Déchargement du cache KV avec NetApp

NetApp artificial intelligence solutions

NetApp
August 20, 2026

Sommaire

Déchargement du cache KV avec NetApp

La première vague d'investissements dans l'IA d'entreprise s'est concentrée sur l'entraînement et l'optimisation des modèles, sans toutefois les déployer à grande échelle. À mesure que les organisations passent de l'expérimentation à la production, l'attention se porte désormais sur l'inférence en temps réel : assistants de recherche, chatbots, copilotes et flux de travail d'agents avec lesquels les utilisateurs interagissent en permanence. Dans ces environnements, l'utilisation des GPU augmente rapidement, non pas parce que chaque requête nécessite un entraînement complet, mais parce que chaque question complémentaire, chaque nouvel échange et chaque utilisateur simultané augmentent la charge sur la pile d'inférence.

Introduction

Un schéma se dessine rapidement : le GPU effectue une quantité surprenante de tâches répétitives, recalculant l'attention sur des jetons déjà traités. Cette répétition n'est pas uniquement un problème d'optimisation ; il s'agit d'un problème d'architecture mémoire. La réponse de l'industrie est une stratégie de mémoire hiérarchisée articulée autour du cache KV.

Qu'est-ce qu'un cache KV ?

En termes simples, la mise en cache KV (clé-valeur) est une technique utilisée pour optimiser l'inférence des grands modèles de langage (LLM) en stockant les valeurs précédemment calculées dans un cache KV afin que ces valeurs n'aient pas besoin d'être recalculées pour chaque nouveau jeton généré, ce qui serait autrement nécessaire.

Remarque : Pour une présentation technique plus détaillée, veuillez consulter "Aperçu de la mise en cache KV Hugging Face".

Qu'est-ce que le déchargement du cache KV ?

La plupart des moteurs d'inférence stockent par défaut le cache clé-valeur (KV) dans la mémoire GPU. Cela fonctionne correctement jusqu'à ce que la demande augmente. La taille du cache KV est proportionnelle à la taille des lots (le nombre d'invites traitées simultanément) et à la longueur des séquences (la longueur de chaque conversation ou flux de génération). À mesure que les fenêtres de contexte s'étendent, que les conversations à plusieurs tours se généralisent et que davantage d'utilisateurs et d'agents consomment les services d'inférence, les besoins en cache KV peuvent rapidement dépasser la mémoire GPU disponible. Dans ce cas, des entrées de cache utiles sont souvent supprimées et le moteur doit recalculer l'attention pour des jetons déjà traités.

Le déchargement du cache KV permet de contourner cette contrainte en déplaçant le cache KV d'une requête précédemment traitée vers une cible externe, hors de la mémoire GPU ou de l'accélérateur, telle que la RAM système, le disque local ou un stockage partagé. Les entrées déchargées peuvent être conservées tant que la capacité le permet et réutilisées lors de la réception d'un préfixe similaire ou d'une requête de suivi, au lieu d'être supprimées lorsque la mémoire GPU est saturée. De fait, ces cibles de déchargement fonctionnent comme un espace d'échange hiérarchisé pour la mémoire GPU : plus lent que la VRAM, mais plus volumineux et plus durable, ce qui augmente la quantité de contexte conversationnel et la charge de travail simultanée que le système peut supporter sans avoir à effectuer de préremplissage répété.

Importance d’un niveau de stockage partagé

Le déchargement du cache KV s’avère déjà utile lorsqu’un moteur d’inférence unique dépasse la capacité de la mémoire GPU. Déplacer les blocs de cache vers la RAM du CPU augmente la capacité d’un serveur. C’est utile, mais cela ne résout qu’une partie du problème en production. Les véritables plateformes d’inférence fonctionnent rarement comme une seule instance de moteur de service. Elles s’exécutent derrière des équilibreurs de charge, évoluent horizontalement et servent de nombreux utilisateurs simultanés et agents. Dans ce contexte, le stockage partagé transforme le cache KV d’une optimisation locale en une capacité de la plateforme.

