WHITEPAPER

MSAP.ai

AI 필수 추론 가속 엔진 vLLM

온프레미스 실무 엔지니어를 위한 도입·튜닝 백서

모델을 학습하는 일과 학습이 끝난 모델을 서비스로 띄우는 일은 서로 다른 문제입니다. 학습은 정해진 데이터로 수 주에서 수 개월에 걸쳐 한 번 크게 돌리는 배치 작업이지만, 서빙은 길이도 다르고 도착 시각도 제각각인 요청을 실시간으로 끊임없이 받아 처리해야 합니다.

2026-07-05 · 작성자: SMART CF Team · msap.ai

목차

AI 필수 추론 가속 엔진 vLLM

1장: LLM 서빙의 병목과 vLLM 도입의 핵심 근거
- 1.1 오픈 LLM 직접 서빙의 세 가지 문제
- 1.2 vLLM 검토가 필요한 핵심 근거

2장: vLLM의 정의와 개발 연혁
- 2.1 vLLM의 정의와 위치
- 2.2 개발 주체와 등장 배경

3장: 추론 서빙의 기초 개념과 핵심 용어
- 3.1 토큰과 생성 단계
- 3.2 성능을 읽는 두 축

4장: 추론 가속의 원리 — PagedAttention과 연속 배칭
- 4.1 PagedAttention의 메모리 관리
- 4.2 연속 배칭과 스케줄링

5장: vLLM 도입 전후의 처리량과 비용 변화
- 5.1 정량 대비
- 5.2 비용과 운영 영향

6장: 온프레미스 설치와 서버 기동 절차
- 6.1 설치와 사전 조건
- 6.2 서버 기동과 API 확인

7장: Gemma4·Qwen3.6 튜닝 파라미터
- 7.1 메모리와 문맥 파라미터
- 7.2 동시성과 병렬화 파라미터

8장: 경쟁 오픈소스 추론 엔진 비교
- 8.1 엔진별 강점과 약점
- 8.2 선택 기준과 사례

9장: 라이선스 검토와 도입 체크리스트
- 9.1 라이선스 유의점
- 9.2 도입 체크리스트와 다음 단계

참고 문헌 (References)

AI 필수 추론 가속 엔진 vLLM

1장: LLM 서빙의 병목과 vLLM 도입의 핵심 근거

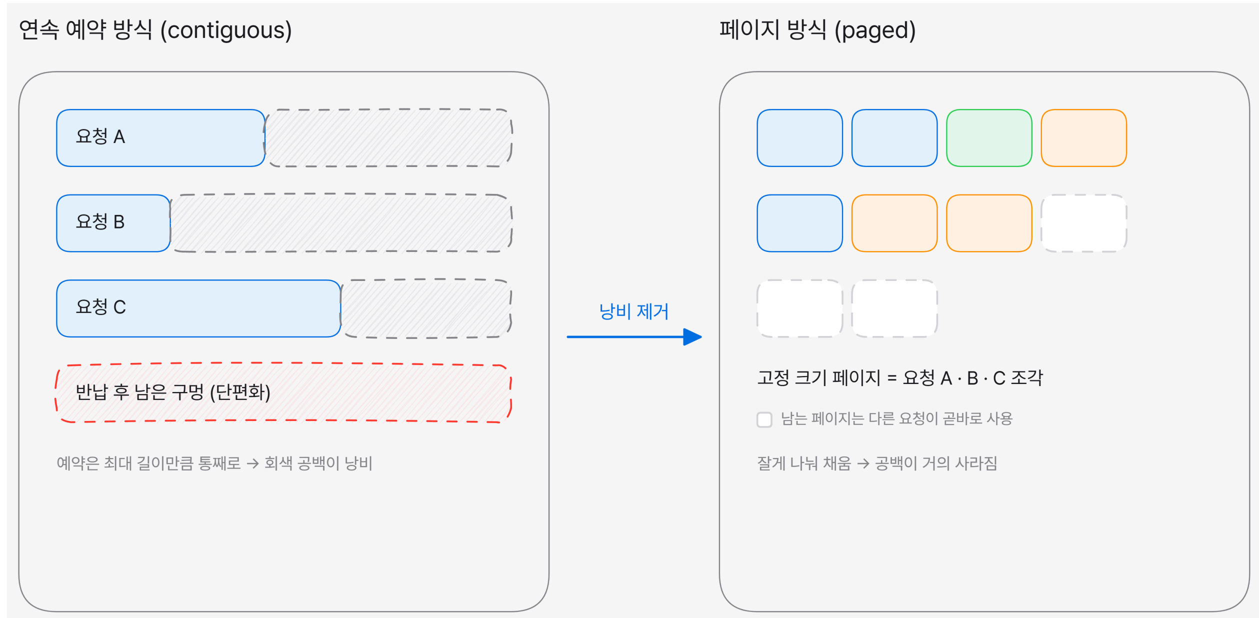
모델을 학습하는 일과 학습이 끝난 모델을 서비스로 띄우는 일은 서로 다른 문제입니다. 학습은 정해진 데이터로 수 주에서 수 개월에 걸쳐 한 번 크게 돌리는 배치 작업이지만, 서빙은 길이도 다르고 도착 시각도 제각각인 요청을 실시간으로 끊임없이 받아 처리해야 합니다. 이 차이를 간과한 채 학습 프레임워크에 딸려 온 순정 추론 코드를 그대로 프로덕션에 올리면, GPU를 넉넉히 사놓고도 실제 처리량은 기대의 몇 분의 일에 그치는 상황을 만나게 됩니다. 이 장은 오픈 LLM을 순정 상태로 서빙할 때 반복해서 나타나는 세 가지 구조적 증상을 먼저 짚고, 그 증상이 왜 곧바로 GPU 비용으로 환산되는지, 그래서 왜 vLLM 같은 전용 서빙 엔진을 검토할 근거가 되는지를 정리합니다. vLLM은 UC Berkeley Sky Computing Lab에서 처음 개발되어 지금은 오픈소스 커뮤니티가 유지관리하는 추론·서빙 엔진이며, 그 핵심 아이디어는 SOSP 2023에서 발표된 PagedAttention 논문에 담겨 있습니다 [S1][S11]. 뒤 장에서 다룰 설치와 튜닝은 모두 이 장에서 정의하는 문제를 해결하기 위한 수단입니다.

1.1 오픈 LLM 직접 서빙의 세 가지 문제

순정 프레임워크로 오픈 LLM을 서빙할 때 엔지니어가 실무에서 가장 먼저 마주치는 증상은 두 가지입니다. 하나는 GPU 메모리가 예상보다 훨씬 빨리 차서 OOM(out of memory)이 나는 것이고, 다른 하나는 GPU 사용률 그래프가 100%를 찍지 못하고 중간중간 골짜기를 만드는 것입니다. 이 둘은 서로 다른 원인에서 나오지만 결과는 같습니다. 같은 카드로 받을 수 있는 동시 요청 수가 줄어들고, 요청당 대기 시간이 늘어납니다. 아래에서는 이 증상의 배후에 있는 KV 캐시 메모리 낭비와 단편화, 그리고 정적 배칭에 따른 GPU 유휴를 현상 그대로 살펴봅니다.

KV 캐시가 메모리를 낭비하는 방식. 트랜스포머 계열 LLM은 토큰을 하나씩 생성할 때마다 앞선 모든 토큰의 키(key)와 값(value) 텐서를 다시 계산하지 않으려고 이를 메모리에 쌓아 둡니다. 이 저장 영역이 KV 캐시입니다. 문제는 순정 구현이 이 KV 캐시를 하나의 요청마다 연속된 큰 메모리 블록으로 미리 잡아 둔다는 데 있습니다. 한 대화가 최대 몇 토큰까지 늘어날지 알 수 없으니, 구현은 안전하게 최대 시퀀스 길이만큼의 공간을 통째로 예약합니다. 그런데 실제 요청은 대부분 그 최대치보다 훨씬 짧게 끝납니다. 짧은 답변 하나에도 최대 길이만큼의 좌석을 예약해 두는 셈이라, 예약된 공간의 상당 부분이 한 번도 쓰이지 않고 비어 있게 됩니다. 게다가 요청마다 예약 크기가 다르니, 요청이 끝나고 반납된 공간은 크기가 제각각인 구멍으로 남습니다. 새 요청이 들어와도 그 구멍들이 이어붙지 않아 넣지 못하는 외부 단편화(external fragmentation)가 생기고, 예약은 했지만 실제로 채우지 못한 내부 단편화(internal fragmentation)도 함께 쌓입니다. PagedAttention은 바로 이 지점을 겨냥합니다. 운영체제의 가상 메모리 페이징처럼 KV 캐시를 고정 크기의 작은 블록으로 쪼개어 필요할 때마다 할당함으로써, 예약해 놓고 버리는 공간을 거의 남기지 않는 near-zero waste에 도달하고, 나아가 같은 프롬프트를 공유하는 요청들 사이에서 KV 캐시를 재사용하기까지 합니다 [S11]. 순정 구현에서 "우리 GPU가 왜 이렇게 금방 OOM이 나는가"라는 질문의 답은 대체로 이 낭비와 단편화에 있습니다. 그리고 이 낭비는 추상적인

비효율에 그치지 않습니다. 같은 모델을 같은 동시성으로 받으려는데 메모리가 부족하면 남는 선택지는 GPU를 더 사는 것뿐이고, 낭비는 곧 카드 추가 구매로 이어집니다 [S1].

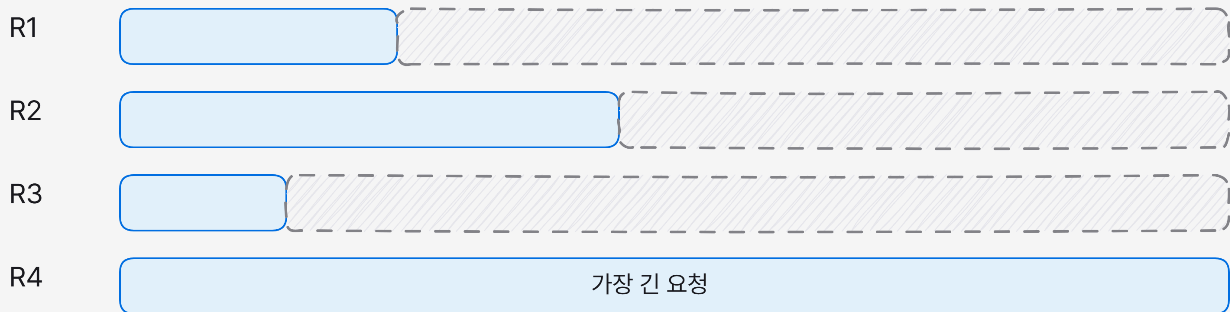


연속 예약 방식과 페이지 기반 KV 캐시 할당의 메모리 점유 구조 비교.

정적 배칭이 GPU를 놀게 하는 방식. 두 번째 증상은 처리 순서에서 나옵니다. GPU는 요청을 하나씩 처리하는 것보다 여러 요청을 한 배치로 묶어 처리할 때 훨씬 효율적입니다. 그래서 순정 서빙도 요청을 모아 배치로 돌리는데, 여기서 정적 배칭(static batching)은 한 배치를 시작하면 그 배치에 묶인 모든 요청이 끝날 때까지 다음 배치를 시작하지 않습니다. 문제는 LLM 요청의 생성 길이 편차가 매우 크다는 데 있습니다. 어떤 요청은 스무 토큰이면 끝나지만 어떤 요청은 수천 토큰을 이어 씁니다. 정적 배칭에서는 짧은 요청들이 진작 끝났는데도 가장 긴 요청 하나가 마무리될 때까지 자리를 비운 채 기다려야 합니다. 그 사이 GPU의 상당수 연산 슬롯은 아무 일도 하지 않고 놉니다. 요청 길이 편차가 큰 실제 트래픽일수록 이 유휴는 심해지고, 사용률 그래프의 골짜기로 나타납니다. vLLM이 채택한 연속 배칭(continuous batching)은 배치가 통째로 끝나기를 기다리지 않고, 요청 하나가 끝나면 그 자리에 대기 중인 새 요청을 곧바로 끼워 넣어 GPU를 계속 채웁니다. vLLM V1 엔진은 여기에 더해 긴 프롬프트를 잘게 나눠 생성과 함께 처리하는 chunked prefill을 기본으로 켜 두어, 새 요청의 프롬프트 처리와 기존 요청의 토큰 생성이 서로를 오래 막지 않도록 조율합니다 [S2]. 유휴 시간은 그 자체로 처리량 손실이고, 처리량이 떨어지면 요청 하나를 처리하는 데 드는 GPU 시간이 늘어나 단가가 올라갑니다 [S4]. 즉 배칭 방식의 선택은 성능 튜닝 이전에 원가 구조의 문제입니다.

정적 배칭 (static batching)

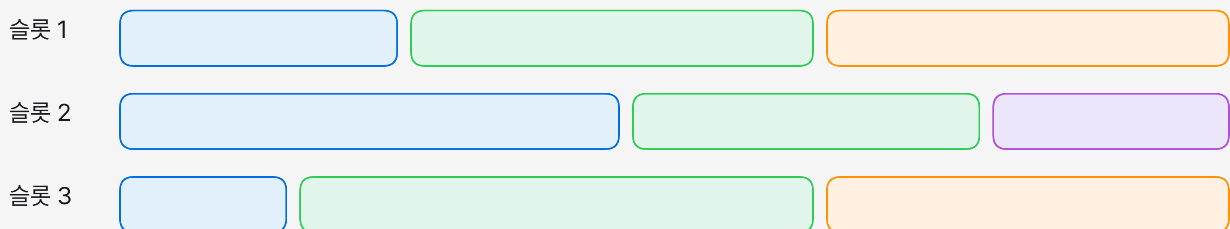
가장 긴 요청이 끝날 때까지 짧은 요청 자리가 유휴로 남음



▲ 빗금 = GPU 유휴 (처리량 손실)

연속 배칭 (continuous batching)

요청이 끝나는 즉시 대기 큐의 새 요청이 그 자리를 채움



▲ 색이 바뀌는 지점 = 끝난 자리에 새 요청 즉시 삽입 → 유휴 거의 없음

시간 →

정적 배칭과 연속 배칭의 GPU 점유 타임라인 비교.

1.2 vLLM 검토가 필요한 핵심 근거

앞 절의 두 증상은 엔지니어에게는 메모리와 사용률 그래프의 문제로 보이지만, 도입을 승인해야 하는 사람에게는 결국 하나의 질문으로 수렴합니다. "같은 GPU로 더 많은 요청을 받을 수 있는가." 이 절은 메모리 낭비와 유휴가 어떻게 비용과 동시 처리량으로 환산되는지를 요약하고, 바쁜 독자가 이 백서에서 자기에게 필요한 장으로 빠르게 이동할 수 있도록 읽기 경로를 안내합니다.

낭비와 유휴는 곧 비용과 처리량입니다. GPU는 도입할 때 카드 값을 한 번 치르고 끝나는 자산이 아니라, 살아 있는 동안 전력과 상면, 냉각을 계속 소모하는 고정비입니다. 그래서 한 장의 GPU가 단위 시간에 처리하는 요청 수, 즉 처리량이 사실상 요청 한 건의 단가를 결정합니다. 앞 절의 KV 캐시 낭비는 같은 메모리에 올릴 수 있는 동시 요청 수를 깎아 처리량의 상한을 끌어내리고, 정적 배칭의 유휴는 이미 올린 요청조차 GPU를 충분히 활용하지 못하게 만들어 실효 처리

량을 다시 깎습니다. 두 손실이 겹치면 카드는 놀고 있는데 대기 큐는 쌓이는, 겉보기에 모순되어 보이는 상황이 벌어집니다. PagedAttention 논문은 vLLM이 동일한 지연 조건에서 앞선 서빙 시스템 대비 2~4배 높은 처리량을 낸다고 보고하며 [S11], Red Hat은 순정 HuggingFace Transformers와 비교해 최대 24배까지 처리량이 오른 사례를 인용합니다 [S1]. 다만 이 두 수치는 비교 기준이 다릅니다. 2~4배는 이미 최적화된 서빙 시스템과의 대비이고, 24배는 아무 최적화도 하지 않은 순정 구현과의 대비이므로, 실제 자기 환경에 대입할 때는 어떤 기준선과 비교한 값인지를 반드시 함께 봐야 합니다. 처리량이 몇 배 오른다는 말은 뒤집으면 같은 처리량을 몇 분의 일의 GPU로 낼 수 있다는 뜻이고, 이것이 도입 판단의 출발점입니다. 구체적인 정량화와 자기 워크로드 기준의 벤치마크는 뒤 5장에서 조건과 함께 다룹니다.

독자별 읽기 경로. 이 백서는 두 부류의 독자를 함께 상정합니다. 온프레미스에서 실제로 모델을 올리고 튜닝하는 엔지니어는 전 장을 순서대로 읽는 것을 권합니다. 특히 GPU 메모리 활용률과 최대 시퀀스 길이, 배치 토큰 예산 같은 실행 파라미터를 다루는 7장과, 자기 워크로드로 성능을 재는 5장이 실무의 중심입니다. 반면 도입을 승인하고 예산을 배정하는 의사결정권자라면 문제와 비용 구조를 정의한 이 1장, 정량 효과를 다루는 5장, 그리고 다른 서빙 엔진과의 선택 기준을 정리한 9장만 읽어도 판단에 필요한 재료를 얻을 수 있습니다. 각 장의 도입 문단은 두 독자가 모두 읽을 수 있도록 기술 세부에 앞서 그 장이 답하려는 질문을 먼저 밝혀 두었습니다. 참고로 vLLM 엔진 자체는 Apache License 2.0으로 배포되지만, 그 위에 올려 서빙하는 모델 가중치의 라이선스는 별개이므로 도입 검토 단계에서 각 모델의 이용약관을 따로 확인해야 합니다 [S10].

