NetApp을 통한 KV 캐시 오프로딩

NetApp artificial intelligence solutions

NetApp
August 20, 2026

목차

NetApp을 통한 KV 캐시 오프로딩

초기 기업 AI 투자 물결은 모델 학습 및 미세 조정을 통한 역량 구축에 집중되었지만, 이를 대규모로 활용하는 데는 초점을 맞추지 않았습니다. 조직이 실험 단계에서 실제 운영 단계로 전환함에 따라, 검색 도우미, 챗봇, 코파일럿, 사용자가 지속적으로 상호 작용하는 에이전트 워크플로 등 실시간 추론이 중심이 되고 있습니다. 이러한 환경에서 GPU 사용률이 급격히 증가하는 이유는 모든 요청이 전체 학습 과정을 거치기 때문이 아니라, 후속 질문, 대화의 새로운 전환, 그리고 동시 접속 사용자 모두가 추론 스택에 부하를 추가하기 때문입니다.

소개

곧 한 가지 패턴이 눈에 띕니다. GPU가 이미 처리한 토큰에 대해 어텐션 연산을 다시 수행하는 등 놀라울 정도로 반복적인 작업을 많이 한다는 것입니다. 이러한 반복은 단순히 튜닝 문제만이 아니라 메모리 아키텍처 문제입니다. 업계에서는 이에 대한 대응으로 KV 캐시를 중심으로 한 계층형 메모리 전략을 사용하고 있습니다.

KV 캐시란 무엇입니까?

간단히 말해, KV(키-값) 캐싱은 이전에 계산된 값을 KV 캐시에 저장하여 새 토큰이 생성될 때마다 해당 값을 다시 계산할 필요가 없도록 함으로써 대규모 언어 모델(LLM) 추론을 최적화하는 데 사용되는 기술입니다.

참고: 보다 자세한 기술적 개요는 "Hugging Face KV 캐싱 개요"을 참조하십시오.

KV 캐시 오프로딩이란 무엇입니까?

대부분의 추론 엔진은 기본적으로 KV 캐시를 GPU 메모리에 저장합니다. 이는 수요가 증가하기 전까지는 잘 작동합니다. KV 캐시 크기는 배치 크기(동시에 처리되는 프롬프트 수)와 시퀀스 길이(각 대화 또는 생성 스트림의 길이)에 비례하여 선형적으로 확장됩니다. 컨텍스트 창이 확장되고, 다중 턴 채팅이 일반화되고, 더 많은 사용자와 에이전트가 추론 서비스를 사용하게 되면 KV 캐시 요구 사항이 사용 가능한 GPU 메모리를 빠르게 초과할 수 있습니다. 이러한 상황이 발생하면 유용한 캐시 항목이 종종 제거되고, 엔진은 이미 처리한 토큰에 대한 어텐션을 다시 계산해야 합니다.

KV 캐시 오프로딩은 이전에 처리된 프롬프트의 KV 캐시를 시스템 RAM, 로컬 디스크 또는 공유 스토리지 등 GPU 또는 가속기 메모리 외부의 외부 대상으로 이동하여 해당 제약을 해결합니다. 오프로딩된 항목은 용량이 허용하는 한 유지되며, GPU 메모리가 가득 찰 때 폐기되는 대신 유사한 접두사 또는 후속 요청이 도착할 때 다시 참조될 수 있습니다. 사실상 이러한 오프로딩 대상은 GPU 메모리를 위한 계층형 스왑 공간처럼 작동합니다. VRAM보다 속도는 느리지만 더 크고 내구성이 뛰어나, 반복적인 사전 채우기 작업 없이 시스템이 유지할 수 있는 대화 컨텍스트 및 동시 작업 부하의 양을 확장합니다.

공유 스토리지 계층의 중요성

KV 캐시 오프로딩은 단일 추론 엔진의 GPU 메모리 용량이 부족해질 때 이미 도움이 됩니다. 캐시 블록을 CPU RAM으로 이동하면 하나의 서버에서 용량을 확장할 수 있습니다. 이는 유용하지만 프로덕션 환경의 문제를 부분적으로만 해결합니다. 실제 추론 플랫폼은 단일 서버 엔진 인스턴스로 실행되는 경우가 드뭅니다. 로드 밸런서 뒤에서 실행되고, 수평 확장이 가능하며, 수많은 동시 사용자와 에이전트를 처리합니다. 이러한 환경에서 공유 스토리지는 KV 캐시를 로컬 최적화에서 플랫폼 기능으로 전환하는 핵심 요소입니다.

