# 제18장. 고급 프리필-디코딩 및 키-값 캐시 튜닝



본 장은 제17장을 기반으로 추론 프리필 및 디코드 단계에 대한 고급 최적화를 심층적으로 다룹니다. 고수준 확장 전략을 바탕으로 단일 디코드 "메가 커널", 지능형 KV 캐시 튜닝 및 GPU 간 공유, prompt 상태의 빠른 GPU-to-GPU 전송, 적응형 리소스 스케줄링, 프리필 및 디코드 작업자 간 동적 라우팅 등 저수준 기법을 다룹니다.

또한 새로운 수준의 성능과 효율성을 제공하는 하드웨어 및 소프트웨어 혁신을 강조할 것입니다. 이러한 기법을 적용하면 디코드 지연 시간을 크게 줄이고 GPU당 처리량을 향상시키며 대규모 환경에서 엄격한 지연 시간 SLO를 충족할 수 있습니다.

## 최적화된 디코드 커널

지금까지 우리는 에서 고수준 시스템 및 클러스터 최적화 전략에 집중해 왔습니다. 초고성능 추론을 확장할 때 고려해야 할 또 다른 기술 세트는 저수준 커널 및 메모리 관리 튜닝, 특히 디코딩 단계에 대한 것입니다.

디코딩 단계는 분산 처리되며 종종 메모리 바운디드에 직면합니다. 이로 인해 연구자와 실무자들은 디코딩 단계를 최대한 빠르게 만들고 특정 하드웨어에 최적화하기 위해 노력해 왔습니다. 이 분야에서 주목할 만한 두 가지 혁신은 FlashMLA(DeepSeek), ThunderMLA(Stanford), FlexDecoding(PyTorch)입니다. 이들은 특히 LLM 워크로드에서 흔히 발생하는 가변 시퀀스 시나리오에서 디코딩 중 트랜스포머의 멀티헤드 어텐션 효율성을 목표로 합니다. 이제 각각을 살펴보겠습니다.

### FlashMLA (DeepSeek)

Flash Multi-Latent Attention, 즉 FlashMLA 는 DeepSeek에서 도입한 최적화된 디코딩 커널입니다. 이는 다음 토큰을 생성하는 트랜스포머 레이어의 포워드 패스인 단일 토큰 디코딩 단계에 특히 초점을 맞춥니다. FlashMLA는 연산 융합과 GPU 메모리 계층 구조의 효율적 활용을 통해 디코딩 속도를 향상시킵니다.

FlashMLA(디코딩)은 추론에 있어 FlashAttention(프리필)이 훈련에 기여하는 것과 유사한 역할을 합니다. 메모리 접근 오버헤드와 지연 시간을 줄여줍니다. FlashMLA를 사용하면 표준 커널 대비 디코딩 단계에서 상당한 지연 시간 감소를 달성할 수 있습니다.

FlashMLA는 여러 어텐션 연산을 하나로 융합하여 산술 집약도를 높입니다. 이를 통해 하나의 융합 커널 실행으로 여러 헤드와 여러 시간 단계를 처리할 수 있습니다. 이는 작은 배치 크기에도 불구하고 수학 유닛을 지속적으로 활용함으로써 디코딩 중 GPU 사용률을 높입니다. 그림 18-1은 Hopper H100 GPU에서 그룹 쿼리 어텐션(GQA) 및 멀티쿼리 어텐션(MQA)과 같은 다른 어텐션 구현 대비 MLA의 산술 집약도 개선을 보여줍니다. (참고: Blackwell은 더 높은 TFLOPs와 HBM 대역폭으로 두 루프라인을 모두 상향 조정합니다.)

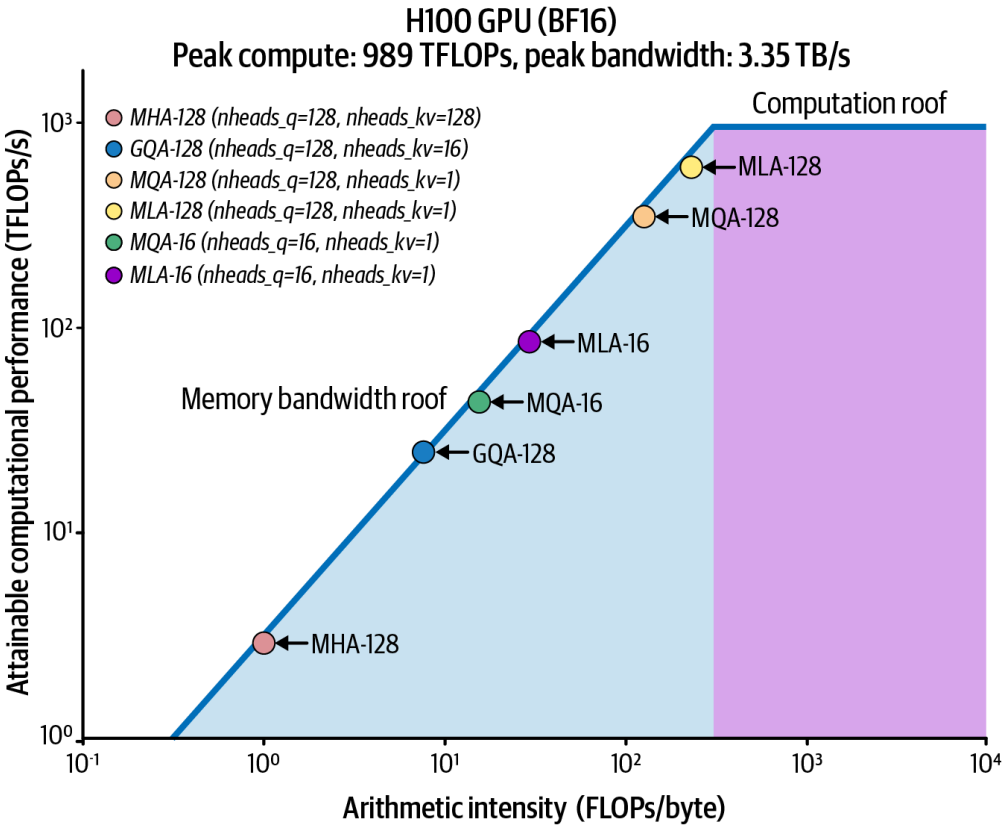


그림 18-1. MLA가 컴퓨팅 바운드 영역에 접근하는 모습 (NVIDIA Hopper H100 아키텍처에서 측정)

FlashMLA의 도입은 디코딩 단계의 병목 현상인 메모리 대역폭과 커널 런치 오버헤드를 비최적 GPU 하드웨어에서도 줄일 수 있음을 보여준 점에서 중요했습니다. 이는 별도의 GPU 커널 런치 횟수를 줄이고 메모리 접근 패턴을 최적화하여 디코딩 작업에 제약된 하드웨어에서 최대한의 성능을 끌어냈습니다.

DeepSeek의 오픈소스 FlashMLA 구현체가 공개되어 채택되고 있습니다. SGLang과 vLLM 모두 DeepSeek 모델에 대한 최상위 지원을 제공합니다. 따라서 상위 아키텍처 변경 없이 토큰당 디코딩 처리량을 높이기 위해 FlashMLA를 평가해야 합니다.

DeepSeek의 오픈소스 FlashMLA가 현대적 추론 서비스 시스템에 통합되었으므로, 상위 아키텍처 변경 없이 각 디코드 작업자의 처리량 증가(또는 토큰당 지연 시간 감소)를 위한 방법으로 이를 검토해야 합니다.

## ThunderMLA (스탠퍼드)

FlashMLA를 기반으로 스탠퍼드의 연구진 은 ThunderMLA를 소개했습니다. 이는 완전 융합형 어텐션 디코딩 "메가커널"로, 전체 피드포워드 블록을 융합하기보다는 디코딩과 스케줄링에 집중합니다. 이 "메가커널"은 여러 커널 런치를 하나로 통합하고 중간 메모리 쓰기를 통합함으로써 런치 오버헤드와 테일 효과를 줄입니다. ThunderMLA는 다양한 워크로드에서 FlashMLA 대비 20~35% 빠른 디코딩 처리량을 보고합니다.

ThunderMLA의 핵심 아이디어는 서로 다른 길이의 시퀀스를 디코딩할 때, 세분화된 스케줄링과 융합 연산을 활용하면 일부 시퀀스가 일찍 완료되는 반면 다른 시퀀스는 GPU를 부분적으로 유휴 상태로 남겨두는 테일 효과를 피할 수 있다는 점입니다. ThunderMLA는 일부 디코딩 스트림이 일찍 완료되더라도 GPU를 계속 바쁘게 유지합니다. 이는 융합 접근법을 사용하여 남은 스트림을 동적으로 압축하고 처리함으로써 달성됩니다.

이러한 이점은 더 큰 L2 캐시와 더 빠른 어텐션 프리미티브를 갖춘 최신 GPU에서 더욱 증폭됩니다. 특히 최신 NVIDIA GPU는 FP8 및 FP4(FP6도 지원하나, 본 문서에서는 FP6이 기존 AI 프레임워크 및 도구에서 널리 사용되지 않으므로 주로 FP8/FP4 형식에 초점을 맞춥니다)에 대한 Transformer Engine 지원을 제공합니다. 더 높은 메모리 대역폭과 결합된 텐서 코어는 ThunderMLA와 같은 커널이 하드웨어 한계에 훨씬 더 가깝게 작동하도록 합니다. 이러한 아키텍처적 진보 덕분에 최신 GPU에서 ThunderMLA는 토큰당 더 낮은 지연 시간을 달성합니다.

## FlexDecoding (PyTorch)

14장에서는 PyTorch의 FlexAttention( )에 대해 논의했습니다. 이 기능은 로컬 윈도우, 블록 스파스 패턴 등 주의(attention)의 임의 스파스 패턴에 대해 융합 커널을 JIT 컴파일할 수 있게 해주며, 사용자 정의 CUDA 코드 작성 없이도 가능합니다. 내부적으로 TorchInductor와 OpenAI의 Triton은 해당 패턴에 허용된 쿼리-키 쌍만 계산하는 융합 커널을 생성합니다. Triton은 주어진 하드웨어에서 유익할 경우 워프 전문화(warp specialization) 및 비동기 복사(asynchronous copies)와 같은 성능 최적화 기법을 자동으로 적용합니다. 그러나 `num_consumer_groups` 와 같은 매개변수를 구성하여 `triton.Config` 을 추가로 맞춤 설정할 수도 있습니다.

FlexDecoding은 `torch.nn.attention.flex_attention` 의 디코딩 백엔드입니다. FlexDecoding은 KV를 인플레이스 관리할 수 있게 하며 FlexAttention과 마찬가지로 마스크 및 바이어스를 지원합니다. 구체적으로 FlexDecoding은 확장되는 KV 캐시를 처리하는 디코딩 단계 전용 커널 (`Q_len=1` )을 컴파일합니다.

실행 시 FlexDecoding 구현은 특수화된 디코딩 커널을 선택하여 여러 디코딩 단계에 걸쳐 재사용합니다. 이는 형상(shape)과 데이터 유형(dtype)이 호환될 때

오버헤드를 최소화하여 긴 시퀀스 LLM 추론을 크게 가속화합니다.

---

재컴파일 작업이 안정화되면 안정적이고 지연 시간이 중요한 디코딩에는 `torch.compile(mode="max-autotune")`를 우선적으로 사용하십시오. 캡처 경계를 좁게 유지(레이어별 또는 어텐션 블록별)하여 불규칙 배치로 인한 그래프 무효화를 줄이십시오. 프리필 및 디코딩에는 Transformer Engine FP8(MXFP8)를 우선적으로 사용하십시오. 정확도가 허용되고 성능이 향상될 경우 FP4(NVFP4)를 고려하십시오. 현재 FP4 지원은 아직 성숙 단계에 있으며, 단기적으로는 8비트 및 16비트 형식보다 성능이 떨어질 수 있습니다. 나머지 FP32 연산에 TF32 폴백을 활성화하려면 계속해서 `torch.set_float32_matmul_precision("high")`을 설정하십시오. FlexAttention의 디코드 백엔드는 그룹화 쿼리 어텐션(GQA) 및 PagedAttention을 포함한 일반적인 성능 향상 기능을 지원합니다.

---

FlexAttention과 FlexDecoding의 핵심 기능에는 중첩된 불규칙 레이아웃 텐서(NJT)에 대한 지원 이 포함됩니다. 이를 통해 디코딩 과정에서 가변 길이 시퀀스(LLM 워크로드에서 흔함)의 불규칙 배치 처리가 가능합니다. 다양한 시퀀스의 불규칙 텐서 표현은 그림 18-2에 표시되어 있습니다.

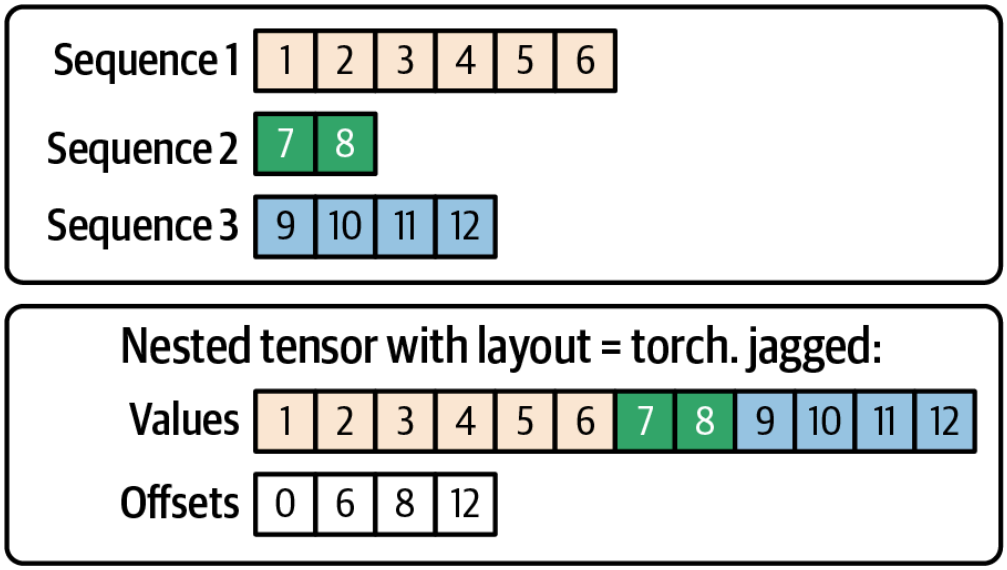


그림 18-2. 중첩된 잼지 텐서(오프셋) 형태의래그드 배치; 세 시퀀스(상단)를 오프셋이 포함된 단일 중첩 잼지 텐서 표현으로 나타냄(하단); 디코드 시 배치 처리를 위해 PyTorch NJT 사용 권장

또한 FlexDecoding은 바이어스 항을 지원하며, 는 논리 블록을 물리적 캐시 레이아웃에 매핑하는 블록 마스크 변환 인터페이스를 사용하여 PagedAttention과 통합됩니다. 이는 그림 18-3과 같이 추가 복사본 생성 없이 논리 KV 블록을 물리적 캐시 레이아웃에 분산시킵니다.

FlexDecoding은 캡처된 텐서를 활용하여 각 반복자 과정에서 특정 마스크 또는 바이어스 값을 변경합니다. 재컴파일 없이도 가능합니다. 또한 PagedAttention과 통합됩니다. vLLM LMCache와 같은 글로벌 KV 캐시를 사용하려면 캐시의 페이지 테이블을 FlexAttention의 BlockMask에 매핑하십시오. 이렇게 하면 논리적 KV 페이지를 물리적 메모리 주소로 실시간 변환합니다.

FlexDecoding을 통해 개발자는 사용자 정의 어텐션 스파서시티 패턴에 대해 Python 수준의 완전한 유연성을 확보합니다. 이는 특히 MoE 모델 추론에 유용합니다. FlexDecoding은 사용자 정의 CUDA 커널을 작성할 필요 없이 거의 최적의 성능을 달성할 수 있게 합니다. 본질적으로, 이는 임의의 어텐션 패턴을 밀집형 어텐션 패턴과 유사하게 최적화할 수 있게 합니다. 새로운 추론 기법이 등장함에 따라 이는 더욱 가치 있게 됩니다.

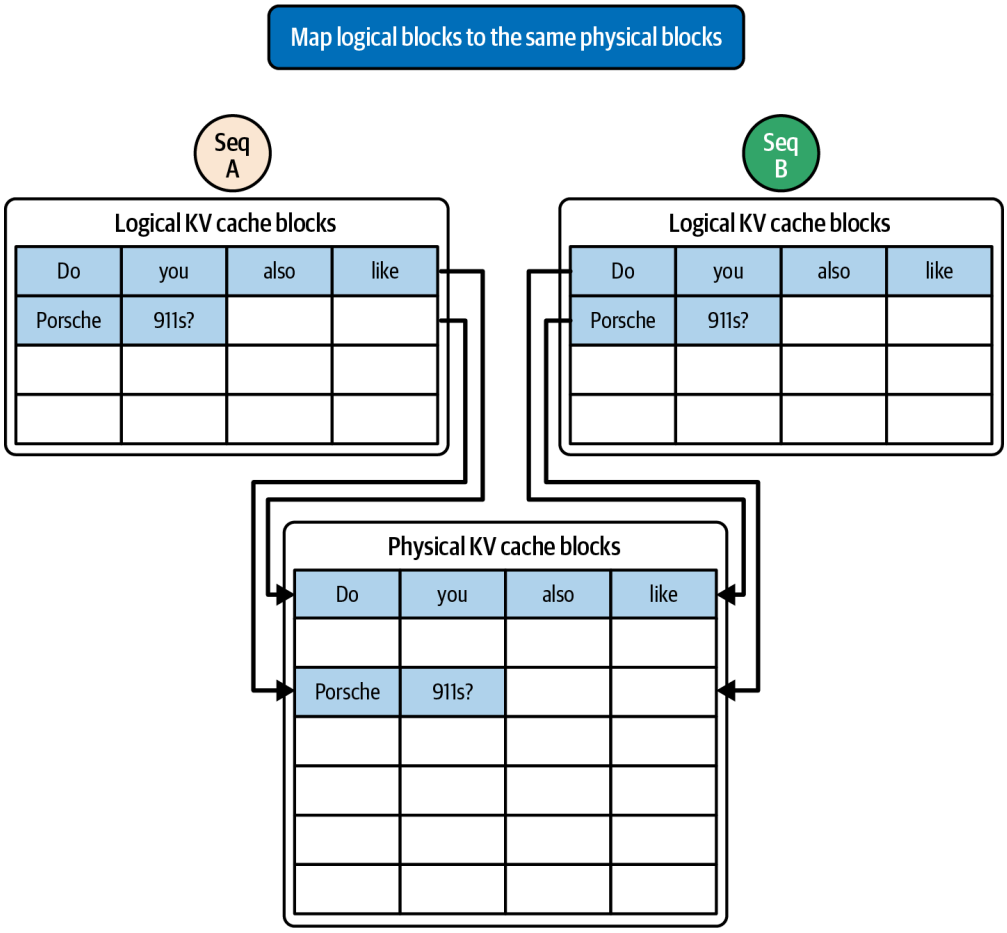


그림 18-3. PagedAttention은 시퀀스 간 최적의 캐시 재사용을 위해 논리적 KV 블록을 물리적 KV 블록으로 분산합니다; 블록 크기를 LMCache 페이지 크기와 정렬하세요—더 큰 페이지(예: 64–128 토큰)는 분산 환경에서 RDMA 오버헤드를 줄입니다

와 같은 디코딩용 융합 어텐션 및 PyTorch의 중첩된 불규칙 텐서(NJT) 배치 지원과 같은 많은 기능이 핵심 PyTorch 라이브러리에서 제공됩니다. 이는 일반적인 패턴에 대해 맞춤형 융합이 덜 필요하게 만듭니다.