- **Le stockage partagé permet la réutilisation entre les instances de service** L’un des principaux avantages du stockage partagé est la possibilité d’accéder aux mêmes fichiers ou objets depuis plusieurs instances et nœuds. Ceci est particulièrement vrai dans le cas d’un déploiement à grande échelle de deux instances de moteur de service ou plus derrière un équilibreur de charge, chacune configurée pour décharger les blocs de cache KV vers le même compartiment partagé ou volume NAS. Par exemple, lorsqu’une instance traite un préfixe d’invite et décharge les blocs de cache résultants, une autre instance peut charger ces blocs lors d’un accès au cache, évitant ainsi de recalculer le même travail de préremplissage. Ceci est important car le trafic en production est distribué : la première question d’un utilisateur peut atteindre l’instance A ; une question de suivi ou un autre utilisateur avec une invite système similaire peut atteindre l’instance B. Sans stockage partagé, chaque instance conserve son propre cache privé.
- La mémoire CPU seule ne suffit pas pour les déploiements multi-instances. La couche CPU est rapide, mais elle reste locale à chaque processus de moteur de service. Une instance ne peut pas accéder aux entrées du cache KV qu’une autre instance a déportées vers la RAM locale de son CPU, sauf si vous mettez en place un canal peer-to-peer, ce qui introduit une latence supplémentaire et une complexité opérationnelle. Le stockage partagé lève cet obstacle. La RAM du CPU demeure précieuse comme couche de transit rapide sur chaque nœud, mais un stockage S3 ou NAS partagé fournit le plan mémoire commun que plusieurs moteurs peuvent lire et écrire. Vous ne faites plus évoluer les GPU de manière isolée, vous évoluez avec une infrastructure de cache partagée.
- **Activation d’une stratégie de conception à trois niveaux** + Une architecture préférable combine :

  - **Mémoire GPU** — cache actif pour les requêtes en cours.
  - **Mémoire du processeur** — mise en mémoire rapide lors de l’entrée des invites dans la file d’attente.
  - **Stockage partagé** — espace de stockage durable, de grande capacité et partageable.

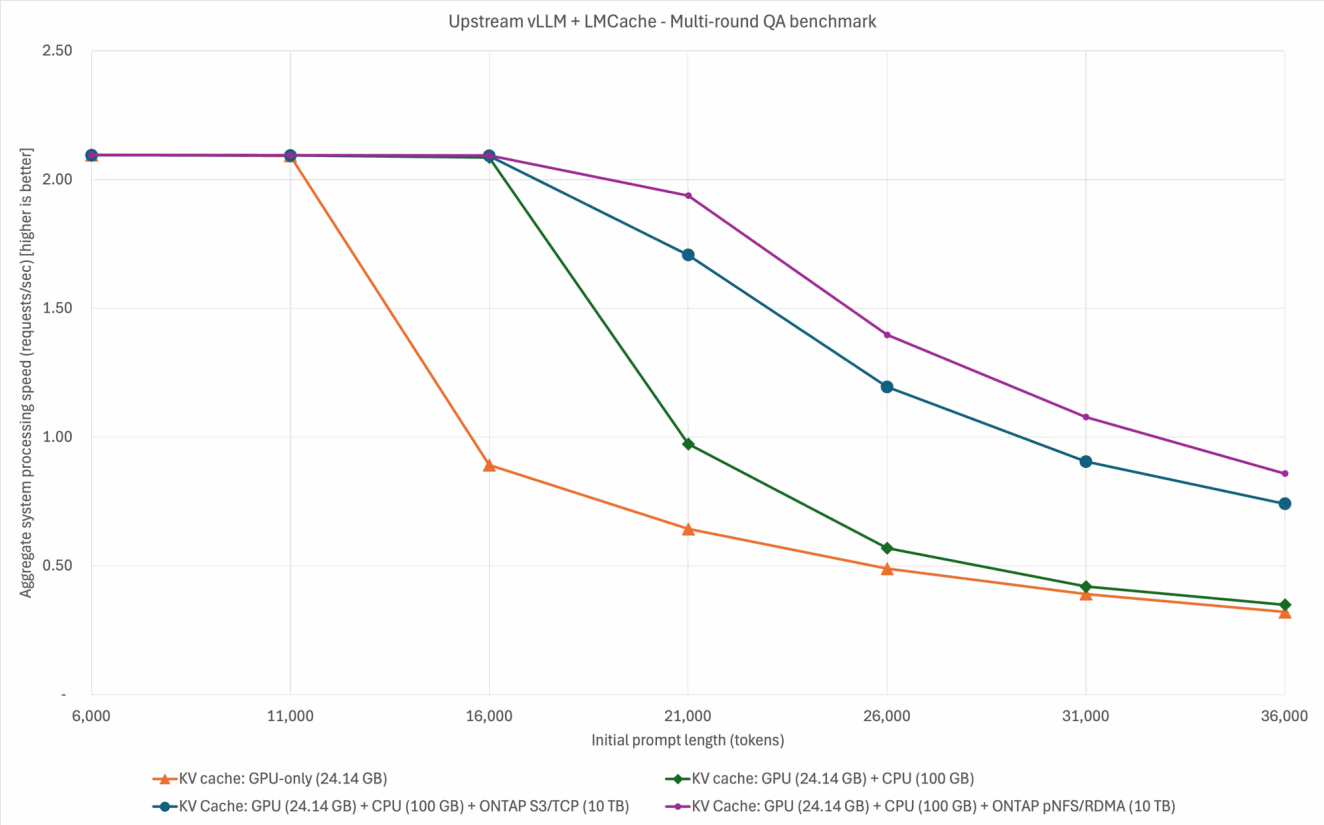
  Grâce à cette architecture, les blocs de cache KV peuvent être transférés de la couche partagée vers la mémoire du processeur au fur et à mesure que de nouvelles requêtes arrivent, permettant ainsi au GPU de se concentrer sur les tâches actives tout en conservant un cache durable sur le stockage.