2장: vLLM의 정의와 개발 연혁

이 장은 vLLM이 정확히 무엇인지, 그리고 누가 언제 어떤 이유로 만들었는지를 사실 그대로 정리합니다. 도입을 검토하는 조직에서 가장 먼저 정리해야 할 질문은 "이 도구가 우리 스택의 어디에 들어가는가"입니다. vLLM은 이미 학습을 마친 대규모 언어 모델을 운영 환경에서 빠르고 저렴하게 서빙하는 오픈소스 추론 엔진입니다. 학습이나 파인튜닝을 대체하는 도구가 아니라, 그 결과물을 실제 서비스로 연결하는 서빙 계층에 위치합니다. 이 구분을 명확히 해 두면 도입 범위를 과대·과소 산정하는 실수를 피할 수 있습니다.

이어지는 두 절은 각각 vLLM의 정의와 기술 스택 내 위치를 다루고(2.1), 개발 주체와 등장 배경을 연구 근거 중심으로 정리합니다(2.2). 특히 vLLM의 성능 우위가 마케팅 문구가 아니라 UC Berkeley의 연구에서 출발했다는 점은 온프레미스 도입을 정당화하는 핵심 근거이므로, 창시 주체와 논문 사실에 출처를 붙여 서술합니다. 전체적으로 이 장은 판단의 재료를 제공할 뿐 설득을 목적으로 하지 않으며, 수치와 조건은 확인된 범위 안에서만 인용합니다.

2.1 vLLM의 정의와 위치

vLLM을 한 문장으로 정의하고, 학습 프레임워크와의 경계를 긋는 절입니다. 많은 팀이 새 도구를 검토할 때 그것이 파이프라인의 어느 단계를 담당하는지부터 혼동하는데, vLLM은 그 경계가 비교적 뚜렷합니다. 이 절에서는 먼저 서빙 전용 엔진이라는 정의를 정하고, 뒤이어 학습·파인튜닝과 서빙의 역할 차이를 대비해 도입 범위의 오해를 줄입니다.

추론 서빙 전용 엔진이라는 정의

vLLM은 이미 학습된 대규모 언어 모델을 빠르고 저렴하게 서빙하는 오픈소스 추론 엔진입니다. 이 한 문장이 vLLM의 성격을 가장 압축적으로 담고 있습니다. 여기서 추론(inference)이란 학습이 끝난 모델에 프롬프트를 입력해 응답 토큰을 생성하는 실행 단계를 가리킵니다. vLLM은 이 실행 단계를 GPU 위에서 효율적으로 처리하는 데 초점을 맞춘 서버 소프트웨어이며, 모델 자체를 만들거나 개선하는 도구가 아닙니다 [S1][S10].

엔진이라는 표현은 vLLM이 단순한 라이브러리 이상의 실행 계층이라는 의미를 담습니다. vLLM은 요청을 받아 배치로 묶고, GPU 메모리를 관리하고, 생성된 토큰을 스트리밍으로 돌려주는 일련의 처리를 자체적으로 수행합니다. 특히 KV 캐시 메모리를 운영체제의 페이징 기법처럼 다루는 PagedAttention 기법을 적용해 메모리 낭비를 거의 0에 가깝게 줄이고, 요청 내부와 요청 사이에서 KV 캐시를 공유합니다 [S11]. 이 메모리 관리 방식이 vLLM을 여느 서빙 도구와 구분 짓는 기술적 핵심입니다.

실무 관점에서 vLLM은 OpenAI 호환 API를 제공하기 때문에, 기존에 상용 API를 호출하던 애플리케이션 코드를 크게 바꾸지 않고도 자체 인프라의 모델로 전환할 수 있습니다. 온프레미스 엔지니어에게 이 정의가 중요한 이유는 도입 검토의 초점이 "어떤 모델을 만들 것인가"가 아니라 "이미 확보한 모델을 어떻게 안정적으로 서비스할 것인가"로 좁혀지기 때문입니다. Red Hat은 순정 Hugging Face Transformers 대비 최대 24배까지 처리량이 향상된다고 인용하지만, 이 수

치는 순정 실행 대비 기준이라는 조건을 함께 읽어야 정확합니다 [S1]. 처리량 향상의 성격과 조건은 3장 이후에서 정량적으로 다룹니다.

학습 프레임워크와의 구분

vLLM의 위치를 오해 없이 잡으려면 학습·파인튜닝 단계와 서빙 단계를 나누어 볼 필요가 있습니다. 대규모 언어 모델을 실제 서비스로 만드는 과정은 크게 두 국면으로 나뉩니다. 하나는 데이터로 모델의 가중치를 만들거나 조정하는 학습·파인튜닝 국면이고, 다른 하나는 그렇게 완성된 모델을 사용자 요청에 응답시키는 서빙 국면입니다. vLLM은 이 중 두 번째 국면만을 담당합니다 [S1].

학습 국면에서는 PyTorch, DeepSpeed, Hugging Face Transformers의 학습 API, Megatron-LM 같은 프레임워크가 사용됩니다. 이들은 역전파와 옵티마이저 상태 관리, 분산 학습을 다루며, 산출물은 새로운 모델 가중치입니다. 반면 서빙 국면에서 vLLM이 다루는 것은 이미 고정된 가중치이며, 관심사는 지연 시간과 처리량, GPU 메모리 효율입니다. 아래 표는 두 국면의 역할 차이를 정리한 것입니다.

구분	학습·파인튜닝	서빙(vLLM 담당)
목적	모델 가중치 생성·조정	완성된 모델로 응답 생성
대표 도구	PyTorch·DeepSpeed·Transformers 학습 API	vLLM
산출물	새 모델 가중치	사용자 응답 토큰
주요 지표	손실·정확도·수렴	지연 시간·처리량·메모리 효율
실행 특성	역전파·옵티마이저	순전파 추론·배치·KV 캐시 관리

이 구분에서 도출되는 실무적 함의가 하나 있습니다. vLLM을 도입한다고 해서 모델을 자체적으로 만들거나 개선하는 역량이 생기는 것은 아니라는 점입니다. 파인튜닝이 필요한 조직은 별도의 학습 파이프라인을 유지해야 하며, vLLM은 그 결과물을 이어받아 운영하는 역할에 집중합니다. 또한 vLLM 엔진 자체는 Apache License 2.0으로 배포되지만, 이 라이선스는 엔진에만 적용됩니다 [S10]. 실제로 서빙하는 모델 가중치의 이용 조건은 각 모델의 약관을 별도로 확인해야 합니다. 예컨대 Gemma나 Qwen 계열 모델은 저마다의 이용약관을 따르므로, 도입 범위를 산정할 때 엔진 라이선스와 모델 라이선스를 분리해 검토하는 편이 안전합니다.

2.2 개발 주체와 등장 배경

이 절은 vLLM을 누가 만들었고 어떤 연구가 그 기원이 되었는지, 그리고 지금은 어떤 구조로 유지되는지를 사실 중심으로 정리합니다. IT 담당자에게 이 정보가 중요한 이유는 도구의 지속성과 신뢰 근거를 판단하는 재료이기 때문입니다. 성능 주장이 특정 벤더의 마케팅이 아니라 공개된 학술 연구에서 출발했다는 사실, 그리고 오늘날 특정 기업에 종속되지 않은 커뮤니티가 유지관리한다는 사실은 온프레미스 장기 운영을 검토할 때 안정성 근거로 작용합니다.

UC Berkeley Sky Computing Lab과 PagedAttention 논문

vLLM은 UC Berkeley의 Sky Computing Lab에서 최초로 개발되었습니다 [S1][S10]. 기술적 기원은 PagedAttention이라는 메모리 관리 기법을 제안한 논문으로 거슬러 올라갑니다. 해당 논문의 제목은 *"Efficient Memory Management for Large Language Model Serving with PagedAttention"*이며, 대표 저자는 Woosuk Kwon입니다. 공저자로는 Zhuohan Li, Ying Sheng, Lianmin Zheng, 그리고 Ion Stoica 등이 참여했고, 이 논문은 시스템 분야의 최고 학회 중 하나인 SOSP 2023에서 발표되었습니다 [S11]. 아래 표는 창시와 논문에 관해 확인된 사실을 정리한 것입니다.

항목	내용
최초 개발 주체	UC Berkeley Sky Computing Lab [S1][S10]
기원 논문	*Efficient Memory Management for Large Language Model Serving with PagedAttention* [S11]
대표 저자	Woosuk Kwon (공저 Zhuohan Li·Ying Sheng·Lianmin Zheng·Ion Stoica 등) [S11]
발표 학회	SOSP 2023 [S11]
엔진 라이선스	Apache License 2.0 [S10]

PagedAttention의 핵심 착상은 운영체제의 가상 메모리 페이징에서 왔습니다. 기존 서빙 방식은 각 요청의 KV 캐시를 연속된 메모리 블록에 미리 잡아 두어 상당한 낭비가 발생했으나, PagedAttention은 KV 캐시를 작은 페이지 단위로 나눠 필요할 때만 할당함으로써 낭비를 거의 0에 가깝게 줄였습니다 [S11]. 이 논문은 동일한 지연 조건에서 FasterTransformer와 Orca 같은 당시 대표 서빙 시스템 대비 2~4배의 처리량을 보고했습니다 [S11]. IT 담당자 관점에서 이 대목의 의미는 분명합니다. vLLM의 성능 우위는 제품 홍보에서 나온 주장이 아니라 공개 검증을 거친 학술 연구에 근거를 두고 있으며, 논문과 코드 저장소가 모두 공개되어 있어 주장의 출처를 직접 확인할 수 있습니다.

PagedAttention 논문(SOSP 2023)에서 vLLM 엔진 공개, 그리고 커뮤니티 유지 구조로 이어지는 연혁 흐름.

오픈소스 커뮤니티 유지 구조

vLLM은 UC Berkeley에서 출발했지만, 현재는 특정 기업이나 연구실이 단독으로 소유하는 프로젝트가 아니라 오픈소스 커뮤니티가 유지관리하는 프로젝트입니다 [S1][S10]. 엔진은 Apache License 2.0으로 배포되어 상업적 활용과 수정, 재배포에 폭넓은 자유를 허용합니다 [S10]. 이 라이선스 조건은 온프레미스 환경에서 소스를 직접 빌드하거나 필요에 맞게 패치해 운영하려는 조직에게 특히 우호적입니다.

커뮤니티 유지 구조가 도입 안정성 측면에서 갖는 의미는, 프로젝트의 존속이 한 벤더의 사업 판단에 좌우되지 않는다는 점입니다. 이 대목은 경쟁 서빙 도구와 비교할 때 더 뚜렷해집니다. Hugging Face의 Text Generation Inference(TGI)는 현재 유지보수 모드로 전환되어 마이너 수정과 문서 갱신 위주로만 관리되며, Hugging Face 자체가 프로덕션 서빙에 vLLM, SGLang, llama.cpp, MLX 같은 대안을 권장하는 상황입니다 [S6]. 이처럼 활발한 개발이 진행되는 프로젝트와 유지보수 단계에 들어선 프로젝트를 구분하는 것은 장기 운영 계획에서 중요한 판단 기준이 됩니다.

실무 담당자는 도입 전에 공개 코드 저장소에서 릴리스 이력과 기여 활동을 직접 확인할 것을 권합니다. 저장소의 릴리스 노트는 어떤 모델 아키텍처가 새로 지원되었는지, 어떤 성능 개선과 버그 수정이 반영되었는지를 사실로 보여 주며, 이는 마케팅 자료보다 프로젝트의 건강도를 정확하게 드러냅니다. 정리하면 vLLM은 검증된 연구에서 출발해 개방형 라이선스로 배포되고 커뮤니티가 지속적으로 개선하는 서빙 엔진이며, 이 세 가지 성격이 온프레미스 도입의 기술적·조직적 안정성을 함께 뒷받침합니다.

3장: 추론 서빙의 기초 개념과 핵심 용어

이 장은 뒤이어 다룰 PagedAttention 원리(4장)와 파라미터 튜닝(7장)을 읽어 나가기 위한 용어의 토대를 정리합니다. LLM 추론 서빙에서 반복해서 등장하는 개념 몇 가지는 이름만 보면 추상적이지만, 실제로는 GPU 메모리와 응답 속도를 좌우하는 매우 구체적인 대상입니다. 여기서는 각 용어를 한 줄 정의로 분명히 하고, 실무 감각을 잡을 수 있는 비유를 곁들여 설명합니다. 비유는 이해를 돕기 위한 보조 장치일 뿐이며, 뒷장에서는 같은 개념을 수치와 명령으로 다시 다룹니다. 이 장에서 다지는 개념은 크게 두 묶음입니다. 하나는 모델이 텍스트를 처리하는 단위와 단계이고, 다른 하나는 그 처리의 성능을 읽는 두 축입니다.

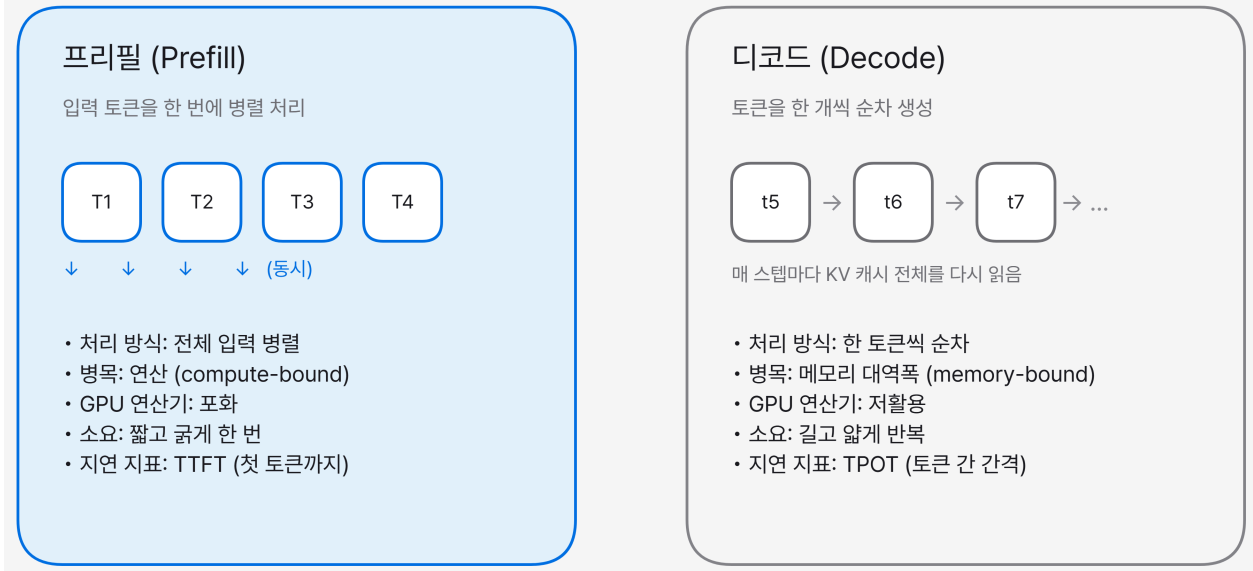
3.1 토큰과 생성 단계

LLM은 문장을 통째로 다루지 않고 토큰(token)이라는 작은 단위로 쪼개어 처리합니다. 토큰은 대략 글자 조각에 해당하는 단위로, 한 단어가 여러 토큰으로 나뉘기도 하고 짧은 단어 하나가 한 토큰이 되기도 합니다. 모델의 입력과 출력은 모두 이 토큰의 열로 표현되며, 서빙 시스템이 부담하는 계산량과 메모리는 결국 이 토큰이 몇 개인지에 따라 결정됩니다. 이 절에서는 토큰을 다루는 두 단계인 프리필과 디코드를 구분하고, 그 과정에서 문맥을 기억하는 KV 캐시가 왜 메모리의 주 소비원이 되는지를 정리합니다.

3.1.1 토큰과 프리필·디코드

한 번의 추론 요청은 성격이 뚜렷이 다른 두 단계를 거칩니다. 첫 단계는 프리필(prefill)로, 사용자가 넣은 입력 프롬프트 전체를 한 번에 읽어 들여 내부 상태를 만드는 단계입니다. 둘째 단계는 디코드(decode)로, 그 상태를 바탕으로 답변을 한 토큰씩 순차적으로 생성하는 단계입니다. 프리필은 입력에 담긴 여러 토큰을 병렬로 한꺼번에 계산하므로 GPU의 연산 능력을 채워 쓰는 계산 중심 단계입니다. 반면 디코드는 이전에 만든 토큰에 이어 다음 토큰 하나를 만드는 일을 반복하므로, 매 걸음마다 메모리에서 앞선 문맥을 읽어 오는 메모리 대역폭 중심 단계가 됩니다 [S1].

두 단계의 자원 소모 방식이 다르다는 사실은 튜닝의 출발점입니다. 프리필은 긴 입력을 한꺼번에 처리하느라 순간적으로 연산이 몰리고, 디코드는 출력 길이만큼 걸음을 반복하느라 시간이 길어집니다. 그래서 첫 응답이 나오기까지의 시간과, 그 뒤로 토큰이 이어지는 속도는 서로 다른 요인에 지배됩니다. vLLM의 V1 엔진은 이 두 단계를 한 배치 안에서 섞어 처리하는 chunked prefill을 기본으로 활성화해, 긴 프리필이 디코드 걸음을 오래 막아 세우는 상황을 완화합니다 [S2]. 이 배경을 알아 두면 7장에서 배치 토큰 예산 같은 파라미터를 조정할 때 어느 단계를 겨냥한 조정인지 분간할 수 있습니다.