---

LLM 워크로드에서 흔히 발생하는 불규칙한 시퀀스를 배치 처리할 때는 NJT 레이아웃을 선호하십시오.

---

이러한 커널 수준의 발전은 매우 기술적이며 GPU, 네트워크, 메모리의 모든 성능을 활용합니다. 이러한 소프트웨어 최적화는 동일한 하드웨어에서도 디코딩 성능을 크게 향상시킬 수 있습니다. 초대형 시스템을 설계할 때는 가능하면 이러한 최적화된 커널을 통합해야 합니다. 하드웨어 기반 CUDA 추적을 사용하여 Nsight Systems로 오버랩과 커널 효율성을 반드시 검증하십시오. 또한 특정 메모리 및 링크 메트릭을 위해 Nsight Compute를 사용하십시오.

이러한 고급 커널 중 일부를 활성화하려면 사용자 정의 라이브러리 설치나 특수 CUDA 커널 활성화가 필요할 수 있습니다(특히 최신 기술의 경우). 그러나 이러한 기술은 일반적으로 출시 직후 PyTorch 및 주요 추론 엔진에서 지원됩니다. 사용자 정의 리소스 설치가 필요하더라도, 이는 디코드 작업자 풀에서 지연 시간 감소와 GPU 수 절감으로 직접 연결되므로 노력할 가치가 있습니다.

---

# KV 캐시 활용도 및 관리 조정

분리(Disaggregation)를 위해서는 클러스터 전반에 걸쳐 KV 캐시 를 최우선 공유 리소스로 취급해야 합니다. KV 캐시가 더 오래 유지되고 노드 간 이동이 가능해짐에 따라, 고성능 추론 시스템은 KV 캐시의 저장 및 공유 방식을 개선했습니다.

특히, 분산 KV 캐시 풀과 요청 간 접두사 재사용은 강력한 기술이 되었습니다. 또한, 최신 GPU 및 HBM 세대의 메모리 대역폭 개선 사항을 주시하는 것이 중요합니다. KV 캐시 성능 향상의 맥락에서 이 각각을 논의해 보겠습니다.

## 분리형 KV 캐시 풀

분리된 KV 캐시 풀은 각 GPU가 현재 처리 중인 요청에 대한 KV만 저장하는 방식( ) 대신, 개별 GPU로부터 KV 저장을 분리합니다. 대신 클러스터 전체 GPU 메모리에 데이터를 분산 저장합니다.

이 풀은 Grace Blackwell 및 Vera Rubin 플랫폼의 통합 CPU/GPU 메모리를 포함한 CPU 메모리로도 오프로드할 수 있습니다. NVMe SSD와 같은 영구 저장소로도 오프로드 가능합니다.

분산형 KV 캐시 풀을 사용하면, 프리필이 prompt에 대한 KV 텐서를 계산할 때 또는 디코드가 KV 텐서를 확장할 때 KV 블록이 여러 컴퓨팅 노드에 분산된 방식으로 저장됩니다. 이는 분산형 KV 풀 연구를 바탕으로 수정된 그림 18-4에 표시되어 있습니다.

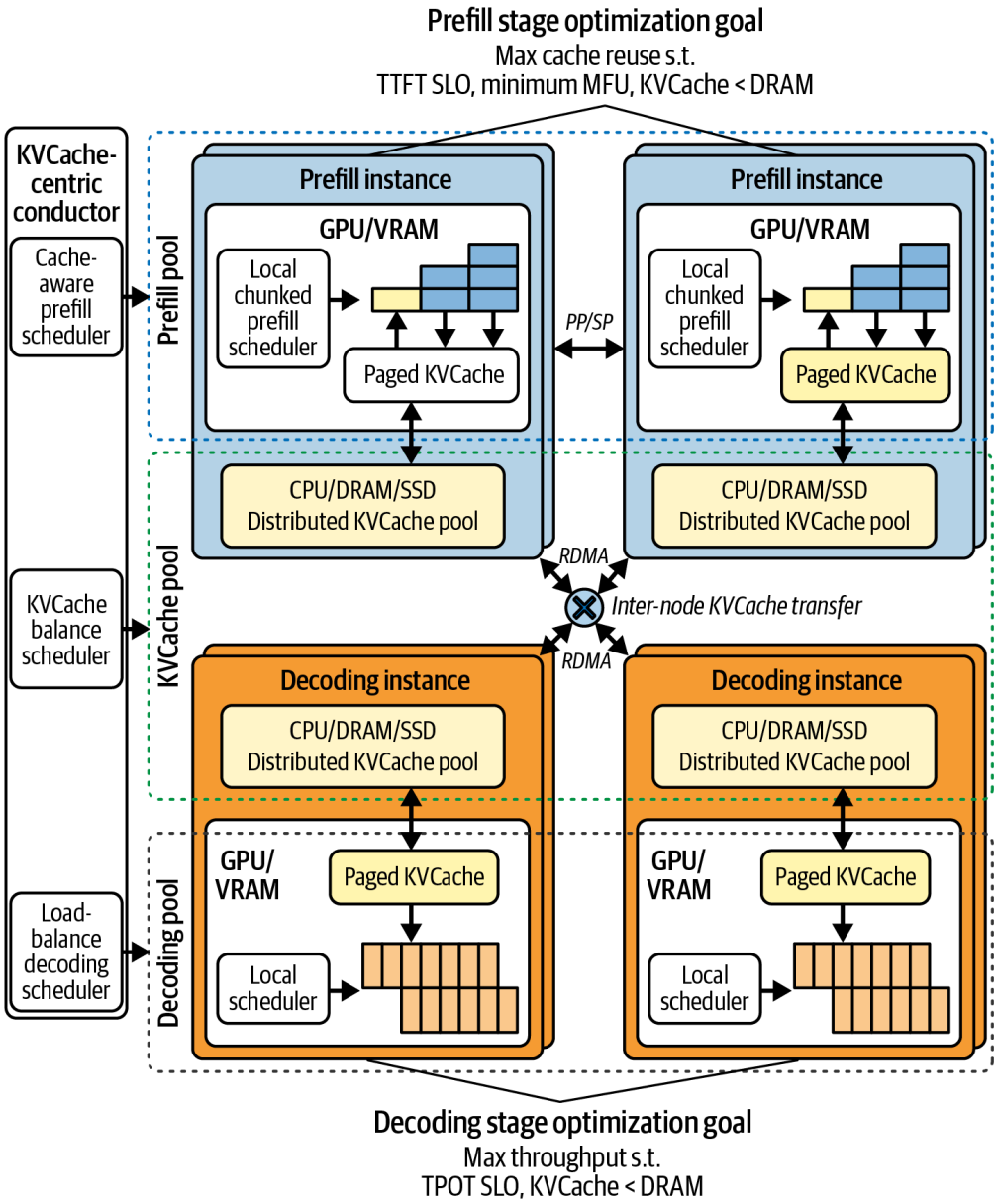


그림 18-4. 분산(분리 )된 KV 캐시 풀 (출처: https://oreil.ly/2xtK-)

매우 긴 250,000토큰 컨텍스트(예: 다수의 대화 턴이 포함된 채팅 세션)를 고려해 보자. 80개의 레이어와 32개의 헤드를 가진 700억 매개변수 트랜스포머 모델을 사용하며, 각 헤드의 차원은 128이다. 이는 토큰당 엄청난 KV 캐시 공간을 생성한다.

각 토큰은 모델의 숨겨진 차원(`num_heads` × `head_dim`)과 동일한 길이의 키와 값 벡터를 생성합니다. 본 모델의 경우 이를 4,096으로 가정합니다. 이로 인해 레이어당 8,192개의 부동 소수점 데이터가 발생합니다. 80개 레이어를 합산하면 토큰당 655,360개의 부동 소수점 KV 데이터가 생성됩니다. 16비트 정밀도(부동 소수점당 2바이트)를 가정하면 토큰당 약 1.31MB가 필요합니다. 이를 250,000 토큰으로 확장하면 약 328GB가 소요됩니다—KV 데이터만으로도!

---

이 계산은 대규모 모델의 토큰별 KV 크기를 기준으로 합니다. 이는 vLLM과 같은 엔진에서 FP8 KV 캐시가 널리 채택되어 메모리 사용량을 줄이고 배치 기회를 늘리는 이유를 보여줍니다.

---

KV 캐시를 FP8으로 양자화하고 선택적 레이어 캐싱 같은 기법을 사용하면, 이 250,000 토큰 prompt의 메모리 사용량을 100~150GB 수준으로 줄일 수 있습니다. 단일 GPU는 모델 가중치 등 보유해야 할 다른 모든 데이터와 함께 모든 토큰의 KV를 수용할 용량을 갖추지 못할 가능성이 높습니다. 특히 다중 회화(multi-turn conversation)가 지속될 경우 더욱 그렇습니다. 따라서 시스템은 컨텍스트를 잘라내거나(truncate) 이전 토큰에 대한 비용이 많이 드는 KV 재계산을 유발해야 할 것입니다.

그러나 분산형 KV 풀을 사용하면 컨텍스트의 오래된 KV 부분을 GPU에서 제거하여 클러스터 전반에 분산된 KV 캐시 풀(CPU DRAM 또는 NVMe 스토리지)로 이동시킬 수 있습니다. 이후 필요할 때 해당 데이터를 GPU 메모리로 다시 불러옵니다.

분리된 KV 캐시 풀은 다단계 메모리 계층 구조를 구현합니다. GPU 장치 메모리는 활성 KV 캐시를 보유하는 반면, CPU 호스트 RAM(또는 NVMe 스토리지)은 오버플로 백업 저장소 역할을 합니다. 현대적인 추론 엔진은 KV 캐시를 CPU 메모리나 NVMe로 오프로드할 수 있습니다. 이는 OS의 가상 메모리 하위 시스템과 유사하게 GPU 메모리를 효과적으로 가상화합니다.

이 설계는 GPU 메모리와 KV 캐시 풀 간에 KV 블록을 비동기적으로 페이징함으로써 컴퓨팅 파이프라인을 중단시키지 않고 초장기 컨텍스트를 가능하게 합니다. 이는 이 책 전반에 걸쳐 논의된 바와 같이 통신과 계산의 중첩이 양호하다는 가정 하에 성립합니다.

또한 요청 상태를 개별 GPU로부터 분리함으로써, 시스템은 글로벌 KV 캐시 풀을 활용해 클러스터 내 다중 노드에 걸쳐 KV 데이터를 동적으로 분할하여 부하를 적응적으로 분산시킬 수 있습니다. 이는 대규모 추론 클러스터에서 확장성을 단순화하고 결함 격리 성능을 향상시킵니다.

또한 모든 디코드 노드가 글로벌 KV 풀에 접근할 수 있으므로, 장애 조치나 부하 분산이 필요한 경우 어떤 디코드 노드든 모든 요청의 디코딩에 참여할 수 있습니다. 이는 스케줄러가 관련 KV 캐시 블록이 위치한 곳과 가장 가까운 디코드 노드를 선택할 수 있도록 유연성을 더합니다.

서버 A에 특정 접두사 KV 블록이 DRAM에 캐시된 경우, 해당 서버 A에서 디코딩을 스케줄링하는 것이 더 빠를 수 있습니다. 해당 서버가 해당 블록을 GPU로 신속하게 불러올 수 있기 때문입니다. 반면 서버 B는 네트워크를 통해 KV 블록을 가져와야 합니다.

---

이는 계산 대상 데이터에 가장 가까운 컴퓨팅 노드를 선택하는 고전적인 분산 시스템 모범 사례를 설명합니다. 이를 통해 시스템은 비용이 많이 드는 데이터 이동을 최소화합니다.

---

효율적인 KV 캐시 스케줄러는 네트워크 토폴로지 외에도 풀 내 KV 블록의 분포를 고려하여 프리필 및 디코딩 작업을 할당할 수 있습니다. 따라서 프리필 노드는

클러스터 전체에서 접근 가능한 분산 메모리 공간으로 구현된 풀에 KV 데이터를 배치할 수 있습니다.

클러스터 측 공유 메모리 공간에 KV 캐시가 배치되면 모든 디코딩 노드가 데이터를 검색할 수 있습니다. 이는 매번 프리필에서 디코딩으로의 직접 전송을 스케줄링할 필요가 없도록 합니다.

풀에서 KV 데이터를 검색하기 위한 추가 경로로 인해 약간의 오버헤드가 발생하지만, 모든 디코드 노드가 모든 KV 캐시 데이터에 접근할 수 있으므로 유연성이 높아집니다. 또한 특정 프리필 노드로부터 직접 데이터를 받지 않은 디코드 노드도 필요 시 풀에서 KV 데이터에 접근할 수 있음을 의미합니다.

디코드 노드가 중단되거나, 어떤 이유로든 생성 중간에 요청을 이동해야 하는 경우에도 KV 데이터는 손실되지 않습니다. 데이터는 풀에 상주하며, 다른 노드가 이를 가져와 저장된 KV를 사용하여 중단된 지점부터 계속 처리할 수 있습니다. 이는 결함 내성을 향상시킵니다.

글로벌 KV 캐시 풀은 요청 간 캐시 지속성도 제공합니다. 따라서 두 요청이 일부 접두사를 공유할 경우, 해당 접두사에 대한 KV는 한 번만 계산되어 클러스터 전체에서 재사용될 수 있습니다. 요청이 서로 다른 디코드 서버로 분배되더라도 마찬가지입니다.

요약하면, 분산형 KV 캐시 풀은 메모리(또는 저온 저장소)를 컴퓨팅 성능과 교환합니다. 더 큰 KV 캐시를 저장함으로써 시스템은 다양한 시나리오에서 KV 데이터 재계산을 피할 수 있습니다. 이 접근 방식은 DRAM이나 SSD에서 데이터를 재사용하는 것이, 이차 시간 복잡도($O(N^2)$)를 가진 대규모 어텐션 행렬 곱셈을 반복적으로 재계산하는 것보다 종종 비용 효율적이라는 점을 활용합니다.

## KV 캐시 재사용과 접두사 공유

앞서 언급했듯이, 공통 접두사를 공유하는 prompt에 대해 요청 간에 캐시된 KV 데이터를 재사용하는 것은 이 유리합니다. 이러한 시나리오는 다중 회화, 공유 시스템 prompt, 첨부 문서 형태로 상당히 자주 발생합니다.

모든 요청마다 해당 접두사에 대한 트랜스포머 어텐션 출력을 재계산하는 대신, 시스템은 접두사에 대한 KV 출력을 저장하고 직접 재사용할 수 있습니다. 본질적으로 이는 입력의 해당 부분에 대한 사전 채우기 계산을 생략하여 상당한 시간과 GPU 사이클을 절약합니다.

적절한 KV 캐시 중심 스케줄러는 작업을 할당할 때 '접두사 캐시 히트 길이', 즉 이 prompt의 토큰 중 캐시 풀에 이미 존재하는 토큰 수를 살펴 접두사 캐시 히트를 고려합니다. 실제로 새로운 요청이 들어오고 그 첫 *N개* 토큰이 KV 풀의 캐시된 접두사와 일치하면 시스템은 해당 KV 데이터를 재사용하기로 결정할 수 있습니다.

vLLM은 PagedAttention 메커니즘을 통해 KV "페이지"의 글로벌 해시 테이블을 사용해 자동 접두사 캐싱 을 구현합니다. 여기서 고유한 16토큰 컨텍스트 블록마다 해시가 할당됩니다. 새 요청이 저장된 블록(해시 기준)과 일치하는 접두사를 필요로 하면, 재계산 대신 해당 KV 텐서를 직접 복사할 수 있습니다.

동일한 컨텍스트가 다시 나타나면 시스템은 메모리에서 이를 제공합니다. 본질적으로 컨텍스트의 KV를 해싱을 통해 콘텐츠로 조회 가능한 재사용 가능한 데이터로 취급합니다. 구현체는 일반적으로 이러한 캐시된 컨텍스트를 관리하고 필요 시 제거하기 위해 전역 "prompt 트리"를 유지합니다. 이는 가장 빈번하게 재사용되는 접두사를 최적화합니다.

효과적인 KV 재사용의 핵심은 동일하거나 중복되는 접두사를 식별하는 것입니다. 일반적으로 시스템은 단순성을 위해 정확한 일치에 집중합니다. 즉, 처음 *N개의* 토큰이 정확히 일치하면 해당 청크를 재사용합니다. 부분 접두사 중복을 결합하는 것은 캐시를 어떻게든 병합해야 하므로 더 복잡하며, 이는 항상 직관적이지 않습니다. 따라서 일반적인 캐싱은 정확한 접두사 캐싱을 사용합니다.

그러나 여기에는 장단점이 있습니다. 다수의 사용자 KV 캐시를 무기한 저장하면 많은 메모리를 소모할 수 있습니다. 시스템은 KV 블록에 대해 LRU(최소 최근 사용)와 같은 제거 정책을 구현하여 재사용 가능성이 낮은 캐시를 삭제해야 합니다. 이렇게 하면 새로운 캐시를 위한 공간이 확보됩니다. 스케줄러는 재사용 가능성에 따라 유지할 캐시를 결정할 수도 있습니다. 핵심은 메모리 제약 내에서 캐시 적중률을 극대화하는 것입니다.

