MODULE 10 · 심화

로컬 서빙의 다음 카드 — 대규모 기술이 우리 책상에 내려앉을 때

모듈 8에서 세계 최전선의 인프라(프리필·디코드 분리)를 구경했습니다. 그 글의 원문은 정직하게 이렇게 선을 긋습니다 — **"GPU 한두 장 규모에는 통합 서빙이 여전히 정답"**이라고. 그럼 우리는 구경만 하면 될까요? 아닙니다. 그 최전선 기술 중 **세 가지는 이미 우리 책상 규모로 내려앉았습니다.** 로컬 모델 연구의 흐름이 어디로 가는지, 우리가 지금 쓸 수 있는 건 무엇인지 정리합니다. (2026년 하반기 기준)

01 흐름이 바뀌었다 — "맞는 모델 찾기"에서 "일의 성격 읽기"로

로컬 모델 연구의 질문이 달라지고 있습니다. 예전 질문은 **"내 장비에 어떤 모델이 들어가나"**였습니다 — 모델 무게와 KV 캐시를 더해 메모리에 맞추는 것(모듈 3의 산정법). 그 산정법은 여전히 유효하지만, 최전선이 보여준 새 질문이 하나 얹혔습니다:

> 💡 **비유**
>
> "내 **업무의 성격**은 읽기가 많은가, 쓰기가 많은가, 같은 문서를 반복해서 읽는가?" — 같은 장비라도 업무 성격에 맞춰 운영을 바꾸면 몇 배가 달라진다는 것이, 지난 1년 대규모 인프라가 실증한 교훈입니다. 그리고 그 교훈은 규모와 무관하게 적용됩니다.

또 하나의 인식 전환 — **KV 캐시(읽어둔 내용의 메모장)는 "메모리 남는 자리"가 아니라 1급 자산**이라는 것. 대규모 업계는 이 메모장을 전담 창고에 보관하고 재활용하는 시스템까지 만들었습니다. 우리 규모에서도 이 메모장을 어떻게 다루느냐가 아래 세 카드의 공통 열쇠입니다.

02 카드 ① 프리픽스 캐싱 — 같은 문서를 두 번 읽지 않는다

업무용 AI 요청은 앞부분이 반복됩니다. 시스템 지시문(역할·규칙)은 매번 같고, 같은 규정 문서를 붙여 여러 질문을 하는 일도 흔합니다. **프리픽스 캐싱(prefix caching)**은 그 겹치는 앞부분의 읽기 결과(KV 캐시)를 보관해뒀다가 재활용하는 기술입니다 — 겹치는 만큼 프리필(읽기)을 통째로 건너뜁니다.

- ✔ 대규모 실증: 한 대형 모델 서비스의 코딩 업무에서 **캐시 적중률 약 90%** — 요청 열 개 중 아홉은 읽기 대부분을 재활용했고, 캐시된 입력의 가격은 새 입력의 몇 분의 일이었습니다.
- ✔ **우리도 이미 갖고 있습니다.** llama.cpp 서버는 슬롯별로 직전 대화의 앞부분이 겹치면 자동 재활용하고, Ollama도 같은 원리로 동작합니다. 다만 "그냥 두면 알아서"가 아니라 **설계해야 적중합니다** — 시스템 지시문을 매번 똑같이 유지하고, 자주 쓰는 문서를 프롬프트 앞쪽에 고정 배치하는 습관이 적중률을 만듭니다.
- ✔ 세대가 오래된 GPU일수록 프리필이 약점입니다 — 읽기를 건너뛰게 해주는 이 카드가 **그런 장비에서 오히려 더 값집니다.**

> 💡 **비유**
> 회의 때마다 참석자 전원에게 배경 브리핑을 처음부터 다시 하는 팀과, 브리핑 문서를 한 번 읽혀두고 "지난번 그 문서 기준으로 질문만" 하는 팀 — 프리픽스 캐싱은 후자의 방식입니다.

03 카드 ② 투기 디코딩 — 작은 모델이 초안을 쓰고 큰 모델이 검수한다

쓰기(디코드)가 느린 이유는 한 글자 만들 때마다 모델 전체를 메모리에서 다시 읽기 때문이었습니다(모듈 2). **투기 디코딩(speculative decoding)**은 이 병목을 영리하게 우회합니다 — **작고 빠른 모델이 다음 몇 글자를 미리 추측(초안)하고, 큰 모델이 그 초안을 한 번에 검증**합니다. 맞으면 여러 글자를 한 번의 메모리 왕복으로 확정하고, 틀리면 그 지점부터 큰 모델이 직접 씁니다.

- ✔ 결과 품질은 **큰 모델이 혼자 쓴 것과 동일**합니다 — 틀린 추측은 반드시 걸러지니까요. 빨라질 뿐 나빠지지 않는, 드문 공짜 점심입니다.
- ✔ 최전선에서는 이 아이디어가 모델 안에 내장되는 단계(MTP — 다중 토큰 예측)까지 갔고, 우리 규모에서는 llama.cpp의 드래프트 모델 옵션(`--model-draft`)으로 같은 원리를 쓸 수 있습니다 — 예: 26B 주력 옆에 1B급 초안 모델을 붙이는 구성.
- ✔ 단, 공짜는 아닙니다 — 초안 모델도 VRAM을 먹습니다. VRAM이 빠듯한 장비(예: 24GB급 단일 GPU)

에겐 **"벤치로 이득을 실측한 뒤 판단"**이 정답입니다(늘 그렇듯).

