vLLM: 이론과 활용 가이드

1. 이론적 배경

PagedAttention – **페이지 단위 KV 캐시 관리**

PagedAttention 은 **대용량 KV 캐시 메모리를 고정 크기 "블록(페이지)"들로 분할하여 필요할 때만 동적으로 할당**하는 알고리즘이다[1]. 기존에는 각 요청이 생성할 최대 토큰 길이만큼의 연속 메모리를 미리 확보해 두었으나, 이 방식은 대부분 사용되지 않는 KV 메모리 공간을 낭비하게 만들었다. vLLM 의 PagedAttention 에서는 **토큰 KV 캐시를 여러 작은 블록으로 쪼개고, 블록이 가득 찰 때마다 새로운 블록을 할당**하는 방식으로 메모리 단편화를 줄였다[2]. 덕분에 **필요한 만큼만 GPU 메모리를 사용**하여 더 많은 요청을 동시에 처리할 수 있고, 전체 메모리 효율도 향상된다[2]. 또한 PagedAttention 은 모든 Attention 연산을 하나의 글로벌 CUDA 커널로 **퓨즈(Fuse)** 하여 실행함으로써, HuggingFace Transformers 의 naive 구현처럼 여러 번의 `torch.bmm`, `softmax`, `dropout` 호출로 인한 **GPU 커널 런칭 오버헤드**를 제거한다[3]. (실제로 PagedAttention 에서는 `paged_attention_kernel` 이라는 단일 커널로 한 번에 모든 어텐션 연산을 수행하며, 이는 기존 PyTorch 연산들을 개별 호출하는 것보다 훨씬 효율적이다[4].) PagedAttention 기술은 2023 년 발표된 vLLM 논문[5]에도 도입되어 있으며, **메모리 파편화 제거와 대규모 배치 최적화**를 통해 vLLM 의 고성능 기반을 제공한다[6].

Continuous Batching – **연속 배치 스케줄링**

LLM 배치 처리 방식 비교: 위에서부터 개별 요청 처리(Static Single), 일반적인 동적 배칭(Dynamic Batching), 연속 배칭(Continuous Batching)의 개념도. 연속 배칭(Continuous Batching)은 **한 번에 들어온 요청 묶음을 모두 처리할 때까지 기다리지 않고**, 일부 요청이 완료되면 **즉시 새로운 요청을 해당 빈자리에 투입**하여 연속적으로 처리하는 기법이다[7]. 그림에서 보듯, 기존의 정적 배칭은 한 배치의 모든 시퀀스 출력이 끝날 때까지 다음 작업을 대기하지만, **연속 배칭은 작업** 한 스텝(iteration) **단위로 완료된 시퀀스를 배치에서 빼고 새로운 시퀀스를 즉시 추가**한다[7]. 이 **Iteration-level 지속적 스케줄링**을 통해 GPU 의 유휴 시간을 최소화하고, **완료된 요청은 곧바로 응답을 반환하면서도 남은 GPU 자원으로**

새로운 요청을 처리하게 된다[7]. 그 결과 **전체 처리량(Throughput)이 향상되고 지연시간(Latency)이 감소**하여 더 많은 동시 사용자 요청을 빠르게 처리할 수 있다[7]. Continuous Batching 개념은 2022 년 Orca 논문에서 제안되어 vLLM 등에 적용된 기법으로, vLLM 은 이를 통해 **배치 대기열 지연을 제거하고 GPU 활용도를 극대화**하였다.

KV 캐시 공유 – 프리픽스 중복 계산 제거

vLLM 은 **KV 캐시의 중복 사용을 줄이기 위해 Automatic Prefix Caching(프리픽스 캐싱) 기능**을 제공한다. **여러 요청이 동일한 프롬프트 앞부분(프리픽스)을 공유할 경우 이미 계산된 해당 KV 캐시 블록들을 재사용**함으로써 불필요한 연산을 피하는 원리이다[8]. vLLM 엔진은 들어오는 요청들의 토큰 시퀀스를 해시로 식별하여, **이전에 처리한 동일한 토큰 시퀀스 블록이 있으면 새로운 요청에서 이를 그대로 가져다 활용**한다[8]. 이를 통해 매 요청마다 처음부터 프롬프트 토큰을 다시 처리하지 않아도 되므로 **가장 비용이 큰 프리필(prefill) 단계 연산을 생략**할 수 있다[8]. 그 결과 **첫 토큰 응답 시간(TTFT)이 획기적으로 단축**되고 GPU 연산 여력이 절약되어 **전체 처리량도 높아지는 효과**가 나타난다[9]. 예를 들어 약 **1 만 토큰 길이의 프롬프트를 가진 요청을 한 번 처리한 후** 동일한 프롬프트의 두 번째 요청을 보낼 경우, **첫 토큰 생성까지 걸리는 시간이 4.3 초에서 0.6 초로 크게 줄었다**는 보고도 있다[10]. 이러한 **프리픽스 KV 캐시 공유** 기법은 다중 턴 대화(bot 챗 등)나 에이전트 루프 같이 **매 요청마다 공통 맥락(prompt)이 누적되는 시나리오**에서 특히 효과적이다 – 대화 이력 전체를 매번 다시 계산하지 않고 새로운 사용자 입력 부분만 처리하면 되므로 대화 길이가 길어져도 응답 지연이 늘어나지 않게 된다.