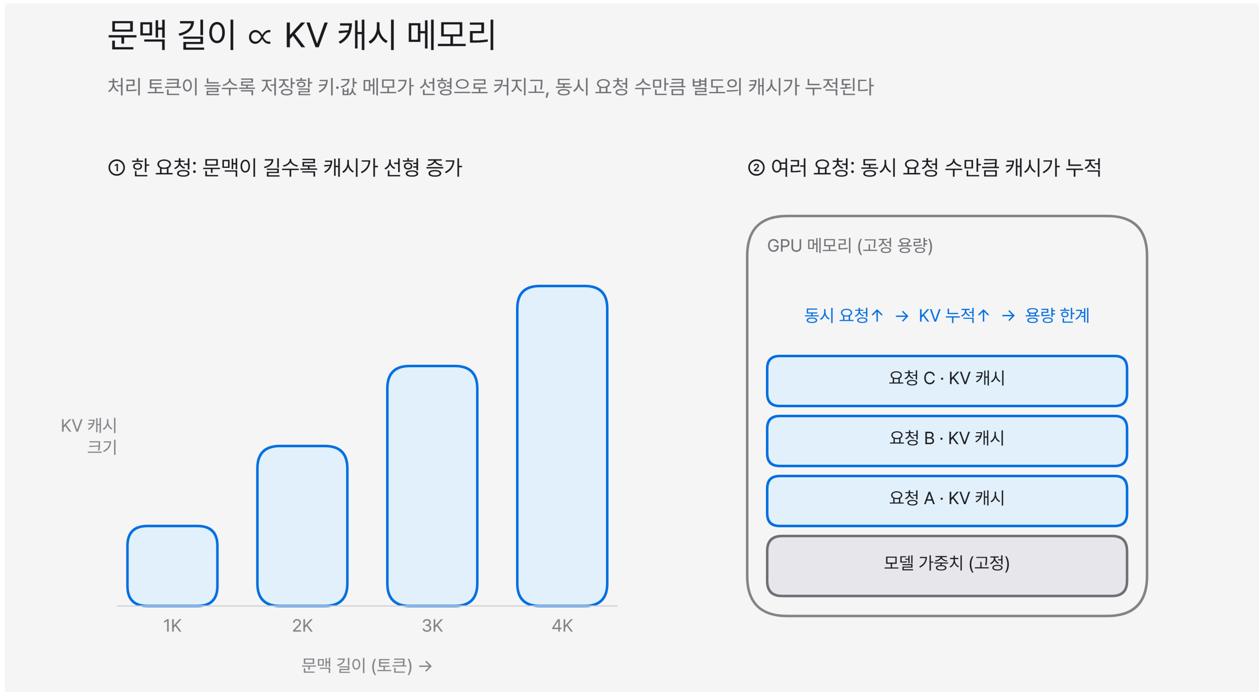


프리필과 디코드의 계산 특성 대비 — 프리필은 입력 토큰을 병렬로 한 번에 읽는 계산 중심 단계, 디코드는 토큰을 한 개씩 이어 쓰는 메모리 중심 단계.

3.1.2 KV 캐시의 역할과 메모리 비례

디코드가 매 걸음 앞선 문맥을 다시 읽어 온다고 했는데, 그 문맥을 담아 두는 저장소가 바로 KV 캐시(KV cache)입니다. KV 캐시는 앞서 처리한 토큰들의 계산 결과(키와 값)를 보관하는 메모리로, 다음 토큰을 만들 때 앞의 모든 토큰을 처음부터 다시 계산하지 않도록 해 줍니다. 대화가 이어지는 동안 지금까지의 맥락을 적어 두는 대화 메모라고 이해하면 됩니다. 이 메모가 있어야 디코드는 걸음마다 전체를 되짚지 않고 다음 한 토큰만 계산하면 됩니다.

핵심은 이 KV 캐시의 크기가 문맥 길이에 비례해서 커진다는 점입니다. 다루는 토큰이 많아질수록, 즉 프롬프트가 길고 생성이 길수록 적어 둘 메모가 그만큼 늘어납니다. 그래서 KV 캐시는 GPU 메모리를 가장 많이 잡아먹는 주된 소비원이 되며, 동시에 여러 요청을 받을수록 요청마다 별도의 메모가 쌓여 부담이 가중됩니다 [S1]. 이 지점이 바로 vLLM이 해결하려는 문제의 정중앙입니다. 4장에서 다룰 PagedAttention은 이 KV 캐시를 잘게 나눠 관리해 거의 낭비 없이(near-zero waste) 쓰고 요청 사이에 공유까지 가능하게 만드는 기법입니다 [S11]. 지금은 KV 캐시가 문맥 길이에 비례해 부풀고 그만큼 GPU 메모리를 지배한다는 사실만 확실히 잡아 두면 됩니다. 7장에서 `--max-model-len`으로 한 요청이 쓸 수 있는 최대 시퀀스 길이를 정하는 것도, `--kv-cache-dtype fp8`로 KV 캐시 자체를 압축 저장해 메모리를 약 절반으로 줄이는 것도 모두 이 비례 관계를 손보는 조정입니다 [S12].



문맥 길이와 KV 캐시 크기의 비례 관계 — 처리 토큰이 늘수록 저장할 키·값 메모가 선형으로 증가하고, 동시 요청 수만큼 별도의 캐시가 누적된다.

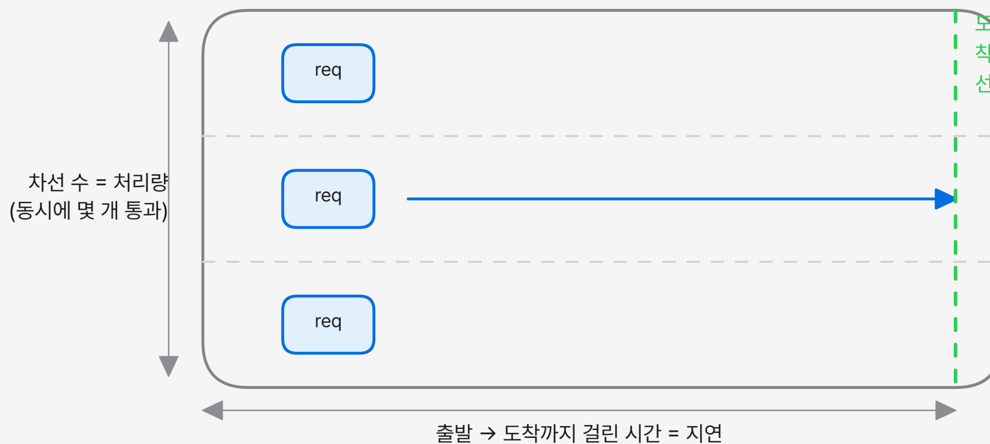
3.2 성능을 읽는 두 축

추론 서빙의 성능은 하나의 숫자로 요약되지 않습니다. 같은 시스템이라도 어느 관점에서 보느냐에 따라 좋게도 나쁘게도 읽힙니다. 이 절에서는 성능을 읽는 두 개의 독립된 축인 처리량과 지연을 구분하고, 그 둘을 잇는 동시성과 배치의 관계를 정리합니다. 이 구분을 명확히 해 두면 뒷장에서 소개하는 성능 수치가 정확히 어느 축의 이득인지 오해 없이 읽을 수 있습니다.

3.2.1 처리량과 지연의 구분

처리량(throughput)은 시스템이 단위 시간에 처리하는 토큰이나 요청의 총량이고, 지연(latency)은 한 요청이 처리되는 데 걸리는 시간입니다. 둘은 서로 다른 축이라 한쪽이 좋다고 다른 쪽까지 좋다고 볼 수 없습니다. 고속도로에 빗대면 처리량은 차선 수에 가깝고 지연은 한 대가 목적지에 도착하기까지의 시간에 가깝습니다. 차선을 넓히면 같은 시간에 더 많은 차가 지나가지만, 그렇다고 개별 차량의 도착 시간이 반드시 빨라지지는 않습니다.

이 구분이 실무에서 중요한 이유는 vLLM이 내세우는 이득이 주로 처리량 축에 있기 때문입니다. vLLM은 동일 지연 조건에서 앞선 서빙 시스템 대비 2~4배의 처리량을 보고했고, 순정 HF Transformers와 견주면 최대 24배까지 언급됩니다 [S11][S1]. 다만 이 두 수치는 비교 기준이 다릅니다. 2~4배는 이미 고도로 최적화된 서빙 시스템과의 대비이고, 24배는 최적화되지 않은 순정 실행과의 대비입니다 [S11][S1]. 그래서 성능 수치를 읽을 때는 어느 축(처리량인지 지연인지)의 값인지, 무엇과 견준 값인지를 함께 확인해야 합니다. 이 습관은 5장 이후 벤치마크를 해석할 때 특히 유용합니다.

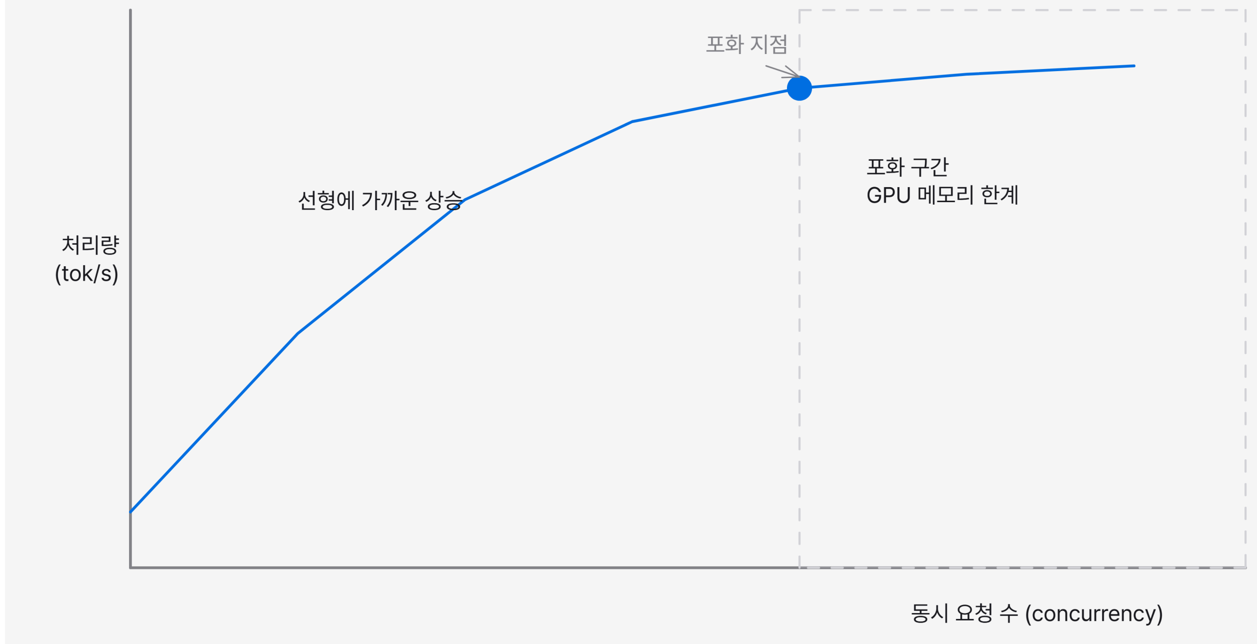


처리량과 지연의 독립성 — 처리량은 단위 시간당 처리량(차선 수), 지연은 한 요청의 소요 시간(도착 시간)으로 서로 다른 축을 이룬다.

3.2.2 동시성과 배치의 관계

앞의 두 축을 잇는 개념이 동시성과 배치입니다. 동시성은 시스템이 같은 시점에 처리하는 요청의 수이고, 배치(batch)는 그 여러 요청을 한 묶음으로 모아 GPU에서 함께 계산하는 방식입니다. 식당에 빗대면 동시에 받는 테이블 수가 동시성이고, 그 테이블들의 주문을 한 번에 조리하는 것이 배치입니다. 테이블을 더 받으면 주방은 바빠지지만 재료와 화구를 효율적으로 나눠 써 시간당 처리하는 손님은 늘어납니다.

배치를 키우면 GPU가 놀지 않고 계산을 채워 처리량이 오르지만, 그만큼 요청마다 KV 캐시가 쌓여 메모리 부담이 함께 커집니다. 즉 동시성은 처리량과 메모리를 동시에 밀어 올리는 손잡이라, 어디까지 키울지는 GPU 메모리 한도 안에서 정해집니다. vLLM V1 엔진은 배치 안 동시 요청 수의 기본값(`--max-num-seqs`)을 1024로 두어 V0의 256보다 넉넉하게 잡습니다 [S12]. 실제로는 모델 크기와 메모리에 맞춰 이 값을 낮추기도 하는데, 앞의 온프렘 예시에서 단일 GPU에 큰 모델을 올릴 때 이 값을 8까지 좁히는 것이 그런 조정입니다 [S12]. 동시성을 올릴수록 처리량은 어느 지점까지 가파르게 오르다가 메모리 한계에 다가서면서 완만해지는데, 이 곡선의 모양을 감각으로 잡아 두면 7장의 동시성 파라미터 튜닝을 훨씬 수월하게 이해할 수 있습니다.



동시성과 처리량의 관계 개념도 — 동시 요청 수를 늘리면 처리량이 오르다가 GPU 메모리 한계 부근에서 증가폭이 둔화된다.

4장: 추론 가속의 원리 — PagedAttention과 연속 배칭

vLLM이 같은 GPU에서 더 많은 요청을 더 빠르게 처리하는 근거는 두 가지 손실을 각각 제거하는 데 있습니다. 하나는 KV 캐시가 차지하는 GPU 메모리의 낭비이고, 다른 하나는 배치 처리 과정에서 발생하는 GPU 연산 자원의 유휴입니다. 이 장은 두 손실이 왜 생기는지, 그리고 vLLM이 각각을 어떤 방식으로 없애는지를 원리 수준에서 설명합니다. PagedAttention은 메모리 낭비를 해결하고, 연속 배칭은 GPU 유휴를 해결합니다. 두 기법은 독립적으로 동작하지만, 함께 적용될 때 처리량과 동시성이 크게 올라갑니다. 앞의 3장에서 KV 캐시가 무엇이고 왜 메모리를 점유하는지를 다뤘다면, 이 장은 그 KV 캐시를 낭비 없이 담아내는 방법과, 담긴 요청들을 쉼 없이 굴리는 방법을 이어서 풀어냅니다.

4.1 PagedAttention의 메모리 관리

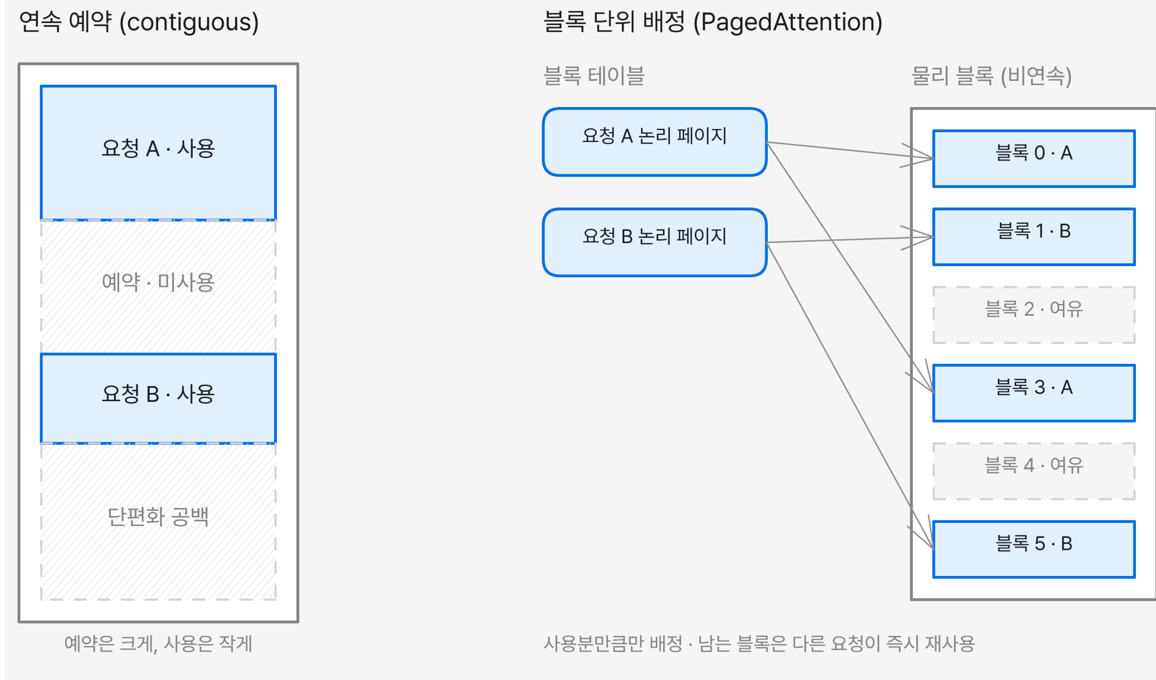
KV 캐시는 각 요청이 생성하는 토큰마다 조금씩 늘어나는 자료입니다. 문제는 요청이 시작될 때 최종 길이를 알 수 없다는 점입니다. 전통적인 서빙 시스템은 이 불확실성에 대비해 요청이 도달할 수 있는 최대 길이만큼의 메모리를 연속된 한 덩어리로 미리 예약했습니다.
PagedAttention은 이 예약 방식 자체를 바꿉니다. UC Berkeley Sky Computing Lab에서 발표한 논문 *"Efficient Memory Management for Large Language Model Serving with PagedAttention"*(SOSP 2023, 대표 저자 Woosuk Kwon)이 이 기법의 1차 근거입니다[S11]. 이 절에서는 그 관리 방식을 운영체제의 가상 메모리 페이징에 빗대어 설명하고, 그 결과로 낭비가 어떻게 거의 사라지는지를 다룹니다.

