

# Ch08 sLLM 서빙하기

## ~ 효율적인 추론을 위한 전략과 실습 ~



2025-04-19

송태영

# Agenda

- **효율적인 배치 전략**
  - 일반 / 동적 / 연속 배치 비교
- **트랜스포머 연산 최적화**
  - 플래시어텐션, 상대적 위치 인코딩 등
- **효율적인 추론 전략**
  - 커널 퓨전, 페이지어텐션, 추측 디코딩
- **실습: vLLM 프레임워크**
  - 오프라인/온라인 서빙 실습
  - n8n 연동
- **요약 및 마무리**

# 8.1 효율적인 배치 전략

-  **요지:**
  - LLM은 한 토큰씩 생성
  - 입력마다 생성 길이가 달라 추론 시간 불균형 발생
  - 문제점:
    - 일부 요청이 끝나도 대기 상태 유지
    - GPU 자원 낭비
-  **해결 전략:**
  - 8.1.1 일반 배치
  - 8.1.2 동적 배치
  - 8.1.3 연속 배치

# 8.1.1 일반 배치 (Static Batching)

- 📌 **설명:**
  - 정해진 개수(N)의 요청을 한 번에 처리
  - 생성이 끝난 요청도 모두 완료될 때까지 대기
- 📈 **단점:**
  - 빠르게 끝난 요청도 지연됨
  - 추론 후반부에는 배치 크기 감소 → GPU 효율 저하

T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	T <sub>4</sub>	T <sub>5</sub>	T <sub>6</sub>	T <sub>7</sub>	T <sub>8</sub>
S <sub>1</sub>	S <sub>1</sub>	S <sub>1</sub>	S <sub>1</sub>				
S <sub>2</sub>	S <sub>2</sub>	S <sub>2</sub>					
S <sub>3</sub>	S <sub>3</sub>	S <sub>3</sub>	S <sub>3</sub>				
S <sub>4</sub>	S <sub>4</sub>	S <sub>4</sub>	S <sub>4</sub>	S <sub>4</sub>			

(a)

T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	T <sub>4</sub>	T <sub>5</sub>	T <sub>6</sub>	T <sub>7</sub>	T <sub>8</sub>
S <sub>1</sub>	S <sub>1</sub>	S <sub>1</sub>	S <sub>1</sub>	S <sub>1</sub>	END		
S <sub>2</sub>	S <sub>2</sub>	S <sub>2</sub>	S <sub>2</sub>	S <sub>2</sub>	S <sub>2</sub>	S <sub>2</sub>	END
S <sub>3</sub>	S <sub>3</sub>	S <sub>3</sub>	S <sub>3</sub>	END			
S <sub>4</sub>	S <sub>4</sub>	S <sub>4</sub>	S <sub>4</sub>	S <sub>4</sub>	S <sub>4</sub>	END	

(b)

그림 8.1 한 번 받은 배치 데이터가 모두 끝날 때까지 처리하는 일반 배치 전략

(출처: <https://www.anyscale.com/blog/continuous-batching-llm-inference>)

## 8.1.2 동적 배치 (Dynamic Batching)

- 📌 **설명:**
  - 일정 시간 동안 요청을 모아 동시에 처리
  - 대기 시간 약간 ↑, GPU 처리율 향상
- 📌 **상황 예시:**
  - 요청 A(1ms), 요청 B(2ms) → 함께 처리
- ⚠️ **한계:**
  - 생성 길이 차이 → 여전히 배치 크기 감소
  - 연속 배치로 해결