HuggingFace 대비 성능 우위 원리

위 세 가지 핵심 기술(PagedAttention, Continuous Batching, Prefix KV Caching)을 통해 **vLLM 은 기존의 HuggingFace Transformers 기반 추론 대비 월등한 성능**을 달성한다. 실제로 vLLM 연구진의 발표에 따르면 **vLLM 은 HuggingFace Transformers 나 TGI(Text Generation Inference) 대비 최대 24 배 이상의 추론 처리량 향상**을 보였고, **KV 캐시 메모리 낭비도 크게 절감**하였다[11]. 이러한 성능 우위의 원리는 **GPU 자원의 효율적 활용과 불필요한 연산 제거**로 요약된다. 예를 들어 HuggingFace Transformers 의 기본 구현은 시퀀스마다 별도의 KV 메모리를 통째로 할당하고, 토큰 생성 시에도 토치 연산들을 개별 호출하는 등 **메모리와 연산 측면에서 비효율적인 부분**이 있다[12]. 반면 vLLM 은

메모리를 필요한 만큼만 페이지로 나눠 쓰고, 요청들을 끊김 없이 연속 처리하며, FlashAttention 같은 퓨즈드 커널**과 CUDA 그래프 최적화**를 통해 매 토큰 생성 시 발생하는 오버헤드를 줄였다[13][7]. 그 결과 **GPU 메모리 활용률을 높이면서도 불필요한 연산을 최소화**하여 높은 처리량과 빠른 응답 속도를 구현한 것이다.

2. 중급 활용

Docker 를 이용한 vLLM 서버 구성 (GPU 환경)

vLLM 은 **공식 Docker 이미지(vllm/vllm-openai)**를 제공하고 있어, 온프레미스 환경에서 쉽게 컨테이너로 LLM 서비스를 배포할 수 있다[14]. 이 이미지는 OpenAI 호환 API 서버 기능을 내장하고 있으며, GPU 가속을 위해 Docker 호스트에 **NVIDIA Container Toolkit** 이 설치되어 있어야 한다. 컨테이너 실행 시 `--gpus all` 옵션을 지정하면 GPU 를 컨테이너에 할당할 수 있다[15]. 예를 들어, 아래와 같이 Docker 로 vLLM 서버를 실행할 수 있다:

```
$ docker run -d --gpus all -p 8000:8000 --ipc=host \
    -v ~/.cache/huggingface:/root/.cache/huggingface \
    -e HUGGING_FACE_HUB_TOKEN=<your_hf_token> \
    vllm/vllm-openai:v0.11.0 \
    --model <모델-이름 또는 경로>
```

위 명령에서 `-v` 옵션은 호스트의 HuggingFace 캐시 디렉토리를 컨테이너와 공유하여 **모델 다운로드 및 로드 시간을 단축**하고 중복 다운받는 것을 방지한다[16]. `HUGGING_FACE_HUB_TOKEN` 환경변수는 접근 제한된 HuggingFace 모델을 사용할 경우 필요한 토큰을 전달하는 것으로, 공개 모델이라면 생략 가능하다. 또한 `--ipc=host` 옵션을 주면 **컨테이너가 호스트와 IPC 자원을 공유**하여, 다중 프로세스 간 메모리 공유를 활용하는 vLLM 의 성능을 더욱 높일 수 있다[17]. (`--ipc=host` 는 GPU 가속 어플리케이션에서 프로세스 간 CUDA 메모리 공유에 유용하며, vLLM 도 대용량 KV 캐시를 프로세스 사이에서 다룰 때 이 이점을 활용한다.)

vLLM 0.11.0 버전부터는 **FlashInfer** 라는 고속 샘플링 커널이 도입되었는데, 일부 구형 GPU 에서는 이 최적화가 호환되지 않아 문제가 생길 수 있다. 예를 들어 NVIDIA T4 와 같이 Compute Capability 가 낮은 카드에서는 FlashInfer 사용 시 오류나 비정상 종료가 보고되어, Docker 환경에서 **FlashInfer 를 비활성화**해야 할 수도 있다[18]. 이 경우 위 실행 명령에 환경변수 `VLLM_USE_FLASHINFER_SAMPLER=0` 를 추가하여 FlashInfer 기반 샘플러를

끄면 된다[18]. (FlashInfer 는 샘플링 가속을 위한 고성능 CUDA 커널인데, Turing 세대 GPU 등에서는 미지원이므로 이러한 설정이 필요하다.)

