変更することなく、KVキャッシュのオフロードに対応したなじみのあるストレージ基盤を提供します。

著作権に関する情報

商標に関する情報