특정 프리필 노드가 이미 로컬 GPU 메모리나 로컬 DRAM 캐시에 필요한 KV의 일부를 보유하고 있다면, 데이터 전송을 최소화하기 위해 해당 노드로 요청을 라우팅하는 것이 유리할 수 있습니다. 이는 항상 컴퓨팅이 가능한 곳으로 데이터를 끌어오는 대신, 데이터가 있는 곳으로 컴퓨팅을 보내는 데이터 인식 스케줄링의 예시입니다.

이는 분산 시스템의 지역성 인식 스케줄링과 유사합니다. 앞서 라우팅 논의에서 이 점을 언급한 바 있습니다. 가능하다면 요청을 해당 접두사를 생성한 서버로 라우팅해야 합니다. 이는 캐시 적중 가능성을 극대화합니다.

분산 처리라는 더 넓은 맥락에서, 프리픽스 캐싱은 다수의 요청에 걸쳐 키-값(KV)에 대한 통합된 뷰를 유지하고 이를 글로벌 풀과 같은 공유 가능한 장소에 저장함으로써 지원됩니다. 이는 요청별 또는 노드별 고립된 접근 방식과 대조됩니다.

이는 동일한 prompt가 서로 다른 시간에 다른 노드로 전송될 경우 분산화로 인해 발생할 수 있는 재계산 오버헤드를 줄이는 데도 도움이 됩니다. 글로벌 KV 저장소나 조정된 캐싱을 사용하면, 사용자의 요청이 서로 다른 디코드 서버에 도달하더라도 서로의 캐시된 작업 결과를 활용할 수 있습니다.

## 최적화된 KV 캐시 메모리 레이아웃

저수준 혁신 의 또 다른 영역은 KV 캐시 메모리 레이아웃 최적화입니다. 각 시퀀스의 모든 과거 토큰에 대한 키와 값을 저장하는 KV 캐시는 다중 동시 디코딩 스트림에서 매우 커질 수 있습니다. 각 스트림이 사용하는 메모리는 대략 `num_layers` × 2 × `sequence_length` × `d_head` 에 비례하기 때문입니다.

계층형 캐싱과 같은 기법이 유용한 이유는 모든 키-값 쌍을 항상 GPU 메모리에 보관할 필요가 없기 때문입니다. 오래된 KV 캐시 부분은 CPU로 스왑하거나 압축할 수도 있습니다.

디코드 지연 시간을 낮게 유지하는 것이 중요하므로, 대부분의 설계에서는 빠른 접근을 위해 활성 KV 캐시를 GPU 메모리에 보관합니다. 이 경우 메모리 레이아웃과 접근 방식을 조정할 수 있습니다.

DeepSeek의 FlashMLA는 페이지를 KV 캐시로 사용하며, 활성 시퀀스에 대해 연속적인 메모리 접근이 가능하도록 고정 크기 블록(페이지)으로 캐시를 할당합니다. 이는 캐시 미스와 DRAM 트래픽을 줄여줍니다.

또한 일부 시스템은 컨텍스트 윈도우 이동 등으로 prompt 접두사가 더 이상 고려되지 않을 경우 접두사 압축을 구현합니다. 이때 KV 캐시 관리자는 해당 KV 항목을 삭제하거나 압축할 수 있습니다. 이는 컨텍스트 윈도우가 슬라이드하는 긴 대화에서 더 관련성이 높지만, 극도로 긴 시퀀스의 경우 메모리와 대역폭을 절약할 수 있습니다.

---

이 제거/압축 기법은 모델이 슬라이딩 윈도우 또는 기타 제한된 어텐션 패턴을 사용할 때 안전합니다. 그러나 전체 콘텐츠 윈도우(또는 검색 후크)에 걸쳐 완전한 어텐션을 유지하는 레이어에는 신중한 평가 없이 적용해서는 안 됩니다.

---

*POD-Attention이라 불리는* 또 다른 기법은 HBM 트래픽을 줄이기 위해 주의력 계산을 유사하게 재구성합니다. 구체적으로 SM 인식 스레드 블록(또는 협력적 스레드 배열[CTA]) 스케줄링을 사용합니다. 이는 *런타임 작업 바인딩*을 구현하여 각 SM에서 실행되는 CTA에 프리필 또는 디코드 작업 중 하나를 동적으로 할당합니다. 이는 그림 18-5에 표시되어 있습니다.

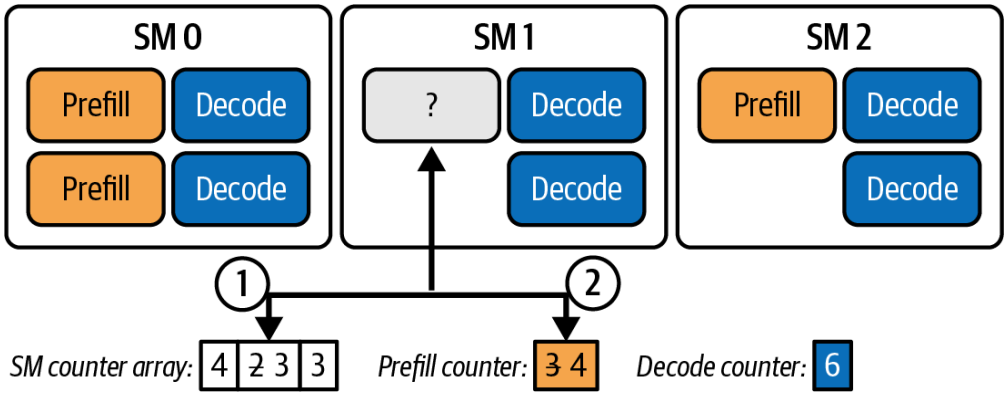


그림 18-5. SM 인식 스레드 블록(CTA) 스케줄링으로 프리필 작업과 SM의 디코드 작업을 매칭하여 메모리 이동 최소화

따라서 각 단계에 대해 별도의 커널을 정적으로 실행하는 대신, 단일 커널이 두 워크로드를 모두 처리할 수 있는 충분한 수의 CTA를 실행합니다. 런타임에 각 CTA는 자신이 속한 SM을 확인하고 SM별 카운터를 사용하여 해당 SM에서 실행 중인 다른 작업에 따라 프리필 또는 디코딩 중 어떤 작업을 실행할지 결정합니다.

SM 인식 스케줄링 로직은 실행 시 프리필과 디코딩 작업을 매칭하려고 시도합니다. 이는 고립된 메모리 트래픽 급증을 방지하고 리소스 수요를 균일하게 분배합니다.

구체적으로, POD-Attention은 프리필과 디코딩 작업을 동일한 SM에 배치하여 융합 커널이 로컬리티를 개선하고 중복 HBM 트랜잭션을 줄일 수 있도록 합니다. 이는 메모리 이동을 최소화하고 대역폭 활용도를 극대화하며, 각 SM에서 컴퓨팅 바운디드 및 메모리 바운디드 워크로드의 균형을 맞춥니다. POD-Attention은 적절한 SM 인식 CTA 스케줄링을 통해 동일한 SM에 프리필 및 디코드 작업을 배치함으로써 완전한 중첩을 실현하여 어텐션 성능을 최대 약 29%까지 향상시킬 수 있습니다.

POD-Attention의 동적 바인딩은 하드웨어의 CTA-SM 할당과 소프트웨어의 CTA 역할 할당(프리필 또는 디코드)을 분리합니다. 이러한 혁신은 메모리 이동을 최소화하고 시스템 성능을 극대화하기 위한 하드웨어와 소프트웨어의 공동 설계에 대한 관심이 증가하고 있음을 보여줍니다.

## GPU 및 CPU-GPU 슈퍼칩 개선 사항

새로운 하드웨어의 메모리 대역폭 향상도 고려해야 합니다. 더 높은 메모리 대역폭과 더 큰 L2 캐시는 메모리 제약 디코드 단계의 성능에 직접적인 이점을 제공합니다.

NVIDIA의 Grace Blackwell GB200 NVL72 시스템 은 36개의 Grace CPU와 72개의 Blackwell GPU를 탑재한 랙 스케일 플랫폼으로, KV 캐시를 위한 수십 테라바이트의 메모리를 갖춘 단일 논리 디코드 유닛을 지원합니다. 약 30TB의 통합 메모리를 보유한 이 하드웨어는 매우 큰 컨텍스트를 메모리에 유지하는 데 이상적입니다. 이러한 컨텍스트는 수백만 토큰 규모에 이를 수 있습니다.

이러한 플랫폼에서는 통합 메모리 사용량이 큽니다. 그러나 지연 시간이 중요한 디코딩의 경우, 활성 키와 값은 여전히 GPU HBM에 상주해야 합니다. 따라서 Grace CPU 메모리(HBM이 아닌 LPDDR5X)를 하위 계층 캐시나 매우 오래된 토큰용으로 사용해야 합니다. 컨텍스트가 사용 가능한 HBM을 초과할 때는 프리필 및 키-값 오프로딩이 여전히 중요합니다—NVL72와 같은 시스템에서도 마찬가지입니다.

요약하면, 최대 추론 성능을 완전히 달성하려면 거시적 수준의 분리와 미시적 수준의 최적화를 병행해야 합니다. FlashMLA/ThunderMLA와 같은 고급 디코드 커널, 효율적인 메모리 레이아웃(페이지 캐시 등), 최신 GPU 아키텍처는 효율적이고 확장 가능한 디코드를 구현할 것입니다.

# 프리필과 디코드 간 빠른 KV 캐시 전송

분산 추론 의 핵심 요구사항은 프리필 작업자에서 디코드 작업자로 KV 캐시를 빠르고 효율적으로 전송하는 것입니다. 이 전송이 느리다면 프리필과 디코드를 병렬화하여 절약한 시간이 데이터 이동 대기 시간으로 상쇄될 수 있습니다.

이 섹션에서는 전송 오버헤드를 최소화하기 위해 사용되는 기술을 논의합니다. 그런 다음 고속 상호 연결을 사용하고 추가 KV 복사를 피함으로써 시스템이 핸드오프를 구현하는 방법을 설명합니다.

## KV 캐시 크기

프리필 출력은 주로 모든 프롬프트 토큰에 대한 KV 캐시로 구성됩니다. 이는 상당한 양의 데이터가 될 수 있습니다. *L개의* 레이어를 가지며 각 레이어마다 *차원 d의 h개* 어텐션 헤드를 갖고, *N개* 토큰의 prompt를 가진 모델을 고려해 보십시오. KV 캐시 크기는 대략 2 × L × N × (h × d)이며, 여기서 2라는 계수는 키와 값 모두에 해당합니다.

실제 크기는 정밀도(FP16 대 INT8 등)와 모델 특성에 따라 달라지지만, 매우 큽니다. 예를 들어, 40층 모델에 64 크기의 16개 헤드와 1,000토큰 prompt를 사용하면 약 40,000개의 KV 벡터가 생성됩니다. 이는 수백 MB에 달하는 데이터일 수 있습니다. 토큰 수가 5,000개라면 5배 더 커집니다.

이 정도의 데이터를 네트워크로 전송하는 것은 단순하게 처리할 경우 상당한 지연 시간을 유발할 수 있습니다. 예를 들어, 단순한 접근 방식은 프리필 작업자에서 KV를 CPU 메모리로 복사한 후 TCP로 전송하거나, 심지어 디코딩 프로세스가 로드하도록 디스크에 쓰는 것일 수 있습니다. 이는 매우 느릴 수 있으며, 큰 prompt의 경우 수백 밀리초에 달할 수 있습니다. 목표는 전송 시간을 단 몇 밀리초로 줄이는 것입니다. 이를 통해 프리필과 디코딩이 진정한 병렬 처리를 통해 중첩될 수 있습니다.

저지연 KV 데이터 전송 시간을 달성하려면 일반적으로 작은 PagedAttention 블록을 더 큰 버퍼로 통합하고 CPU 소켓 대신 GPUDirect RDMA 기반 경로로 이동해야 합니다.

---

## 제로 카피 GPU-to-GPU 전송

현대적인 분산 시스템은 고성능 컴퓨팅( ) 기반 제로 카피 GPU-to-GPU 전송 기술을 사용합니다. 실제로 이는 고속 패브릭을 통한 원격 직접 메모리 액세스(RDMA)를 활용하는 것을 의미합니다. 예를 들어, 랙/노드 간 전송에는 인피니밴드(InfiniBand)를 사용할 수 있으며, 단일 노드(다중 GPU) 플랫폼 내에서는 NVLink/NVSwitch를 통해 GPU 메모리에 직접 쓰기를 수행할 수 있습니다. 이러한 방법은 CPU 메모리를 경유하지 않고 GPU 간에 데이터를 직접 전송합니다.

추론을 위한 NVIDIA의 고성능 GPU-to-GPU 전송 라이브러리는 *NVIDIA Inference Xfer Library* (NIXL)라고 합니다. NIXL은 제로 카피 GPU↔GPU 및 GPU↔스토리지 데이터 이동을 위한 플러그인 아키텍처(예: NVLink, UCX 패브릭, GPUDirect Storage)를 제공합니다.

NIXL은 RDMA 방식 전송을 간소화하여, 사용 가능한 고속 패브릭(예: InfiniBand 또는 NVLink 기반 연결)을 통해 한 GPU가 다른 GPU의 메모리에 직접 쓰기를 허용합니다. 즉, 프리필 GPU가 디코드 작업 GPU의 메모리에 KV 텐서를 직접 주입할 수 있습니다.

RDMA 기반 프로토콜은 CPU를 우회하고 GPU 상호 연결 대역폭을 최대한 활용합니다. NVIDIA Dynamo 및 오픈 소스 vLLM과 LMCache 통합과 같은 시스템은 NIXL에 의존합니다. 구체적으로, 이들은 NIXL을 사용하여 NVLink 또는 RDMA를 통해 원격 GPU 메모리에 KV 텐서를 직접 기록합니다. 현대 GPU 상호 연결은 매우 높은 대역폭을 제공하며, 링크 유형과 경합에 따라 1GB 전송이 수 밀리초에서 수십 밀리초 내에 완료될 수 있습니다.

실제 구현에서는 데이터 전송과 계산을 중첩시켜 낮은 전송 시간을 달성합니다. 예를 들어 RDMA를 사용하면 디코딩 GPU가 다른 시퀀스에 대한 토큰 생성을 계속하는 동안 프리필 작업자가 비동기적으로 KV 데이터를 메모리 버퍼에 기록할 수 있습니다. 프리필이 데이터를 푸시하거나(RDMA 쓰기 푸시 모델), 디코딩이 데이터를 풀링할 수 있으며(RDMA 읽기), 이는 설계에 따라 달라집니다. 어느 쪽이든 데이터 경로에 CPU 개입이 필요하지 않습니다.

빠른 KV 전송을 위한 일반적인 전략으로는 프리필 측 푸시, 디코딩 측 풀, 공유 메모리(CUDA IPC) 버퍼, 커넥터/큐 추상화, 비차단 중첩 등이 있습니다. 각각에 대해 살펴보겠습니다:

*프리필 측 푸시*

프리필 작업자는 prompt 처리를 완료하면 디코드 작업자의 GPU에 예약된 버퍼로 KV 데이터를 RDMA 쓰기를 직접 시작합니다. 이는 비차단 방

식으로 수행될 수 있습니다. 즉, 프리필 작업자는 전송을 시작한 후 백그라운드에서 DMA가 진행되는 동안 다른 작업으로 넘어갈 수 있습니다.

*디코딩 측 풀*

또는 디코딩 작업자가 디코딩을 시작할 준비가 되면 프리필 GPU 메모리에서 직접 RDMA 읽기를 수행할 수도 있습니다. 푸시 방식이든 풀 방식이든 최종 결과는 동일합니다(CPU 복사 작업 없음). 일부 구현에서는 조정 작업을 송신자에게 넘기기 위해 푸시 방식을 선호할 수 있으며, 다른 구현에서는 수신자가 타이밍을 제어할 수 있도록 풀 방식을 선호할 수 있습니다.

*공유 메모리(IPC) 버퍼*

프리필과 디코딩이 동일한 머신(서버 내 서로 다른 GPU)에서 수행되는 경우, CUDA 프로세스 간 통신을 사용하여 메모리 핸들 또는 PCIe 바를 공유할 수 있습니다. 이는 동일한 호스트에서 NVLink 또는 NVSwitch를 활용한 효과적인 복사 방식입니다. 네트워크를 거치지 않는 로컬 제로-카피 전송의 변형입니다.

*커넥터/큐 추상화*

vLLM 구현은 전송 메커니즘을 논리적 인터페이스(파이프 또는 룩업 버퍼) 뒤에 추상화합니다. 프리필 프로세스는 이 버퍼에 KV를 배치하거나 사용 가능 신호를 보내고, 디코딩 측이 이를 가져옵니다. 내부적으로는 RDMA 또는 고성능 퍼블리시-서브스크라이브 메시지(Dynamo의 경우 제어 신호용 NATS)를 사용할 수 있습니다. 핵심은 논리적 인계를 전송 방식과 분리하여 RDMA, 공유 메모리 등 다양한 전송 방식을 유연하게 적용할 수 있도록 하는 것입니다 .

*비차단 오버랩*

앞서 언급했듯이 최적화된 시스템은 KV 전송과 진행 중인 디코딩 계산을 중첩합니다. 예를 들어, Dynamo의 디코딩 작업자는 새 요청에 대한 KV 데이터를 GPU 메모리에 쓰는 프리필 작업자가 있는 동안 다른 요청에 대한 토큰 생성을 계속합니다. 이는 전송 지연 시간의 상당 부분을 숨깁니다. 따라서 약 5ms의 KV 전송을 디코딩 계산과 중첩하여 요청의 첫 번째 생성 토큰에 실질적으로 순수한 지연 시간 증가 없이 처리할 수 있습니다.

이러한 방법을 사용하면 KV 전송은 수 밀리초 단위로 수행될 수 있습니다. 이는 프리필 작업자에서 해당 KV를 실제로 계산하는 데 필요한 수백 밀리초보다 훨씬 짧은 시간입니다. 따라서 프리필 → 전송 → 디코딩 파이프라인은 디코딩이 프리필 완료 직후 거의 즉시 시작될 수 있어 긴 대기 시간 없이 우수한 병렬성을 달성합니다.