- **L’économie de l’inférence, pas seulement la vitesse** Du point de vue de la production, l’importance du stockage partagé ne réside pas uniquement dans la latence. Elle concerne le contrôle de la mémoire, la concurrence et le coût. Les moteurs de service comme vLLM gèrent efficacement le cache KV à l’intérieur d’un seul nœud. Les frameworks de déchargement du cache KV, tels que LMCache, transforment le cache KV en un actif portable et partageable pouvant circuler entre la mémoire du processeur et le stockage. Les plateformes de stockage partagé comme NetApp ONTAP et StorageGRID fournissent le niveau durable qui rend ce partage possible entre plusieurs instances. Avec LMCache connecté au stockage partagé, plusieurs instances vLLM peuvent accéder aux mêmes entrées de cache KV déchargées, ce qui améliore le débit et réduit les calculs redondants à mesure que la concurrence augmente. Cela réduit le gaspillage de cycles GPU lors des préremplissages répétés, ce qui est directement pertinent pour les chatbots, les assistants de recherche, les agents et toute charge de travail comportant des threads multi-tours, des invites système partagées ou des modèles de contexte récurrents.
- **Optimisation des performances d’inférence et de l’investissement GPU** + Ce test compare les

performances d'inférence en fonction de la longueur des conversations et de la charge simultanée. Lorsque le cache KV est stocké uniquement dans la mémoire GPU, la marge disponible est rapidement épuisée et le débit chute (courbe orange). Grâce à une architecture à trois niveaux (GPU pour les tâches actives, CPU pour la préparation rapide et stockage partagé pour un cache durable et partageable), la plateforme prend en charge davantage de sessions et évite cette chute brutale de performances. En pratique, cette architecture hiérarchisée vous permet d'optimiser l'utilisation des mêmes GPU en réduisant les calculs de préremplissage répétitifs.

  - **En termes simples :**
    - **Niveau GPU** — gère le travail actif en cours.
    - **Niveau CPU** — conserve le cache récemment utilisé à proximité du moteur de service.
    - **Niveau de stockage partagé** — préserve le cache entre les serveurs et libère la mémoire GPU/CPU pour les nouvelles sessions.

Figure 1 - Benchmark d'inférence multi-tours



Le test de performance indique que l'ajout de stockage partagé à la mémoire du processeur ne réduit pas les performances ; il étend la plage de fonctionnement utilisable avant l'apparition d'une dégradation du débit. La hiérarchisation ajoute efficacement des couches de mémoire pour l'inférence, réduisant ainsi le besoin d'ajouter des GPU à chaque augmentation du nombre de sessions, de la longueur du contexte ou du nombre d'utilisateurs simultanés.

- **Adapté aux entreprises : protocoles standard, aucun client propriétaire** + Le stockage partagé est important, mais tous les stockages partagés ne se valent pas. NetApp prend en charge le déchargement du cache KV à l'aide de protocoles et d'outils standard du secteur :
  - **StorageGRID S3**

- **NFS / pNFS pour ONTAP NAS**

Aucun client d'inférence propriétaire personnalisé n'est requis. Les équipes d'exploitation peuvent utiliser les mêmes protocoles de stockage qu'elles exploitent déjà pour d'autres services de données, avec une protection des données, une sécurité et une mobilité multicloud familières.