다양한 모델 로딩 및 포맷 지원

vLLM 은 **HuggingFace Transformers 호환 포맷**을 기반으로 동작하므로, **HuggingFace 허브에 올라온 대부분의 LLM 모델을 바로 로딩하여 사용할 수 있다**[19]. Docker 컨테이너 실행 시 `--model` 옵션에 **모델 이름**(예: `THUDM/chatglm2-6b`)이나 **지역 경로**를 지정하면 해당 모델 가중치를 자동으로 불러온다[20]. 모델을 처음 로드할 때는 인터넷이 연결되어 있으면 HuggingFace Hub 에서 다운로드하며, 앞서 언급한 캐시 디렉토리를 통해 이후부터는 오프라인에서도 재사용이 가능하다.

vLLM 은 **양자화된 경량 모델**도 지원한다. 예를 들어 4 비트 양자화된 GPTQ 또는 AWQ 형식 모델의 경우, `--quantization=gptq` 혹은 `--quantization=awq` 플래그를 함께 지정하면 해당 방법으로 quantize 된 모델 weights 를 올바르게 불러온다[21]. 양자화 모델은 정밀도를 약간 희생하는 대신 **모델 메모리 크기를 줄여 GPU 메모리 사용량을 낮추므로, 더 큰 KV 캐시 확보**나 **동시 처리 증가, 속도 향상**에 이점이 있다[22]. 실제로 vLLM 은 INT4/INT8 등 저비트 모델을 활용하여 **동일한 GPU 에서 더 높은 동시 처리량**을 얻을 수 있음을 보여주고 있다. 양자화 방식으로는 GPTQ/AWQ 외에도 FP4(4 비트 부동소수)나 FP8 등의 포맷이 vLLM 0.11 에서 확대 지원되며(추론 시 전용 커널 사용), **W4A8** (가중치 4 비트-활성 8 비트) 양자화도 지원된다. 다만 일부 양자화 형식의 경우 아직 최적화가 완벽하지 않아 **비양자화 모델보다 속도가 크게 개선되지 않을 수 있음**을 참고해야 한다 (예: GPTQ 모델의 경우 현재 커널 최적화가 진행 중이라 속도가 FP16 모델 대비 이점이 크지 않을 수 있다).

또한 **Llama.cpp 계열 GGML/GGUF 포맷**의 모델도 vLLM 에서 실험적으로 지원한다. vLLM 0.11 기준으로 **단일 파일로 된 GGUF 모델을 로드하는 기능**이 도입되었으며, 이를 통해 HuggingFace 포맷으로 변환하지 않은 경량 모델도 사용할 수 있다[23]. 다만 GGUF 지원은 아직 **실험적(experimental) 기능**으로, 멀티파일 GGUF 는 하나로 병합하여야 하고 성능 최적화도 진행 중이다. (반면 vLLM 은 **CPU-only 실행은 공식 지원하지 않으며**, GPU 없이 GGML 모델을 구동하려면 Llama.cpp 등을 사용하는 것이 적합하다. vLLM 은 기본적으로 GPU 상의 고속 추론을 목표로 설계되었다는 점에 유의한다.)

vLLM 이 지원하는 **모델 아키텍처**로는 GPT-Neo/GPT-J 계열, GPT-2/3 계열, LLaMA 1&2, Mistral, Falcon, Baichuan, GLM, Bloom 등 **다양한 시리즈의 Decoder 모델**이 포함된다[19]. 최신 7B~70B 급 오픈소스 모델들은 대부분 vLLM 에서 실행 가능하며, Continual Batching 과 PagedAttention 의 혜택을 받을 수 있다. Encoder-Decoder 구조의 일부 모델도 (예: T5, BART) 제한적으로 지원되지만, v0.11 시점에 BART 지원은 일시 제거되었으므로 향후 버전 업데이트를 확인해야 한다. 최신 문서의 **지원 모델 목록**을 참고하면 현재 지원되는 아키텍처와 사용 가능한 태스크(텍스트 생성, 임베딩 등)를 확인할 수 있다.