- 공유 스토리지를 통해 여러 인스턴스에서 재사용 가능 공유 스토리지의 주요 이점 중 하나는 여러 인스턴스와 노드에서 동일한 파일이나 객체에 액세스할 수 있다는 것입니다. 이는 특히 로드 밸런서 뒤에 두 개 이상의 서버 엔진 인스턴스가 확장 배포된 경우, 각 인스턴스가 KV 캐시 블록을 동일한 공유 버킷 또는 NAS 볼륨으로 오프로드하도록

구성된 경우에 더욱 중요합니다. 예를 들어, 한 인스턴스가 프롬프트 접두사를 처리하고 결과 캐시 블록을 오프로드하면, 다른 인스턴스는 캐시 적중 시 동일한 사전 채우기 작업을 다시 계산하는 대신 해당 블록을 로드할 수 있습니다. 이는 프로덕션 트래픽이 분산되어 있기 때문에 중요합니다. 사용자의 첫 번째 질문은 인스턴스 A로 전송될 수 있고, 후속 질문이나 유사한 시스템 프롬프트를 사용하는 다른 사용자는 인스턴스 B로 전송될 수 있습니다. 공유 스토리지가 없으면 각 인스턴스는 자체적인 개인 캐시 환경을 유지해야 합니다.

- **CPU** 메모리만으로는 멀티 인스턴스 배포에 충분하지 않습니다**.** CPU 계층은 빠르지만 각 서비스 엔진 프로세스에 로컬로 할당됩니다. 한 인스턴스가 다른 인스턴스가 로컬 CPU RAM으로 오프로드한 KV 캐시 항목에 액세스하려면 피어 투 피어 채널을 구현해야 하며, 이는 추가적인 지연 시간과 운영 복잡성을 초래합니다. 공유 스토리지는 이러한 장벽을 제거합니다. CPU RAM은 각 노드에서 빠른 스테이징 계층으로서 여전히 유용하지만, 공유 S3 또는 NAS는 여러 엔진이 읽고 쓸 수 있는 공통 메모리 공간을 제공합니다. 더 이상 GPU를 개별적으로 확장하는 것이 아니라 공유 캐시 인프라를 통해 확장할 수 있습니다.
- **3**단계 설계 전략 구현 + 바람직한 아키텍처는 다음을 결합합니다:
  - **GPU** 메모리 — 진행 중인 요청을 위한 활성 캐시입니다.
  - **CPU** 메모리 — 프롬프트가 대기열에 들어갈 때 빠른 스테이징이 가능합니다.
  - 공유 스토리지 — 내구성이 뛰어나고 용량이 큰 공유 백업 스토리지입니다.

  이 아키텍처를 사용하면 새로운 요청이 도착할 때 공유 계층에서 CPU 메모리로 KV 캐시 블록을 스테이징할 수 있으므로 GPU는 활성 작업에 집중하면서도 스토리지에 영구 캐시를 유지할 수 있습니다.