KV 캐시 데이터를 전송할 때 조각화와 오버헤드를 피하도록 주의하십시오. 예를 들어, vLLM의 PagedAttention은 KV 캐시를 고정 크기 토큰 블록(일반적으로 블록당 16개 토큰)으로 저장합니다. KV 블록은 상대적으로 작습니다(단, 블록당

바이트 수는 헤드 수, 헤드 차원, 레이어 수, dtype에 따라 증가합니다). 수천 개의 작은 KV 페이지를 RDMA로 무분별하게 전송하면 각 전송에 고정된 지연 시간과 프로토콜 오버헤드가 발생하여 과도한 오버헤드가 발생합니다. 이는 대역폭 활용도 저하로 이어집니다.

---

현대적인 LLM 엔진은 블록당 8, 16, 32, 64 또는 128 토큰과 같은 여러 페이지 크기를 지원합니다. 더 큰 페이지 크기는 RDMA를 통해 KV를 이동할 때 전송 오버헤드를 줄일 수 있습니다. 이는 더 큰 콜레이트된 버퍼와 더 적은 작업 큐 요소(WQE)로 인해 지속적 링크 처리량이 향상되기 때문입니다. 가능한 경우 RDMA 쓰기당 ≥ 128 토큰 페이지를 콜레이트하십시오. 전송을 전용 CUDA 스트림에서 중첩하도록 하십시오. 비차단 스트림을 우선적으로 사용하고 이벤트 펜스를 활용하십시오. 중첩을 확인하기 위해 항상 Nsight Systems와 같은 도구로 자질을 평가하십시오. LMCache는 RDMA에서 콜레이션 후 7.5k 토큰 KV에 대해 ~20ms → ~8ms를 보고합니다.

---

LMCache 확장 기능은 전송 전에 KV 페이지를 큰 연속 버퍼로 통합함으로써 이러한 비효율성을 해결합니다. 기본적으로 GPU 메모리에서 작은 청크들을 하나의 큰 청크로 모은 다음, 해당 큰 버퍼를 단일 전송으로 보냅니다.

예를 들어, 7,500개 토큰의 KV 캐시를 470개의 작은 전송으로 보낼 때 20ms가 소요된다면, 이를 더 큰 블록(예: 128개 토큰 페이지)으로 모으면 전송 시간이 8ms로 단축됩니다. 이 간단한 배치 최적화는 네트워크 파이프를 가득 채우고 패킷당 오버헤드를 줄여줍니다.

빠른 GPU-to-GPU KV 전송을 위한 시스템 구성 방법을 살펴보겠습니다. 다음은 NIXL 전송 채널을 사용하는 LMCache의 프리필-디코드 모드 구성 예시입니다:

```
# Prefill server config (lmcache-prefiller-config.yaml)
enable_pd: true
transfer_channel: "nixl"
pd_role: "sender"               # this instance sends KV data
pd_proxy_host: "decode-host"    # PD proxy / decode coordinator
pd_proxy_port: 7500             # control-plane port on the proxy/decoder
# size the buffer to the KV you plan to transfer
# FP8/FP4 KV should shrink it significantly
pd_buffer_size: 1073741824      # 1 GiB transfer buffer size
pd_buffer_device: "cuda"        # buffer stays in GPU memory
```

여기서 프리필 서버는 RDMA 송신기로 설정됩니다. 디코드 호스트의 포트 7500을 대상으로 하며, KV 전송을 위해 할당된 1GB GPU 버퍼를 사용합니다. 디코드 서버는 해당 포트에서 수신기로 설정되며, 동일한 1GB GPU 버퍼를 사용합니다. 구성은 다음과 같습니다:

```
# Decode server config (lmcache-decoder-config.yaml)
enable_pd: true
```

```
transfer_channel: "nixl"
pd_role: "receiver"             # this instance receives KV
pd_peer_host: "0.0.0.0"         # bind address for NIXL peer
pd_peer_init_port: 7300         # NIXL handshake/control port
pd_peer_alloc_port: 7400        # NIXL allocation/data port
pd_buffer_size: 1073741824      # 1 GiB (match sender unless you plan to s
pd_buffer_device: "cuda"        # keep buffer in GPU memory
nixl_backends: [UCX]            # UCX backend is sufficient for disagg
```

이 구성은 프리필이 CPU 개입 없이 디코드 GPU 메모리에 직접 KV 캐시를 쓰도록 허용합니다(전송당 최대 1GB). 양측 모두 제로 카피 작업을 위해 전송 버퍼를 GPU 메모리에 유지합니다.

전송 버퍼 크기를 설정할 때는 pd_buffer_size = 1GB로 시작하십시오. 이는 약 700억 매개변수, 80개 레이어, 32개 헤드, 128차원 헤드를 가진 모델에서 ~4~8k 토큰으로 추정되는 FP16 KV 캐시 크기입니다. prompt가 ~7.5k 토큰을 초과할 경우 2GB를 사용하십시오. dtype과 헤드 수를 기준으로 확장할 수 있습니다: bytes ≈ 2 × L × N × (H × Dh) × bytes_per_val. 전송 전에 페이지를 반드시 정렬하십시오. 이는 소규모 IO 비효율을 방지합니다.

---

KV 캐시를 FP8 또는 FP4로 양자화하면 토큰당 바이트 수가 감소하므로 고정 토큰 수에 필요한 전송 버퍼도 줄어듭니다. 따라서 버퍼당 더 많은 토큰을 전송하거나 버퍼 크기를 그에 맞게 줄일 수 있습니다. 1~2GiB 버퍼는 대부분의 배포 환경에서 작동하지만, 위 KV 공식으로 계산한 후 256MB 단위로 올림 처리하십시오. FP8 또는 FP4 KV 사용 시 버퍼 크기를 비례적으로 축소할 수 있습니다. 전송할 최대 콜레이션 페이지 그룹을 기준으로 항상 검증하십시오. 최적의 링크 활용을 위해 콜레이션된 128개 이상 토큰 페이지에는 GPUDirect RDMA를 우선 적용하십시오.

---

실제 환경에서는 다음 셸 스크립트를 사용해 CLI로 디코드 서버를 실행할 수 있습니다. 이는 이기적 분할을 줄이고 더 큰 버퍼에서의 랜데뷰를 촉진합니다:

```
# Example decode worker
# (select device by index or UUID)

UCX_RNDV_THRESH=16384
UCX_MAX_EAGER_RAILS=1
UCX_TLS=cuda_ipc,rc,rdmacm,cuda_copy,cuda_ipc,tcp \
CUDA_VISIBLE_DEVICES=1 \
LMCACHE_CONFIG_FILE=lmcache-decoder-config.yaml \
python run_vllm_decoder.py --port 8200
```

다른 GPU에서 프리필 서버를 시작할 때도 유사하게 구성 파일을 사용합니다. 이 설정은 키-값 전송에 표준 TCP 소켓 대신 노드 간 InfiniBand RDMA 또는 노드 내 NVLink 피어투피어 통신을 사용하도록 보장합니다.

단일 노드, 다중 GPU 실행 시 CUDA IPC를 활성화해야 합니다. 노드 간 실행 시 RDMA를 선호하십시오. LMCache/vLLM 작업자를 위한 일반적인 UCX 구성은 다음과 같습니다: `UCX_TLS=rc,rdmacm,cuda_copy,cuda_ipc,tcp` 그리고 패브릭에 RoCE/IB 무손실 설정(ECN/PFC)이 적용되었는지 확인하십시오. 노드 간 RDMA의 경우, 대형 KV 버퍼는 랜데뷰를 사용하고 소형 KV 버퍼는 이거를 사용하도록 `UCX_RNDV_THRESH=16384` 를 고려하십시오. 항상 `ucx_info -f` 로 확인하십시오.

---

RDMA와 적절한 버퍼링이 구현되면, 핸드오프 지연 시간은 상호 연결 및 페이지 크기에 따라 한 자릿수에서 수십 밀리초 수준으로 단축될 수 있습니다. 예를 들어, 7,500 토큰 컨텍스트에서 LMCache는 다수의 소규모 전송 시 약 20밀리초, 대규모 블록으로 콜레이션한 후에는 약 8밀리초를 기록했습니다. 구체적으로 RDMA 전 16토큰 페이지를 ≥128토큰 슬래브로 콜레이션하는 것이 권장됩니다. 이는 패킷당 오버헤드 감소에 도움이 됩니다.

요약하면, 분산 시스템은 빠른 상호 연결과 스마트한 데이터 병합을 통해 프리필 → 디코딩 전환을 원활하고 빠르게 수행해야 합니다. 핸드오프 시간을 최소화하는 것이 중요한 이유는 핸드오프가 느리면 병렬화 단계의 이점을 상쇄하기 때문입니다.

---

`export PYTHONHASHSEED=0` 를 설정하여 다중 프로세스 실행 시 KV 청크 라우팅에 결정론적 해시를 사용하십시오.

---

# 커넥터 및 데이터 경로 설계

제로 카피 최적화를 기반으로, 프리필과 디코드 노드가 단순히 비트 이동을 넘어 전송을 종단 간에 어떻게 조정하는지 살펴보겠습니다. 프리필 및 디코드 작업자는 종종 스케줄러나 라우터를 사용하여 통신합니다. 실제로 이 스케줄러는 NVIDIA Dynamo에서 사용되는 중앙 집중식 구성 요소로 구현되거나, SGLang에서 사용되는 분산 조정 방식으로 구현되는 경우가 많습니다.

예를 들어 NVIDIA Dynamo는 디코드 작업자가 새로운 prompt 작업을 큐에 푸시하면 프리필 작업자가 이를 소비하는 글로벌 스케줄링 큐를 구현합니다. 이 설계에서 디코드 노드는 그림 18-6의"Put RemovePrefillRequest"(6단계)에 표시된 대로 prompt 처리를 위한 요청을 큐에 등록합니다.

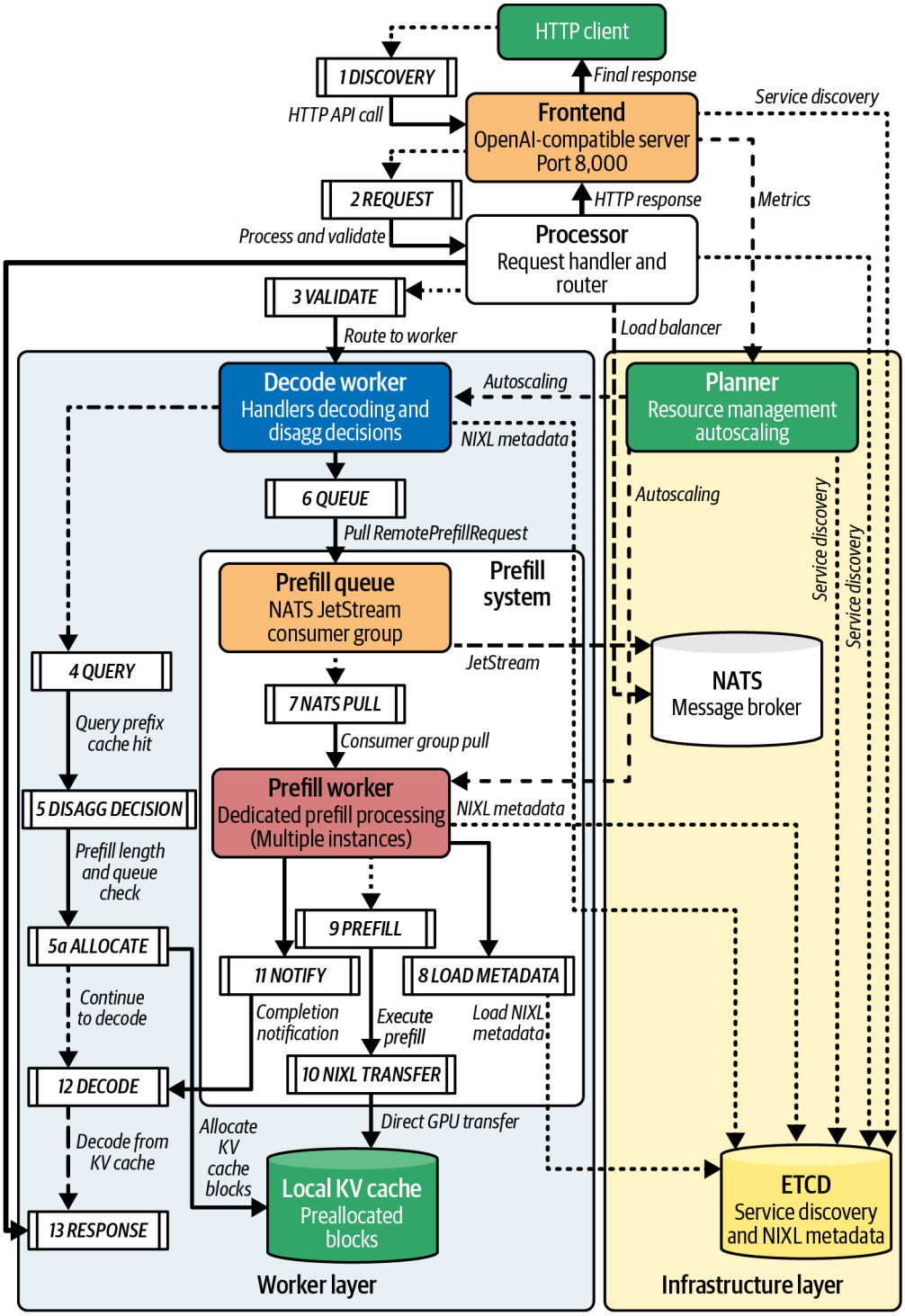


그림 18-6. NVIDIA Dynamo의요청 라이프사이클; 디코딩은 prompt를 가져오고 프리필은 NIXL을 사용하여 KV를 푸시함

프리필 노드는 이 요청을 수신하고, 완료 시 요청에 포함된 디코딩 노드의 원본 ID 또는 응답 대상 ID를 통해 결과를 보낼 정확한 디코딩 노드를 파악합니다. 이후 키-값 쌍은 NIXL RDMA를 통해 해당 디코딩 작업자의 GPU로 직접 전송됩니다.

vLLM + LMCache 구현에서는 보다 분산된 접근 방식을 사용합니다. 디코딩 및 프리필 프로세스는 각 요청의 키-값 쌍을 위해 파이프 또는 버퍼를 사용해 직접 채널을 설정합니다. 내부적으로는 요청 시작 시 협상된 일대일 TCP 또는 RDMA 연결을 사용할 수 있습니다. 글로벌 큐 대신 각 요청이 자체 전송 채널을 설정합니다. 두 접근 방식 모두 장단점이 있습니다. 글로벌 큐는 부하 분산과 장애 처리에 더 간단합니다. 직접 채널은 대기열 지연을 최소화할 수 있습니다.

어떤 패턴을 사용할지 결정할 때는 워크로드와 인프라 제약 조건을 고려하십시오. 견고한 다중 테넌트 부하 분산과 쉬운 장애 전환이 필요하며, 약간의 대기열 지연은 감수할 수 있다면 일반적으로 글로벌 큐 모델이 더 적합합니다. 반대로, 꼬리 지연 시간 요구 사항이 엄격하고 디코드-프리필 쌍이 비교적 안정적이며 고속 상호 연결을 보유한 경우, 요청별 직접 채널 접근 방식이 홉 수와 지터를 최소화할 수 있습니다.

---

실제 적용 시 예상되는 요청 혼합 하에서 두 설계 모두 벤치마킹하십시오. prompt 길이, 동시성 수준, 실패 시나리오를 다양하게 변경하여 SLO에 가장 적합한 지연 시간-처리량 균형을 제공하는 설계를 확인하십시오.

---

핵심 설계 목표는 파이프라인을 비차단 및 고처리량으로 만들어 한 요청이 디코딩되는 동안 다른 prompt가 프리필을 시작할 수 있도록 하는 것입니다. 동시에 다른 요청의 KV가 전송 중일 수 있습니다. 따라서 다른 단계에서 작업이 있을 경우 어떤 단계도 유휴 상태에 머물지 않습니다. 바로 이 점이 분산화가 대규모에서 전체 처리량을 향상시키는 정확한 이유입니다—모든 단계가 병렬로 바쁘게 유지되기 때문입니다.

---

종종 프리필에서 생성된 첫 번째 토큰의 로짓 값은 명시적으로 전송되지 않습니다. 디코드 작업자가 키-값 쌍(KV)에서 직접 첫 번째 토큰의 확률을 재계산할 수 있기 때문입니다. 일부 시스템은 디코드 작업자의 추가 연산 시간을 수백 마이크로초 단축하기 위해 첫 번째 토큰의 출력을 전송하기도 합니다. 그러나 다른 시스템은 단순성을 유지하며 KV만 전송하고 디코드 작업자가 해당 최종 레이어를 재계산하도록 합니다.

---

이 파이프라인이 장애에 강건하도록 하는 것이 중요합니다. 디코딩 노드가 생성 도중 실패하면 앞서 논의한 글로벌 KV 캐시 풀을 통해 다른 노드가 저장된 KV를 사용해 중단된 지점부터 작업을 이어갈 수 있습니다. 마찬가지로 프리필 노드가 prompt 도중 실패하면 해당 prompt는 다른 곳에서 재시도될 수 있습니다. 커넥터 설계는 이러한 장애를 우아하게 처리하여 한 노드의 실패가 전체 요청을 오류로 만들지 않도록 해야 합니다.

---

이러한 라우터들은 일반적으로 하트비트 확인과 타임아웃을 사용합니다. 프리필에서 디코딩으로의 전송이 중단될 경우 요청을 재할당하거나 안전하게 중단할 수 있도록 하기 위함입니다.

---

# 프리필 및 디코딩을 위한 이종 하드웨어와 병렬화 전략

분리형 아키텍처 의 강력한 장점 중 하나는 프리필 및 디코딩 클러스터 각각의 요구사항에 가장 적합한 서로 다른 하드웨어(심지어 서로 다른 모델 병렬 구성까지)를 자유롭게 선택할 수 있다는 점입니다. 통합된 단일 아키텍처 배포 환경에서는 일반적으로 두 단계 모두에 동일한 하드웨어 유형과 구성만 사용됩니다. 분리형 아키텍처를 사용하면 다음에 설명하는 바와 같이 단계별로 하드웨어와 전략을 혼합하여 적용할 수 있습니다.