Déployez le déchargement du cache KV

Le stockage partagé étend la capacité du cache KV et permet sa réutilisation entre les instances de service. Pour utiliser ce niveau en production, vous configurez un moteur d'inférence et un framework de déchargement du cache KV. Les sections suivantes décrivent comment implémenter cette pile à l'aide d'options d'outillage courantes.

vLLM

vLLM est un moteur d'inférence LLM open source performant et populaire. Dans cette solution, c'est le moteur qui reçoit les requêtes utilisateur, génère les réponses et gère le cache KV dans la mémoire GPU pendant l'inférence. vLLM est modulaire : vous pouvez choisir le framework de déchargement que vous souhaitez utiliser. Les sections suivantes décrivent comment implémenter une architecture de service basée sur vLLM avec les frameworks de déchargement les plus courants.

LMCache (mode en processus)

LMCache est un framework open source populaire de déchargement du cache clé-valeur. Cette section décrit comment implémenter une pile de service basée sur vLLM qui utilise LMCache pour le déchargement du cache clé-valeur. Dans cette implémentation, LMCache fonctionne avec vLLM pour déplacer le cache clé-valeur de la mémoire GPU vers des niveaux de stockage plus importants tels que la RAM du processeur, le disque local ou un stockage partagé (comme StorageGRID S3 ou un montage ONTAP NAS).

Dans une configuration par défaut, vLLM stocke le cache KV uniquement dans la mémoire GPU. Lorsque la charge système augmente, cela devient un goulot d'étranglement. Dans cette solution, vLLM est associé à LMCache, où vLLM sert le modèle, tandis que LMCache décharge et réutilise le cache entre le CPU et le stockage partagé NetApp. LMCache se connecte à vLLM via un connecteur ; vous l'activez au démarrage avec l'option --kv-transfer-config de vLLM, afin que vLLM et LMCache enregistrent le cache sur le stockage et le rechargent lors d'un accès au cache.

Le mode « in-process » est la méthode d'origine d'exécution de LMCache avec vLLM. Lorsque vous utilisez le mode « in-process », LMCache s'exécute au sein du processus vLLM. Les versions ultérieures de LMCache ont ajouté le mode « multiprocess », qui est actuellement la méthode recommandée pour une meilleure prise en charge des fonctionnalités et des performances accrues. Cette section couvre le mode « in-process », qui est marqué comme obsolète dans la documentation actuelle de LMCache. Pour les nouveaux déploiements, envisagez d'utiliser plutôt le mode « multiprocess ».

Pour ce déploiement, vous allez :

- Installez vLLM avec LMCache
- Démarrez vLLM avec le connecteur LMCache activé
- Déployez une configuration à trois niveaux (GPU + CPU + stockage partagé) avec deux instances vLLM (sur des GPU séparés) derrière un routeur vLLM, toutes deux utilisant le même backend de stockage partagé.

Prérequis

- Serveur GPU Linux avec pilotes NVIDIA (y compris CUDA) et au moins un GPU (deux GPU ou plusieurs serveurs pour la configuration complète à deux instances + routeur)
- Python et uv installés
- Docker et NVIDIA Container Toolkit installés (requis pour le routeur vLLM à l'étape 4)
- Accès réseau pour télécharger le modèle (par exemple, Hugging Face) et pour atteindre votre point de terminaison de stockage

Backend S3 StorageGRID

- Compartiment S3 créé pour l'utilisation du cache KV
- Informations d'identification en lecture/écriture pour le compartiment
- Connectivité réseau du serveur GPU au point de terminaison S3
- Requêtes S3 de type hébergé virtuel activées