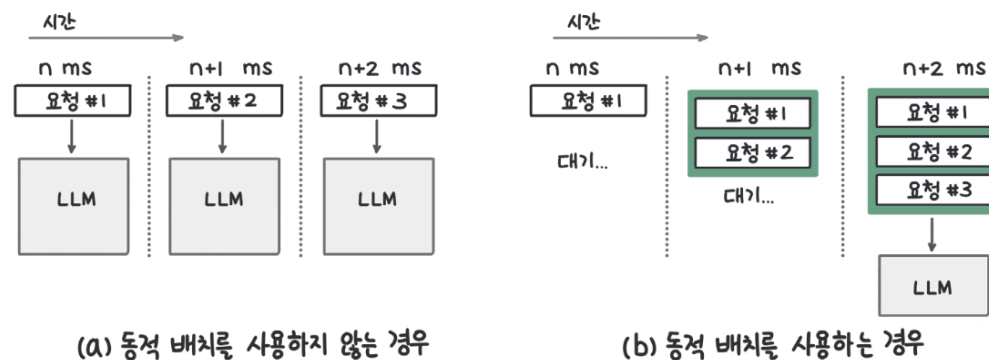


그림 8.2 일정한 시간 동안의 요청을 묶어서 처리하는 동적 배치 전략

(출처: <https://clear.ml/blog/increase-huggingface-triton-throughput-by-193/>)

## 8.1.3 연속 배치 (Continuous Batching)



### 설명:

- 생성 완료된 요청은 즉시 제거
- 새로운 요청을 빈 자리에 추가
- 배치 크기 유지 → GPU 자원 최적 활용



### 장점:

- 응답 지연 최소화
- 서버 처리량 ↑

T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	T <sub>4</sub>	T <sub>5</sub>	T <sub>6</sub>	T <sub>7</sub>	T <sub>8</sub>
S <sub>1</sub>	S <sub>1</sub>	S <sub>1</sub>	S <sub>1</sub>	S <sub>1</sub>	END		
S <sub>2</sub>	S <sub>2</sub>	S <sub>2</sub>	S <sub>2</sub>	S <sub>2</sub>	S <sub>2</sub>	S <sub>2</sub>	END
S <sub>3</sub>	S <sub>3</sub>	S <sub>3</sub>	S <sub>3</sub>	END			
S <sub>4</sub>	S <sub>4</sub>	S <sub>4</sub>	S <sub>4</sub>	S <sub>4</sub>	S <sub>4</sub>	END	

(a)

T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	T <sub>4</sub>	T <sub>5</sub>	T <sub>6</sub>	T <sub>7</sub>	T <sub>8</sub>
S <sub>1</sub>	S <sub>1</sub>	S <sub>1</sub>	S <sub>1</sub>	S <sub>1</sub>	END	S <sub>6</sub>	S <sub>6</sub>
S <sub>2</sub>	S <sub>2</sub>	S <sub>2</sub>	S <sub>2</sub>	S <sub>2</sub>	S <sub>2</sub>	S <sub>2</sub>	END
S <sub>3</sub>	S <sub>3</sub>	S <sub>3</sub>	S <sub>3</sub>	END	S <sub>5</sub>	S <sub>5</sub>	S <sub>5</sub>
S <sub>4</sub>	S <sub>4</sub>	S <sub>4</sub>	S <sub>4</sub>	S <sub>4</sub>	S <sub>4</sub>	END	S <sub>7</sub>

(b)

그림 8.3 생성이 끝나면 새로운 문장을 추가하는 연속 배치 전략

(출처: <https://www.anyscale.com/blog/continuous-batching-llm-inference>)

## 8.2 효율적인 트랜스포머 연산

### 문제 인식:

- 트랜스포머의 핵심인 **Self-Attention** 연산은 계산량 & 메모리 사용량이 큼
- 특히 긴 시퀀스 처리 시 병목 심화

### 해결책:

- 8.2.1 플래시어텐션
- 8.2.2 플래시어텐션 2
- 8.2.3 상대적 위치 인코딩
- **목표:** 동일 성능 유지하면서 속도 & 메모리 최적화

# 8.2.1 플래시어텐션 (FlashAttention) - 1

## • 왜 필요한가?

- 기존 Self-Attention 연산은
  - 어텐션 행렬( $N \times N$ )을 전부 계산 & 저장
  - GPU 메모리 대량 소비 + 속도 병목 발생
- 특히 문제되는 부분:
  - 마스크, 소프트맥스, 드롭아웃 등의 연산이
  - 큰 메모리 이동을 수반 → 느림
  - 행렬 곱보다도 느린 경우 다수