## 연산 최적화 대 메모리 최적화 하드웨어

프리필 단계는 높은 컴퓨팅 처리량, 다량의 TFLOPS, 전용 텐서 코어, 높은 클럭 속도를 갖춘 GPU 의 이점을 누립니다. 상당한 메모리 대역폭도 도움이 되지만, prompt의 KV 캐시에 필요한 용량을 초과하는 막대한 HBM 용량은 반드시 필요하지 않습니다.

반면 디코드 단계는 다량의 토큰에 해당하는 KV를 처리하므로 대용량 메모리와 메모리 대역폭 모두에서 이점을 얻습니다. 극한의 컴퓨팅 성능은 필요하지 않지만, 높을수록 유리합니다.

이를 통해 각 단계에 서로 다른 세대의 GPU를 사용할 가능성이 열립니다. 예를 들어, 프리필 클러스터에는 최신 고성능 GPU를 사용하고 디코드 클러스터에는 충분한 메모리 대역폭을 갖춘 구형 또는 비용 효율적인 GPU를 고수하는 설계가 가능합니다.

이렇게 하면 최신 GPU(예: 최신 텐서 코어)의 잠재력을 완전히 활용하지 못하는 디코드 작업에 낭비하는 것을 피할 수 있습니다. 프리필 작업은 GPU 연산 유닛을 최대화하여 더 많은 전력을 소모하는 반면, 동일한 GPU에서 디코드 작업은 훨씬 적은 전력을 사용합니다.

### 처리량 및 비용 이점

이종 하드웨어 간에 단계를 분할하면 비용당 처리량과 와트당 처리량을 개선할 수 있습니다. Splitwise 연구에서 단계별 전용 하드웨어를 사용한 한 구성은 동종 기준 대비 20% 낮은 비용으로 1.4배 높은 처리량을 달성했습니다.

고정된 비용/전력 예산 하에서 최대 성능을 목표로 한 다른 구성에서는 동일한 비용과 전력으로 2.35배 더 높은 처리량을 달성했습니다. 구체적으로, 이 연구에서는 프리필에 4개의 H100(고성능 컴퓨팅)을, 디코딩에 4개의 A100(고메모리)을 사용했습니다. 이 혼합 구성은 동일한 비용/전력으로 8개의 GPU로 구성된 동질 시스템(모든 H100 또는 모든 A100)보다 약 2.35배 높은 RPS를 달성했습니다.

대안으로, 이질적 시스템은 디코딩 작업을 비용 효율적인 GPU로 오프로드함으로써 기준 처리량에 도달하기 위해 전체 GPU 수를 줄일 수 있음을 발견했습니다(예: 8개 대신 5~6개). 이는 비용 절감 기회를 강조하며 각 GPU 유형을 가장 효과적인 영역에 배치하는 가치를 보여줍니다. 구체적으로, 컴퓨팅 성능 대비 비용 효율이 가장 높은 GPU(예: Blackwell 또는 Rubin 세대)에서 컴퓨팅 바운디드 작업을 수행하고, 메모리 대역폭이 적절한 비용 효율적인 구형 GPU(예: Hopper 또는 Ampere)에 메모리 바운디드 작업을 할당할 수 있습니다.

Splitwise 평가에서는 이종 GPU 간 상태 전송 오버헤드를 고려했습니다. 본 테스트는 NVSwitch 패브릭을 통해 KV 데이터를 전송했으며, 서로 다른 세대의 GPU 간에도 최소한의 오버헤드만 발생했습니다. 이는 NVSwitch 및 NVLink와 같은 고대역폭 상호 연결이 혼합 GPU 환경에서도 성능에 미미한 영향으로 프리필/디코딩 분리를 가능하게 함을 시사합니다.

또 다른 시스템인 HexGen-2는 이기종 GPU에 분산된 추론을 할당하는 것을 최적화 문제로 다루는 분산 추론 프레임워크입니다. 이 스케줄러는 자원 할당, 단계별 병렬화 전략, 통신 효율성을 함께 최적화합니다.

Llama 2 70B와 같은 모델에 대한 실험에서 HexGen-2는 동일 가격대의 최첨단 시스템 대비 최대 2배(평균 약 1.3배)의 서비스 처리량 향상을 보였습니다. 또한 고성능 기준 시스템과 유사한 처리량을 달성하면서도 약 30% 적은 비용을 사용합니다. 이러한 개선은 GPU 유형 혼합과 작업 분할 최적화를 통해 이루어졌습니다. 이는 기본적으로 Splitwise가 개념적으로 수행한 작업을 자동화한 방식입니다.

이러한 결과는 분산 처리가 단순히 속도만을 위한 것이 아님을 입증합니다. 효율성과 적은 자원으로 더 많은 성과를 내는 것이 핵심입니다. 클라우드 환경에서 추론을 배포할 때, 이는 수백만 또는 수십억 명의 최종 사용자를 지원하는 대규모 추론 서비스의 GPU 시간 비용을 수백만 달러 규모로 절감할 수 있습니다.

예를 들어, 최상위 GPU 8대 대신 6대의 GPU(프리필 + 디코딩 혼합)로 동일한 트래픽을 처리할 수 있습니다. 해당 서비스의 하드웨어 비용을 약 25% 절감할 수 있는 셈입니다. 따라서 디스어그리게이션은 동일한 하드웨어로 더 많은 사용자에게 서비스를 제공할 수 있게 합니다. 특히 최신 GPU의 경우 공급이 제한적인 경우가 많기 때문에 이는 매우 중요합니다.

에너지 효율성도 미국을 비롯한 일부 지역의 전력 제약 고려 시 중요합니다. Splitwise는 디코딩 작업을 저전력 GPU에서 실행함으로써 속도가 약간 감소하더라도 더 나은 전력 효율성을 입증했습니다.

프리필(prefill) 작업과 디코딩 작업을 서로 다른 하드웨어 유형에 할당함으로써 각 단계를 어디서 어떻게 실행할지 선택하여 성능을 높이고 비용을 절감할 수 있습니다. 분산 처리 방식은 각 단계가 독립적이므로 이러한 유연성을 제공합니다.

요약하면, Splitwise, HexGen-2 및 관련 이종 배포 연구의 평가 결과 분리가 순수 속도 향상 외에도 비용 최적화에 활용될 수 있음을 보여줍니다. 하드웨어를 워크로드에 맞추면 쿼리당 비용을 크게 줄이면서 동시에 고정 예산 내에서 성능을 향상시킬 수 있습니다.

대규모 서비스의 경우 경제적 타당성을 유지하는 데 이는 매우 중요합니다. 단점으로는 여러 GPU 유형을 관리해야 하므로 시스템 복잡성이 다소 증가한다는 점이 있습니다. 또한 GPU 성능이 일치하지 않기 때문에 클러스터 구성 유연성과 프리필 및 디코드 작업 간 GPU의 동적 재할당에 제한이 따릅니다. 그러나 대부분의 경우 각 단계에 다른 하드웨어를 사용하는 것이 효율성 향상으로 인한 이점을 상쇄할 만큼 가치가 있을 수 있습니다.

### 단계별 모델 병렬화

이질성과 단계별 전문화의 또 다른 형태는 각 단계별로 GPU 간에 서로 다른 모델 병렬화 방식(예: 텐서 병렬, 파이프라인 병렬 등)을 선택하는 것입니다. 이는 메모리 제약으로 인해 GPU 간에 분할된 매우 큰 모델에 적용됩니다.

전통적인 설정에서는 프리필 및 디코드 단계 모두 텐서 병렬화나 파이프라인 병렬화를 사용하여 고정된 병렬화 전략으로 모델을 실행하고 여러 GPU에 모델을 분할할 수 있습니다. 그러나 프리필에 최적화된 병렬화 전략이 디코드에도 동일하게 적용되지는 않을 수 있습니다.

예를 들어, 프리필 단계는 *N개의* prompt 토큰을 통과하는 대규모 전방 통과(forward pass)로, 높은 수준의 병렬화에 유리합니다. 다수의 GPU에 걸쳐 텐서 병렬화(TP)를 사용하면 계산을 더 빠르게 수행하고 TTFT를 줄일 수 있습니다.

GPU 동기화 오버헤드는 한 번에 처리 가능한 대량의 토큰에 분산됩니다. 이는 TTFT에 중요한 이 단계의 실제 소요 시간을 단축시킵니다.

파이프라인 병렬화(PP)를 활용하면 프리필 속도를 더욱 높이고 처리량을 증가시킬 수도 있습니다. 이는 모델 레이어를 여러 GPU에 분산시키고 prompt를 다중 파이프라인 단계로 스트리밍하는 방식입니다.

반면 디코딩 단계는 순차적이며 단계별 지연 시간에 민감합니다. 디코딩에 GPU를 지나치게 많이 사용하면 오히려 출력 토큰당 소요 시간(TPOT) *지연* ( *토큰 간 지연* ITL)이 악화될 수 있습니다. 각 토큰 단계마다 추가적인 다중 GPU 통신 오버헤드가 필요하기 때문입니다. 따라서 한 번에 분할할 수 있는 계산량은 한 토큰 분량(추측 디코딩 사용 시 몇 토큰)에 불과하므로 가속화 잠재력이 제한됩니다.

분리(Disaggregation)를 통해 이러한 접근법을 혼합하여 한 단계에는 TP를, 다른 단계에는 PP를 사용하거나 각 기법의 적용 정도를 다르게 할 수 있습니다. 예를 들어, 프릴(prefill) 단계는 8개의 GPU에 걸쳐 실행되는 다중 GPU 병렬 처리( `TP=8` )로 실행하여 prompt 지연 시간을 최소화할 수 있습니다. 그런 다음 디코딩을 텐서 병렬 처리( `TP=1` ) 또는 단일 GPU로 실행하여 토큰당 처리량을 극대

화하고 단계 간 지연을 최소화할 수 있습니다. 이 방식으로 각 단계의 처리량과 지연을 별도로 조정할 수 있으며, 이는 그림 18-7에 표시되어 있습니다.

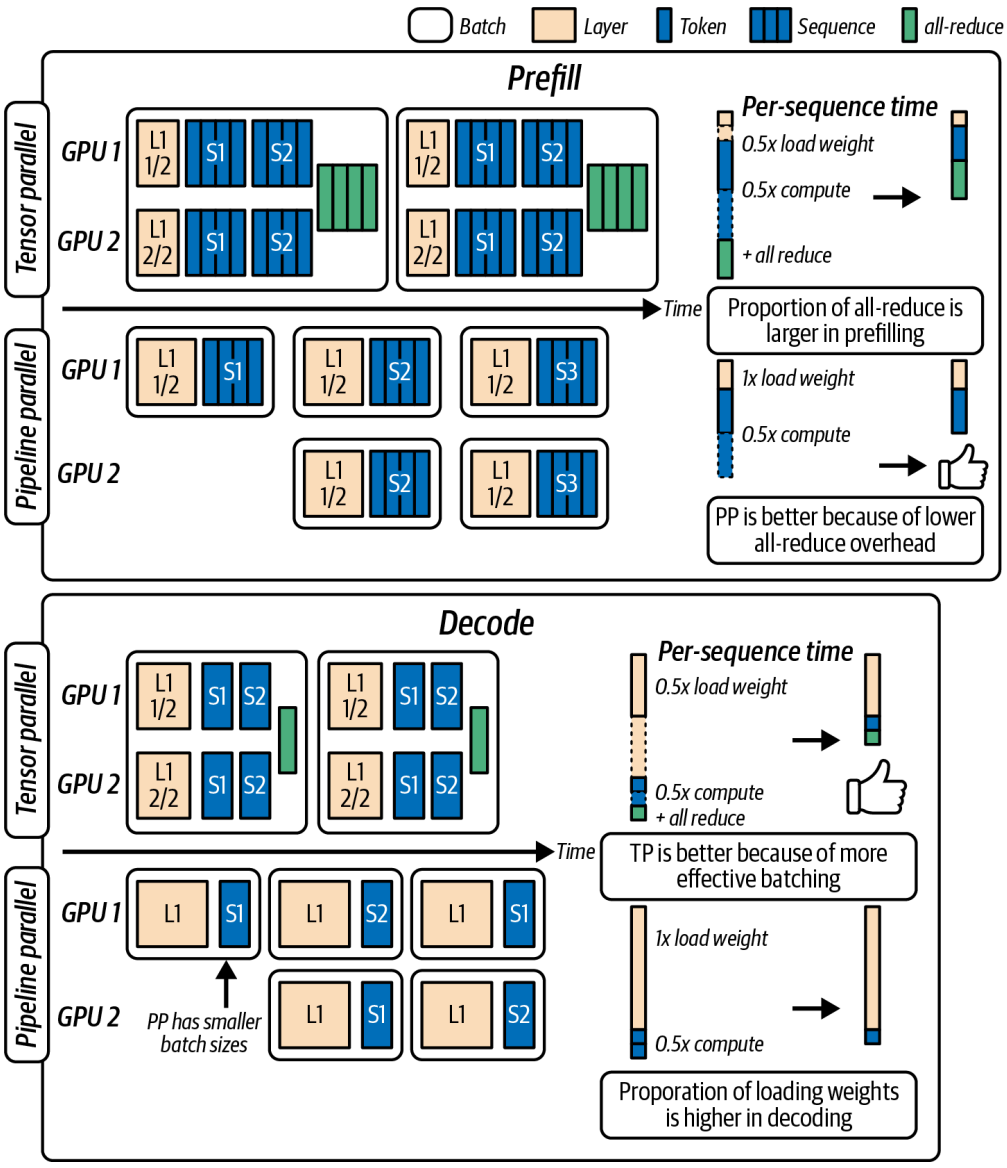


그림 18-7. 프리필 및 디코딩에 대한 다양한 병렬 처리 전략사용 (출처: https://oreil.ly/1-Ti0)

여기서 텐서 병렬화의 추가적인 올-리듀스 통신 오버헤드는 프리필 단계에서 더 두드러집니다. 많은 수의 토큰이 병렬로 처리되기 때문입니다. 따라서 프리필 작업 부하에는 파이프라인 병렬화가 더 효율적이므로 이를 선택합니다.

---

텐서 병렬화와 파이프라인 병렬화 모두 프리필에 효과적일 수 있습니다. 다음 예시는 프리필에 파이프라인 병렬화를 사용합니다. 그러나 특정 클러스터에서는 높은 텐서 병렬도가 TTFT를 줄일 수 있습니다. 최적의 선택은 네트워크 대역폭, 집합적 지연 시간, 모델 형태에 따라 달라집니다.

---

그러나 디코딩 단계에서는 파이프라인 병렬화가 GPU 간 토큰 전달 과정에서 더 많은(비록 규모는 작지만) 전방 전달을 유발할 수 있습니다. 이는 단일 토큰 생성을 위해 GPU 내부 및 외부로 많은 데이터 이동이 필요함을 의미합니다. 따라서 디코딩 작업 부하에 더 적합한 텐서 병렬화를 선택합니다.

그러나 이로 인해 복잡성이 발생합니다. 프리필과 디코딩 단계에서 모델의 병렬화 방식이 다르기 때문에 KV 캐시의 형식도 달라져야 합니다. 예를 들어 프리필 단계가 PP를 사용하므로 TP = 1이고 GPU가 4개라면, 각 GPU는 풀사이즈 KV 텐서를 보유합니다.

반면 디코딩 단계에서 TP = 4를 사용한다고 가정해 보겠습니다. 이 경우 데이터가 모델의 히든 크기에 따라 분할되어야 하므로 각 GPU는 KV 텐서의 1/4만 처리합니다. 이를 처리하기 위해 NVIDIA Dynamo와 같은 시스템은 전송 과정에서 KV 전치(또는 변환)를 수행할 수 있습니다. 본질적으로, 이는 `[TP_p parts]` 형식의 KV 캐시를 `[TP_d parts]` 형식으로 변환 및 재배열합니다. 여기서 `TP_p` 는 프리필의 병렬 처리도이고 `TP_d` 는 디코드의 병렬 처리도입니다.

Dynamo는 NIXL에서 읽은 후 디코드 작업자 메모리에 쓰기 전에 이 전치를 실시간으로 수행하는 고성능 커널을 포함합니다. 이렇게 하면 수신 디코드가 예상하는 레이아웃으로 KV 캐시 데이터를 받게 됩니다.

이 전치의 오버헤드는 네트워크 전송에 비해 작을 수 있습니다. 특히 NVLink 처리량 덕분에 이러한 데이터 재구성을 신속하게 처리할 수 있기 때문입니다. 이 경우 각 단계에 최적화된 서로 다른 병렬화 전략을 사용함으로써 계산 자원을 절약할 수 있으므로 쉽게 정당화됩니다.

단계별 병렬 처리의 예를 살펴보겠습니다. 텐서(TP), 파이프라인(PP), 데이터(DP), 시퀀스(SP, 시퀀스를 GPU 간 분할) 등 다양한 병렬 처리 방식을 적용할 수 있는 대규모 모델을 가정해 보겠습니다. 각 단계에 별도의 병렬 처리 구성을 선택할 수 있습니다. 표 18-1은 프리필 단계의 병렬 처리 구성 예시를 보여줍니다.

표 18-1. 프리필 병렬화 예시

| 병렬 처리 전략 | 심볼 | 값 | 설명 |
|---|---|---|---|
| 텐서 병렬 처리 | `TP_p` | 2 | 모델의 가중치 텐서를 2개의 GPU에 분할하여 관리 가능한 통신 오버헤드로 프리필 지연 시간을 절반으로 줄입니다. |
| 파이프라인 병렬 처리 | `PP_p` | 2 | 모델 레이어를 두 개의 파이프라인 단계로 분할하여, 심층 모델의 경우 각 단계를 통해 마이크로배치를 스트리밍합니다. |
| 시퀀스 병렬 처리 | `SP_p` | 1 | 매우 큰 컨텍스트를 처리하는 경우가 아니라면 입력 시퀀스를 GPU 간에 분할하지 않습니다(시퀀스 샤딩 없음). |
| 컨텍스트 병렬 처리 | `CP` | 1 | 최적화 후 메모리에 들어갈 수 있는 경우 전체 컨텍스트를 단일 GPU에 유지하십시오(컨텍스트 수준 분할 금지). |
| 데이터 병렬 처리 | `DP_p` | 1 (또는 2) | GPU당 하나의 모델 복제본을 사용합니다(또는 가중치 복제를 통한 배치 prompt 처리량 두 배를 위해 두 개). |

이 수치는 GPU 간 오버헤드를 최소화하면서도 여러 GPU를 활용해 대규모 prompt를 가속화합니다. 다음으로, 표 18-2에 제시된 디코딩 단계의 병렬화 전략 예시를 살펴보겠습니다.