- 속도뿐 아니라 추론의 경제성 프로덕션 관점에서 공유 스토리지의 중요성은 단순히 지연 시간 단축에만 있는 것이 아닙니다. 메모리, 동시성, 그리고 비용에 대한 제어가 중요합니다. vLLM과 같은 서비스 엔진은 단일 노드 내에서 KV 캐시를 효율적으로 관리합니다. LMCache와 같은 KV 캐시 오프로드 프레임워크는 KV 캐시를 CPU 메모리와 스토리지를 넘나들며 이동하고 공유할 수 있는 휴대 가능한 자산으로 만들어 줍니다. NetApp ONTAP 및 StorageGRID와 같은 공유 스토리지 플랫폼은 여러 인스턴스에서 실질적인 공유를 가능하게 하는 내구성 있는 계층을 제공합니다. LMCache가 공유 스토리지에 연결되면 여러 vLLM 인스턴스가 동일한 오프로드된 KV 캐시 항목에 액세스할 수 있으므로 처리량이 향상되고 동시성이 증가함에 따라 중복 계산이 줄어듭니다. 이는 챗봇, 검색 도우미, 에이전트 및 다중 턴 스레드, 공유 시스템 프롬프트 또는 반복적인 컨텍스트 패턴을 사용하는 모든 워크로드에 직접적으로 관련된 반복적인 사전 채우기에 낭비되는 GPU 사이클을 줄여줍니다.
- 추론 성능 향상 및 **GPU** 투자 효율 극대화 + 이 벤치마크는 대화 길이와 동시 로드 증가에 따른 추론 성능을 비교합니다. KV 캐시를 GPU 메모리에만 유지할 경우, 사용 가능한 여유 공간이 빠르게 소진되어 처리량이 급격히 감소합니다(주황색 선). 활성 작업용 GPU, 빠른 스테이징을 위한 CPU, 그리고 내구성이 뛰어나고 공유 가능한 캐시를 위한 공유 스토리지로 구성된 3계층 설계를 통해 플랫폼은 더 많은 세션을 유지하고 동일한 급격한 성능 저하를 방지할 수 있습니다. 실제로 계층화는 반복적인 사전 채우기 계산을 줄여 동일한 GPU에서 더 많은 작업을 처리하는 데 도움이 됩니다.
  - 간단히 말해서**:**
    - **GPU** 계층 — 진행 중인 활성 작업을 처리합니다.
    - **CPU** 계층 — 최근 사용한 캐시를 서비스 엔진 가까이에 유지합니다.
    - 공유 스토리지 계층 — 서버 간 캐시를 유지하고 새 세션을 위해 GPU/CPU 메모리를 확보합니다.

  그림 1 - 다중 라운드 추론 벤치마크

벤치마크에 따르면 CPU 메모리와 함께 공유 스토리지를 추가해도 성능이 저하되지 않으며, 처리량 저하가 나타나기 전 사용 가능한 작동 범위가 확장됩니다. 계층화는 추론을 위한 메모리 계층을 효과적으로 추가하여 세션 수, 컨텍스트 길이 또는 동시 접속자 수가 증가할 때마다 GPU를 추가해야 하는 필요성을 줄여줍니다.

- 엔터프라이즈급 적합성**:** 표준 프로토콜 사용**,** 독점 클라이언트 없음 + 공유 스토리지는 중요하지만 모든 공유 스토리지가 동일한 것은 아닙니다. NetApp은 업계 표준 프로토콜 및 툴링을 사용하여 KV 캐시 오프로딩을 지원합니다.
  - **StorageGRID S3**
  - **ONTAP NAS**용 **NFS / pNFS**

  별도의 독자적인 추론 클라이언트가 필요하지 않습니다. 운영팀은 기존에 다른 데이터 서비스에 사용하던 것과 동일한 스토리지 프로토콜을 활용할 수 있으며, 익숙한 데이터 보호, 보안 및 멀티클라우드 이동성을 그대로 사용할 수 있습니다.

KV 캐시 오프로딩 배포

공유 스토리지는 KV 캐시 용량을 확장하고 여러 서비스 인스턴스에서 재사용할 수 있도록 합니다. 프로덕션 환경에서 이 계층을 사용하려면 추론 엔진과 KV 캐시 오프로딩 프레임워크를 구성해야 합니다. 다음 섹션에서는 일반적인 도구 옵션을 사용하여 이 스택을 구현하는 방법을 설명합니다.

vLLM

vLLM은 널리 사용되는 고성능 오픈소스 LLM 추론 엔진입니다. 이 솔루션에서 vLLM은 사용자 요청을 수신하고, 응답을 생성하며, 추론 중에 GPU 메모리 내의 KV 캐시를 관리하는 역할을 합니다. vLLM은 플러그인 방식으로 설계되어 있어 원하는 오프로딩 프레임워크를 선택하여 사용할 수 있습니다. 다음 섹션에서는 널리 사용되는 오프로딩 프레임워크를 활용하여 vLLM 기반 서빙 스택을 구현하는 방법을 설명합니다.

LMCache(프로세스 내 모드)

LMCache는 널리 사용되는 오픈 소스 KV 캐시 오프로딩 프레임워크입니다. 이 섹션에서는 LMCache를 사용하여 KV 캐시 오프로딩을 수행하는 vLLM 기반 서버 스택을 구현하는 방법을 설명합니다. 이 구현에서 LMCache는 vLLM과 연동하여 KV 캐시를 GPU 메모리에서 CPU RAM, 로컬 디스크 또는 공유 스토리지(StorageGRID S3 또는 ONTAP NAS 마운트 등)와 같은 더 큰 계층으로 이동합니다.