배치 처리 및 Throughput 최적화 방법

vLLM 은 내부적으로 입력 요청들을 **동적 배치**로 처리하지만, 사용자도 몇 가지 **엔진 파라미터 튜닝**을 통해 처리량을 최적화할 수 있다. 우선 `vllm serve` 실행 시 설정 가능한 **`--max-num-seqs` 와 `--max-num-batched-tokens`** 값이 있다. `max_num_seqs` 는 한 번에 병렬 생성할 **최대 시퀀스 개수**이며, `max_num_batched_tokens` 는 한 iteration 에서 처리할 **총 토큰 수 한도**를 의미한다[24]. 이 두 값은 곧 한 스텝에 모델이 처리하는 배치의 크기를 결정하며, **값을 높이면 한 번에 더 많은 작업을 투입**하여 GPU 활용도를 높일 수 있다[24]. 단, 너무 높게 설정하면 **KV 캐시로 사용되는 메모리가 급증**하여 오히려 메모리 부족을 초래할 수 있으므로 GPU VRAM 용량에 맞게 조절해야 한다[24]. 예를 들어 VRAM 여유가 많고 응답 지연보다 처리량이 중요하다면 `max_num_seqs` 와 `max_num_batched_tokens` 를 늘려서 대량 요청을 동시에 처리하도록 하고, 반대로 짧은 지연이 중요하거나 메모리가 부족하면 이 값을 줄여 1 회 배치량을 낮출 수 있다. 한 실험 보고에 따르면 **A100 80GB GPU 환경**에서 `max_num_seqs=256`, `max_num_batched_tokens=512` 로 설정했을 때 **최고의 처리량**을 얻었다는 사례도 있다[25]. 적절한 값은 모델 크기와 GPU 사양, 요청 패턴에 따라 다르므로, **vLLM 로그에 표시되는 GPU 메모리 활용도와 토큰 처리 상황**을 보면서 튜닝하는 것이 좋다[26].

vLLM 엔진 스케줄러는 기본적으로 **연속 배치(continuous batching)**로 동작하여 GPU 자원을 빈틈없이 사용하지만, 추가로 `--async-scheduling` 옵션을 통해 **스케줄러의 CPU 병목을 줄이는 비동기 스케줄링** 모드를 사용할 수도 있다[27]. 이 모드는 내부 구현상 **스케줄링 작업을 엔진 실행과 겹쳐 수행**함으로써 대기 시간을 줄이는 실험적 기능이다. 그러나 v0.11.0 버전에서는 해당 옵션을 활성화할 경우 일부 상황(캐시 preemption 등)에서

출력 토큰이 깨지는 **버그**가 보고되었으므로[28], 현재 버전에서는 `--async-scheduling` 옵션을 **사용하지 않는 것이 권장**된다. (이 버그는 다음 버전에서 수정될 예정[28].)

그 외에 **GPU 메모리 활용 관련 옵션**으로 `--gpu-memory-utilization` 환경 변수가 있다. 기본값은 0.8 (80%)로 설정되어 있어 vLLM 엔진이 GPU VRAM 의 일정 비율만 사용하도록 여유를 남겨두는데, 만약 더 큰 KV 캐시 확보가 필요하면 이 값을 높여 메모리 사용량을 늘릴 수 있다[29]. 반대로 여러 인스턴스를 한 GPU 에 올리거나 할 경우 이 비율을 낮출 수도 있다. 이처럼 **엔진의 병렬 처리 한계치와 메모리 사용 한도**를 조절함으로써, 온프레미스 환경에서 **Throughput(초당 토큰 처리량)**을 요구에 맞게 높이거나 지연 시간을 조절할 수 있다.

3. 고급 활용

스케줄러 파라미터 상세 조정 및 우선순위

vLLM 의 **스케줄러**는 기본적으로 **FCFS(선입선출)** 방식으로 요청을 처리하지만, 필요에 따라 **Priority 스케줄링**으로 설정해 특정 요청에 가중치를 줄 수도 있다 (예: 중요 요청에 높은 우선순위 부여). 이러한 정책 변경은 `vllm.LLM(engine_args=...)` 초기화 시 config 를 조정하여 적용한다. 또한 **요청별 세부 한도 설정**으로 각 요청에 `max_tokens` (최대 생성 토큰 수), `temperature`, `top_p` 등의 샘플링 파라미터를 개별 부여할 수 있는데, vLLM 엔진은 각 요청의 `max_tokens` 를 확인하여 **KV 캐시 블록을 동적으로 관리**한다. 과거 Orca 엔진은 요청이 들어올 때 `max_tokens` 길이만큼 KV 메모리를 통으로 예약했지만, vLLM 은 PagedAttention 을 통해 **토큰이 실제 생성될 때마다 한 블록씩 할당**하기 때문에 메모리 활용이 훨씬 효율적이다[30]. 따라서 `max_tokens` 는 vLLM 에서 메모리 예약의 상한선으로 참고되지만, 미리 거대한 메모리를 점유하지는 않는다.