표 18-2. 디코드 병렬화 전략 예시

| 병렬 처리 전략 | 기호 | 값 | 설명 |
|---|---|---|---|
| 텐서 병렬 처리 | `TP_d` | 1 (기본값)<br>...<br>*N* (GPU 수) | 간편성과 최소 동기화 오버헤드를 위해 TP_d = 1을 기본값으로 설정합니다.<br>TP_d = N GPU 수를 설정하면 소규모 배치에서 작은 GEMM의 효율성을 높이거나 단일 GPU로 모델을 수용할 수 없는 경우에 효율성을 높일 수 있습니다. |
| 파이프라인 병렬 처리 | `PP_d` | 1 | 파이프라인 병렬 처리는 단일 토큰 디코딩에 버블을 추가하므로 PP_d = 1은 유휴 단계를 방지합니다. |
| 시퀀스 병렬 처리 | `SP_d` | 1 | 출력 시퀀스를 여러 GPU에 분할하는 경우는 드뭅니다.<br>SP_d = 1은 매우 긴 출력을 처리하지 않는 한 각 디코딩 스트림을 로컬로 유지합니다. |
| 데이터 병렬 처리 | `DP_d` | 1 | 디코딩 스트림당 GPU당 하나의 모델 복제본.<br>단일 스트림에 대해 복제하기보다는 별도의 복제본을 사용하여 병렬 요청을 처리하십시오. |

---

예를 들어 모델이 단일 Blackwell B200에 맞지 않는 경우, PP 사용 대신 디코딩에 대해 `TP_d` = 2 또는 4를 선호하십시오. 이는 파이프라인 버블을 피하는 데 도움이 됩니다.

---

이상적으로는 각 디코드 스트림이 단일 GPU에서 실행되어 GPU 간 오버헤드를 피합니다. 이는 모델이 GPU 메모리에 완전히 들어갈 때만 가능합니다. 이 경우 텐서 병렬화 정도( `TP_d` )는 1이 되어 디코드 중 텐서 병렬화를 전혀 사용하지 않음을 의미합니다. 모델이 메모리에 들어가지 않을 경우 텐서 병렬화 정도를 높일 수 있습니다(예: `TP_d` = *N*, 여기서 *N*은 GPU 수).

소규모 배치 처리를 위해 시스템이 작은 GEMM 연산을 발행하는 경우에도 텐서 병렬 처리 증가가 유용합니다. 이는 작은 행렬 곱셈을 여러 장치에 분산하면 통신 지연이 연산 뒤로 숨겨져 전체 처리량이 높아질 수 있기 때문입니다.

이 값들은 예시일 뿐이지만, 핵심은 분리를 통해 prompt 측에서 TTFT 목표 달성을 위해 자원을 별도로 구성할 수 있다는 점입니다. 동시에 디코딩 측에서는 최종 사용자에게 토큰을 스트리밍하는 처리량 및 지연 시간 목표를 달성하기 위해 자원을 독립적으로 조정할 수 있습니다.

이렇게 하면 두 병렬 처리 전략이 서로 간섭하지 않습니다. 통합 시스템에서 이를 시도한다면, 두 단계 모두에 대해 차선책인 하나의 타협 전략을 선택해야 할 것입니다.

### 프리필과 디코딩에 대한 서로 다른 정밀도

일부 추론 엔진은 프리필과 디코딩 간 서로 다른 정밀도 사용을 허용합니다. 예를 들어, FP8, INT8 또는 FP4와 같은 낮은 정밀도로 프리필을 수행하여 속도를 높일 수 있습니다. 동시에 생성 정확도 향상이 필요한 경우 더 높은 정밀도로 디코딩할 수 있습니다.

일반적으로 프리필 단계에서 계산된 KV 캐시가 디코드 단계에서 사용 가능하도록 두 단계 모두 동일한 정밀도로 실행해야 합니다. 그러나 이전 섹션에서 설명한 병렬화 변환과 유사한 변환을 적용할 수 있습니다. 전송 전에 FP16에서 INT8/FP8/FP4로 양자화 및 압축을 선택할 수 있습니다. 이후 수신 측에서 필요 시 다시 변환하면 됩니다.

예를 들어, 전송 속도를 높이기 위해 네트워크를 통해 낮은 정밀도 데이터를 전송할 수 있습니다. 또는 사용 가능한 FLOPS 등에 따라 송신기 또는 수신기에서 변환을 수행하도록 선택할 수도 있습니다.

이는 고급 개념입니다. 그러나 하드웨어 유형, GPU 수, 정밀도 등 거의 모든 측면이 분산형 설정에서 각 단계별로 독립적으로 조정될 수 있음을 보여줍니다.

## GPU-CPU 협업을 통한 하이브리드 프리필

지금까지 프리필 및 디코드 단계 모두 GPU( )에서 실행된다고 가정했습니다. 심지어 서로 다른 유형의 GPU를 사용할 수도 있습니다. 그러나 극한의 규모—또는 매우 큰 모델과 prompt—에서는 CPU가 GPU의 부하를 분산시킬 수 있는지 평가해 볼 가치가 있습니다.

현대 CPU는 신경망 연산에서 GPU보다 훨씬 느리지만, 풍부한 RAM, GPU 메모리 대역폭 경쟁 없음 , 극도로 긴 시퀀스나 토큰화, 패딩 등 트랜스포머가 잘 처리하지 못하는 작업에 대한 유연성 등 다른 장점을 지닙니다.

하이브리드 프리필 전략을 사용하면 프리필 계산의 일부를 CPU에서 수행합니다. 한 가지 시나리오는 초장문 prompt에 대한 CPU 오프로딩입니다. 대용량 문서 첨부 파일에서 수만 개의 토큰으로 구성된 prompt를 생각해 보십시오. 강력한 GPU조차도 메모리 제약으로 인해 이렇게 큰 prompt를 처리하는 데 어려움을 겪을 수 있습니다.

극도로 긴 prompt의 경우, 시스템은 긴 시퀀스를 저장할 수 있는 대용량 RAM을 갖춘 CPU 워커에서 모델의 초기 레이어를 수행하도록 선택할 수 있습니다. 이후 중간 결과를 GPU로 스트리밍하거나, 지연 시간이 문제가 되지 않는다면 프리필

전체를 CPU에서 수행할 수도 있습니다. 디코드 GPU는 이후 CPU 워커로부터 거대한 KV 캐시를 수신하게 됩니다.

---

대화형 추론에서는 흔하지 않지만, 장기 실행되는 "딥 리서치" 작업과 같은 일부 배치 또는 오프라인 파이프라인에서는 매우 긴 텍스트에 대해 CPU 전처리를 사용할 수 있습니다.

---

CPU 오프로딩의 실용적 용도는 백그라운드 또는 저우선순위 프리필 작업 처리입니다. 예를 들어, LLM 서비스는 비대화형 요청의 오프라인 처리를 위해 매우 큰 prompt 제출을 허용할 수 있습니다. 이러한 작업은 CPU 전용 워커에 할당되어 최종적으로 빠른 토큰 생성을 위해 디코드 GPU로 전달될 수 있습니다. CPU가 느리기 때문에 지연 시간은 길어지지만, 오프라인 작업이므로 수용 가능할 수 있습니다. 또한 이 구성은 더 상호작용적인 워크로드를 위해 GPU 리소스를 확보합니다.

하이브리드 프리필은 Grace Blackwell과 같은 CPU-GPU 슈퍼칩에서 더 흔합니다. 이러한 칩에서는 칩 간 상호 연결이 매우 빠릅니다. 또한 CPU 메모리가 GPU 메모리에 비해 방대하다는 점을 활용합니다.

GPU에서 빠르게 접근할 수 있는 CPU 메모리에 거대한 키-값 캐시를 저장한다고 상상해 보십시오. 하이브리드 프리필은 CPU 메모리를 활용해 입력 토큰을 버퍼링하거나 사전 처리하는 동안 GPU는 복잡한 트랜스포머 레이어에 집중합니다.

그레이스 블랙웰 슈퍼칩은 CPU가 메모리와 초기 레이어를 관리하고 GPU가 시퀀스 조각에 대한 밀집형 어텐션을 처리하도록 함으로써 방대한 컨텍스트를 처리할 수 있습니다. 그레이스 CPU는 HBM에 들어가지 않는 KV 캐시 데이터를 CPU DDR 메모리로 스필하는 데에도 사용될 수 있습니다. 이는 GPU가 지원할 수 있는 컨텍스트 길이를 효과적으로 확장합니다.

트랜스포머를 여러 장치에 분할 처리할 수 있습니다. GPU에서 첫 *N개* 레이어를 실행하여 대부분의 토큰을 처리하고 시퀀스를 압축한 후, 메모리가 풍부한 CPU로 다음 *M개* 레이어를 오프로드합니다. 마지막으로 남은 레이어를 GPU로 다시 가져와 최종 출력을 생성합니다.

이러한 레이어 분할 기법은 상당한 데이터 이동과 오케스트레이션 복잡성을 추가하므로, 초장문 컨텍스트나 심각한 GPU 메모리 제약과 같은 극히 드문 경우에만 정당화되어야 합니다. 그럼에도 극한의 추론 시나리오에서 하드웨어 한계를 어떻게 확장할 수 있는지 보여줍니다.

분산형 아키텍처에서 CPU를 활용하려면 세 번째 작업자 유형인 'CPU 사전 채우기 작업자'를 도입해야 합니다. 스케줄링 로직은 GPU 사전 채우기 작업자, CPU 사전 채우기 작업자, 로컬 디코딩 GPU 사전 채우기 중 세 가지 옵션 중 하나를 선택할 수 있습니다. 이 결정은 prompt 길이 또는 우선순위 같은 요소에 따라 달라집니다.

예를 들어 정책은 다음과 같을 수 있습니다: `prompt_length` 가 5,000을 초과하면 CPU 프리필 작업자로 라우팅합니다. 속도는 느리지만 적어도 GPU를 점유하지 않고 대용량 메모리를 사용할 수 있다는 점을 고려한 선택입니다. 이 경우 디코딩 단계는 키-값(KV)을 더 오래 기다려야 합니다. 극단적인 경우, 완전히 오프라인 상태라면 디코딩 자체도 CPU에서 수행될 수 있습니다.

일반적으로 CPU 오프로드는 TTFT(전체 처리 시간)를 증가시키므로, 일반적인 지연 시간에 민감한 요청에는 사용되지 않습니다. 이는 확장성과 안전망 기능에 가깝습니다. 활용될 경우, 시스템은 이 경로가 얼마나 자주 사용되는지 모니터링해야 합니다. 잦은 CPU 오프로드는 오히려 GPU 용량 증설이나 모델 최적화가 필요함을 나타낼 수 있기 때문입니다.

CPU 오프로드는 또한 GPU가 모두 바쁠 때 초장입력이나 버스트 같은 에지 케이스를 처리할 수 있게 합니다. 완전히 실패하는 대신 느린 CPU로 대체함으로써 이를 수행합니다. 그러나 처리 속도가 너무 느려 SLO 요구사항을 초과할 경우 신속한 실패가 최선임을 명심하십시오.

비용 측면에서, CPU 코어는 GPU 시간보다 저렴하므로 일부 작업에 CPU를 사용하는 것이 더 경제적일 수 있습니다. 일부 클라우드 공급자는 LLM 서비스를 위해 GPU와 CPU 인스턴스를 혼합하여 운영할 수 있습니다. CPU는 GPU를 활용하기 전에 입력 전처리 또는 소규모 모델 추론을 수행합니다.

요약하면, 코어 분산 논리가 GPU 간 작업 분배에 집중하는 반면, 견고한 초대규모 추론 시스템은 특정 작업 부하에 대해 CPU를 창의적으로 활용할 수 있습니다. 하드웨어가 Grace Blackwell과 같은 밀접하게 결합된 CPU-GPU 슈퍼칩 설계로 진화함에 따라 GPU와 CPU 사용의 경계는 모호해질 것입니다.

효율적인 스케줄링은 사용 가능한 모든 컴퓨팅 자원을 고려해야 합니다. 그러나 기본 원칙은 동일합니다. 중간 길이의 시퀀스에 대한 대규모 병렬 연산을 포함해 GPU가 가장 잘하는 작업에는 GPU를 사용하십시오. 그리고 극도로 긴 시퀀스, 메모리 집약적 작업 또는 우선순위가 낮은 작업처럼 GPU가 비효율적일 수 있는 경우에는 CPU를 사용하십시오.

# SLO 인식 요청 관리 및 결함 내결함성

초대규모 환경에서 SLO 목표를 달성하려면 효율적인 확장 및 스케줄링만으로는 부족합니다. 때로는 현재 부하 상태에서 SLO를 위반할 수 있는 작업을 거부하거나 연기해야 할 때도 있습니다. 이는 문케이크의 초기 거부 사례에서 언급한 바 있습니다.

## 조기 거부(접수 제어)

조기 거부(조기 거절 또는 입장 제어) 는 17장에서 소개되었습니다. 간단히 말해, 시스템이 지연 시간 목표 내에 요청을 처리할 수 없다고 예측할 경우 오류 반환

또는 "나중에 다시 시도해 주세요" 응답으로 신속하게 실패를 처리한다는 의미입니다( ). 이 예측은 현재 대기열 길이, 최근 처리량, 또는 응답 시간을 예측하는 경량 머신러닝 모델을 기반으로 할 수 있습니다.

조기 거부는 요청을 대기열에 넣었다가 SLO 마감 시간을 놓치는 방식과 대비됩니다. 이는 추론 시스템이 처리하는 요청이 보장 사항을 충족하도록 함으로써 유효 처리량(goodput)을 유지합니다.

Mooncake에서 조기 거부( ) 전략은 프리필(prefill) 및 디코드(decode) 클러스터의 예상 부하를 기준으로 들어오는 요청을 평가합니다. 예를 들어, 현재 많은 양의 긴 시퀀스를 처리 중인 디코드 클러스터를 생각해 보십시오.

새로운 요청이 도착하면, 예상 출력 길이를 기준으로 디코딩 활용도가 안전 임계값을 초과할지 여부를 시스템이 판단합니다. 이때 시스템은 그림 18-8과 같이 해당 요청을 즉시 거부하거나 연기하고 GPU 노드의 리소스를 확보합니다.

이를 통해 요청이 대기열에 머물다 처리 시간이 지연되어 지연 시간 SLO를 초과하고, 부족한 컴퓨팅 또는 메모리 대역폭 자원을 소모하여 다른 모든 요청을 늦추는 상황을 방지합니다. 이는 조용히 수락했다가 지연 시간 보장을 실패하는 것보다 신속하게 "너무 바쁨" 응답을 반환하는 것이 더 낫다는 사실을 강조합니다.

이는 웹 서버가 극심한 과부하 시 부하를 분산시켜 남은 요청을 허용 가능한 지연 시간으로 계속 처리하는 방식(공포의 HTTP 503 오류)과 유사합니다. 이는 모든 요청을 시간 초과 처리하는 것보다 낫습니다.

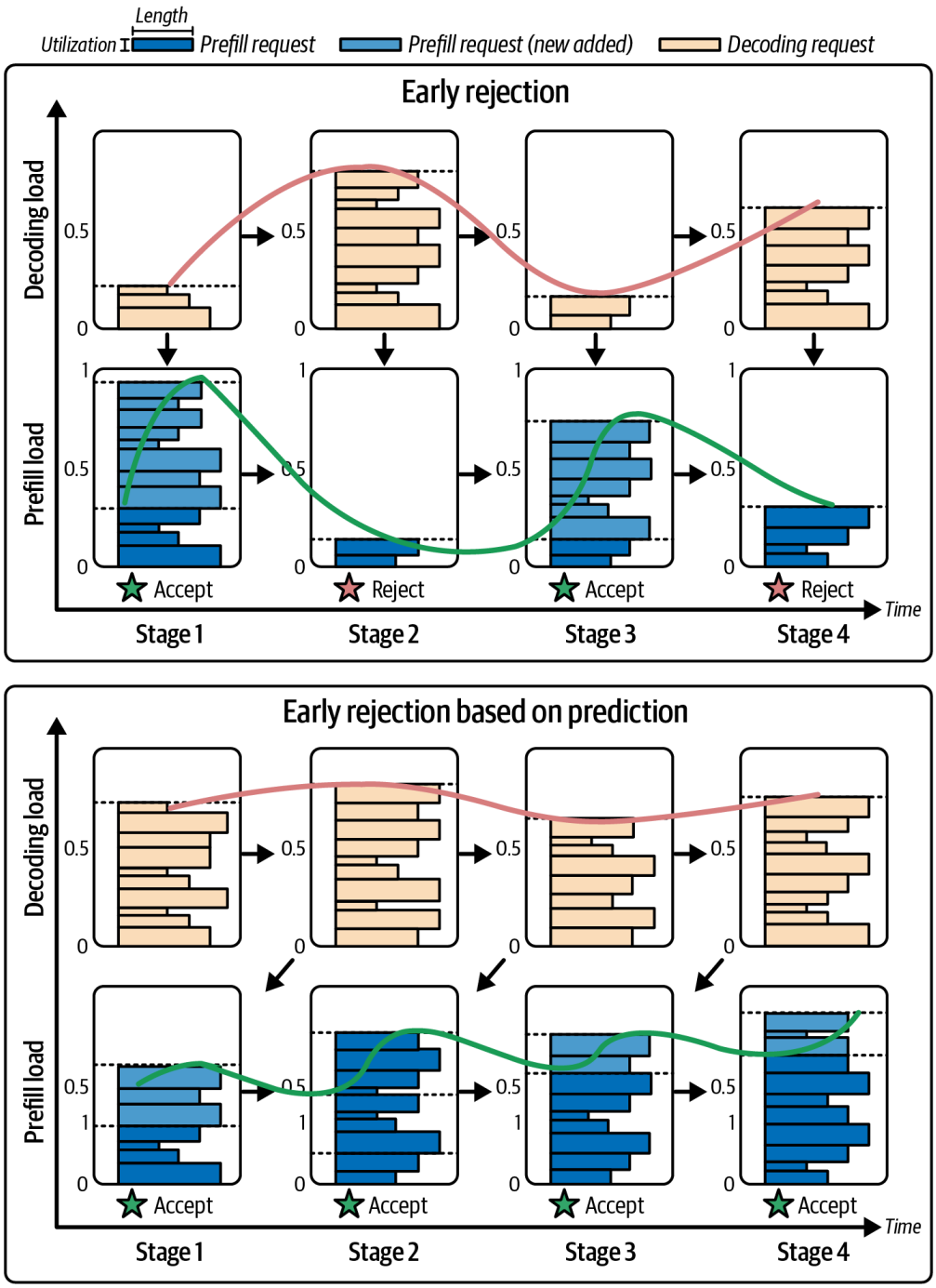


그림 18-8. 조기 거절 적용 시인스턴스 부하 (출처: https://oreil.ly/2xtK-)

LLM 서비스에서는 요청 크기와 처리 시간이 크게 달라지므로, 명확한 입장 제어 단계를 통해 수락된 요청의 성능을 유지하는 것이 중요합니다. 잘 설계된 입장 제어는 시스템이 수락한 부하에 대해 TTFT(최소 처리 시간)와 TPOT(평균 처리 시간) 목표를 모두 충족할 수 있는 상태를 유지합니다.