> 💡 **비유**
>
> 초벌 번역가가 빠르게 초안을 쓰고 수석 번역가가 훑으며 승인 도장을 찍는 번역 사무소 — 수석이 직접 다 쓰는 것보다 훨씬 빠르지만, 최종 품질은 수석의 기준 그대로입니다. 어딘가 낯익지 않나요? 우리 검증 게이트의 "생성과 검증 분리"와 같은 사고방식이 속도 최적화에 쓰인 것입니다.

04 카드 ③ KV 오프로딩 — 메모장을 책상 서랍으로

VRAM이 부족할 때 모델 일부를 일반 메모리(RAM)로 내리는 오프로딩은 모듈 1에서 배웠습니다. 같은 일을 **KV 캐시에도** 할 수 있습니다 — 당장 안 쓰는 대화의 메모장을 RAM으로 내려두고, 필요할 때 다시 올리는 것. 대규모 업계는 이걸 SSD까지 계층화한 전용 창고 시스템(Mooncake 등, 모듈 8)으로 키웠고, 우리 규모에서는 llama.cpp의 옵션으로 원시적 형태를 쓸 수 있습니다.

- ✔ 쓰임새: 동시 사용자를 늘리고 싶은데 KV 자리가 모자랄 때, 속도를 조금 내주고 자리를 버는 교환.
- ✔ 주의: RAM은 VRAM보다 수십 배 느립니다 — 자주 쓰는 캐시까지 내려가면 오히려 전체가 느려집니다. **교환의 손익은 역시 실측으로만 확인됩니다.**
- ✔ 시스템 RAM이 넉넉한 장비(예: 맥 스튜디오급 64GB+ 통합 메모리)라면, 이 카드는 "쓸 일이 오면 쓸 수 있는 예비 카드"로 기억해두면 됩니다.

05 세 카드를 우리 지도에 놓으면

카드	무엇을 아끼나	우리 환경에서	발동 조건
프리픽스 캐싱	읽기(프리필) 반복	**지금 바로** — 프롬프트 앞부분 고정 설계만 하면 자동	같은 지시문·문서를 반복 사용하는 업무(우리 태스크 전부)
투기 디코딩	쓰기(디코드) 왕복	가능 — 단 초안 모델의 VRAM 비용과 교환	긴 출력이 많고 VRAM에 여유가 생겼을 때, 벤치 후

KV 오프로딩	VRAM 자리	예비 카드 — RAM 넉넉, 속도와 교환	동시 사용자가 늘어 KV가 모자랄 때

공통점이 보이시나요. 셋 다 **"어떤 모델이냐"가 아니라 "어떻게 돌리느냐"의 기술**입니다. 로컬 모델 연구의 무게중심이 모델 고르기에서 **운영 설계**로 옮겨가고 있다는 것 — 이것이 이번 모듈의 진짜 요지이고, 하드웨어까지 이 방향으로 갑니다(차세대 칩은 읽기용과 쓰기용을 아예 따로 설계합니다, 모듈 8 참조). 장비가 낡아도 운영 설계는 배울 수 있고, 그 설계 감각은 어떤 장비로 갈아타도 따라갑니다.

우리 연구회에선

출처: CV-Learn "효율적 AI 인프라"(모듈 8과 같은 글 — 이번엔 "소규모에 내려앉는 것" 관점으로 다시 읽음). 캐시 적중률 90% 등 수치는 해당 글이 정리한 업계 공개 실측이며, 우리 환경 수치는 아닙니다 — **우리 숫자는 우리 벤치가 만듭니다.**

↗ 심화 학습

더 깊이: llama.cpp 공식 문서의 서버 옵션(`--model-draft`, 캐시 관련 플래그) 설명과 vLLM 블로그의 automatic prefix caching 글. · 옵션 이름과 동작은 버전에 따라 바뀝니다 — 우리 서버 도커 이미지 기준으로 확인 후 사용하세요.

06 퀴즈 — 인접 개념을 가려내기

CHECK

이해했는지 확인해 봅시다

틀려도 괜찮습니다 — 오답을 고르면 왜 아닌지 설명이 나오고, 정답을 찾을 때까지 다시 고를 수 있습니다. 점수는 첫 시도 기준입니다.

문제 1 / 5

Q1. 프리픽스 캐싱이 아끼는 것은?

① 쓰기(디코드) 단계의 메모리 왕복

② 요청 앞부분이 이전과 겹칠 때의 읽기(프리필) 계산

③ 모델 파일이 차지하는 VRAM

④ 답변의 총 길이

← 이전
모듈 9 · 분산 클라우드

다음 →
모듈 11 · 에이전트를 믿을 수 있는 선까지