vLLM 엔진 V1 에서는 **Preemption(선점)** 기능도 포함되어 있다. 만약 동시 생성 중인 시퀀스들로 인해 **KV 캐시 메모리가 부족**해지는 경우, 우선순위가 낮은 일부 시퀀스의 KV 블록을 **CPU 로 스왑하거나(discard/recompute 모드)** 일시적으로 제거함으로써 공간을 확보한다[31][32]. 이러한 선점 알고리즘은 **LRU(Least Recently Used)** 방식으로 가장 마지막에 사용된 캐시부터 내보내는 식이며, 해당 시퀀스들의 처리는 잠시 중단되었다가 나중에 다시 재개된다[33]. 이 동작은 완전히 자동으로 이루어지며, 설정된

`max_num_batched_tokens` 한도 내에서 최대한 많은 요청을 수용하기 위한 것이다. 다만 선점이 빈번하게 발생하면 오히려 재계산 오버헤드로 성능이 떨어질 수 있으므로, **GPU 메모리 모니터링을 통해 선점이 일어나지 않을 적절한 배치 크기 설정**이 중요하다.

OpenAI 호환 API 서버 고성능 구성 (비동기 요청 처리)

vLLM 이 제공하는 API 서버는 **OpenAI 의 REST API 프로토콜과 호환**되며 (`/v1/completions`, `/v1/chat/completions` 엔드포인트 등), 이를 통해 기존 OpenAI API 사용 코드를 거의 수정 없이 자체 호스팅된 LLM 에 연결할 수 있다. vLLM 서버는 **Uvicorn 기반의 HTTP 서버**로 구동되며 기본적으로 단일 프로세스에서 동작하지만, 엔진 내부에서 다중 요청을 효율적으로 스케줄링하므로 **별도의 애플리케이션 레벨 비동기 처리를 하지 않아도** 다수의 동시 요청을 처리할 수 있다. 예를 들어 OpenAI API 의 `stream=True` 옵션을 활용하면 vLLM 서버도 **스트리밍으로 토큰을 실시간 전송**해주는데, 연속 배칭 덕분에 첫 토큰이 준비되는 대로 스트림을 시작하면서도 뒤에서는 다음 토큰들을 꾸준히 생성하는 식으로 **지연을 최소화**한다. 클라이언트 측에서는 **여러 요청을 비동기(Async)**로 보낼 경우 자동으로 vLLM 엔진이 이를 받아 **동적 배칭**하므로, 단일 요청씩 순차 호출하는 것보다 훨씬 높은 처리량을 달성할 수 있다.

고성능 설정을 위해 고려할 점으로는, **GPU 당 하나의 vLLM 서버 프로세스**를 운영하는 것이 좋다는 것이다. 하나의 vLLM 인스턴스가 이미 GPU 를 최대 활용하여 병렬 처리를 수행하므로, 동일 GPU 에 여러 vLLM 프로세스를 띄우면 오히려 메모리 경쟁과 컨텍스트 스위칭으로 성능이 떨어질 수 있다. 만약 **서버 노드에 여러 개의 GPU** 가 있다면, `--tensor-parallel-size` 옵션을 사용하지 않고 **GPU 마다 별도의 vLLM API 인스턴스**를 띄우는 방식(예: 컨테이너를 GPU 개수만큼 실행하고 각기 다른 포트 할당)도 고려할 수 있다. 이 경우 클라이언트 요청을 GPU 별로 라운드로빈 분배하는 로드밸런서를 두면 된다. vLLM 은 멀티스레드로 동작하므로 Uvicorn 의 workers 설정을 늘릴 필요는 거의 없지만, CPU 코어가 매우 많은 환경에서는 `--workers <n>`로 Uvicorn 프로세스를 늘려볼 수도 있다. 다만 **vLLM 엔진은 Python 레벨이 아닌 C++ 백엔드에서 병렬화**되므로, 일반적인 웹 서버처럼 워커 프로세스를 늘리는 효과가 크지 않을 수 있다.

정리하면, **OpenAI API 호환 모드**로 vLLM 서버를 운영할 때에는 별다른 튜닝 없이도 연속 배칭과 prefix 캐싱 등 엔진 최적화로 고성능이 나며, 클라이언트는 가능하면 **Streaming**

사용 및 **동시 요청수를 높이는** 방향으로 구현하여 엔진의 이점을 극대화하는 것이 좋다. (예를 들어 짧은 요청 여러 개를 모아서 동시에 보내면 vLLM 이 이를 하나의 배치로 묶어 처리할 수 있어 GPU 활용도가 높아진다.)