4.1.1 가상 메모리 페이징 비유

전통적 방식의 손실은 예약 단위가 크고 연속이라는 데서 나옵니다. 한 요청이 최대 4,096토큰까지 자랄 수 있다고 하면, 실제로 512토큰만 쓰고 끝나도 나머지 3,584토큰 몫의 메모리는 그 요청이 살아 있는 동안 다른 요청이 손댈 수 없습니다. 이렇게 예약해 두고 쓰지 않는 공간이 곧 내부 단편화입니다. 게다가 연속된 한 덩어리를 요구하기 때문에, 자잘한 빈 공간이 여기저기 흩어져 있어도 큰 요청 하나를 담을 자리가 없으면 받지 못하는 외부 단편화까지 겹칩니다.

PagedAttention은 운영체제가 물리 메모리를 다루는 방식을 KV 캐시에 그대로 적용합니다[S11][S1]. 운영체제는 프로세스에 메모리를 통째로 연속으로 주지 않습니다. 대신 메모리를 고정 크기 페이지로 잘게 나누고, 프로세스가 요구하는 만큼만 페이지를 배정하며, 그 페이지들이 물리적으로 떨어져 있어도 페이지 테이블로 논리 주소와 물리 주소를 이어 붙입니다. vLLM은 KV 캐시를 고정 크기 블록(page) 단위로 나누고, 요청이 토큰을 생성하는 만큼만 블록을 하나씩 배정합니다. 배정된 블록들은 GPU 메모리 안에서 연속일 필요가 없습니다. 어느 블록이 어느 요청의 몇 번째 구간에 해당하는지는 블록 테이블이 관리하며, 어텐션 연산은 이 테이블을 따라 흩어진 블록을 읽어 계산합니다[S11][S9].

이 전환의 효과는 두 단편화를 동시에 없앤다는 데 있습니다. 요청은 실제로 쓰는 만큼만 블록을 받으므로 미리 예약해 두고 노는 공간이 사라지고, 블록이 연속일 필요가 없으므로 흩어진 빈 블록도 모두 재사용됩니다.



전통적 연속 예약 방식과 PagedAttention의 블록 방식을 나란히 대비하는 도식. 왼쪽은 요청마다 최대 길이만큼 연속 예약해 회색 빈 칸(단편화)이 크게 남는 모습, 오른쪽은 KV 캐시를 고정 크기 블록으로 나눠 실제 사용분만큼만 배정하고 블록 테이블이 흩어진 블록을 잇는 모습.

4.1.2 KV 캐시 공유와 near-zero 낭비

블록 단위 관리가 주는 두 번째 이점은 공유입니다. 서로 다른 요청이 같은 내용의 KV 캐시를 필요로 할 때, 전통적 방식은 각 요청이 자기 몫을 따로 계산해 따로 저장했습니다. PagedAttention에서는 같은 내용을 담은 블록을 여러 요청이 함께 가리킬 수 있습니다[S11]. 대표적으로 병렬 샘플링처럼 한 프롬프트에서 여러 개의 출력을 동시에 만드는 경우, 공통된 프롬프트 구간의 블록은 한 벌만 두고 여러 출력이 공유합니다. 요청 내에서든 요청 사이에서든 같은 블록을 재사용하므로, 실제 GPU에 올라가는 KV 캐시의 양이 줄어듭니다.

이렇게 미리 예약해 두고 노는 공간을 없애고 같은 블록을 공유하는 결과로, PagedAttention은 KV 캐시 메모리의 낭비를 거의 0에 가깝게(near-zero waste) 낮춥니다[S11]. 낭비가 줄면 같은 GPU 메모리에 더 많은 요청의 KV 캐시가 들어갑니다. 이는 곧 동시에 받을 수 있는 요청 수가 늘고, 요청 하나가 다룰 수 있는 컨텍스트 길이에 여유가 생긴다는 뜻입니다. 즉 메모리 낭비를 없앤 것 자체가 목적이 아니라, 그 여유가 동시성과 컨텍스트 용량으로 환산된다는 점이 이 기법의 실무적 가치입니다. 참고로 낭비 감소 폭을 백분율 같은 구체 수치로 제시하는 대신 near-zero라는 정성 표현으로 서술하는 이유는, 정확한 수치는 배포 조건과 워크로드에 따라 달라지기 때문입니다. 인용 시에는 논문 원문에 명시된 조건을 함께 확인하는 편이 안전합니다.

4.2 연속 배칭과 스케줄링

메모리 낭비를 없앴다면, 남은 손실은 시간 축에서 발생합니다. 여러 요청을 묶어 GPU에 태우는 배치 처리 방식에 따라 GPU가 일하지 않고 노는 구간이 생기기 때문입니다. 이 절에서는 요청 단위가 아니라 iteration(생성 반복) 단위로 스케줄링하는 연속 배칭이 그 유휴를 어떻게 없애는지, 그리고 vLLM의 최신 엔진에서 기본으로 동작하는 chunked prefill과 선점 방식이 무엇인지를 설명합니다.

4.2.1 iteration 단위 스케줄링

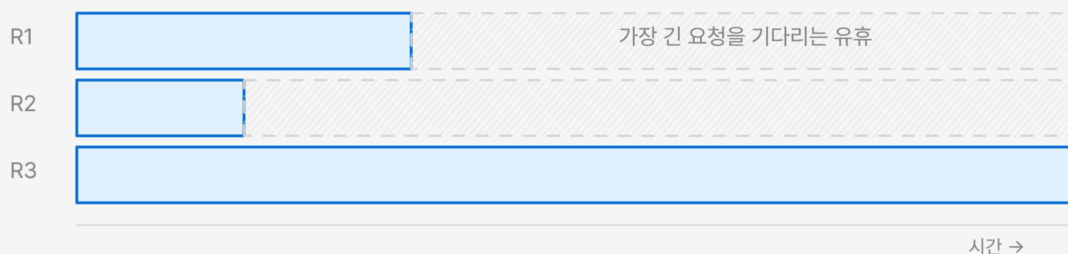
정적 배칭(static batching)은 요청을 정해진 크기의 묶음으로 모아 함께 시작하고, 그 묶음의 모든 요청이 끝날 때까지 다음 묶음을 시작하지 않습니다. 문제는 요청마다 생성 길이가 제각각이라는 점입니다. 한 묶음에 20토큰만 생성하는 요청과 500토큰을 생성하는 요청이 섞여 있으면, 짧은 요청은 진작 끝났는데도 가장 긴 요청이 마칠 때까지 그 자리를 붙들고 있습니다. 먼저 식사를 마친 손님이 나가도 같은 테이블의 다른 손님이 다 먹을 때까지 새 손님을 못 받는 식당과 같아서, 그 시간 동안 GPU의 연산 자원이 놀게 됩니다.

연속 배칭(continuous batching)은 스케줄링 단위를 요청에서 iteration으로 내립니다[S4][S1]. vLLM은 토큰을 한 번 생성하는 매 반복마다 배치 구성을 다시 정합니다. 어떤 요청이 종료 조건을 만족해 끝나면 그 즉시 자리를 반환하고, 대기 중이던 다음 요청을 비어 있는 자리에 곧바로 끼워 넣습니다. 묶음 전체가 끝나기를 기다릴 필요가 없으므로, 짧은 요청이 붙들던 유휴 구간이 사라지고 GPU는 쉬지 않고 다음 일을 이어받습니다. 식당으로 치면 테이블 단위가 아니라 자리 단위로 회전율을 관리하는 셈입니다. 이렇게 유휴를 제거하는 것이 처리량 향상의 직접적인 원인입니다.

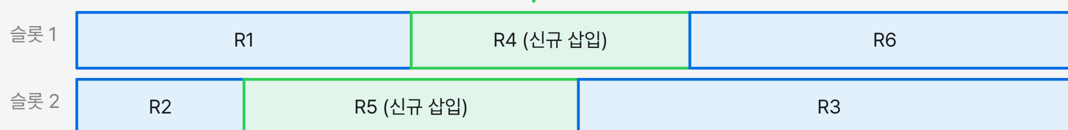
정적 배칭 vs 연속 배칭 — GPU 점유 타임라인

iteration 단위로 끝난 요청을 반환하고 대기 요청을 그 자리에 삽입해 유휴를 없앤다

정적 배칭 — 배치가 끝날 때까지 대기



연속 배칭 — 끝난 자리에 대기 요청 삽입



iteration 경계마다 스케줄러가 큐를 확인 → 빈 슬롯 즉시 충전 (유휴 거의 없음)

정적 배칭과 연속 배칭의 실행 타임라인을 위아래로 대비하는 도식. 위쪽 정적 배칭은 묶음 내 짧은 요청이 끝난 뒤 긴 요청을 기다리는 회색 유휴 구간이 넓게 남고, 아래쪽 연속 배칭은 요청

이 끝나는 즉시 대기 요청이 빈 자리를 채워 유휴가 거의 없는 모습.

4.2.2 chunked prefill과 선점 방식

요청 처리는 프롬프트 전체를 한 번에 읽어 첫 KV 캐시를 채우는 프리필(prefill) 단계와, 이후 토큰을 하나씩 생성하는 디코드(decode) 단계로 나뉩니다. 프리필은 프롬프트가 길수록 한 번에 큰 연산을 요구합니다. 긴 프리필 하나가 배치를 통째로 점유하면, 이미 생성 중이던 다른 요청들의 디코드가 그동안 밀려 응답이 뚝뚝 끊깁니다. chunked prefill은 이 긴 프리필을 여러 조각으로 잘라, 디코드 작업과 같은 배치에 섞어 조금씩 처리합니다. 긴 프롬프트가 들어와도 진행 중인 요청의 디코드가 계속 흐르므로, 처리량과 응답 지연 사이의 균형이 좋아집니다. vLLM의 V1 엔진에서는 이 chunked prefill이 기본으로 활성화되어 있습니다[S2][S12].

한편 GPU 메모리가 부족해지면 vLLM은 실행 중인 요청 일부를 잠시 물러나게 하는 선점(preemption)을 수행합니다. 선점된 요청의 KV 캐시를 어떻게 처리하느냐에 따라 두 방식이 있습니다. 하나는 캐시를 CPU 메모리로 옮겨 두었다가 되돌리는 SWAP이고, 다른 하나는 캐시를 버렸다가 요청이 다시 실행될 때 프리필을 다시 계산하는 RECOMPUTE입니다. V1 엔진의 선점 기본 모드는 RECOMPUTE입니다[S2][S12]. 여기서 유의할 점은 이 기본값들이 엔진 버전에 따라 다르다는 것입니다. 아래 표는 V0과 V1의 주요 기본값 차이를 정리한 것으로, 운영 환경에서 어느 엔진이 동작 중인지에 따라 기대 동작이 달라지므로 배포 전에 확인하는 편이 좋습니다.

항목	V0 기본값	V1 기본값	실무 영향
chunked prefill	비활성(옵트인)	기본 활성	긴 프롬프트 유입 시에도 디코드가 흘러 TTFT·응답 안정성 개선
선점 모드	SWAP 계열	RECOMPUTE	메모리 압박 시 캐시를 재계산으로 복원
`--max-num-seqs`	256	1024	동시에 처리하는 요청 수 상한이 큼

이 기본값 변화는 최신 엔진일수록 긴 프롬프트와 높은 동시성을 별도 튜닝 없이도 안정적으로 감당하도록 설계가 옮겨 갔음을 보여 줍니다. 요약하면 이 장이 다룬 두 축은 명확히 나뉩니다. PagedAttention은 공간 축의 손실인 메모리 낭비를 블록 단위 관리로 없애고, 연속 배칭과 그에 딸린 스케줄링 기법은 시간 축의 손실인 GPU 유휴를 iteration 단위 처리로 없앱니다. 두 가속의 원리를 이해하면, 뒤이어 다룰 실무 파라미터들이 왜 그런 방식으로 메모리와 배치를 조절하는지도 자연스럽게 이어집니다.

5장: vLLM 도입 전후의 처리량과 비용 변화

앞 장까지가 vLLM이 무엇을 어떻게 처리하는지에 관한 설명이었다면, 이 장은 그 구조가 실제 운영 지표로 환산되었을 때 무엇이 달라지는지를 다룹니다. 결론부터 말하면, 같은 GPU에 더 많은 요청을 태우고 같은 트래픽을 더 적은 GPU로 감당하게 되는 변화입니다. 다만 여기서 인용하는 처리량 배수는 어떤 대상과 비교했는지에 따라 크게 달라지므로, 수치를 볼 때는 배수 자체보다 비교 조건을 먼저 확인해야 합니다. 이 장은 처리량 향상 수치와 그 조건(5.1), 그리고 그 향상이 GPU 비용과 응답 지연으로 이어지는 경로(5.2)를 순서대로 정리합니다. 의사결정 관점에서 이 장의 핵심 질문은 하나입니다. "이 향상이 우리 트래픽에서 몇 대의 GPU 절감으로 환산되는가"입니다.

5.1 정량 대비

정량 대비에서 가장 흔한 오해는 "24배 빨라진다"는 식의 조건 없는 인용입니다. vLLM의 처리량 향상 수치는 실제로 여러 개가 존재하며, 각 수치는 서로 다른 비교 대상을 놓고 측정된 결과입니다. 이 절은 대표 수치 두 개의 비교 조건을 먼저 명확히 하고, 그다음 그 향상의 근본 원인인 KV 캐시 낭비 감소를 다룹니다. 조건을 병기하지 않은 배수는 의사결정 근거로 쓸 수 없다는 점을 전제로 읽어 주시기 바랍니다.

5.1.1 처리량 향상 수치와 비교 조건

vLLM의 처리량 향상은 대표적으로 두 가지 수치로 인용됩니다. 두 수치는 배수의 크기가 크게 다르지만, 이는 vLLM이 상황에 따라 다르게 동작해서가 아니라 **비교 대상이 다르기 때문**입니다.

첫째, PagedAttention 논문(SOSP 2023)은 **동일 지연 조건에서 당시 최고 수준(SOTA)의 서빙 시스템인 FasterTransformer 및 Orca 대비 2~4배** 높은 처리량을 보고합니다[S11]. 여기서 비교 대상은 이미 배치·스케줄링을 최적화한 전문 서빙 엔진입니다. 즉 이 2~4배는 "잘 만든 서빙 시스템과 비교해도 그만큼 더 낫다"는 의미이며, vLLM의 순수한 알고리즘적 우위를 가장 보수적으로 보여 주는 수치입니다.

둘째, Red Hat은 **순정 HuggingFace Transformers 대비 최대 24배** 높은 처리량을 인용합니다[S1]. 여기서 비교 대상은 서빙에 특화되지 않은 순정 추론 라이브러리입니다. HF Transformers는 연구·프로토타입에는 훌륭하지만 연속 배치나 KV 캐시 관리 같은 서빙 최적화를 기본 탑재하지 않으므로, 그 대비의 배수가 커지는 것은 자연스럽습니다. 이 24배는 "서빙 최적화가 전혀 없는 기준선에서 vLLM으로 옮기면 최대 그만큼 달라질 수 있다"는 의미로 읽어야 합니다.

두 수치를 나란히 표로 정리하면 조건 차이가 분명해집니다.

표 5-1. 비교 대상별 vLLM 처리량 향상 수치와 조건

향상 수치	비교 대상	비교 대상의 성격	측정 조건	출처
2~4배	FasterTransformer, Orca	당시 SOTA 서빙 엔진(배치·스케줄링 최적화 완료)	동일 지연 조건에서의 처리량	[S11]
최대 24배	순정 HF Transformers	서빙 미최적화 순정 추론 라이브러리	Red Hat 인용(상한값)	[S1]

의사결정 관점에서 두 수치는 각각 다른 상황에 대응합니다. 이미 전문 서빙 엔진을 쓰고 있다면 기대할 수 있는 개선폭은 2~4배 쪽에 가깝습니다. 반대로 아직 순정 HF Transformers나 그에 준하는 미최적화 경로로 모델을 서빙하고 있다면 개선폭은 두 자릿수 배수까지 열려 있습니다. "최대 24배"의 "최대"라는 표현을 흘려 읽지 말아야 하며, 이는 평균이나 보장치가 아니라 특정 조건에서의 상한이라는 점을 분명히 해 둡니다. 엔지니어 검토 단계에서는 자사의 현재 서빙 스택이 두 비교 대상 중 어디에 가까운지를 먼저 판별한 뒤, 해당 배수 구간을 시작점으로 삼는 것이 안전합니다.