입력:  $Q, K, V$

$$O = \text{드롭아웃}(\text{소프트맥스}(\text{마스크}(QK^T)))V$$

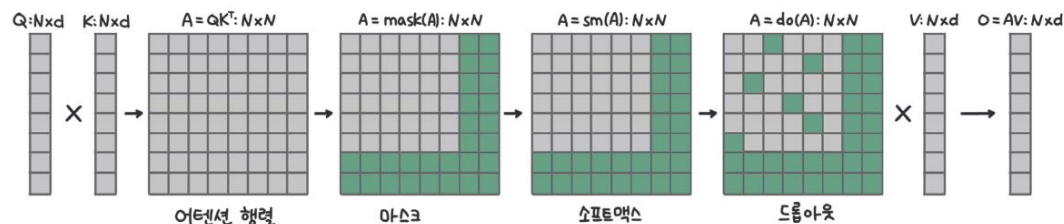


그림 8.4 셀프 어텐션의 연산 과정

(출처: [https://www.youtube.com/watch?v=FTHvfKXWqtE&ab\\_channel=StanfordMedAI](https://www.youtube.com/watch?v=FTHvfKXWqtE&ab_channel=StanfordMedAI))

## ⚙️ 병목 원인은 계산량이 아니라 메모리 I/O

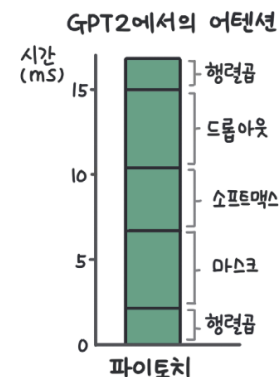


그림 8.5 셀프 어텐션의 각 연산에 걸리는 시간  
(출처: <https://arxiv.org/abs/2205.14135>)

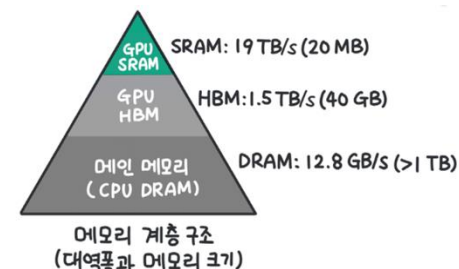






그림 8.6 GPU 메모리의 계층 구조(출처: <https://arxiv.org/abs/2205.14135>)



## 8.2.1 플래시어텐션 (FlashAttention) - 2

-  어떻게 개선했나?
  -  블록 단위 연산
    - 전체 어텐션 행렬 저장 X
    - SRAM에서 블록 단위 연산만 수행
  -  소프트맥스도 부분 연산 후 결합 (Tiling Softmax)
    - 전체를 직접 계산하지 않고, 블록 별 max, sum 값을 활용해 전체 softmax 값 도출
  -  드롭아웃 등도 연산 내에 통합 (커널 퓨전)
    - 중간 결과 저장/복사 최소화

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \rightarrow \text{softmax}(x) = \frac{f(x^{(1)}), f(x^{(2)})}{l(x^{(1)}) + l(x^{(2)})}$$

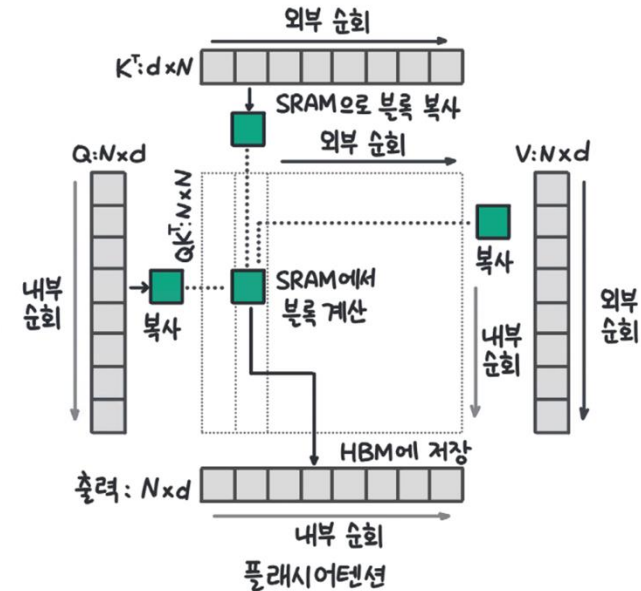


그림 8.7 플래시어텐션에서 어텐션 연산 과정을 개선한 방식(출처: <https://arxiv.org/abs/2205.14135>)