이는 동시 작업에 효과적인 제한을 설정합니다. 더 많은 작업을 수행하면 SLO를 위반할 경우 시도조차 하지 않습니다. 이는 스케일링 정책과 반드시 결합되어야 합니다. 예를 들어, 거절 대신 스케일 업을 선택할 수 있습니다. 그러나 스케일 업은 일반적으로 즉각적이지 않거나 최대 용량에 도달한 상태라면, 장기적으로 거절이 가장 안전하고 성능이 우수한 옵션입니다.

## 서비스 품질(QoS)

SLO 인식 관리 의 또 다른 측면은 서비스 품질(QoS) 차별화입니다. 일부 요청에는 우선순위 수준이 포함될 수 있습니다. 예를 들어 유료 고객은 무료 계층 고객보다 우선순위가 높습니다. 또는 대화형 쿼리는 오프라인 배치 요청보다 우선순위가 높습니다.

스케줄러는 다른 요청이 대기하거나 드롭되더라도 특정 요청을 우선 처리하여 항상 SLO를 충족시킬 수 있습니다. 디스어그리게이션은 프리필 및 디코드 작업자의 일부를 고우선순위 작업에 전용함으로써 이를 지원할 수 있습니다. 예를 들어, 클러스터 용량의 10%를 프리미엄 티어 요청 전용으로, 30%를 스탠다드 티어 전용으로 할당할 수 있습니다.

이러한 계층적 접근 방식은 유료 계층(프리미엄 및 스탠다드)에 여유 용량을 보장합니다. 이를 통해 해당 요청이 낮은 우선순위(무료 계층) 요청으로 인해 지연되지 않도록 합니다. NVIDIA Dynamo의 QoS 구성 예시를 통해 이러한 계층적 스케줄링 방식을 설명합니다:

```
# configs/qos.yaml

scheduler:
  # Define QoS classes and their reserved capacity
  qos_classes:
    - name: premium
      reserved_fraction: 0.10   # reserve 10% of prefill/decode workers
      priority: 100
    - name: standard
      reserved_fraction: 0.30   # reserve 30% for standard tier
      priority: 50
    - name: free
      reserved_fraction: 0.60   # share remaining capacity (free tier)
      priority: 10

# Map incoming requests to QoS classes by label
request_router:
  routes:
    - match:
        header: "x-customer-tier: premium"
      qos: premium
    - match:
        header: "x-customer-tier: standard"
      qos: standard
    - match:
        header: "x-customer-tier: free"
      qos: free
    - match:
        # default fallback
      qos: free
```

이 계층적 접근 방식은 예약된 용량을 활용하여 우선순위가 높은 요청을 낮은 지연 시간으로 수용하고 처리할 수 있게 합니다. 시스템이 우선순위가 높은 요청 처리에 바쁠 경우, 우선순위가 낮은 요청은 대기열에 추가되거나 잠재적으로 거부될 수 있습니다.

지연 시간 모니터링과 피드백 루프는 이러한 유형의 시스템에서 TTFT 및 TPOT p99와 p99.9를 지속적으로 모니터링하는 데 유용합니다. 시스템이 이러한 지표가 SLO 한계에 접근하는 것을 감지하면, 스케일링이나 신규 부하 거부, 최대 출력 길이 제한을 통한 요청 성능 저하, 또는 일시적인 부하 감소와 같은 조치를 트리거할 수 있습니다.

분산형 시스템에서는 프리필 큐와 디코딩 큐를 별도로 관찰할 수 있습니다. 이를 통해 병목 현상이 발생하는 측을 정확히 파악할 수 있습니다. 프리필이 지연되면 새 prompt 수신을 중단하거나, 디코딩이 포화 상태일 경우 긴 출력 추론 요청을 제한하는 등 시스템이 상황에 맞게 조정할 수 있습니다.

---

ChatGPT 사용 중 "무작위" 오류가 발생한 경험이 있을 수 있습니다. 이는 여러 원인이 있을 수 있으나, 시스템에 과부하가 걸렸을 때 특정 유형의 요청에 대해 이러한 "서킷 브레이커"가 작동하여 부하를 분산시키는 것이 원인 중 하나일 수 있습니다.

---

## 결함 내결함성

결함은 에서 견고한 추론 시스템을 운영하는 또 다른 측면입니다. 분산형 시스템에서 디코딩 노드가 생성 중간에 장애 발생 시, 앞서 논의한 KV 캐시 풀링을 통해 KV가 풀에 저장되어 있으므로 다른 노드에서 복구가 가능합니다. 이렇게 하면 다른 디코딩 작업자가 잔여 토큰 생성을 이어갈 수 있습니다 ( 약간의 지연 발생 가능).

KV가 저장되지 않았다면 시스템은 다른 노드에서 프리필을 재계산한 후 디코드를 계속해야 합니다. 이 때문에 vLLM 같은 시스템은 분산이 엄격히 필요하지 않더라도 주기적으로 KV 캐시를 체크포인트하거나 풀로 복사합니다. 이는 장애로부터 보호하기 위한 조치입니다.

프레임워크 수준의 KV 스냅샷 외에도, GPU 작업자를 호스팅하는 Linux 프로세스는 5장에서 논의한 바와 같이 cuda-checkpoint와 사용자 공간 체크포인트/복원(CRIU)을 통해 일시 중지 및 스냅샷될 수 있습니다. 이렇게 하면 체크포인트를 동일한 GPU 칩 유형의 다른 노드에 복원하여 선점이나 장애 시 작업 손실을 최소화할 수 있습니다.

---

추론 엔진의 콜드 스타트 지연 시간을 줄이려면, 매번 시작 시 그래프를 재컴파일하고 가중치를 재로드하는 대신 cuda-checkpoint를 사용하여 사전 예열된 GPU 메모리를 복원하세요.

---

체크포인트는 지연 시간에 영향을 줄 수 있지만, 최소한 요청은 완료될 수 있습니다. 프리필 노드 장애는 KV 데이터가 해당 노드에서 제거된 상태이므로, 계산 완료 후 디코드 워커나 KV 캐시 풀로 데이터를 전송한 후 발생할 경우 영향이 상대적으로 적습니다. 그러나 prompt 작업 중 프리필 워커가 장애를 일으키면 해당 prompt 작업은 다른 프리필 노드에서 재시도되어야 합니다. 아키텍처는 이러한 유형의 장애를 우아하게 처리하여 단일 노드 장애로 인해 전체 요청이 중단되거나 대규모 연쇄 지연이 발생하지 않도록 해야 합니다.

요약하면, SLO 인식 요청 관리는 극한 상황에서도 시스템이 처리하기로 선택한 부하에 대해 성능 보장을 유지하도록 합니다. 이는 어떤 요청을 처리할지, 어떤 요청을 버릴지, 어떤 요청을 지연시킬지 지능적으로 결정함으로써 이루어집니다. 조기 거부는 SLO 제약 조건 내에서 과부하를 방지하고 높은 처리량을 유지하기 위해 접수 시점에 부하 예측을 활용하는 구체적인 예시입니다.

SLO 인식 요청 관리와 결함 내결함성은 지금까지 논의한 다른 모든 전략(예: 확장, 스케줄링, 캐싱)과 결합되어 프로덕션 환경에서 엄격한 SLO 목표를 유지하는 데 기여합니다.

분리(Disaggregation)는 명확한 제어 지점을 제공함으로써 이를 가능하게 합니다. 프리필(prefill)과 디코드(decode) 메트릭을 별도로 관찰하고 필요한 곳에 특정적으로 수용 제어를 적용할 수 있습니다. 예를 들어 프리필 클러스터가 지연되면 새 prompt 수신을 중단하거나, 디코드 클러스터가 용량에 도달하면 긴 출력 허용을 중단할 수 있습니다.

그 결과, 부하가 용량을 초과해도 성능이 급격히 저하되지 않는 예측 가능하고 안정적이며 적응력 있는 추론 서비스를 구현할 수 있습니다. 대신 초과 부하를 우아하게 거부하고 변화하는 조건을 처리하기 위해 신속하게 재조정합니다.

## 동적 스케줄링 및 부하 분산

사전에 프리필(prefill)과 디코드(decode)에 할당할 프리필-디코드( ) 리소스 비율을 결정할 수 있습니다. 그러나 이후 작업 부하 변화에 따라 이러한 리소스 분배를 동적으로 조정해야 합니다.

순전히 정적 구성으로 운영할 경우, 작업 부하 구성비가 변해 prompt 처리 비중이 높아지거나 생성 처리 비중이 높아져야 할 때 고정된 비율은 최적화되지 못합니다. 프리필과 디코딩 용량의 최적 균형은 시간이 지남에 따라 변화합니다. 특정 순간에는 신규 요청이 대량으로 유입되어 프리필 비중을 크게 늘려야 할 수 있습니다. 이후에는 장시간 지속되는 추론 생성 작업이 많아져 디코딩 비중을 크게 늘려야 할 수도 있습니다.

적응형 스케줄링 및 부하 분산 메커니즘은 어느 단계도 병목 현상이 되지 않도록 시스템을 지속적으로 조정하는 것을 목표로 합니다. 이는 변화하는 조건에서도 처리량을 높게 유지합니다.

vLLM, SGLang, NVIDIA Dynamo와 같은 현대적 추론 엔진은 플랫폼에 부하 모니터링 및 동적 작업자 할당 기능을 통합합니다. 또한 많은 클라우드 추론 플랫폼은 유사한 원리를 사용하는 맞춤형 자동 확장기 및 라우터를 보유하고 있습니다.

## 적응형 리소스 스케줄링 및 핫스팟 방지

분리형 설정 의 한 가지 문제는 프리필 클러스터와 디코드 클러스터 간의 부하 불균형입니다. 현재 부하 구성에 대해 프리필 작업자와 디코드 작업자의 비율이 잘못 구성되면 한 쪽은 포화 상태에 이르는데 다른 쪽은 활용도가 낮은 상태로 남을 수 있습니다.

예를 들어, 프리필에 비해 디코드 작업자가 충분하지 않으면 디코드 작업이 대기열에 쌓이게 됩니다. 이로 인해 prompt가 빠르게 처리되고 있음에도 TPOT(평균 대기 시간)이 증가합니다. 반대로 디코드가 과도하게 프로비저닝되었지만 프리필이 부족하면 새 요청이 시작을 기다리게 됩니다. 이로 인해 TTFT(전체 처리 시간)가 높아지는 반면 많은 디코드 GPU가 유휴 상태로 남아 있게 됩니다.

이상적인 시나리오는 양측이 용량에 근접하여 작업하지만 과부하 상태가 아닌 경우입니다. 이렇게 하면 시스템은 프리필과 디코딩 모두를 바쁘게 유지하면서도 어느 쪽에도 대기열이 증가하지 않습니다.

정적 프리필-디코드 작업자 할당은 실제 환경에서 충분하지 않습니다. 워크로드가 하루 종일 크게 변동할 수 있기 때문입니다. 장기간 운영되는 대규모 추론 시스템에서는 입력 길이 및 출력 길이의 조합이 시간대별로 크게 달라질 수 있습니다.

예를 들어, 한 시간 동안은 긴 질문에 짧은 답변을 생성하는 작업(예: 요약)이 많을 수 있습니다. 이 경우 프리필 리소스는 더 많이, 디코드 리소스는 더 적게 필요합니다. 다음 시간에는 짧은 질문에 긴 답변을 생성하는 작업(예: 추론, 웹 검색)이 발생할 수 있습니다. 이 경우 프리필은 적게 필요하지만 디코드는 훨씬 더 많이 필요합니다.

다시 말해, 한 시나리오에서 효과적이었던 고정된 프리필 대 디코드 비율이 다른 시나리오에서는 최적이지 않을 수 있습니다. 따라서 프리필 기반 추론( `x_p` )과 디코드 기반 추론( `y_d` )의 최적 구성은 워크로드에 따라 달라지며 시간이 지남에 따라 변동합니다.

이를 해결하기 위해 고급 추론 시스템은 부하를 재분배하거나 인스턴스를 실시간으로 재할당할 수 있는 적응형 스케줄링 알고리즘을 활용할 수 있습니다. 다음으로 이러한 접근법 몇 가지를 살펴보겠습니다.

### TetriInfer의 2단계 스케줄러

의 연구용 프로토타입 스케줄러인 TetriInfer는 두 가지 세분화 수준에서 작동합니다. 첫째, 개별 요청 수준에서 현재 부하를 기반으로 들어오는 요청을 특정 프리필 및 디코드 인스턴스에 할당합니다. 이는 우리가 기대하는 일반적인 라우팅

입니다. 둘째, 클러스터 전체의 리소스 사용률을 모니터링하고 병목 현상이 발생할 수 있는 위치를 예측합니다. 이를 통해 핫스팟을 방지하기 위해 작업을 선제적으로 이동시킵니다.

예를 들어, 스케줄러가 한 디코드 노드에 매우 긴 시퀀스들이 대량으로 대기열에 쌓이는 것을 감지할 수 있습니다. 문제가 발생하기 전에, 스케줄러는 해당 시퀀스 일부를 일반적으로는 라우팅하지 않을 다른 디코드 노드로 이동시킵니다. 이 노드가 더 가용성이 높기 때문입니다. 이렇게 하면 부하가 균형 있게 분산됩니다. 그림 18-9는 기존 시스템과 TetriInfer의 실행 타임라인 및 아키텍처 비교를 보여줍니다.

여기서 볼 수 있듯이, 큐 길이, GPU 사용률 추세 등을 통해 리소스 사용량을 예측함으로써 TetriInfer의 2단계 스케줄러는 클러스터 전반에 걸쳐 부하를 완화하고 단일 노드가 과부하되는 것을 방지합니다.

---

TetriInfer라는 이름은 GPU 시간을 간섭 없이 채우기 위해 테트리스 조각처럼 요청을 "패킹"한다는 점을 암시합니다.

---

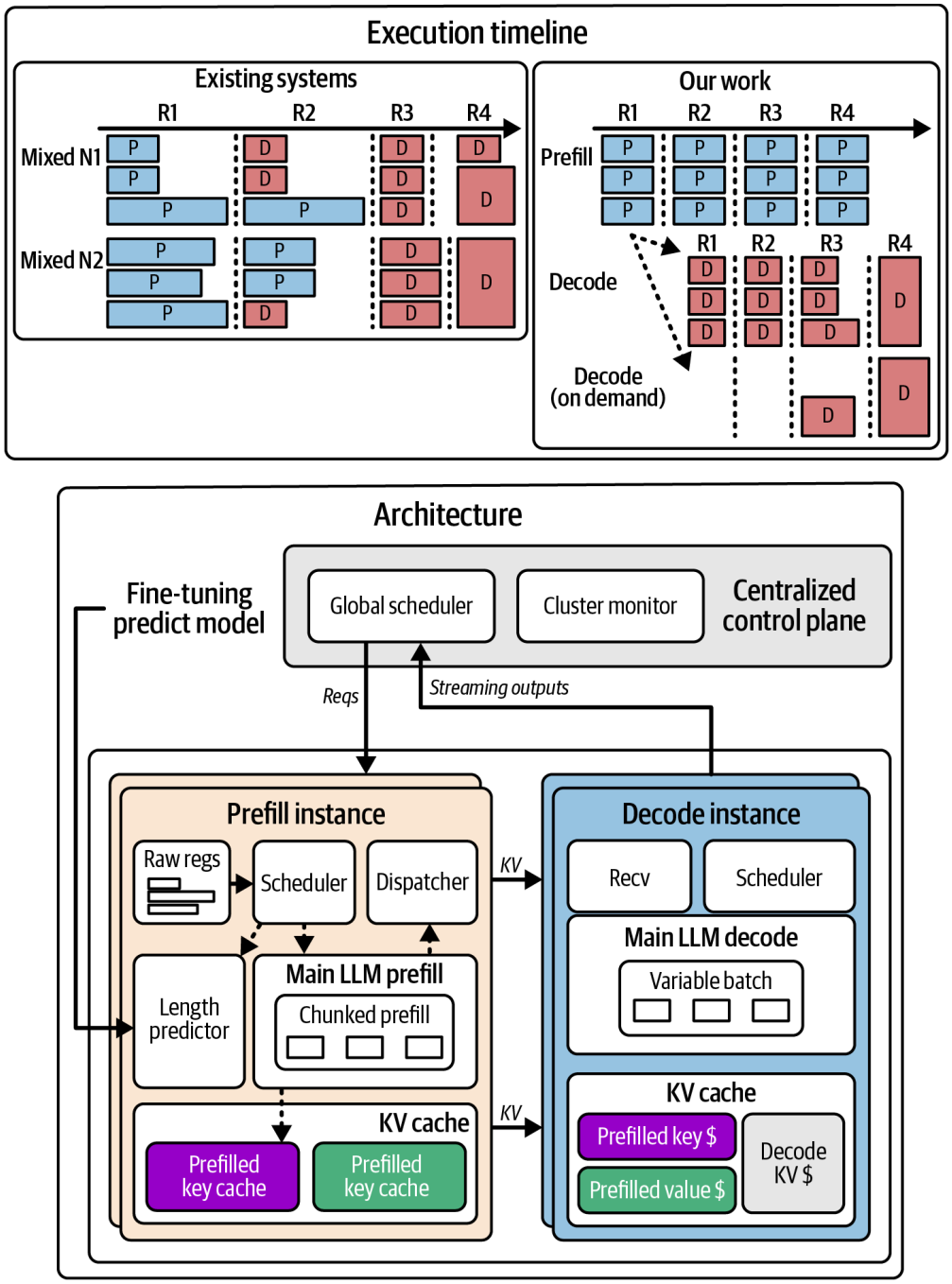


그림 18-9. 기존 시스템과 TetriInfer 아키텍처비교 (출처: *https://oreil.ly/_3KGj*)

## Arrow의 적응형 인스턴스 확장

의 동적 리소스 스케줄링을 위한 또 다른 기법은 Arrow (인기 있는 Arrow 데이터 형식과 혼동하지 말 것)입니다. 이는 분산 시스템이 작업 부하 변화에 대한 반응이 종종 지연된다는 사실을 활용하는 적응형 인스턴스 스케일링 기법입니다. 예를 들어, 입력 대 출력 분포가 변경되면 사전 채우기 작업자와 디코딩 작업자의 정적 수가 즉시 조정되지 않습니다. 이로 인해 한 쪽이 병목 현상이 되어 일시적인 유효 처리량 손실이 발생합니다.