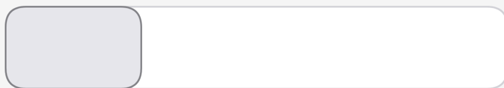
같은 vLLM, 비교 대상에 따라 달라지는 배수

배수의 크기는 vLLM 성능이 아니라 비교 기준선의 차이에서 나온다

대 SOTA 서빙 엔진

조건: 이미 최적화된 프로덕션 서빙 엔진과 동일 조건 비교

2~4배

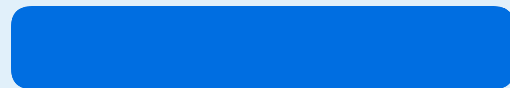


높은 기준선 → 보수적 배수

대 순정 HF Transformers

조건: 배칭·KV 최적화가 없는 순정 파이프라인과 비교

최대 24배



낮은 기준선 → 큰 배수

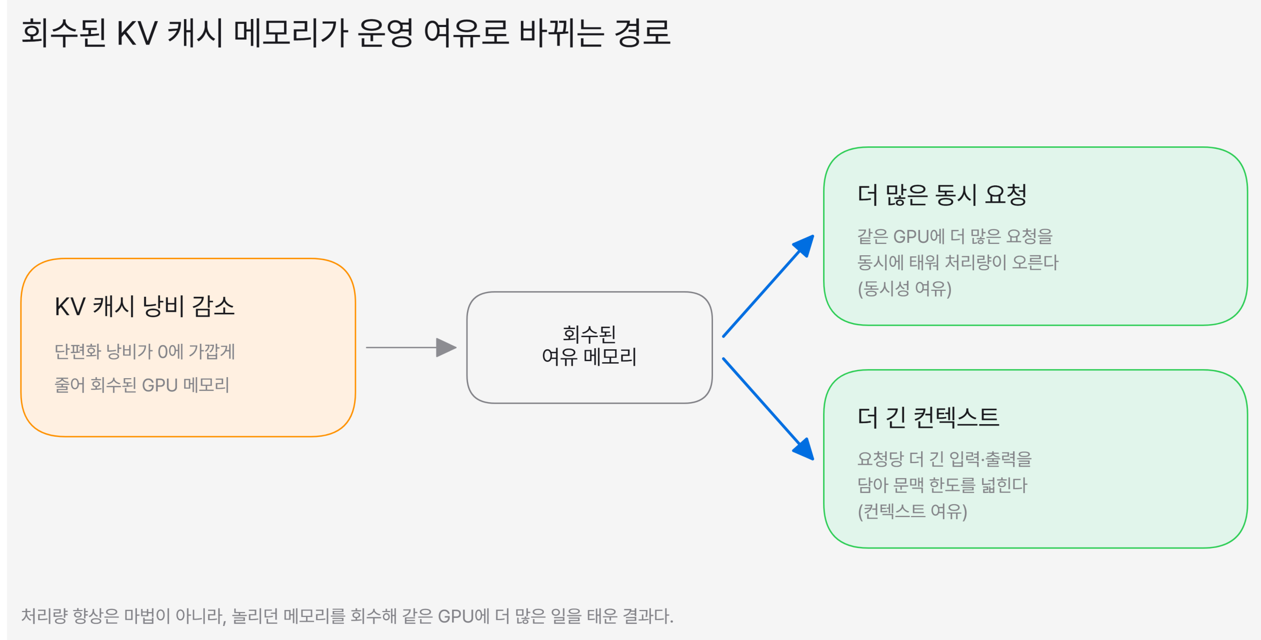
막대 길이는 배수 크기를 나타냄. 두 배수는 서로 다른 비교 조건의 결과이므로 직접 비교 대상이 아니다.

동일한 vLLM이라도 비교 대상(SOTA 서빙 엔진 vs 순정 HF)에 따라 인용 배수가 2~4배에서 최대 24배까지 벌어짐을 보여 주는 대비도.

5.1.2 KV 캐시 낭비 감소

앞의 처리량 향상은 상당 부분 KV 캐시 메모리를 얼마나 알뜰하게 쓰느냐에서 나옵니다. PagedAttention은 KV 캐시를 고정 크기 블록 단위로 관리해 **거의 0에 가까운 낭비(near-zero waste)**를 달성하고, 요청 내부는 물론 요청 사이에서도 KV 캐시를 공유합니다[S11]. 기존 방식이 요청마다 최대 길이를 미리 잡아 두어 상당량의 메모리를 놀리던 것과 대비됩니다. (참고로 "기존 60~80% 낭비가 4% 미만으로 줄었다"는 식의 구체 수치가 일부 자료에 돌지만, 확보한 소스에서 그대로 확인되지 않아 이 백서에서는 인용하지 않습니다. 검증된 표현은 "거의 0에 가까운 낭비"까지입니다.)

이 낭비 감소가 왜 비용 논거가 되는지가 핵심입니다. GPU 메모리는 유한하고, 그중 상당 부분이 KV 캐시에 쓰입니다. 낭비되던 메모리가 회수되면 그 여유 공간은 두 방향으로 환산됩니다. 하나는 **더 많은 동시 요청**(같은 GPU가 받을 수 있는 요청 수 증가)이고, 다른 하나는 **더 긴 컨텍스트**(요청당 처리 가능한 토큰 길이 증가)입니다. 실무 파라미터로 보면 이는 배치 내 동시 요청 수를 좌우하는 `--max-num-seqs`가 V1 엔진에서 기본 1024로(V0의 256 대비) 크게 상향된 배경과도 맞닿아 있습니다[S12]. 같은 하드웨어에서 더 많은 좌석을 받을 수 있게 된 것입니다.



회수된 KV 캐시 메모리가 "더 많은 동시 요청" 또는 "더 긴 컨텍스트"라는 두 갈래의 여유 자원으로 환산되는 경로도.

의사결정 관점에서 이 항의 결론은 단순합니다. 처리량 향상은 마법이 아니라, 놀리던 메모리를 회수해 같은 GPU에 더 많은 일을 태운 결과라는 것입니다. 이 인과를 이해하면 다음 절의 비용 환산이 왜 성립하는지도 자연스럽게 따라옵니다.

5.2 비용과 운영 영향

처리량이 오른다는 사실 자체는 비용 논거가 아닙니다. 그 향상이 GPU 대수 절감, 동시 사용자 증가, 응답 지연 변화로 어떻게 환산되는지를 자사 트래픽에 대입해야 비로소 도입 승인의 근거가 됩니다. 이 절은 먼저 처리량을 비용으로 환산하는 계산 틀을 가정과 함께 제시하고(5.2.1), 이어 처리량 이득과 단일 요청 지연이 별개라는 점을 분명히 합니다(5.2.2).

5.2.1 GPU 비용과 동시 사용자

처리량 향상은 두 가지 방식으로 비용에 반영됩니다. 트래픽이 고정되어 있다면 **더 적은 GPU로 같은 트래픽을 감당**하므로 GPU 대수·전력·상면 비용이 줄고, GPU 대수가 고정되어 있다면 **같은 GPU로 더 많은 동시 사용자**를 받으므로 사용자당 인프라 원가가 낮아집니다. 어느 방향이든 처리량 배수가 그대로 비용 지렛대로 작동합니다.

다만 절감폭을 계산할 때는 반드시 가정을 명시해야 합니다. 아래 예시는 **설명을 위한 가상 시나리오**이며, 실제 값은 모델·시퀀스 길이·트래픽 패턴에 따라 달라집니다.

표 5-2. 처리량 향상의 비용 환산 예시(전부 가정)

항목	도입 전(가정)	도입 후(가정)	비고
비교 기준선	순정 HF 서빙	vLLM	배수는 이 대비에 종속
가정한 처리량 향상	1배(기준)	8배(24배 상한의 보수적 1/3 적용)	**가정**: 상한 아닌 보수치 사용
목표 트래픽	동일	동일	트래픽 고정 시나리오
필요 GPU 대수	8대(가정)	1대	처리량 8배 → 대수 1/8 (선형 근사)
사용자당 원가	기준	약 1/8	대수 절감이 원가로 직결

이 표의 숫자는 어느 것도 벤치마크 실측이 아니며, "처리량 배수 → GPU 대수 → 원가"라는 **환산 논리의 골격**만을 보여 줍니다. 실제 사업 계획에 넣을 때는 세 가지를 자사 값으로 교체해야 합니다. 첫째, 배수는 현재 서빙 스택이 5.1.1의 두 비교 대상 중 어디에 가까운지에 따라 2~4배 또는 그 이상 구간에서 고릅니다. 둘째, 처리량과 GPU 대수의 관계는 완전 선형이 아니므로(네트워크·스케줄링 오버헤드 존재) 선형 근사에 여유를 둡니다. 셋째, 절감 대수를 계산할 때 가용성 확보용 예비 GPU와 트래픽 피크 대응분은 별도로 남겨 둡니다.

경쟁 포지셔닝도 비용 판단에 영향을 줍니다. HuggingFace가 자사 TGI를 유지보수 모드로 전환하고 vLLM·SGLang·llama.cpp·MLX를 권장 경로로 안내하고 있다는 점은[S6], 이미 서빙 스택을 갖춘 조직이 앞으로 vLLM 계열로 수렴할 가능성이 높다는 신호입니다. 다만 동시성이 매우 높은 구간에서는 TensorRT-LLM의 원시 처리량 우위도 함께 검토해야 하며[S6], vLLM은 저~중 동시성에서 운영 단순성을 무기로 비용 대비 효율이 좋은 선택지라는 것이 현재의 일반적 정리입니다.

5.2.2 응답 지연과 사용자 경험

여기서 가장 흔히 발생하는 기대치 오조정을 짚어야 합니다. **vLLM이 개선하는 것은 주로 처리량(throughput)이지, 단일 요청의 지연(latency)이 저절로 좋아지는 것은 아닙니다.** 처리량은 "단위 시간에 처리한 전체 요청·토큰의 양"이고, 지연은 "한 요청이 응답을 받기까지 걸린 시간"입니다. 배치를 키우고 동시성을 높이면 시스템 전체 처리량은 오르지만, 개별 요청 입장에서는 오히려 대기가 생겨 지연이 나빠질 수도 있습니다.

vLLM은 이 트레이드오프를 완화하는 장치를 갖추고 있습니다. V1 엔진의 chunked prefill 기본 활성화와 배치 토큰 예산 조정이 대표적입니다[S2][S12]. 배치 토큰 예산을 작게 잡으면 토큰 간 지연이 줄고, 크게 잡으면 첫 토큰까지의 시간(TTFT)이 줄어드는 식으로, 처리량과 지연 사이의 균형점을 파라미터로 옮길 수 있습니다[S2][S12]. 그러나 이것은 어디까지나 균형점의 이동이지, 처리량과 지연을 동시에 무제한 개선하는 것은 아닙니다.

표 5-3. 처리량·지연 트레이드오프 요약

지표	정의	vLLM 도입 효과	주의
처리량	단위 시간당 전체 처리량	향상(5.1의 배수)	비용 논거의 핵심
단일 요청 지연	한 요청의 응답 시간	자동 개선 아님	배치·동시성↑ 시 악화 가능
균형 조정	둘 사이의 배분	파라미터로 이동 가능	동시 최대화는 불가

의사결정 관점의 결론은 이렇습니다. 지연에 민감한 서비스(대화형 챗봇의 체감 응답 속도 등)를 담당하는 조직이라면, vLLM 도입을 "지연 단축 프로젝트"가 아니라 "처리량·비용 효율 프로젝트"로 정의하고 기대치를 설정하는 편이 정확합니다. 지연 목표(예: TTFT 상한, 토큰 간 지연 상한)는 별도의 SLO로 두고, 그 목표를 만족하는 범위 안에서 처리량을 최대화하도록 파라미터를 조정하는 순서가 맞습니다. 처리량과 비용에서 얻는 이득이 크다는 사실과, 개별 사용자의 응답 지연이 얼마나 개선되는지는 서로 다른 질문이며, 이 둘을 분리해서 검증해야 도입 후 기대치 불일치를 피할 수 있습니다.

6장: 온프레미스 설치와 서버 기동 절차

이 장은 사내 GPU 서버에 vLLM을 설치하고 OpenAI 호환 API 서버로 기동하는 최소 절차를 다룹니다. 앞선 장들이 vLLM이 무엇을 왜 하는지를 설명했다면, 여기서는 손을 움직여 실제로 서버를 띄우는 데 집중합니다. 다루는 범위는 패키지 설치, GPU·드라이버 요건 확인, 모델 가중치의 로컬 확보, `vllm serve` 기동, 그리고 OpenAI 호환 엔드포인트로 첫 응답을 받아보는 검증까지입니다. 성능 튜닝 파라미터는 7장에서 별도로 다루므로, 이 장에서는 "일단 한 번 성공적으로 뜬다"를 목표로 삼습니다.

전제는 하나입니다. 온프레미스 환경에서는 모델 가중치를 미리 로컬에 확보해 두어야 합니다. 폐쇄망이라면 외부 허브에서 즉석 다운로드가 불가능하므로, 가중치 준비가 다른 모든 단계의 선행 조건이 됩니다. 아래 명령 예시는 그대로 복사해 붙여 넣을 수 있도록 구성했고, 각 줄이 무엇을 하는지 짧게 덧붙였습니다. 명령에 등장하는 `<model>` 자리에는 사내에서 실제로 운용하는 모델 식별자 또는 로컬 경로를 넣으면 됩니다.

6.1 설치와 사전 조건

이 절은 vLLM 패키지를 설치하고, 설치가 요구하는 GPU·드라이버 요건을 점검하며, 온프레미스 전제에 맞춰 모델 가중치를 로컬로 확보하는 절차를 정리합니다. 요건을 먼저 확인하는 이유는 단순합니다. 최소 요건을 충족하지 못한 상태에서 서버를 띄우면 기동 후반부에서야 오류를 만나 시간을 낭비하게 되므로, 설치 시점에 조기 실패를 유도해 문제를 앞당겨 확인하는 편이 낫습니다.

6.1.1 패키지 설치와 GPU 요건

vLLM은 파이썬 패키지로 배포되므로 설치 자체는 한 줄이면 끝납니다. 다만 CUDA 런타임과 GPU 드라이버가 미리 갖춰져 있어야 하고, 파이썬 환경을 격리해 두는 편이 나중에 의존성 충돌을 피하는 데 유리합니다.

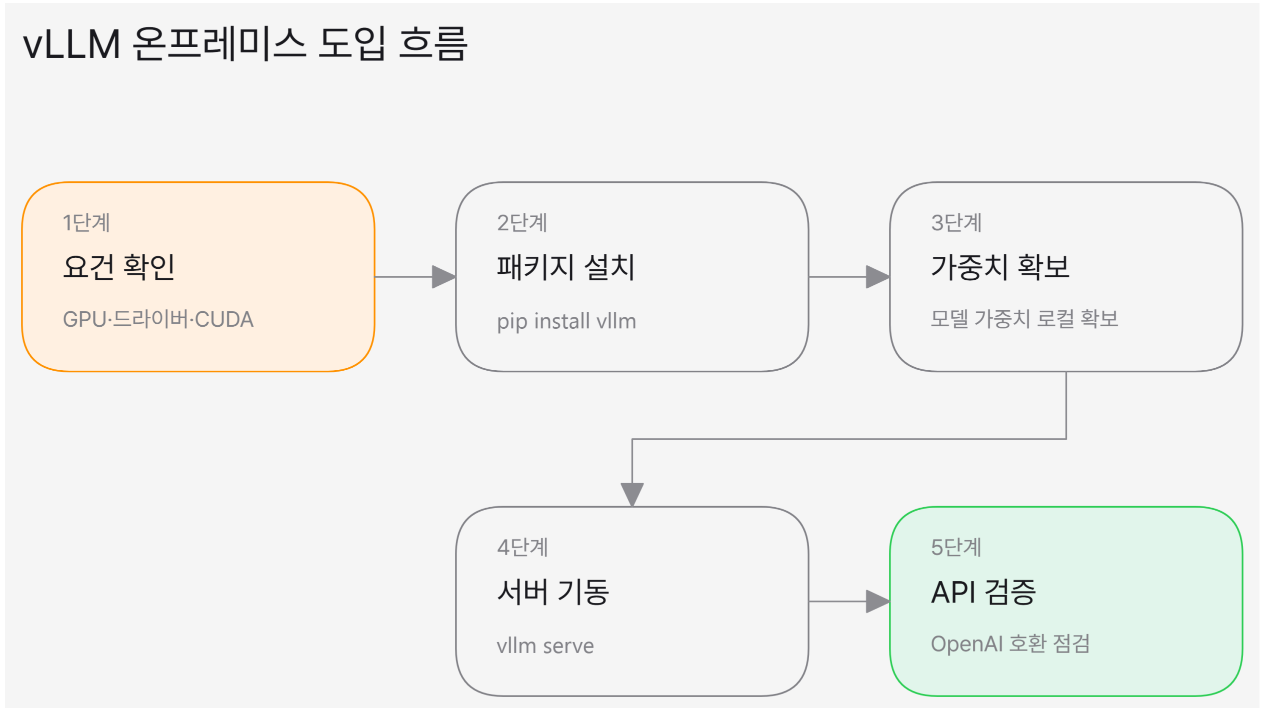
```
# 파이썬 가상환경 생성·활성화 (의존성 격리)
python -m venv .venv
source .venv/bin/activate

# vLLM 설치 — CUDA 빌드 휠을 내려받아 설치
pip install vllm

# 설치 확인 — 버전이 출력되면 정상
python -c "import vllm; print(vllm.__version__)"
```

각 줄의 의미는 이렇습니다. 첫 두 줄은 프로젝트 전용 가상환경을 만들고 켭니다. 세 번째 줄 `pip install vllm` 이 실제 설치이며, 이 명령은 CUDA용으로 미리 빌드된 휠(wheel)을 내려받습니다. 마지막 줄은 파이썬에서 vLLM을 불러와 버전 문자열이 정상 출력되는지 확인하는 최소 점검입니다.