## 8.2.2 플래시어텐션 2 - 1

### 📌 주요 개선 사항:

1. 비행렬 연산 최소화 → 행렬 연산에 최적화된 GPU 특성 활용
2. 시퀀스 방향 병렬화 추가 → 더 많은 SM 사용 가능

### 📊 결과:

- 기존 대비 약 2배 속도 향상
- GPU 활용률 최대 70% 이상까지 도달

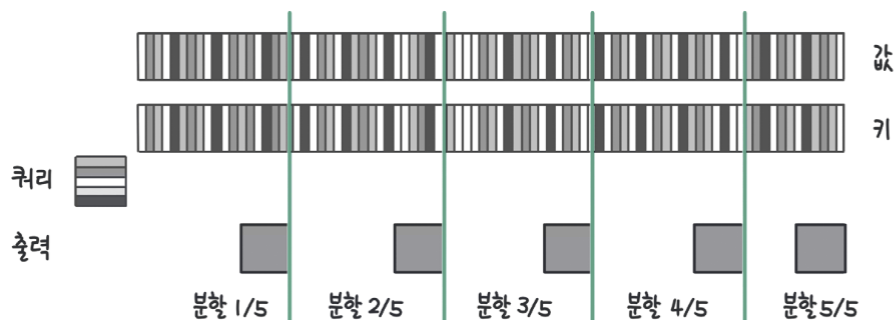


그림 8.15 시퀀스 길이 방향의 병렬 연산(출처: <https://pytorch.org/blog/flash-decoding/>)

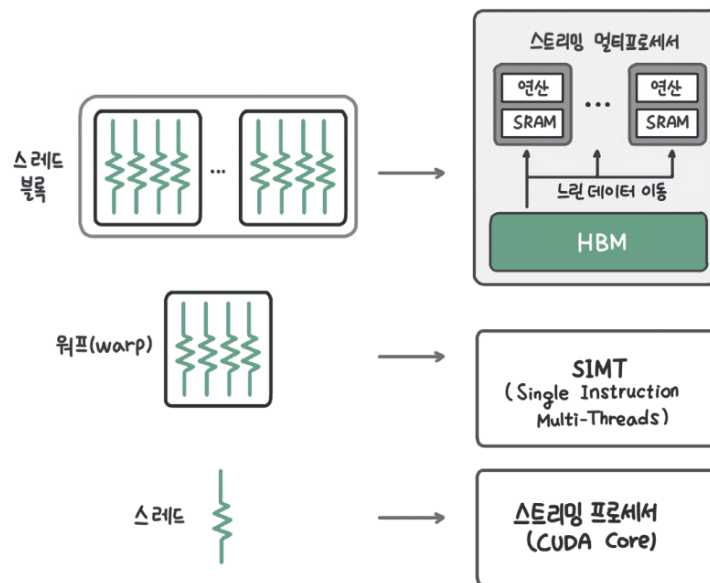


그림 8.14 GPU에서 계산되는 단위와 GPU의 물리적 구조 대응 관계

## 8.2.2 플래시어텐션 2 - 2

### 😞 문제가 생기는 경우

- 예: 배치 크기 2, 헤드 수 8 → 일꾼 16개 (일꾼 = GPU SM(Streaming Multiprocessor))
- 근데 A100 GPU는 일꾼 100개 이상이 있어요!
- 그럼 84개는 놀고 있는 거죠... 😞

### 💡 해결책: 시퀀스 길이로도 쪼개자!

- FlashAttention 2는 각 시퀀스를 더 작게 나눠서 일꾼 수를 늘립니다.