Arrow는 입력 토큰 속도와 출력 토큰 속도, 그리고 클러스터 내 각 작업자 풀의 백로그를 측정하여 워크로드를 지속적으로 분석합니다. 그런 다음 그림 18-10과 같이 작업자 할당을 동적으로 조정합니다.

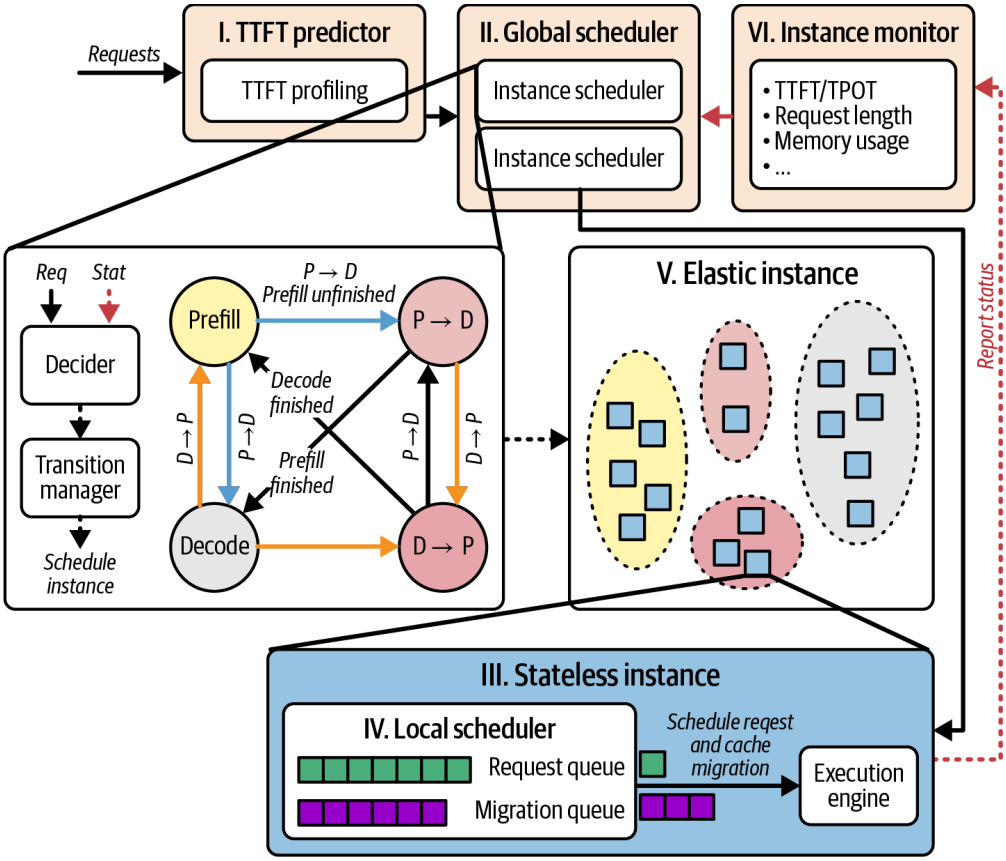


그림 18-10. Arrow 아키텍처

클라우드 환경에서는 출력 부하가 증가할 때 디코드 인스턴스를 추가로 가동할 수 있습니다. 입력 중심 워크로드가 감지되면 디코드를 축소하고 프리필 인스턴스를 더 많이 가동할 수도 있습니다.

Arrow의 설계에는 요청 스케줄링(어떤 노드가 어떤 요청을 처리할지 결정, 다른 스케줄링 기법과 유사)과 인스턴스 스케줄링(프리필 또는 디코딩 인스턴스의 시작/종료 시점 결정)이 모두 포함됩니다.

Arrow는 프리필 및 디코드 작업자 수를 조정 가능한 매개변수로 취급하여 거의 실시간으로 조정 및 확장할 수 있습니다. Arrow의 설계는 LLM 추론 스케줄러 내부로 자동 확장 로직을 도입합니다.

예를 들어, 입력은 많지만 출력은 적은 요청이 대량으로 유입되면 프리필 노드가 병목 현상을 일으킬 수 있습니다. 이 경우 Arrow는 증가하는 프리필 대기 시간이나 상승하는 TTFT 백분위수를 감지하고 일부 디코드 작업자를 프리필 작업자로 전환하거나 추가 프리필 인스턴스를 시작하여 급증하는 요청을 처리하기로 결정합니다.

반대로, 매우 긴 답변(예: 추론 체인)이 필요한 출력 집약적 요청이 대량으로 유입되면, TPOT 상승과 긴 디코딩 대기열 형성을 통해 디코딩 클러스터가 지연되기 시작할 수 있습니다. 이 경우 Arrow는 디코딩 단계에 더 많은 GPU 작업자를 할당할 수 있습니다. prompt 도착 속도가 느려졌다면 일부 프리필 노드를 일시적으로 유휴 상태로 전환하는 방식이 될 수 있습니다. 그렇지 않다면 단순히 새로운 디코딩 노드를 추가할 수도 있습니다.

고정된 수의 GPU를 가진 정적 온프레미스 클러스터에서는 동적 확장이 프리필 작업에 할당된 일부 GPU에 역할 전환을 지시하여 디코딩 풀에 잠시 참여하도록 하는 작업 재할당을 수반합니다. 각 컴퓨팅 노드가 프리필 작업자와 디코딩 작업자 모두로 실행되도록 구성된 경우 가능합니다.

역할 전환 시 오버헤드가 발생할 수 있습니다. 모델이 병렬화 전략이나 양자화 선택 등에 따라 서로 다른 모델 파티션을 로드해야 할 수 있기 때문입니다. 일부 설계는 모든 모델 가중치를 모든 GPU에 로드한 상태로 유지하고 필요에 따라 서로 다른 작업을 할당합니다. 이는 클러스터를 유연한 풀로 효과적으로 처리하여, 특정 시점에 일부 작업자는 프리필을 수행하고 다른 작업자는 디코드를 수행하도록 합니다.

이는 시간 공유 방식의 통합 클러스터와 유사하지만, 한 노드가 프리필 작업 수행에 일정 시간을 할당한 후 디코드 작업 수행으로 전환하는 거친 수준의 세분화입니다.

클라우드 배포 환경에서 동적 확장성은 각 역할에 필요한 포드와 노드를 추가하거나 제거하기 위해 쿠버네티스 수평 포드 확장기(Horizontal Pod Autoscaler)나 클러스터 확장기(Cluster Autoscaler) 같은 자동 확장기와 연동하는 것을 의미하기도 합니다. 예를 들어, Arrow는 프리필 부하가 지속적으로 높을 경우 새로운 프리필 포드를 시작하도록 트리거할 수 있습니다. 이는 필요에 따라 각 측면이 확장되거나 축소되는 완전히 탄력적이고 분산된 프리필 및 디코딩 추론 클러스터를 구현합니다.

실제 환경에서 새 GPU 포드의 확장에는 수십 초 이상 소요될 수 있습니다. 따라서 시스템은 용량 확보를 기다리는 동안 부하를 분산해야 할 수 있습니다. 다음에서 논의할 Mooncake의 적응형 전략이 바로 이러한 상황을 처리하는 방식입니다.

### Mooncake 적응형 전략

Arrow 외에도 Mooncake 시스템 은 적응형 전략을 강조합니다. Mooncake는 수요 관리와 공급 관리를 보완하는 예측 기반 접근 제어인 *'조기 거부( early rejection)*'를 도입했습니다.

구체적으로, 시스템이 SLO 내에서 새 요청을 처리할 충분한 용량이 없다고 예측하면 해당 요청을 수락하여 SLO 위반 가능성을 초래하기보다 요청 자체를 거부합니다. 이는 확장보다는 동적 *부하 분산* 형태이지만, 과부하를 방지하는 것을 목표로 합니다.

이전 섹션에서 문케이크의 조기 거부 접근법을 이미 다루었습니다. 여기서 핵심은 적응이 수요 측에서 일부 요청을 제한하거나 거부하는 것뿐만 아니라 공급 측에서 자원을 추가하거나 재배분함으로써도 가능하다는 점입니다.

동적 스케일링의 핵심 이점은 워크로드 변동에도 높은 유효 처리량(goodput)을 유지하는 것입니다. 프리필-디코드 비율을 자동으로 조정함으로써, 한쪽 GPU가 유휴 상태인 반면 다른 쪽은 과부하 상태인 불일치 기간이 장기화되는 것을 방지합니다.

이상적으로는 두 유형의 노드가 균등하게 활용됩니다. 이는 병목 현상이 발생하자마자 완화하여 지연 시간을 개선할 뿐만 아니라, 한 유형의 GPU가 유휴 상태로 방치되는 동안 다른 유형이 과부하되는 상황을 방지함으로써 효율성도 높입니다. 대신 자원을 재할당하거나 작업을 이동시켜 균형을 유지합니다.

성과 측면에서 적응형 시스템은 다음과 같은 주장을 할 수 있습니다. "적응형 스케일링 적용 후, 정적 시스템이 달성하지 못한 트래픽 패턴 전반에서 90% 이상의 SLO(서비스 수준 목표) 준수를 유지했으며, 급증 시점에 자원을 신속히 재배치하여 X% 더 많은 요청을 처리했습니다."

구체적으로, Arrow의 결과는 극도로 변동하는 워크로드 시나리오에서 비적응형 시스템 대비 최대 5.6배 높은 요청 처리율을 언급합니다. 일반적인 개선 효과는 더 낮았지만 여전히 유의미했습니다. 정확한 개선 효과는 워크로드 변화의 극심함에 따라 달라집니다.

이 모든 것은 분산화가 위상 간섭 문제를 제거함을 보여줍니다. 또한 동적 분산화는 고정 할당으로 인한 위상 불균형이라는 다음 제약 조건을 제거합니다. 이러한 적응성을 구현하려면 스케줄러가 프리필 큐 길이, 디코드 큐 길이, TTFT/TPOT 백분위수 등 메트릭을 지속적으로 모니터링해야 합니다. 이러한 메트릭은 K8s 배포 환경에서 Prometheus와 커스텀 컨트롤러를 통해 제어 대시보드에 입력되는 경우가 많습니다. 이후 알고리즘이 의사 결정을 자동화할 수 있습니다.

정교한 추론 시스템은 ARIMA와 같은 예측 모델 및 기타 예측 기법을 활용하여 시간대 패턴을 기반으로 트래픽 급증과 이동을 예측합니다. 과거 사용 패턴을 바탕으로 오후 9시에 긴 출력 작업이 급증할 것으로 예측되면, 스케줄러는 적시에 더 많은 디코드 용량을 선제적으로 할당할 수 있습니다. 궁극적인 목표는 수동 개입이나 재구성 없이 SLO 내에서 처리되는 요청의 비율을 높게 유지하는 것입니다.

## 동적 리소스 스케일링

예측 스케줄링 개념을 기반으로 한 적응형 인스턴스 스케줄링( )은 부하에 대응하여 프리필(prefill)과 디코드(decode) 간 자원 분배를 변경하는 데 특화되어 있습니다. 예를 들어, Arrow의 적응형 인스턴스 스케줄링은 프리필 작업자와 디코드 작업자의 수를 지속적으로 조정합니다. 추론 시스템에서 부하를 균형 있게 분배하는 몇 가지 방법은 다음과 같습니다:

*탄력적 인스턴스*

쿠버네티스 또는 유사한 환경에서는 각 배포에 대한 자동 확장 정책을 정의할 수 있습니다. 예를 들어, 프리필 GPU 평균 사용률을 70%로 유지하도

록 지정할 수 있습니다. GPU 사용률이 이보다 높아지면 시스템은 프리필 작업자 풀에 포드나 노드를 추가합니다. 사용률이 낮아지고 디코딩 사용률이 여전히 높을 경우, 시스템은 프리필에서 디코딩으로 포드 하나를 이동시킬 수 있습니다. 이는 규칙 기반이거나 알고리즘적일 수 있으며, 예를 들어 각 간격마다 최적의 구성을 구해 새로운 $X$ 및 $Y$ 카운트를 선택하는 방식이 있습니다.

*인스턴스 "플립" 메커니즘*

TetriInfer 논문에서는 필요 시 일부 노드가 역할을 전환할 수 있는 "인스턴스 플립"을 설명합니다. 이를 위해서는 모델이 두 역할을 모두 처리할 수 있도록 적절히 로드, 샤딩 및 양자화되어야 합니다.

*탄력성을 위한 상태 비저장성*

Arrow는 상태 비저장 인스턴스를 활용합니다. 따라서 작업자에는 장기 세션 상태가 저장되지 않습니다. 이로 인해 작업자를 자유롭게 재배정할 수 있습니다. 활성 요청의 키-값 캐시는 디코딩 노드가 토큰 생성 중간에 프리필로 전환될 때 복잡성을 유발하므로, 역할 전환 전 디코딩 작업이 완료될 때까지 기다려야 하는 경우가 많습니다.

*안정성과 진동*

역할을 빠르게 전환하면 진동 및 스래싱이 발생할 수 있습니다. 따라서 역할 전환이 너무 빈번해지지 않도록 최소 시간 제한을 적용하는 것이 일반적입니다.

부하 분산 외에도 고려해야 할 또 다른 측면은 멀티테넌시 또는 혼합 워크로드입니다. 클러스터가 서로 다른 모델이나 작업을 처리하는 경우, 수요에 따라 단일 모델 범위를 넘어 GPU를 재할당할 수도 있습니다.

분리형 아키텍처의 모듈식 구성은 유휴 디코드 GPU를 활용해 다른 소규모 모델의 추론 작업을 실행할 수 있게 합니다. 이는 본 문서 작성 시점에는 아직 주류가 아닌 확장 기능이지만, 추론 엔진이 진화하고 설계 유연성이 높아짐에 따라 개념적으로 가능합니다.

결론적으로, 초대형 추론 시스템은 현재 워크로드에 맞춰 자원 할당을 지속적으로 조정하는 피드백 루프가 필요합니다. TetriInfer, Arrow, Mooncake 등은 이러한 피드백 루프를 활용하고 부하 변화에 적응할 때 상당한 성능 향상을 보입니다. 이는 분산화가 간섭을 제거하는 반면, 적응형 분산화는 불균형을 제거한다는 점을 강조합니다. 동적 환경에서 최상의 성능을 위해서는 두 가지 모두 필요합니다.

## 핵심 요약

이 장에서는 통합 메가커널, 효율적인 메모리 할당, 빠른 데이터 전송, 프리필/디코딩 분리, KV 캐시 풀, 동적 스케일링, 지속적인 SLO 인식 등 다양한 기법을 다

루었습니다. 주요 내용은 다음과 같습니다:

*디코드 단계 가속화*

FlashMLA, ThunderMLA, FlexDecoding 등 융합 어텐션 커널을 사용하면 디코딩 단계를 크게 가속화할 수 있습니다. 이러한 커널은 단일 토큰 처리량과 GPU 활용도를 향상시킵니다.

*KV 캐시를 최우선 요소로 취급*

분리(disaggregation)를 통해 GPU 간 KV 캐시를 공유하세요. 중복 계산을 피하기 위해 접두사를 재사용하세요. 이는 글로벌 캐시 풀과 해싱을 통해 가능해집니다.

*프리필과 디코드 작업자 간 오버헤드를 거의 제로에 가깝게 유지하기 위해 노력하기*

GPUDirect RDMA 및 NIXL을 활용한 고속 GPU-to-GPU 전송을 활용하세요. 컴퓨팅/전송 작업을 중첩시켜 프리필과 디코딩 작업자 간 오버헤드를 거의 제로 수준으로 줄일 수 있습니다.

*각 단계에 특화된 하드웨어와 병렬 처리 채택*

프리필과 디코드를 분리하면 단계별 전용 하드웨어와 병렬 처리가 가능합니다. 예를 들어 프리필에는 고성능 GPU 또는 다중 GPU 노드를 사용하고, 디코드에는 메모리 용량이 큰 GPU 또는 단일 GPU를 활용합니다. 목표는 비용 절감과 처리량 증대입니다.

*적응형 및 동적 알고리즘을 활용하여 시스템 최적화*

가변적 부하 하에서 지연 시간 보장을 유지하고 모든 GPU를 과부하 없이 완전히 활용하려면 적응형 스케줄링과 SLO 인식 제어(예: 조기 거부 및 동적 스케일링)가 필요합니다.

# 결론

초대규모 LLM 추론에는 고수준 적응형 리소스 관리와 저수준 커널 및 메모리 최적화를 아우르는 종합적 접근이 필요합니다. 본 장에서 제시된 기법들을 결합함으로써, 고도로 최적화된 추론 배포는 현대 하드웨어에서 최대 처리량을 달성하면서 엄격한 지연 시간 보장을 충족할 수 있습니다.

하드웨어가 지속적으로 진화하고 GPU(예: 메모리 증설 및 전용 추론 코어)와 소프트웨어 프레임워크(예: 동적 라우팅 및 유연한 역할 할당)가 더욱 정교해짐에 따라 이러한 최적화 효과는 누적되어 더욱 강력해질 것입니다. 이를 통해 추론 엔진은 더 큰 모델, 더 긴 컨텍스트, 더 많은 사용자를 효율적으로 처리할 수 있게 될 것입니다.