설치가 성공해도 GPU 요건이 맞지 않으면 기동 단계에서 막힙니다. vLLM은 최신 GPU에서 fp8 양자화 같은 메모리 절감 기능을 활용할 수 있는데, 예컨대 fp8 가중치 압축은 H100·H200 계열에서 지원되며 메모리를 2~4배 줄이고 처리량을 최대 1.6배까지 끌어올립니다 [S12]. 이런 기능을 쓰려면 그에 맞는 GPU 세대와 드라이버가 필요하므로, 설치 직후 `nvidia-smi` 로 장착된 GPU와 드라이버 버전을 먼저 확인하는 것이 좋습니다. 참고로 v1 엔진에서는 chunked prefill이 기본 활성화되어 있고, 동시 요청 수를 결정하는 `--max-num-seqs` 의 기본값이 1024로 상향되어 있습니다(v0는 256) [S2][S12]. 이 값들은 기동 시 자동 적용되므로 지금 단계에서 신경 쓸 필요는 없지만, 첫 기동에서 예상보다 많은 GPU 메모리를 잡는 이유를 이해하는 데 도움이 됩니다.



vLLM 온프레미스 도입 흐름 — 요건 확인 → 패키지 설치 → 가중치 확보 → 서버 기동 → API 검증.

6.1.2 모델 가중치 로컬 확보

온프레미스, 특히 폐쇄망에서는 서버 기동 전에 모델 가중치가 반드시 로컬 디스크에 존재해야 합니다. vLLM은 기본적으로 모델 식별자를 받아 허브에서 내려받으려 시도하지만, 외부 접속이 차단된 환경에서는 이 경로가 막히므로 가중치를 미리 확보한 뒤 로컬 경로를 직접 지정하는 방식으로 운용합니다 [S7][S8].

가중치 준비는 다음 체크 목록으로 점검하면 빠짐없이 진행할 수 있습니다.

- 모델 파일 일체(가중치, 토크나이저, config)가 한 디렉터리 안에 모여 있는지 확인합니다.
- 인터넷 연결이 있는 별도 장비에서 내려받은 뒤 폐쇄망으로 옮기는 경우, 파일 무결성을 해시로 대조합니다.
- 서버가 허브를 조회하지 않도록 오프라인 모드 환경 변수를 설정합니다.

- 모델 가중치의 이용약관을 확인합니다. vLLM 엔진 자체는 Apache License 2.0이지만, 엔진 라이선스와 서빙하는 모델 가중치의 라이선스는 별개입니다 [S10]. Gemma·Qwen 같은 오픈 모델은 각 모델의 이용약관을 따로 확인해야 하며, 이 라이선스 검토는 9장에서 더 자세히 다룹니다.

오프라인 운용의 핵심은 서버가 외부 허브를 조회하지 않게 만드는 것입니다.

```
# 폐쇄망 오프라인 모드 — 허브 조회 차단
export HF_HUB_OFFLINE=1

# 가중치가 놓인 로컬 디렉터리 (예시)
ls /models/local-llm
# config.json  tokenizer.json  model-00001-of-00003.safetensors ...
```

첫 줄은 서버가 기동 중 외부 허브를 조회하지 않고 로컬 파일만 사용하도록 강제합니다. 둘째 명령은 가중치 디렉터리에 필요한 파일이 갖춰졌는지 눈으로 확인하는 용도입니다. config와 토크나이저 파일이 함께 있어야 하며, `.safetensors` 형식의 가중치 조각들이 모두 옮겨졌는지 확인합니다. 이 디렉터리 경로가 다음 절에서 `vllm serve` 에 넘길 값이 됩니다.

6.2 서버 기동과 API 확인

이 절은 확보한 가중치로 `vllm serve` 를 실행해 서버를 띄우고, 그 결과 노출되는 OpenAI 호환 엔드포인트에 요청을 보내 정상 응답을 확인하는 단계를 다룹니다. 목표는 첫 기동을 한 번 성공시키는 경험을 확보하는 것입니다. 세부 튜닝 옵션은 뒤로 미루고, 최소 명령으로 서버가 뜨는지부터 확인합니다.

6.2.1 vllm serve 기동

기동 명령의 골격은 `vllm serve <model>` 한 줄입니다. `<model>` 자리에는 앞 절에서 준비한 로컬 가중치 경로를 넣습니다. 이 명령 하나로 vLLM은 모델을 메모리에 올리고 OpenAI 호환 HTTP 서버를 기본 8000번 포트로 띄웁니다 [S8].

```
# 최소 기동 — 로컬 경로 모델을 8000번 포트로 서빙
vllm serve /models/local-llm
```

첫 기동은 이 한 줄이면 충분합니다. 여기에 익숙해진 뒤, 실제 온프레미스 운용에서 자주 쓰는 시작값을 얹어 갑니다. 아래는 단일 GPU 환경에서 메모리 활용도와 컨텍스트 길이를 명시한 예시입니다 [S8].

```
vllm serve /models/local-llm ₩
 --served-model-name local-llm ₩
 --quantization fp8 ₩
 --gpu-memory-utilization 0.9 ₩
 --max-model-len 32768 ₩
 --port 8000
```

각 옵션의 의미는 이렇습니다. `--served-model-name` 은 API 요청에서 부를 모델 이름을 지정합니다(로컬 경로 대신 짧은 별칭 사용). `--quantization fp8` 은 가중치를 fp8로 압축해 메모리를 아끼며, 앞서 언급한 대로 지원 GPU 세대에서만 유효합니다. `--gpu-memory-utilization 0.9` 는 실행기가 선점할 GPU 메모리 비율로, 기본값이 0.9이고 KV 캐시 여유를 늘리려면 단일 인스턴스 기준 0.95까지 올릴 수 있습니다 [S12]. `--max-model-len 32768` 은 프롬프트와 생성을 합친 최대 시퀀스 길이이며, 메모리가 부족하면 이 값을 낮춰 KV 캐시 할당을 줄입니다. `--port 8000` 은 서버가 노출할 포트입니다. 여러 GPU에 모델을 분산해야 한다면 여기에 `--tensor-parallel-size 8` 처럼 GPU 수를 더해 레이어 파라미터를 샤딩합니다 [S8][S12]. 이 값들의 튜닝 근거와 조합 전략은 7장에서 상세히 설명하므로, 지금은 서버가 정상 기동해 로그 끝에 준비 완료 메시지가 뜨는 것까지만 확인합니다.

6.2.2 OpenAI 호환 엔드포인트 점검

서버가 뜨면 vLLM은 OpenAI Chat Completions API와 동일한 형식의 엔드포인트를 노출합니다. 이 호환성 덕분에 기존에 OpenAI API를 쓰던 클라이언트 코드를 엔드포인트 주소만 바꿔 거의 그대로 재사용할 수 있습니다 [S2]. 첫 응답은 `curl` 로 직접 눈으로 확인하는 것이 가장 빠릅니다.

```
curl http://localhost:8000/v1/chat/completions ₩
 -H "Content-Type: application/json" ₩
 -d '{
   "model": "local-llm",
   "messages": [{"role": "user", "content": "안녕하세요, 자기소개 해주세요."}]
 }'
```

요청의 각 부분은 이렇습니다. 대상 URL `/v1/chat/completions` 가 OpenAI 호환 채팅 엔드포인트입니다. `"model"` 값은 앞서 `--served-model-name` 으로 지정한 이름과 일치해야 하며, `"messages"` 는 OpenAI와 동일한 역할·내용 구조를 따릅니다. 정상이라면 다음과 같은 JSON이 돌아옵니다.

```
{
 "id": "chatcmpl-...",
 "object": "chat.completion",
 "model": "local-llm",
 "choices": [
   {
     "index": 0,
     "message": { "role": "assistant", "content": "안녕하세요. ..." },
     "finish_reason": "stop"
   }
 ],
 "usage": { "prompt_tokens": 18, "completion_tokens": 42, "total_tokens": 60 }
}
```

`choices[0].message.content`에 생성된 답변이 담기고, `usage`에 토큰 사용량이 함께 옵니다. 이 응답 구조가 OpenAI API와 동일하므로, 기존 SDK나 클라이언트 라이브러리에서 base URL을 `http://localhost:8000/v1`로 바꾸기만 하면 코드 대부분을 손대지 않고 붙일 수 있습니다. 여기까지 응답이 정상적으로 확인되면 온프레미스 vLLM 서버의 기본 기동이 완료된 것입니다. 이후의 처리량·지연 튜닝과 다중 GPU 구성은 7장에서 이어집니다.

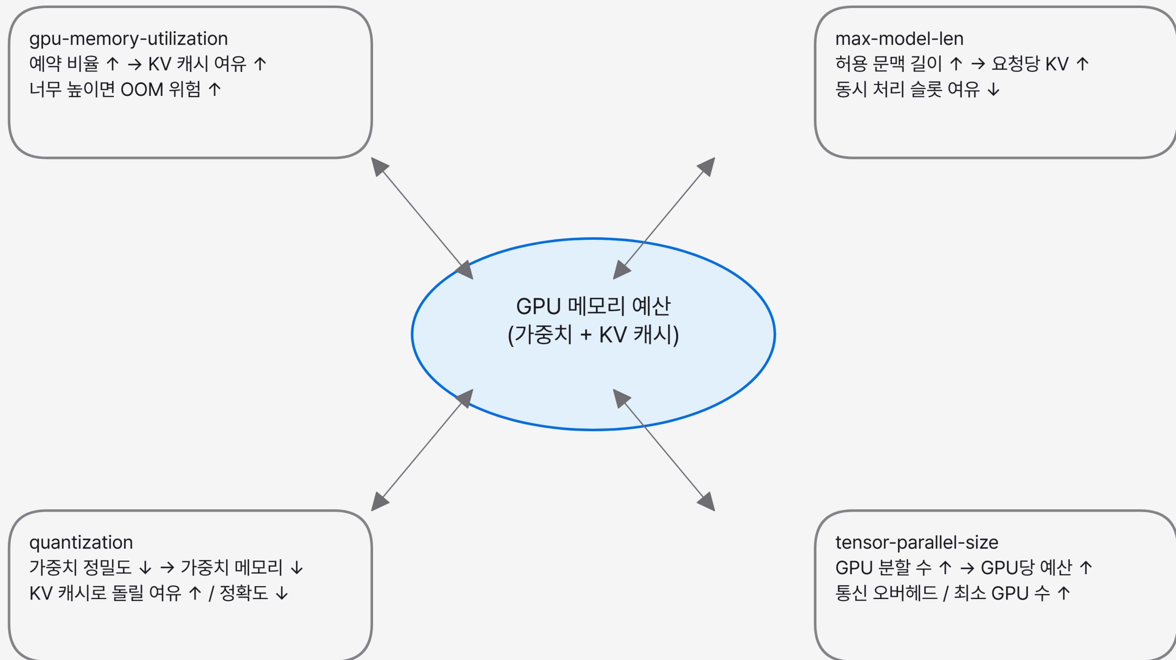
7장: Gemma4·Qwen3.6 튜닝 파라미터

앞선 장에서 vLLM이 왜 빠른지, 어디에 배치하는지를 다뤘다면 이 장은 실제로 손을 대는 곳입니다. 사내에서 서빙하는 두 계열, 즉 Gemma4와 Qwen3.6을 기동할 때 결과를 좌우하는 파라미터는 사실 한 줌입니다. 나머지는 대부분 기본값으로 두어도 무방합니다. 이 장에서는 그 한 줌의 파라미터를 하나씩 "무엇을 조절하는 손잡이인가"라는 관점으로 풀고, 사내 하드웨어에서 바로 쓸 수 있는 시작값을 제시합니다.

핵심은 이 손잡이들이 독립적이지 않다는 점입니다. GPU 메모리를 문맥 길이에 더 쓰면 동시 처리할 수 있는 요청 수가 줄고, 양자화로 메모리를 아끼면 그만큼을 동시성이나 문맥으로 되돌릴 수 있습니다. 파라미터 하나를 올렸다가 OOM(메모리 부족)으로 프로세스가 죽는 일은 대개 이 연동 관계를 놓쳐서 생깁니다. 그래서 값 자체보다 값들 사이의 트레이드오프를 이해하는 것이 이 장의 목표입니다.

아래 표는 이 장에서 다루는 파라미터 전체를 역할과 시작값으로 정리한 것입니다. 개별 설명은 이어지는 절에서 이어갑니다.

파라미터	조절하는 손잡이	시작값
`--gpu-memory-utilization`	vLLM이 선점하는 GPU 메모리 비율(0~1). 올리면 KV 캐시·처리량이 늘지만 과하면 OOM	기본 **0.9**, 단일 인스턴스 **0.95** [S12]
`--max-model-len`	한 요청이 쓸 수 있는 최대 시퀀스 길이(프롬프트+생성)	모델 config 상속(고정 기본값 없음), 부족 시 축소 [S2][S3]
`--quantization`	가중치 저정밀도 압축으로 메모리 절감	`fp8` (H100/H200), 메모리 2~4배↓·처리량 최대 1.6배↑ [S3][S7][S8]
`--kv-cache-dtype fp8`	KV 캐시 자체를 fp8로 저장	메모리 약 50%↓ → 긴 문맥·동시성↑ [S7][S12]
`--max-num-seqs`	배치 내 동시 요청 수	**V1 기본 1024**(V0는 256) [S12]
`--max-num-batched-tokens`	한 스텝의 배치 토큰 예산	온라인 8192·오프라인 16384 [S2][S12]
`--tensor-parallel-size`	모델 레이어를 여러 GPU에 샤딩	GPU 수에 맞춰 2·4·8 [S2][S12]



7.1 메모리와 문맥 파라미터

GPU 메모리는 이 장의 모든 결정이 나눠 갖는 예산입니다. 가중치가 먼저 자리를 차지하고, 남은 공간을 KV 캐시가 씁니다. KV 캐시가 넉넉해야 긴 문맥과 많은 동시 요청을 감당하므로, 실무 튜닝의 절반은 "가중치와 활성화가 쓰고 남은 메모리를 얼마나 KV 캐시로 확보하느냐"입니다. 이 절에서는 그 예산의 크기를 정하는 `--gpu-memory-utilization` 과 `--max-model-len`, 그리고 예산을 벌어들이는 `--quantization` 과 `--kv-cache-dtype` 를 다룹니다.

7.1.1 gpu-memory-utilization과 max-model-len

`--gpu-memory-utilization` 은 vLLM이 한 GPU에서 통째로 선점할 메모리 비율입니다. GPU를 얼마까지 빌릴지 정하는 임대 비율이라고 보면 됩니다. 값이 높을수록 KV 캐시로 쓸 공간이 늘어 처리량이 올라가지만, 여유를 지나치게 깎으면 순간적인 활성화 메모리 급증에 OOM으로 프로세스가 죽습니다. 기본값은 **0.9**입니다 [S12]. GPU 한 장을 vLLM 한 인스턴스가 독점하고 다른 프로세스가 없다면 **0.95**까지 올려 KV 캐시를 더 확보할 수 있습니다 [S12]. 반대로 같은 카드에 다른 워크로드가 함께 산다면 0.9보다 낮춰 충돌을 피합니다.

`--max-model-len` 은 한 요청이 쓸 수 있는 최대 시퀀스 길이, 즉 프롬프트와 생성 토큰을 합친 상한입니다. 한 대화가 채울 수 있는 최대 원고지 칸 수라고 보면 됩니다. 이 값이 KV 캐시 할당을 직접 좌우합니다. 길이 상한이 클수록 요청 하나가 잡을 수 있는 캐시 블록이 커지고, 그만큼 동시에 수용 가능한 요청 수는 줄어듭니다. 고정된 기본값은 없고 모델 config에서 상속합니다 [S2][S3]. config 값이 하드웨어에 비해 과해 기동 시 KV 캐시가 모자라면, 2048·8192·32768 같은

단계로 낮춰 실제 사용 패턴에 맞춥니다 [S3][S7]. 즉 이 두 파라미터는 한 예산을 놓고 겨루는 관계입니다. 문맥을 길게 잡으면 동시성이 깎이고, 동시성을 늘리려면 문맥을 짧게 타협해야 합니다.

7.1.2 quantization과 kv-cache-dtype

앞 항의 트레이드오프는 결국 메모리 예산이 고정되어 있어서 생깁니다. 양자화는 그 예산 자체를 늘리는 손잡이입니다. `--quantization fp8` 은 가중치를 저정밀도로 압축해 저장합니다. 두꺼운 책을 얇은 종이에 인쇄해 책장 공간을 아끼는 것과 같습니다. 사진을 JPEG로 압축할 때처럼 정보를 약간 버리는 대신 용량을 크게 줄이는데, fp8은 이 손실이 대부분의 서빙 작업에서 체감되지 않을 만큼 작으면서 메모리를 **2~4배** 아끼고 처리량은 **최대 1.6배** 끌어올립니다 [S3][S7][S8]. 단 H100·H200 같은 fp8 가속을 갖춘 GPU에서 제값을 합니다.

`--kv-cache-dtype fp8` 은 가중치가 아니라 KV 캐시 자체를 fp8로 저장합니다. 메모장 내용을 압축해 적어두는 셈이라, KV 캐시 메모리를 약 **50%** 줄입니다 [S7][S12]. 여기서 아낀 공간이 이 장의 핵심 트레이드오프로 되돌아옵니다. 절약분을 더 긴 `--max-model-len` 으로 돌려 문맥을 늘릴 수도 있고, 더 많은 `--max-num-seqs` 로 돌려 동시성을 높일 수도 있습니다. 아래는 두 양자화 손잡이를 켜기 전후를 정리한 것입니다.