### 🧩 비유로 설명하자면

- FlashAttention 1: "작업을 문장 단위로 주고, 문장 하나는 한 명이 처리"
- FlashAttention 2: "작업을 문장 안에서도 쪼개서 여러 명이 나눠서 처리"

항목	FlashAttention 1	FlashAttention 2
배치 2, 헤드 8	$2 \times 8 = 16$ 블록	$2 \times 8 \times 4\text{분할} = 64$ 블록
시퀀스 길이	2048	512씩 4조각으로 쪼갬

## 8.2.3 상대적 위치 인코딩 (Relative Positional Encoding)

- 📌 **문제:**
  - 기존 절대 위치 인코딩 → 입력 길이 늘어나면 성능 저하
- 📌 **해결책:**
  - RoPE\*: 토큰 임베딩을 위치별로 회전 (각도 유지)
  - ALiBi\*\*: 어텐션 행렬에 선형 바이어스 적용
- 📈 **효과:**
  - 긴 입력에도 성능 유지
  - ALiBi는 속도 저하도 거의 없음

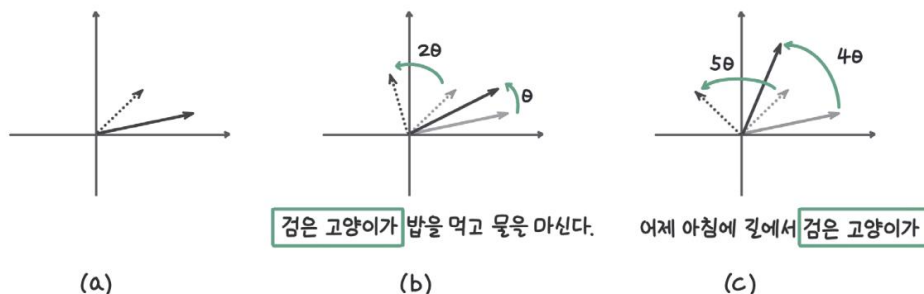


그림 8.18 RoPE의 상대적 위치 인코딩 방식

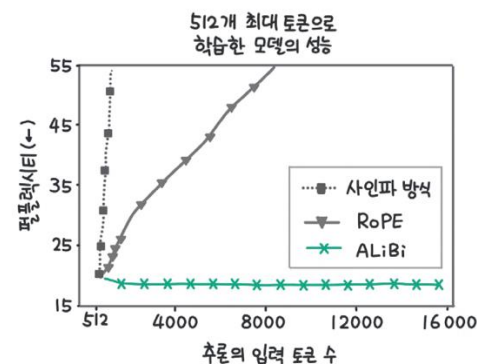


그림 8.20 위치 인코딩 방식에 따른 외삽 성능(출처: <https://arxiv.org/pdf/2108.12409.pdf>)

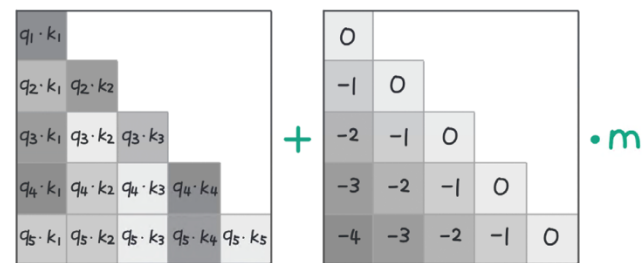





그림 8.19 ALiBi의 상대적 위치 인코딩 방식(출처: <https://arxiv.org/abs/2108.12409>)

\* RoPE (Rotary Positional Encoding)

\*\* ALiBi (Attention with Linear Biases)

## 8.3 효율적인 추론 전략

-  왜 필요한가?
  - LLM 추론에서 속도를 높이려면 계산 효율뿐 아니라 메모리 효율과 연산 최적화가 필수
-  주요 전략들:
  - 8.3.1 커널 퓨전 (Kernel Fusion)
  - 8.3.2 페이지어텐션 (Paged Attention)
  - 8.3.3 추측 디코딩 (Speculative Decoding)
-  공통 목표: 낭비되는 연산 제거 + GPU 최대 활용