기본 설정에서 vLLM은 KV 캐시를 GPU 메모리에만 유지합니다. 시스템 부하가 증가함에 따라 이것이 병목 현상이 됩니다. 이 솔루션에서는 vLLM을 LMCache와 함께 사용하여 vLLM이 모델을 처리하고 LMCache는 CPU와 공유 NetApp 스토리지에 걸쳐 캐시를 오프로드하고 재사용합니다. LMCache는 커넥터를 통해 vLLM에 연결됩니다. vLLM의 --kv-transfer-config 플래그를 사용하여 시작 시 연결 기능을 활성화하면 vLLM과 LMCache는 캐시를 스토리지에 저장하고 캐시 적중 시 다시 로드합니다.

인프로세스 모드는 vLLM에서 LMCache를 실행하는 원래 방식입니다. 인프로세스 모드를 사용하면 LMCache가 vLLM 프로세스 내에서 실행됩니다. 이후 LMCache 버전에서는 멀티프로세스 모드가 추가되었으며, 현재는 기능 지원 및 성능 향상을 위해 이 방식을 권장합니다. 이 섹션에서는 현재 LMCache 문서에서 더 이상 사용되지 않는 것으로 표시된 인프로세스 모드에 대해 설명합니다. 새로운 배포 환경에서는 멀티프로세스 모드를 사용하는 것을 고려하십시오.

이번 배포를 위해 다음 사항을 수행해야 합니다.

- vLLM을 LMCache와 함께 설치
- LMCache 커넥터를 활성화한 상태로 vLLM을 시작합니다.
- 두 개의 vLLM 인스턴스(각각 별도의 GPU에서 실행)를 vLLM 라우터 뒤에 배치하고, 두 인스턴스 모두 동일한 공유 스토리지 백엔드를 사용하는 3계층 구성(GPU + CPU + 공유 스토리지)을 배포합니다.

필수 조건

- NVIDIA 드라이버(CUDA 포함)와 최소 1개의 GPU가 설치된 Linux GPU 서버 (완전한 2개 인스턴스 + 라우터 구성의 경우 2개의 GPU 또는 여러 대의 서버)
- Python 및 uv 설치됨
- Docker 및 NVIDIA Container Toolkit이 설치되어 있어야 합니다(4단계의 vLLM 라우터에 필요).
- 모델(예: Hugging Face)을 다운로드하고 스토리지 엔드포인트에 접근하기 위한 네트워크 액세스

StorageGRID S3 백엔드

- KV 캐시 사용을 위해 S3 버킷이 생성되었습니다.
- 버킷에 대한 읽기/쓰기 자격 증명
- GPU 서버에서 S3 엔드포인트까지의 네트워크 연결
- 가상 호스팅 스타일 S3 요청 활성화됨

ONTAP pNFS/NFS 백엔드

- GPU 서버로 내보낸 ONTAP 볼륨
- vLLM이 실행되는 각 서버에 마운트 경로(예: /mnt/kvcache)가 생성됩니다. 고성능 pNFS over RDMA(RoCE)를 위한 마운트 명령 예시는 다음과 같습니다.

```
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 및 LMCache 설치

가상 환경을 생성하고 vLLM 및 LMCache를 설치하십시오. 자세한 내용은 LMCache "설치 설명서"를 참조하십시오. 사용 중인 CUDA 버전에 맞는 올바른 설치 경로를 따르는 것이 매우 중요합니다.

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

이 시점에는 추론 엔진(vLLM)과 캐시 계층(LMCache)이 있습니다.

[[2-configure-lmcache]]
=== 2. LMCache 구성

vLLM은 자체적으로 KV 캐시를 오프로드하지 않습니다. vLLM의 KV 전송 커넥터를 통해 LMCache를 활성화해야 합니다. CPU 및 공유 스토리지 계층 구성을 정의하는 YAML 파일(lmcache-config.yaml)을 생성합니다. LMCache는 vLLM 추론 시작 시퀀스의 일부로 이 YAML 파일을 읽습니다.

샘플 코드 조각: **StorageGRID S3** 공유 계층