구성	KV 캐시 메모리	처리량	되돌릴 수 있는 여유
양자화 없음(bf16)	기준	기준	없음
`--quantization fp8`	가중치 2~4배↓	최대 1.6배↑	문맥·동시성 확대
`--kv-cache-dtype fp8` 추가	캐시 약 50%↓	유지~소폭↑	문맥·동시성 추가 확대

두 옵션은 함께 켜는 것이 보통이며, 사내 Gemma4 기동 명령이 실제로 둘 다 켠 형태입니다.

7.2 동시성과 병렬화 파라미터

7.1이 메모리 예산을 다뤘다면 이 절은 그 예산 위에서 처리량과 지연을 조율하는 손잡이, 그리고 모델이 GPU 한 장에 들어가지 않을 때의 병렬화를 다룹니다. 여기서부터는 워크로드 성격, 즉 사용자가 응답을 실시간으로 기다리는 온라인인지 대량 배치를 밤새 돌리는 오프라인인지에 따라 값이 갈립니다.

7.2.1 max-num-seqs와 배치 토큰 예산

`--max-num-seqs` 는 한 배치에서 동시에 처리할 요청 수의 상한입니다. 식당이 동시에 받는 테이블 수라고 보면 됩니다. vLLM V1 엔진의 기본값은 **1024**로, 구형 V0의 256에서 크게 올랐습니다 [S12]. 이 최신 기본값을 인지하는 것이 중요한데, 오래된 문서나 예전 감각으로 낮게 잡아두면 하드웨어 여력을 놀리게 됩니다. 다만 값이 크다고 무조건 좋은 것은 아닙니다. 동시 요청이 많아지면 KV 캐시 압박이 커지고, 예산이 모자라면 요청이 대기열에서 밀립니다.

`--max-num-batched-tokens` 는 한 스텝에서 처리할 토큰 총량의 예산입니다. 한 번에 조리하는 재료 총량인 셈입니다. 이 값이 작으면 토큰 간 지연(생성 중 지연)이 줄어 대화형 응답이 매끄럽

고, 크면 첫 토큰까지의 지연(TTFT)이 줄어 대량 처리에 유리합니다 [S2][S12]. 그래서 워크로드별로 권장 범위가 갈립니다.

워크로드	--max-num-seqs	--max-num-batched-tokens	우선 지표
온라인(대화형)	상황에 맞게 하향 조정	**8192**	낮은 토큰 간 지연
오프라인(배치)	기본 1024 유지·상향	**16384**	높은 처리량·낮은 TTFT

온라인 서비스는 응답 리듬이 중요하므로 토큰 예산을 작게, 오프라인 배치는 총처리량이 중요하므로 크게 잡는 것이 출발점입니다.

7.2.2 tensor-parallel-size와 모델별 시작값

`--tensor-parallel-size`는 모델이 GPU 한 장에 다 들어가지 않을 때, 레이어 파라미터를 여러 GPU에 나눠 싣는 손잡이입니다. 큰 짐을 여럿이 나눠 드는 것과 같습니다. 값은 사용할 GPU 수에 맞춰 2·4·8로 잡습니다 [S2][S12]. 단일 카드에 들어가는 모델이라면 1로 두는 것이 통신 오버헤드가 없어 가장 효율적입니다. 카드 여러 장에 걸쳐 샤딩할 때만 이 값을 올립니다.

아래는 사내 하드웨어 기준의 온프렘 시작 명령입니다. 사내에서 서빙하는 계열은 Gemma4와 Qwen3.6이며, 아래 명령의 파라미터 구성은 각각 단일 GPU 워크스테이션과 단일·다중 H100 환경을 반영합니다.

Gemma4 (RTX PRO 6000, 단일 GPU) [S7]:

```
vllm serve <gemma4-model> ₩
 --dtype bfloat16 ₩
 --quantization fp8 ₩
 --kv-cache-dtype fp8 ₩
 --gpu-memory-utilization 0.95 ₩
 --tensor-parallel-size 1 ₩
 --max-num-seqs 8 ₩
 --max-model-len 32767 ₩
 --enable-chunked-prefill ₩
 --enable-prefix-caching
```

이 구성은 카드 한 장을 독점하므로 `--gpu-memory-utilization`을 0.95로 올리고, fp8 양자화와 fp8 KV 캐시로 메모리를 벌어 그 여유를 32767 토큰의 긴 문맥에 투입한 형태입니다. GPU가 한 장이라 `--tensor-parallel-size`는 1입니다.

Qwen3.6 27B (단일 H100) [S8]:

```
vllm serve <qwen3.6-27b-path> ₩
 --served-model-name Qwen/Qwen3.6-27B ₩
 --quantization fp8 ₩
 --gpu-memory-utilization 0.9 ₩
```

```
--max-model-len 32768 ₩
--port 8000
```

Qwen3.6의 더 큰 397B MoE 모델을 8×H100에 올릴 때는 위 명령에 `--tensor-parallel-size 8` 을 더해 레이어를 여덟 장에 샤딩합니다 [S8]. 27B가 단일 카드에 들어가 병렬화 없이 도는 것과 달리, MoE 규모는 한 장에 담기지 않으므로 샤딩이 필수입니다.

정리하면, 시작값은 표에 있는 그대로 두되 세 가지 연동 관계만 기억하면 됩니다. 첫째, `--gpu-memory-utilization` 과 `--max-model-len` 은 같은 메모리 예산을 놓고 겨룹니다. 둘째, fp8 양자화와 fp8 KV 캐시는 그 예산을 벌어 문맥이나 동시성으로 되돌립니다. 셋째, `--max-num-seqs` 와 배치 토큰 예산은 온라인·오프라인 성격에 맞춰 지연과 처리량 사이를 조율합니다. 이 세 축을 잡으면 사내 Gemma4·Qwen3.6 기동은 표의 시작값에서 소폭 조정만으로 안정적으로 수렴합니다.

8장: 경쟁 오픈소스 추론 엔진 비교

vLLM만으로 모든 서빙 상황이 정리되지는 않습니다. 오픈소스 추론 엔진은 여러 갈래로 나뉘어 있고, 각 엔진은 특정 하드웨어와 특정 워크로드를 전제로 설계되어 있습니다. 이 장은 TensorRT-LLM·SGLang·TGI·llama.cpp 네 엔진과 vLLM을 성능·유연성·벤더 락인·셋업 난이도라는 네 가지 기준으로 나란히 놓고, 어떤 상황에서 vLLM이 맞고 어떤 상황에서 다른 엔진으로 넘어가야 하는지 판단 근거를 제시합니다.

한 가지 전제를 먼저 밝힙니다. 추론 엔진의 상태는 시점에 따라 바뀝니다. 벤치마크 수치, 유지보수 정책, 기능 커버리지는 릴리스마다 달라지므로, 이 장의 비교는 2026년 상반기 기준입니다[S5][S6]. 실제 도입 시점에는 각 프로젝트의 최신 문서와 릴리스 노트를 다시 확인하시기 바랍니다. 아래 판단은 그 시점의 사실을 근거로 하되, 시간이 지나도 크게 흔들리지 않는 구조적 특성 위주로 정리했습니다.

8.1 엔진별 강점과 약점

이 절에서는 다섯 엔진의 강점과 약점을 정리합니다. 먼저 각 엔진이 무엇을 전제로 만들어졌는지 개요를 잡은 뒤, 네 가지 기준으로 압축한 비교표로 한눈에 대조할 수 있게 합니다. 핵심은 "벤치마크 최고 처리량 1위 엔진"과 "실무 기본 선택 엔진"이 반드시 같지는 않다는 점입니다.

8.1.1 vLLM·TensorRT-LLM·SGLang·TGI 개요

각 엔진은 서로 다른 목적을 우선합니다. 전제 하드웨어까지 함께 보면 선택의 방향이 더 분명해집니다.

vLLM은 넓은 오픈모델 커버리지, PagedAttention 기반 KV 캐시 관리, OpenAI 호환 API, 그리고 큰 커뮤니티 생태계를 강점으로 합니다[S5][S6]. NVIDIA GPU를 기본으로 하되 다양한 가속기를 지원하며, 혼합된 오픈모델을 프로덕션에서 서빙할 때의 기본 선택지입니다. 약점이라면 특정 벤치마크에서 최고 원시 처리량 1위를 항상 차지하지는 않는다는 점인데, 이는 뒤에서 다시 다룹니다[S6].

TensorRT-LLM은 NVIDIA가 자사 GPU에 맞춰 최적화한 엔진으로, 커널 퓨전과 in-flight batching을 통해 원시 처리량에서 가장 앞선 축에 듭니다[S5][S6]. 대신 배포 전 모델을 엔진 아티팩트로 컴파일해야 하고, 이 과정이 상당한 시간(약 28분 수준)을 소요하며 산출된 아티팩트 관리도 복잡합니다[S5]. NVIDIA 하드웨어에 강하게 결합되므로 벤더 락인이 뒤따릅니다.

SGLang은 RadixAttention을 통해 요청 사이에 공유되는 prefix를 재사용하는 데 강합니다[S5][S6]. 구조화 생성(structured generation)도 함께 지원합니다. 프롬프트 앞부분이 반복되는 워크로드에서 큰 이점을 내지만, 그만큼 특정 워크로드에 특화된 성격이 있습니다.

TGI(Text Generation Inference)는 Hugging Face 생태계에 밀착해 있고 토큰 스트리밍을 잘 지원합니다. 다만 현재 **유지보수 모드**에 있습니다. 마이너 수정과 문서 갱신 중심으로만 관리되며, Hugging Face 자체가 신규 서빙에는 vLLM·SGLang·llama.cpp·MLX를 권장하고 있습니다[S6]. 따라서 이미 TGI로 정착한 팀이 아니라면 신규 도입에는 권장하기 어렵습니다.

llama.cpp는 위 네 엔진과 결이 다릅니다. GGUF 양자화 모델을 CPU나 소규모 GPU, 심지어 개인 장비에서도 돌릴 수 있는 경량 엔진입니다. 대규모 동시성 서빙보다는 로컬 실행, 엣지, 개발용 단일 사용자 환경에 맞습니다. 고동시성 프로덕션 서빙을 목표로 한다면 앞의 엔진들과 같은 층위에서 비교할 대상은 아닙니다.

8.1.2 네 가지 기준 비교표

앞의 개요를 성능·유연성·벤더 락인·셋업 난이도라는 네 가지 기준으로 압축하면 아래와 같습니다. vLLM 행을 강조 표기해 기본 선택지가 각 기준에서 어디에 서 있는지 보이게 했습니다.

엔진	성능(원시 처리량)	유연성(모델·기능)	벤더 락인	셋업 난이도
vLLM (본 엔진)	**높음(최고 1위는 아님)**	**매우 높음(넓은 모델·OpenAI 호환)**	**낮음(다중 가속기·오픈)**	**낮음(즉시 서빙)**
TensorRT-LLM	매우 높음(원시 처리량 우위)	중간(NVIDIA 중심)	높음(NVIDIA 결합)	높음(컴파일·아티팩트)
SGLang	높음(공유 prefix에 특화)	높음(구조화 생성)	낮음	중간
TGI	중간	중간(유지보수 모드)	낮음	낮음
llama.cpp	낮음(단일·소규모)	중간(GGUF 위주)	낮음	낮음

표에서 읽어야 할 핵심은 두 가지입니다. 첫째, TensorRT-LLM은 원시 처리량 기준으로 앞서지만 셋업 난이도와 벤더 락인 비용을 함께 치릅니다[S5][S6]. 둘째, vLLM은 네 기준에서 어느 하나도 바닥에 놓이지 않고 고르게 높은 편이며, 특히 유연성과 셋업 난이도에서 유리합니다. 실무의 기본 선택이 vLLM인 이유는 최고점이 아니라 이 균형에 있습니다.

8.2 선택 기준과 사례

강점과 약점을 정리했으니, 이제 실제 선택으로 넘어갑니다. 이 절은 워크로드 성격과 동시성 규모라는 두 축에서 어떤 엔진이 맞는지를 사례로 검증합니다. 워크로드가 엔진 선택을 좌우하는 경우가 많고, 여기에 트래픽 규모가 겹쳐 최종 판단이 결정됩니다.

8.2.1 워크로드별 적합 엔진

같은 GPU라도 워크로드가 다르면 유리한 엔진이 달라집니다. 아래는 대표적인 워크로드와 적합 엔진을 정리한 매핑입니다.

워크로드	특징	적합 엔진	근거
혼합 오픈모델 프로덕션	여러 모델을 표준 API로 서빙	vLLM	넓은 커버리지·OpenAI 호환[S5][S6]
prefix 반복 RAG·멀티턴	프롬프트 앞부분이 자주 재사용	SGLang	공유 prefix 재사용[S5][S6]

워크로드	특징	적합 엔진	근거
에이전트 루프	동일 컨텍스트 위 반복 호출	SGLang	RadixAttention[S6]
표준 NVIDIA 클러스터 고트래픽	안정 배포·최대 처리량	TensorRT-LLM	원시 처리량 우위[S5][S6]
HF 정착 팀의 기존 서비스	이미 TGI 운영 중	TGI(현행 유지)	신규는 비권장[S6]
로컬·엣지·단일 사용자	소규모·경량 실행	llama.cpp	GGUF 경량 실행

여기서 한 가지 조건부 판단을 짚어둡니다. prefix 재사용이 많은 워크로드, 예를 들어 같은 문서 묶음을 반복해서 참조하는 RAG나 긴 시스템 프롬프트를 공유하는 에이전트 루프라면 SGLang의 공유 prefix 재사용이 뚜렷한 이점을 냅니다[S6]. 다만 vLLM도 prefix 캐싱을 지원하므로, 워크로드의 prefix 반복 비중이 그렇게 높지 않다면 vLLM의 넓은 커버리지와 운영 단순성이 더 나은 선택일 수 있습니다. 즉 SGLang은 "prefix 반복이 지배적일 때"라는 조건이 붙습니다.

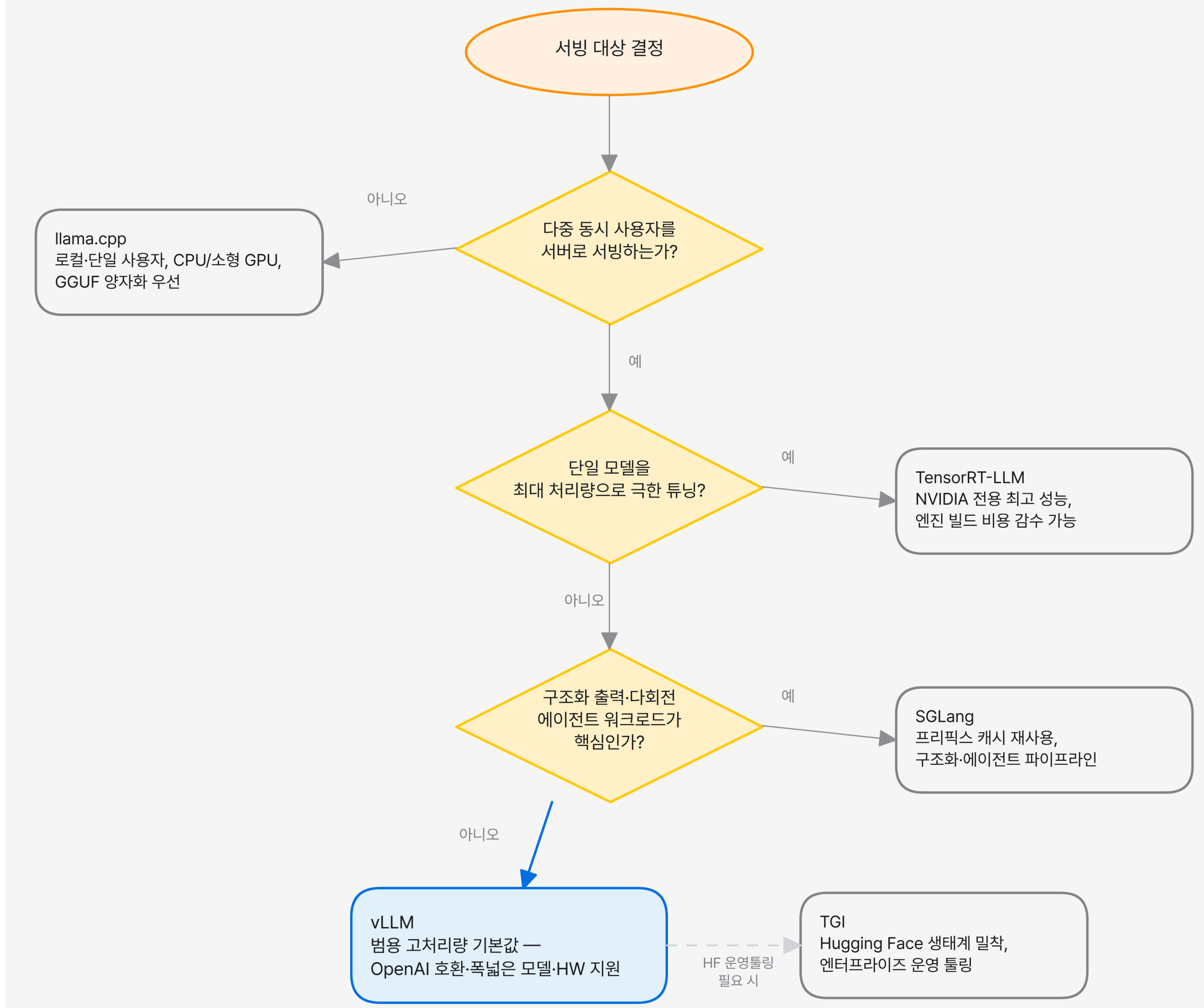
8.2.2 동시성 규모에 따른 선택

마지막 축은 동시성 규모입니다. 트래픽이 커지면 유리한 엔진이 바뀝니다.