## 8.3.1 커널 퓨전 (Kernel Fusion)

- 📌 **핵심 개념:**
  - GPU는 연산을 커널 단위로 수행
  - 여러 커널을 따로 실행하면 데이터 이동 오버헤드 발생
- ✂ **해결 방법:**
  - 자주 함께 쓰이는 연산들을 하나의 커널로 병합  
예: 마스크 → 소프트맥스 → 드롭아웃 → 행렬곱
  - 대표 사례: 플래시어텐션
- 📈 **효과:**
  - 중간 데이터를 GPU 메모리에 저장하지 않음
  - 연산 시간 단축, 메모리 절약

$y = \text{relu}(\text{matmul}(x, w))$

matmul → 커널 1개

relu → 커널 1개

👉 2개의 커널이 실행. 그 사이에 메모리 읽기/쓰기 발생

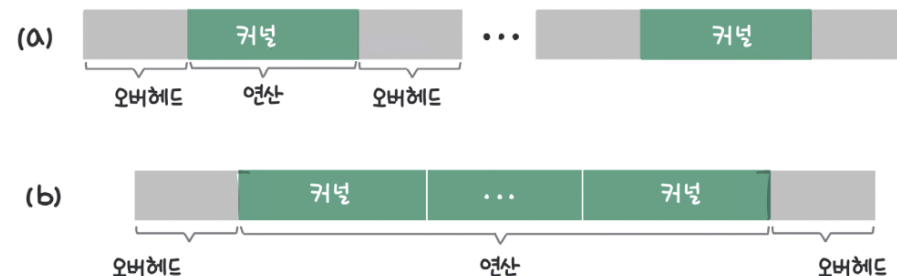


그림 8.22 커널 퓨전이 연산을 효율적으로 만드는 이유

(출처: <https://codeplay.com/portal/blogs/2023/03/20/user-driven-kernel-fusion>)

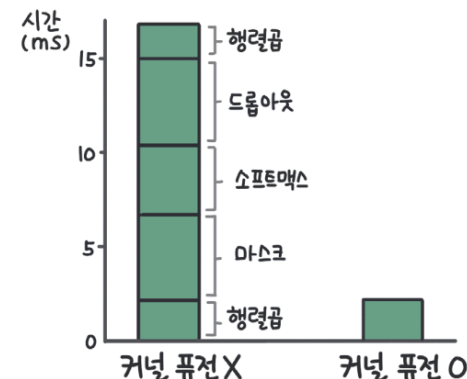


그림 8.23 커널 퓨전을 사용해 연산 시간을 줄인 플래시어텐션(출처: <https://arxiv.org/abs/2205.14135>)

## 8.3.2 페이지어텐션 (PagedAttention)

- 📌 **문제:**
  - KV 캐시를 최대 토큰 수 기준으로 미리 메모리 할당
  - 실제 사용량보다 훨씬 많은 메모리 낭비
- 📌 **해결:**
  - 운영체제의 가상 메모리 기법을 도입
  - KV 캐시를 논리적 블록 ↔ 물리적 블록으로 관리
  - 필요한 만큼만 동적 할당 (예: 4토큰 단위 블록)

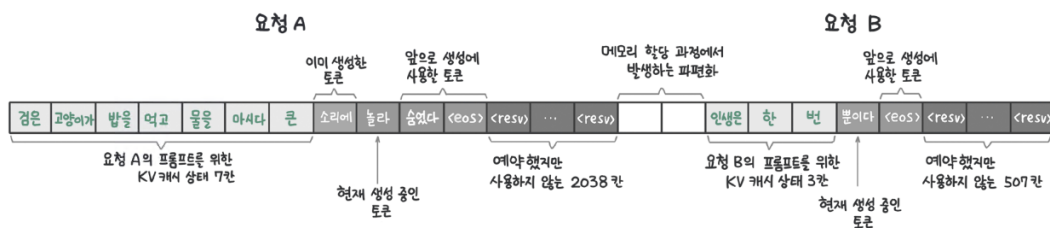


그림 8.25 기존 KV 캐시 저장 방식의 문제점(출처: <https://arxiv.org/pdf/2309.06180.pdf>)