```
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"
```

참고: 버킷 이름, 엔드포인트, 자격 증명 및 `disable_tls`을 환경에 맞게 변경하십시오.

샘플 코드 조각: **ONTAP pNFS/NAS** 공유 계층

```
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
```

3단계 전에 사이트의 NFS/pNFS 절차를 사용하여 ONTAP 내보내기를 마운트하십시오.

[[3-run-two-vllm-instances-shared-cache-across-servers]]
=== 3. vLLM 인스턴스 두 개를 실행합니다(서버 간 캐시 공유).

두 인스턴스 모두에 공통 환경 변수를 설정합니다.

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

인스턴스 **1** (GPU 0, 포트 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
```

인스턴스 **2** (GPU 1, 포트 8002 — 별도 터미널):

```
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
```

동일한 구성 파일과 해시 시드를 사용하여 두 인스턴스가 캐시 블록 ID를 일치시키고 공유 스토리지에서 서로의 오프로드된 데이터를 읽을 수 있도록 합니다. `VLLM_API_KEY`을 두 인스턴스 모두에 설정합니다. 라우터는 이 값을 `OPENAI_API_KEY`로 전달하여 라우팅된 요청이 인스턴스에서 수락되도록 합니다.

참고: 단일 GPU 인스턴스를 사용하여 계층형 스토리지 아키텍처를 구현할 수도 있습니다. 이 경우 4단계를 건너뛰십시오.

[[4-deploy-the-vllm-router-single-entry-point]]
=== 4. vLLM 라우터(단일 진입점)를 배포합니다.

두 인스턴스가 모두 정상적으로 작동하면 클라이언트가 OpenAI 호환 URL 하나를 사용하도록 라우터를 배포하십시오.

```
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
```

트래픽을 http://localhost:8000/v1로 전송합니다. 라우터가 요청을 분산하고, LMCache와 공유 스토리지를 통해 하나의 GPU 내에서뿐만 아니라 여러 인스턴스 간에 캐시를 재사용할 수 있습니다.

예를 들어, 다음과 같이 curl을 사용하여 라우터에 요청을 보낼 수 있습니다(<shared_api_key>을 본인의 API 키로 대체하십시오).

```
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. 계층화가 제대로 작동하는지 확인합니다.

- vLLM 프로세스는 모두 8001번 및 8002번 포트에서 수신 대기하며, 라우터는 8000번 포트에서 응답합니다.
- 라우터를 통해 긴 접두사를 포함한 요청을 보냅니다.
- 공유 스토리지(S3 버킷 또는 ls -ltr /mnt/kvcache)에 객체 또는 파일이 있는지 확인합니다.
- 비슷한 요청을 다시 보내세요. 두 번째 요청은 로그에서 더 빠른 사전 채우기 또는 캐시 적중 동작을 확인할 수 있어야 합니다.
- 라우터를 통해 반복하여 트래픽이 다른 인스턴스에 도달하도록 합니다.

결론

계층형 KV 캐시 아키텍처를 배포하면 KV 캐시가 GPU 메모리에서 CPU 메모리 및 공유 NetApp 스토리지로 이동하여 추론이 사용자 수와 컨텍스트 길이에 따라 확장될 수 있으며, 반복적인 사전 채우기에 GPU 사이클이 낭비되지 않습니다. 공유 스토리지를 통해 캐시는 여러 서비스 인스턴스에서 재사용 가능한 인프라가 되어 기존 GPU 투자에서 더 큰 가치를 얻을 수 있습니다.

NetApp 스토리지는 프로덕션 추론에 필요한 단순 용량 이상의 기능을 제공하기 때문에 이 계층에 매우 적합합니다. S3 및 NFS/pNFS를 통한 표준 액세스, 여러 서비스 엔진 인스턴스가 동시에 사용할 수 있는 공유 캐시, 그리고 내구성, 보호 및 거버넌스를 위해 이미 사용하고 있는 엔터프라이즈 데이터 서비스 등이 여기에 포함됩니다. 별도의 클라이언트가 필요하지 않으며, KV 캐시 오프로딩 프레임워크는 익숙한 프로토콜을 사용하여 StorageGRID S3 또는 ONTAP NAS 마운트에 연결할 수 있습니다.

NetApp은 추론 실행 방식이나 팀의 데이터 관리 방식을 변경하지 않고도 KV 캐시 오프로딩을 위한 익숙한 스토리지 기반을 제공합니다.