커스텀 모델 통합 (파인튜닝된 LLaMA 등)

vLLM 을 사용하면 자체 파인튜닝한 모델도 쉽게 서빙할 수 있다. **파인튜닝된 LLaMA, Mistral 등**이 HuggingFace Transformers 포맷으로 가중치가 저장되어 있다면, 앞서 설명한 대로 `--model` 옵션에 해당 경로 또는 HuggingFace Hub 레포지토리 이름을 넣어주기만 하면 된다. vLLM 은 내부적으로 HuggingFace 의 `AutoModelForCausalLM` 등을 사용하여 모델을 로드하므로, **지원되는 아키텍처의 모델이라면 추가 코드 수정 없이 동작**한다[19]. 예를 들어 LLaMA-2 를 파인튜닝한 모델을 HuggingFace Hub 에 업로드했다면, `--model user/finetuned-llama2-7b` 와 같이 지정하여 곧바로 추론에 활용할 수 있다.

다만 파인튜닝 방법 중 **LoRA 와 같은 경량 어댑터**를 적용한 모델의 경우 주의할 점이 있다. vLLM 0.11.0 에서는 구버전 엔진(V0) 제거와 함께 **런타임에 LoRA 를 병합하여 로드하는 기능이 일시적으로 사라졌다[34].** 따라서 LoRA 로 파인튜닝한 LLM 을 vLLM 에서 사용하려면, **사전에 LoRA 가중치를 원본 모델에 병합한 완전한 모델 가중치를 준비**해야 한다. vLLM 측은 LoRA 통합 로드 기능을 개선된 형태로 재도입할 예정이므로 추후 버전을 확인하는 것이 좋다. (참고로 v0.11 버전에서는 LoRA 관련 클래스들이 제거되었지만[34], 대신 **LoRA 적용 모델의 로딩 속도를 최적화**하는 등의 개선이 이루어져 있다[35].)

다중 GPU 및 분산 환경 설정

한 개의 GPU 메모리에 올릴 수 없는 **초대형 모델(예: 30B~70B 급)**은 vLLM 의 **텐서 병렬화(Tensor Parallelism)** 기능을 사용하여 **여러 GPU 에 나눠 로드**할 수 있다. vLLM 서버 실행 시 `--tensor-parallel-size <N>` 옵션을 주면 모델을 N 개 GPU 로 분할하여 각 GPU 에 일부 레이어씩 할당한다[36]. 예를 들어 4 개의 GPU 가 있는 서버에서 70 억파라미터 모델을 2 개 GPU 씩 나눠서 구동하려면 `--tensor-parallel-size 2` 로 지정하면 된다 (이 경우 남는 2 개 GPU 는 다른 인스턴스로 활용하거나, 한 인스턴스에 `--tensor-parallel-size 4` 로 모든 GPU 를 사용할 수도 있다). **Pipeline 병렬화**도 vLLM 0.11 에서 새롭게 지원되어, 거대 모델을 레이어 단위로 여러 GPU 에 **직렬 연결**하여 배치 크기를 키울 수도 있다[37]. 예를 들어 100 억 단위 이상의 모델을 8 개 GPU 에 올릴 때

Tensor Parallel+Pipeline Parallel 조합을 사용하면 각 GPU 메모리 부담을 줄이면서도 병렬 처리가 가능하다. (vLLM 에서는 HybridAllocator 가 이러한 복합 병렬화 구성을 지원한다[38].)

다중 노드(Distributed) 환경에서도 vLLM 을 활용할 수 있다. 한 서버 노드에서 처리하기 어려운 규모의 트래픽이나 모델 크기는, Ray 등을 활용하여 **여러 노드에 vLLM 인스턴스를 띄우고 분산 요청 처리**를 구현할 수 있다[39]. 예를 들어 2 노드 각 4GPU 씩 총 8GPU 로 하나의 거대 모델을 서빙하거나, 노드별로 동일 모델 인스턴스를 띄워 **클러스터로 확장**하는 시나리오가 가능하다. vLLM 0.11 에서는 `torchrun` 런처, Ray Placement Group 등의 지원이 강화되어 분산 환경에서의 배포가 한층 수월해졌다[39]. 다만 **분산 환경에서 유의할 점은 프리픽스 캐시의 활용률이 떨어질 수 있다**는 것이다. 예를 들어 여러 개의 vLLM 서버 인스턴스를 로드밸런싱할 경우, **연관된 대화 요청들이 서로 다른 노드로 흩어지면** 캐시를 재사용하지 못하고 매번 프리필을 다시 수행하게 된다[40]. 이를 해결하려면 **프리픽스 캐시 친화적인 라우팅** 전략이 필요하다. 하나의 사용자 세션은 항상 동일한 노드로 보내거나, llm-d 와 같은 **프리픽스-캐시 인지형 스케줄러**를 도입하여 관련 요청들을 캐시가 있는 곳으로 보내는 방식이 효과적이다[40][41]. 이러한 고급 분산 설정을 통해 vLLM 의 **캐시 재사용 이점**을 최대한 유지하면서도 대규모 서비스 요구를 충족시킬 수 있다.