동시성 구간	운영 우선순위	권장 엔진	재검토 트리거
저~중 동시성	운영 단순성·빠른 배포	vLLM	트래픽이 지속 급증할 때
급증·고동시성	최대 처리량·안정 배포	TensorRT-LLM	컴파일·락인 비용 수용 가능할 때

저~중 동시성 구간에서는 vLLM이 유리합니다. 컴파일 단계 없이 바로 서빙에 들어가고, 넓은 모델을 표준 API로 다루며, 운영이 단순합니다[S6]. 반면 동시성이 급증해 최대 처리량이 절대적으로 중요해지면 TensorRT-LLM의 원시 처리량 우위가 살아납니다[S6]. 이때는 약 28분 수준의 컴파일 시간, 복잡한 아티팩트 관리, NVIDIA 벤더 락인이라는 비용을 함께 받아들이는 결정을 내리는 셈입니다[S5].

실무적으로는 vLLM으로 시작해 운영 단순성을 확보하고, 트래픽이 지속적으로 급증하는 신호가 관측되면 그 시점에 TensorRT-LLM 전환을 재검토하는 흐름이 자연스럽습니다. 전환은 처리량이라는 이득과 컴파일·락인이라는 비용을 저울에 올린 트레이드오프이며, 트래픽 성장 자체가 그 저울을 다시 꺼내는 트리거가 됩니다. 이 판단 지점을 아키텍처에 미리 표시해 두면, 트래픽이 커졌을 때 급하게 결정하지 않아도 됩니다.


워크로드로 서빙 엔진 고르기
동시성 규모와 운영 제약을 위에서부터 물어 내려가며 후보 엔진을 좁힌다. 아래 잎이 권장 출발점이다.
서빙 대상 결정
다중 동시 사용자를 서버로 서빙하는가?
아니오
llama.cpp
로컬·단일 사용자, CPU/소형 GPU, GGUF 양자화 우선
예
단일 모델을 최대 처리량으로 극한 튜닝?
예
TensorRT-LLM
NVIDIA 전용 최고 성능, 엔진 빌드 비용 감수 가능
아니오
구조화 출력·다회전 에이전트 워크로드가 핵심인가?
예
SGLang
프리픽스 캐시 재사용, 구조화·에이전트 파이프라인
아니오
vLLM
범용 고처리량 기본값 — OpenAI 호환·폭넓은 모델·HW 지원
HF 운영툴링 필요 시
TGI
Hugging Face 생태계 밀착, 엔터프라이즈 운영 툴링

9장: 라이선스 검토와 도입 체크리스트

앞 장까지는 vLLM이 왜 빠른지, 어떤 파라미터로 조율하는지, 경쟁 엔진과 어떻게 다른지를 다뤘습니다. 이 장은 결이 다릅니다. 기술 검토가 끝난 뒤 실제 도입을 승인해야 하는 위치에서 확인할 두 가지, 곧 법무 관점의 라이선스와 실행 관점의 체크리스트를 정리합니다. 라이선스는 도입 여부를 가르는 최종 관문인 경우가 많고, 체크리스트는 도입을 사고 없이 진행하는 경로입니다. 결론부터 말씀드리면 vLLM 엔진 자체의 라이선스는 기업 도입에 거의 걸림돌이 되지 않습니다. 다만 엔진과 모델 가중치를 구분하는 지점에서 한 번 더 확인이 필요하며, MSAP.ai 도입 검토에서도 이 구분이 그대로 결정 프레임이 됩니다.

9.1 라이선스 유의점

라이선스 검토에서 가장 흔한 오해는 "엔진을 도입했으니 라이선스 하나만 보면 된다"는 생각입니다. vLLM은 추론을 실행하는 엔진이고, 그 위에서 실제 답을 생성하는 것은 Gemma나 Qwen 같은 모델 가중치입니다. 이 둘은 서로 다른 주체가 서로 다른 조건으로 배포합니다. 따라서 라이선스도 두 겹으로 확인해야 합니다. 이 절에서는 엔진 라이선스인 Apache License 2.0의 조건을 먼저 정리하고, 이어서 엔진과 모델 가중치 라이선스가 왜 별개인지를 다룹니다.

9.1.1 Apache 2.0 조건

vLLM 엔진은 **Apache License 2.0**으로 배포됩니다[S10]. 실무적으로는 "Apache 2.0으로 상업적 이용 가능"이라는 한 줄로 요약됩니다. 이 라이선스는 허용적(permissive) 계열이라 상업적 사용, 소스 수정, 재배포를 모두 자유롭게 허용하며, 수정한 코드를 공개하도록 강제하지 않습니다. 이 점이 GPL 계열과 결정적으로 다릅니다. GPL은 파생 저작물의 소스 공개를 요구하지만, Apache 2.0은 사내에서 vLLM을 수정해 폐쇄적으로 운영하더라도 소스를 외부에 공개할 의무가 없습니다. 기업 도입 장벽이 낮은 이유가 여기에 있습니다.

두 가지 실무 포인트만 기억하면 됩니다. 첫째, 재배포 시 고지 의무입니다. vLLM을 포함하거나 수정해 외부에 배포한다면 원본 라이선스 사본과 저작권 고지, 그리고 변경 사실 표시를 함께 전달해야 합니다. 사내에서 서비스로만 운영하는 일반적인 도입에서는 재배포에 해당하지 않아 이 의무가 발생하지 않습니다. 둘째, 특허 조항입니다. Apache 2.0에는 기여자가 이용자에게 특허 사용권을 부여하는 조항이 포함되어 있어, 특허 관련 불확실성이 상대적으로 낮습니다. 다만 이용자가 기여자를 상대로 특허 소송을 제기하면 그 특허 사용권이 종료됩니다. 정상적인 기업 이용에서는 문제가 되지 않는 조항입니다.

다음 표는 도입 승인 검토에서 확인할 Apache 2.0의 핵심 조건을 요약한 것입니다.

조건	허용/의무	도입 승인 시 확인 사항
상업적 이용	허용	별도 유료 라이선스 불필요
소스 수정	허용	수정본 소스 공개 의무 없음
재배포	허용	배포 시에만 고지 의무 발생

조건	허용/의무	도입 승인 시 확인 사항
라이선스·고지 유지	의무(재배포 시)	원본 라이선스 사본·저작권 고지 동봉
변경 사실 표시	의무(재배포 시)	수정한 파일에 변경 명시
특허 사용권	부여	기여자 대상 특허 소송 시 사용권 종료

9.1.2 엔진과 모델 가중치 라이선스 분리

여기가 이 장에서 가장 중요한 지점입니다. vLLM 엔진의 Apache 2.0은 엔진 코드에만 적용됩니다. 엔진 위에 올려 실제로 답을 생성하는 모델 가중치, 예를 들어 Gemma나 Qwen은 각 모델 제공사가 정한 별도의 이용약관을 따릅니다[S10]. 자동차에 비유하면 vLLM은 엔진이고 모델 가중치는 연료입니다. 엔진을 자유롭게 쓸 수 있다는 사실이 아무 연료나 마음대로 쓸 수 있다는 뜻은 아닙니다. 연료마다 사용 조건이 따로 있고, 그 조건은 연료 공급사가 정합니다.

실무에서 이 분리를 놓치면 리스크가 됩니다. 어떤 모델 가중치는 상업적 이용을 폭넓게 허용하지만, 어떤 모델은 사용 규모나 용도, 재배포에 제한을 두거나 별도 동의를 요구합니다. 따라서 vLLM 도입을 승인할 때는 "엔진 라이선스는 문제없음"과 "서빙할 각 모델의 가중치 라이선스를 개별 확인함"을 반드시 나눠서 판단해야 합니다. MSAP.ai 도입 맥락에서도 결정 프레임은 동일합니다. 엔진 채택 여부는 Apache 2.0 검토로 빠르게 끝나지만, 실제 서비스에 어떤 모델을 얹을지는 그 모델의 약관을 별도 트랙으로 확인한 뒤에 결정합니다.

구분	대상	라이선스 주체	확인 방식
추론 엔진	vLLM	vLLM 프로젝트(Apache 2.0)[S10]	한 줄 확인, 상업적 이용 가능
모델 가중치	Gemma·Qwen 등	각 모델 제공사	모델별 이용약관 개별 확인

정리하면, 엔진 라이선스는 한 줄로 끝납니다. "Apache 2.0으로 상업적 이용 가능." 실제 리스크 관리 대상은 그 위에 올릴 모델 가중치이며, 이건 서빙할 모델 목록을 정할 때마다 한 건씩 확인하는 절차로 다뤄야 합니다.

9.2 도입 체크리스트와 다음 단계

라이선스 검토를 통과했다면 다음은 실행입니다. 이 절은 검토 순서를 하나의 체크리스트로 제시하고, 파일럿을 거쳐 확산으로 넘어가는 단계와 그 성패를 판정할 지표를 정리합니다. 검토 순서 자체가 리스크를 단계별로 줄여 나가는 경로라는 점이 핵심입니다. 앞 장에서 다룬 GPU·모델·파라미터 논의가 여기서 하나의 실행 순서로 모입니다.

9.2.1 도입 검토 순서

도입 검토는 다음 다섯 단계를 순서대로 밟습니다. 각 단계는 앞 단계가 정해져야 다음으로 넘어갈 수 있으며, 순서를 지키는 것 자체가 뒷단계에서 발생할 문제를 앞에서 걸러 내는 방법입

니다.

1. **GPU 확인** — 보유하거나 확보 가능한 GPU의 종류와 메모리 용량을 먼저 파악합니다. 이 값이 서빙 가능한 모델 크기와 동시 처리량의 상한을 결정합니다. GPU가 정해지지 않으면 그다음 어떤 결정도 내릴 수 없습니다.
2. **모델 선택** — GPU 용량에 맞는 모델을 고르고, 이 시점에 9.1.2의 모델 가중치 라이선스를 함께 확인합니다. 라이선스 확인을 모델 선택과 붙여 두면 나중에 재작업할 위험이 사라집니다.
3. **설치** — vLLM을 설치하고 선택한 모델을 서빙합니다. OpenAI 호환 API를 제공하므로 기존 클라이언트 코드를 그대로 붙일 수 있습니다.
4. **파라미터 조정** — `--gpu-memory-utilization` (기본 0.9), `--max-model-len` , `--max-num-seqs` (V1 기본 1024), `--quantization` , `--tensor-parallel-size` 등을 워크로드에 맞춰 조율합니다 [S12]. 온프렘 시작값은 7장의 예시 명령을 출발점으로 삼습니다. 예를 들어 단일 GPU 에서는 `fp8` 양자화와 `--kv-cache-dtype fp8` 로 메모리를 확보하고, GPU가 여럿이면 `--tensor-parallel-size` 로 샤딩합니다[S12].
5. **검증** — 실제 트래픽과 유사한 부하로 처리량, 지연, 안정성을 측정합니다. 이 측정값이 다음 단계인 파일럿의 기준선이 됩니다.

앞의 세 단계(GPU→모델→설치)는 도입 가능 여부를 판정하고, 뒤의 두 단계(파라미터→검증)는 도입 품질을 다집니다. 이 순서를 지키면 검토 자체가 리스크 감쇄 경로가 됩니다.

9.2.2 파일럿 이후 단계

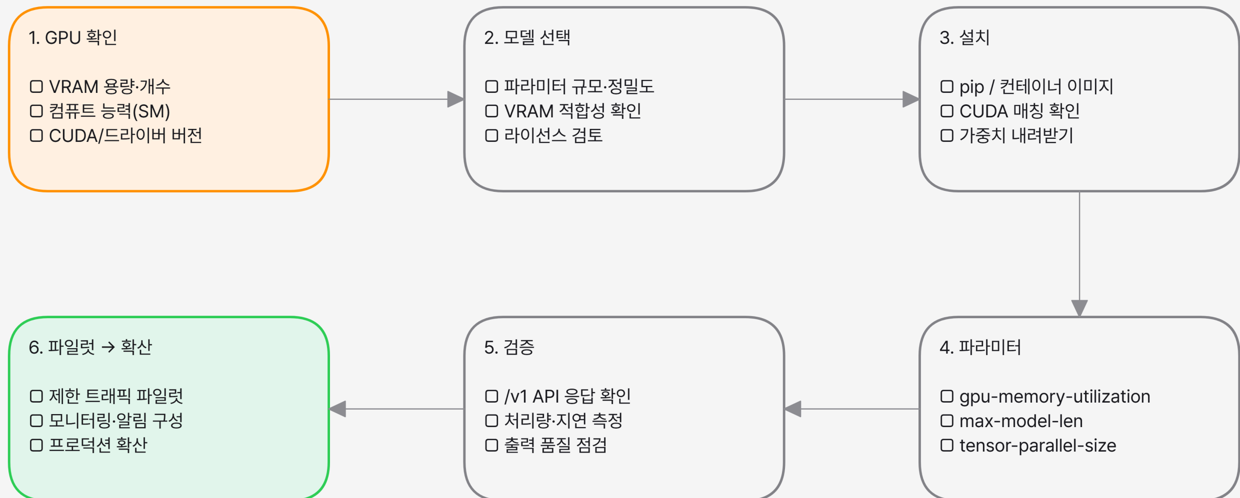
검증까지 마쳤다면 곧바로 전사 확산으로 가지 말고 파일럿을 한 단계 둡니다. 파일럿은 제한된 트래픽과 한정된 사용자 집단에서 실제 운영 조건을 짧게 검증하는 단계입니다. 여기서 도입 성공 여부를 판정할 기준을 미리 정해 두는 것이 중요합니다. 기준 없이 확산하면 도입이 성공했는지조차 판단할 수 없습니다.

점검 지표는 처리량과 비용을 중심에 둡니다. 처리량(초당 처리 요청 수, 토큰 생성 속도)은 같은 하드웨어로 얼마나 많은 요청을 감당하는지를, 비용(요청당 GPU 비용, 목표 지연을 맞추는 데 필요한 GPU 수)은 그 처리량을 내는 데 드는 값을 나타냅니다. 두 지표를 함께 봐야 "빠른데 비싼" 구성과 "적정 속도에 저렴한" 구성을 구분할 수 있습니다. 여기에 목표 지연(TTFT, 토큰 간 지연) 충족 여부를 안정성 조건으로 더합니다.

단계	범위	목표	점검 KPI
파일럿	제한 트래픽·한정 사용자	운영 조건 검증·기준선 확립	처리량, 요청당 비용, 목표 지연 충족률
확산	전사·프로덕션	안정 운영·용량 확장	처리량 유지, 비용 KPI 달성, 장애율

파일럿의 KPI가 목표를 만족하면 확산으로 넘어가고, 확산 단계에서는 같은 지표를 지속 모니터링하며 GPU 용량을 수요에 맞춰 늘립니다. 도입 결정은 이렇게 라이선스 확인(9.1)과 검토 순서

(9.2.1), 그리고 파일럿 지표(9.2.2)라는 세 관문을 순서대로 통과하는 것으로 마무리됩니다. 앞 장들의 기술 논의가 최종적으로 이 결정 프레임 위에서 하나의 도입 판단으로 수렴합니다.



참고 문헌 (References)

- [S1] Red Hat — What is vLLM? https://www.redhat.com/en/topics/ai/what-is-vllm
- [S2] vLLM Docs — Optimization and Tuning https://docs.vllm.ai/en/stable/configuration/optimization/
- [S3] vLLM Docs — Conserving Memory https://docs.vllm.ai/en/latest/configuration/conserving_memory/
- [S4] Spheron — LLM Serving Optimization https://www.spheron.network/blog/llm-serving-optimization-continuous-batching-paged-attention/
- [S5] Yotta Labs — Best LLM Inference Engines 2026 https://www.yottalabs.ai/post/best-llm-inference-engines-in-2026-vllm-tensorrt-llm-tgi-and-sglang-compared
- [S6] LeetLLM — Inference Engine Comparison 2026 https://leetllm.com/blog/llm-inference-engine-comparison-2026
- [S7] Google Codelabs — Gemma 4 on vLLM https://codelabs.developers.google.com/codelabs/cloud-run/cloud-run-gpu-rtx-pro-6000-gemma4-vllm
- [S8] Spheron — Deploy Qwen 3.5 with vLLM https://www.spheron.network/blog/deploy-qwen-3-5-gpu-cloud/
- [S9] RunPod — Introduction to vLLM and PagedAttention https://www.runpod.io/blog/introduction-to-vllm-and-pagedattention
- [S10] vLLM GitHub (Apache-2.0) https://github.com/vllm-project/vllm
- [S11] PagedAttention paper (SOSP 2023) https://arxiv.org/abs/2309.06180
- [S12] ROCm — vLLM V1 performance optimization https://rocm.docs.amd.com/en/latest/how-to/rocm-for-ai/inference-optimization/vllm-optimization.html

AI 필수 추론 가속 엔진 vLLM

CONTACT

WEB
msap.ai
www.msap.ai/

EMAIL
hello@msap.ai

TEL
02-6953-5427
0269535427

YOUTUBE
@msaptv
www.youtube.com/@msaptv

LINKEDIN
linkedin.com/showcas...
www.linkedin.com/showcase/msap-ai/

FACEBOOK
facebook.com/opennaru
www.facebook.com/opennaru

SCAN

MSAP.ai

msap.ai

© 2026 SMART CF, MSAP.ai. All rights reserved.