Backend ONTAP pNFS/NFS

- Volume ONTAP exporté vers le(s) serveur(s) GPU
- Chemin de montage créé (par exemple, /mnt/kvcache) sur **chaque** serveur exécutant vLLM. Exemple de commande de montage pour un pNFS haute performance sur RDMA (RoCE) :

```
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. Installez vLLM et LMCache

Créez un environnement virtuel et installez vLLM et LMCache. Consultez la documentation de LMCache "documentation d'installation" pour plus de détails. Il est essentiel que vous suiviez le chemin d'installation correct pour votre version de CUDA.

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

À ce stade, vous avez le moteur d'inférence (vLLM) et la couche de cache (LMCache).

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

vLLM ne décharge pas le cache KV par lui-même. Vous activez LMCache via le connecteur de transfert KV de vLLM. Créez un fichier YAML (lmcache-config.yaml) qui définit la configuration du processeur et du niveau de stockage partagé. LMCache lit ce fichier YAML dans le cadre de la séquence de démarrage de l'inférence vLLM.

Exemple d'extrait : StorageGRID S3 shared tier

```
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"
```

Remarque : Remplacez le nom du compartiment, le point de terminaison, les informations d'identification et `disable_tls` pour qu'ils correspondent à votre environnement.

Extrait de code : ONTAP pNFS/NAS niveau partagé

```
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
```

Montez l'export ONTAP en utilisant la procédure NFS/pNFS de votre site avant l'étape 3.

[[3-run-two-vllm-instances-shared-cache-across-servers]]
=== 3. Exécutez deux instances vLLM (cache partagé entre les serveurs)

Définissez les variables d'environnement communes sur les deux instances :

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

Instance 1 (GPU 0, port 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
```

Instance 2 (GPU 1, port 8002 — terminal séparé) :

```
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
```

Utilisez le même fichier de configuration et la même valeur de hachage afin que les deux instances partagent la même identité de bloc de cache et puissent lire les données déchargées de l'autre depuis le stockage partagé. Définissez `VLLM_API_KEY` sur les deux instances ; le routeur transmet cette valeur comme `OPENAI_API_KEY` afin que les requêtes routées soient acceptées par les instances.

Remarque : Une seule instance de GPU peut également être utilisée pour implémenter une architecture de stockage hiérarchisée. Dans ce cas, ignorez l'étape 4.

[[4-deploy-the-vllm-router-single-entry-point]]
=== 4. Déployez le routeur vLLM (point d'entrée unique)

Une fois les deux instances opérationnelles, déployez un routeur afin que les clients utilisent une URL compatible avec OpenAI.

```
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
```

Envoyez le trafic à http://localhost:8000/v1. Le routeur distribue les requêtes ; LMCache et le stockage partagé permettent la réutilisation du cache entre les instances, pas seulement au sein d'un seul GPU.

Par exemple, vous pouvez envoyer une requête au routeur en utilisant curl comme suit (remplacez <shared_api_key> par votre clé API respective) :

```
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. Vérifiez que la hiérarchisation fonctionne.

- Les deux processus vLLM écoutent sur les ports 8001 et 8002 ; le routeur répond sur le port 8000.
- Envoyez une requête avec un long préfixe via le routeur.
- Vérifiez que les objets ou les fichiers apparaissent sur le stockage partagé (S3 bucket ou ls -ltr /mnt/kvcache).
- Envoyez une requête similaire à nouveau ; la seconde requête devrait afficher un préremplissage ou un accès au cache plus rapide dans les journaux.
- Répétez l'opération via le routeur afin que le trafic puisse atteindre l'autre instance.

Conclusion

Le déploiement d'une architecture de cache KV hiérarchisée déplace le cache KV de la mémoire GPU vers la mémoire CPU et le stockage partagé NetApp, permettant ainsi à l'inférence de s'adapter à l'augmentation du nombre d'utilisateurs et de la longueur du contexte sans gaspiller de cycles GPU en préremplissages répétés. Le stockage partagé transforme le cache en une infrastructure réutilisable pour toutes les instances de service et vous aide à optimiser votre investissement GPU existant.

NetApp storage est parfaitement adapté à ce niveau car il offre ce dont l'inférence de production a besoin au-delà de la simple capacité brute : un accès standard via S3 et NFS/pNFS, un cache partagé que plusieurs instances de moteur de service peuvent utiliser simultanément, et des services de données d'entreprise sur lesquels vous comptez déjà pour la durabilité, la protection et la gouvernance. Vous n'avez pas besoin d'un client propriétaire ; les frameworks de déchargement de cache KV se connectent à StorageGRID S3 ou à un montage NAS ONTAP en utilisant des protocoles familiers.

NetApp vous offre une base de stockage familière pour le déchargement du cache KV sans modifier la façon dont vous exécutez l'inférence ni la façon dont vos équipes gèrent les données.