4. 최신 변경 사항 요약 (vLLM 0.11.0 기준)

- **V0 엔진 제거 및 V1 통일**: v0.11.0 에서 오래된 V0 엔진 코드베이스가 완전히 제거되어, 이제 **V1 엔진만 사용**된다[42]. 이에 따라 `AsyncLLMEngine`, `LLMEngine` 등의 이전 클래스와 주석 옵션들이 사라졌으며, 엔진 설정은 모두 V1 기반으로 일원화되었다. (BART 등 일부 V0 전용 기능은 임시로 비활성화되었으며, 추후 재지원 예정[43].)

- **KV 캐시 메모리 오프로드 지원**: 대형 컨텍스트 처리 시 **GPU 메모리가 부족하면 KV 캐시를 CPU 메모리로 자동 오프로드**하는 기능이 추가되었다[44]. LRU 정책으로 사용되지 않은 캐시 블록을 PCIe 를 통해 CPU 로 옮겼다가, 다시 필요해지면 불러오는 **스왑 동작**을 수행하여 한정된 VRAM 에서도 긴 문맥을 처리할 수 있다.

- **엔진 기능 개선**: **프롬프트 임베딩 입력 지원**, **샤딩된 가중치 로딩** 등 새로운 기능이 도입되었다[38]. 예를 들어 Prompt Embedding 을 미리 계산해 입력하거나, 수백 GB 에 달하는 모델 가중치를 여러 파일로 분할하여 로드하는 기능이 향상되었다. 또한 **슬라이딩 윈도우 Attention** 지원이 추가되어, 일부 롱폼 모델에서 윈도우 움직이며 문맥을 처리하는 것이 가능해졌다[38].

- **병렬화 및 스케줄러 확장**: **Pipeline 병렬화** 지원이 Hybridallocator 에 추가되어, Tensor Parallel 과 결합해 더 유연한 다중 GPU 활용이 가능해졌다[37]. 서로 다른 hidden size 를 갖는 계층도 혼합 배치가 가능하도록 메모리 할당기가 개선되었다. 또한 **Async 스케줄링 모드가 단일 프로세서 환경에서도 사용 가능**하도록 변경되었으나, 현재 해당 모드 사용 시 출력 오류 이슈가 있어 기본 비활성화 상태이다[28].

- **성능 최적화**: 다양한 최적화로 **추론 속도가 향상**되었다. FlashInfer 모드에서 RoPE 적용을 최적화하여 **디코딩 속도를 2 배 가까이 증가**시켰고, Q/K 행렬 RoPE 계산을 fused 커널로 구현해 **약 11% 추가 성능 향상**을 얻었다[45]. Speculative Decoding(초안 생성)의 CPU 오버헤드를 줄여 해당 경로가 **8 배 빨라졌고**, FlashInfer 기반 speculative decoding 도 **1.14× 속도 증가**가 이루어졌다[45][46]. 내부적으로 **DeepGEMM** 알고리즘을 기본 활성화하여 **약 5.5%의 throughput 개선**을 달성했고, **NCCL 통신 최적화**(symmetric memory)로 다중 GPU Tensor Parallel 사용 시 **추가 3~4%의 성능 향상**이 이루어졌다[47][48].

- **양자화 지원 확대**: **FP8 양자화**가 한층 발전하여 토큰 그룹별 양자화 및 하드웨어 가속 명령 지원, PagedAttention 연동 등이 추가되었다[49]. 또한 **FP4(4 비트 부동소수)** 지원이 도입되고, **W4A8(INT4 가중치-INT8 활성)** 양자화에서 전처리 속도가 개선되었다[49]. 대용량 MoE 모델의 경우 **블록 FP8** 등 특수 압축 포맷도 실험적으로 지원된다.

- **API 및 UX 개선**: OpenAI 호환 API 에서 **전체 토큰에 대한 logprobs 출력**이 가능해졌고 (`logprobs=-1` 설정 시), **Reasoning 단계 스트리밍 이벤트** 등이 추가되어 추론 과정의 피드백을 실시간으로 받을 수 있다[50]. 또 멀티턴 대화에서 함수 호출 툴 모드 등이 개선되고(Hermes 토큰 포맷 지원 등), **XML 파싱** 등 특정

모델(Qwen3-Coder)의 툴콜링 요구사항이 반영되었다[50]. CLI 에서는 `--enable-logging` 플래그 추가, `--help` 출력 개선 등 **사용자 편의 기능**이 늘었다[51]. 구성 파일에서는 Speculative 모델 설정 분리, NVTX 프로파일링 옵션, 가이드된 디코딩 호환성 유지 등의 업데이트가 있다. 또한 지표 출력을 개선하여 KV 캐시 사용량을 GiB 단위로 표시하는 등 모니터링 활용성을 높였다[52].