- 💡 **추가:**
  - 동일 프롬프트를 공유하는 경우에는 참조 카운트로 메모리 절약
  - 이후 분기 시에만 블록을 복사 (copy-on-write)
- 📈 **효과:**
  - 메모리 낭비 최소화
  - 최대 배치 크기 5~6배 증가

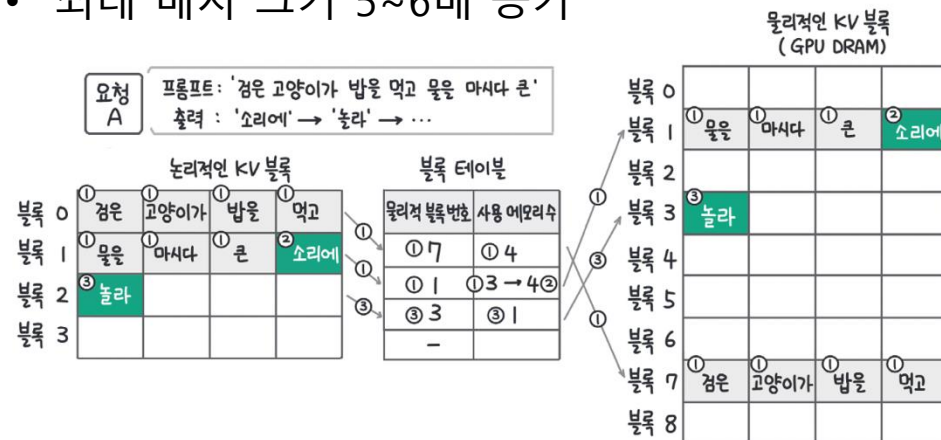


그림 8.26 페이지어텐션의 KV 캐시 관리 방식(출처: PagedAttention 논문)

## 8.3.3 추측 디코딩 (Speculative Decoding)



### 동기:

- 일부 단어는 예측이 쉽고, 일부는 어려움
- 쉬운 건 작은 모델에게 맡기고 → 검증은 큰 모델이!



### 구조:

- 드래프트 모델 (작은) → 먼저 K개 생성
- 타깃 모델 (큰) → 검증 후 승인 or 수정

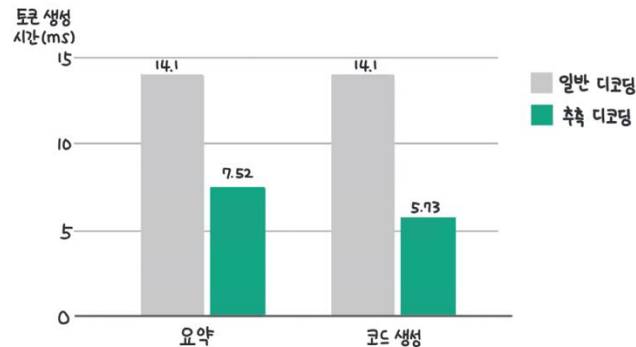


그림 8.29 추측 디코딩을 통한 속도 향상

(출처: [https://www.youtube.com/watch?v=TJ5K1C09Wbs&t=1589s&ab\\_channel=Anyyscale](https://www.youtube.com/watch?v=TJ5K1C09Wbs&t=1589s&ab_channel=Anyyscale))