- **기타 변경 및 알려진 이슈**: 내부적으로 오래된 **토크나이저 그룹 제거**, 멀티모달 캐시 공유 구조 통합 등 리팩토링이 이루어졌다[53]. v0.11.0 에서 **BART 모델 지원이 일시적으로 빠진 상태**이며[54], 이는 V0 엔진 제거 과정에서 발생한 임시 조치로 차기 버전에서 복구될 예정이다. **FlashInfer 샘플러의 비결정성 문제**가 보고되어 일부 배포판(Red Hat AI 등)에서는 기본적으로 FlashInfer 를 꺼놓고 있으며[55], 사용자 환경에서도 동일한 이슈 발생 시 앞서 언급한 방법으로 FlashInfer 를 비활성화하는 것이 권장된다. 전체적으로 vLLM 0.11.0 은 **엔진 구조 정비와 성능 향상**에 중점을 둔 릴리즈로, 약 538 건의 커밋과 200 여 명의 컨트리뷰터가 참여한 대규모 업데이트였다[56]. vLLM 을 사용하는 개발자는 변경 사항에 따른 호환성 확인과 함께, 새로운 최적화 기능들을 활용하여 한층 향상된 LLM 서비스 구현이 가능할 것이다.

---

[1] [2] [5] [6] [7] [11] [19] Meet vLLM: For faster, more efficient LLM inference and serving

https://www.redhat.com/en/blog/meet-vllm-faster-more-efficient-llm-inference-and-serving

[3] [4] [12] [13] [30] [31] [32] [33] LLM Inference: Continuous Batching and PagedAttention · Better Tomorrow with Computer Science

https://insujang.github.io/2024-01-07/llm-inference-continuous-batching-and-pagedattention/

[8] [9] [10] [40] [41] KV-Cache Wins You Can See: From Prefix Caching in vLLM to Distributed Scheduling with llm-d | llm-d

https://llm-d.ai/blog/kvcache-wins-you-can-see

[14] Using Docker - vLLM

https://docs.vllm.ai/en/stable/deployment/docker.html

[15] [16] [17] [18] [20] vLLM by Example: Part 2. Deploying vLLM to Docker and then to... | by John Tucker | Oct, 2025 | Medium

https://john-tucker.medium.com/vllm-by-example-part-2-02403becd22d

[21] [22] Speeding Up Large Language Models: A Deep Dive into GPTQ and AWQ Quantization | by Doil Kim | Medium

https://medium.com/@kimdoil1211/speeding-up-large-language-models-a-deep-dive-into-gptq-and-awq-quantization-0bb001eaabd4

[23] 1) vLLM support for GGUF format models is experimental. 2) I think llama.cpp supports batch inferencing. https://github.com/ggml-org/llama.cpp/tree/master/tools/server - Venkinagaraj - Medium

https://medium.com/@venkinagaraj99/1-vllm-support-for-gguf-format-models-is-experimental-a671b9549a23

[24] Optimization and Tuning - vLLM

https://docs.vllm.ai/en/latest/configuration/optimization/

[25] vLLM Performance Tuning: The Ultimate Guide to xPU Inference ...

https://cloud.google.com/blog/topics/developers-practitioners/vllm-performance-tuning-the-ultimate-guide-to-xpu-inference-configuration

[26] Performance Tuning Guide - verl documentation - Read the Docs

https://verl.readthedocs.io/en/latest/perf/perf_tuning.html

[27] vllm.config.scheduler

https://docs.vllm.ai/en/v0.10.2/api/vllm/config/scheduler.html

[28] [35] [45] [46] [47] [56] Releases · vllm-project/vllm · GitHub

https://github.com/vllm-project/vllm/releases

[29] Inside vLLM: Anatomy of a High-Throughput LLM Inference System | vLLM Blog

https://blog.vllm.ai/2025/09/05/anatomy-of-vllm.html

[34] [37] [38] [39] [42] [43] [44] [48] [49] [50] [51] [52] [53] [54] [55] Release notes | Red Hat AI Inference Server | 3.2 | Red Hat Documentation

https://docs.redhat.com/en/documentation/red_hat_ai_inference_server/3.2/html-single/release_notes/release_notes

[36] Distributed Inference and Serving - vLLM

https://docs.vllm.ai/en/v0.8.0/serving/distributed_serving.html