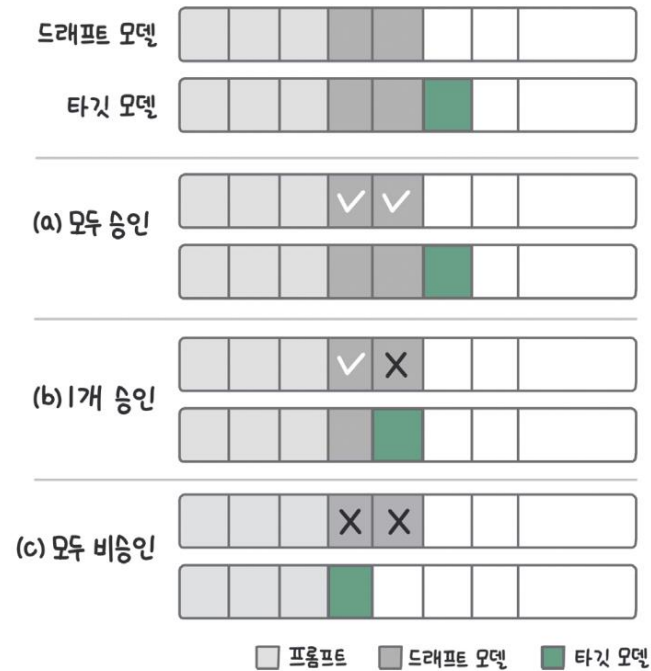


그림 8.28 추측 디코딩 방식의 추론 과정



## 8.4 실습 – vLLM 프레임워크 개요

### 실습 목표:

- vLLM을 활용해 학습된 LLM 모델을 효율적으로 서빙
- 오프라인 & 온라인 추론 모두 실습

### vLLM 특징:

- Hugging Face 모델 서빙 최적화
- 배치 처리 & KV 캐시 & PagedAttention 지원
- 지원 모델 리스트: [docs.vllm.ai](https://docs.vllm.ai)

### 실습 환경:

- Colab 환경 기반
- 필요한 라이브러리: transformers, vllm, datasets, accelerate, 등

## 8.4.1 오프라인 서빙 실습

### 개념:

- 사전 정의된 데이터셋을 대상으로 일괄 추론 수행
- → 배치 크기에 따라 추론 속도 측정

### 실습 흐름 요약:

1. Hugging Face에서 ko\_text2sql 데이터셋 로딩
2. make\_prompt() 함수로 입력 포맷 생성
3. transformers.pipeline()을 활용한 텍스트 생성
4. batch\_size를 1 → 2 → 4 ... 32로 바꾸며 성능 측정

### 관찰 포인트:

- 배치 크기 증가에 따른 추론 속도 변화

## 8.4.2 온라인 서빙 실습

- 📌 개념:
  - 사용자 요청이 들어올 때마다 모델 추론 수행
  - 실시간 응답을 위한 최적화 필요
- 💡 실습 구성:
  1. vllm.entrypoints.api\_server 모듈로 API 서버 실행
  2. 사용자 입력 프롬프트에 대한 HTTP POST 요청
  3. 응답 시간, 추론 결과 확인
- 📌 주요 기능:
  - --model, --tokenizer, --port, --max-model-len 등 다양한 CLI 옵션 제공
- 💡 배치 전략(연속 배치), KV 캐시 자동 적용됨

# 실습 요약 & 정리

- 📌 실습 요약:

항목	오프라인 추론	온라인 추론
입력	사전 정의됨	실시간
처리	대량 배치	소량/단건
목적	처리량 ↑	응답속도 ↑
적용 기술	배치 크기 튜닝	연속 배치 + 캐시

- 🏠 배운 점:

- vLLM을 활용해 **효율적인 서버 구조** 체험
- 배치 전략과 추론 최적화가 **실제 성능에 미치는 영향** 직접 확인

## 8.5 정리

-  이 장에서 배운 것

구분	핵심 내용
배치 전략	일반 배치, 동적 배치, 연속 배치 비교 → 연속 배치로 대기 시간 최소화
트랜스포머 최적화	FlashAttention 1 & 2, 상대적 위치 인코딩 → 연산 효율 & 입력 길이 확장
추론 전략	커널 퓨전, 페이지어텐션, 추측 디코딩 → GPU 활용 & 속도 개선
서빙 실습	vLLM으로 오프라인/온라인 추론 → 배치 크기와 실시간성의 균형 이해

### 핵심 인사이트

- 추론 속도는 단순 모델 경량화로만 해결되지 않는다  
→ 메모리 효율, 배치 전략, 연산 병합, 캐시 최적화 모두 중요
- 최신 서빙 프레임워크(vLLM 등)는  
→ 이 모든 전략을 통합적으로 적용
- 하지만 tool\_calling 지원은 빨리 좀 ...

Q&A