제17장. 추론을 위한 분산 프리필 및 디코딩확장

이전 장에서 언급했듯이, LLM 추론은 두 가지 별개의 단계로 나눌 수 있습니다: 사전 채우기( ) 단계와 디코딩(decode) 단계입니다. 사전 채우기 단계는 입력 prompt를 처리하여 해당 prompt에 대한 LLM의 내부 키-값(KV) 캐시를 생성하는 반면, 디코딩 단계는 캐시된 값들을 사용하여 출력 토큰을 하나씩(또는 추측적 디코딩의 경우 한 번에 몇 개씩) 생성합니다.

이 두 단계는 근본적으로 다른 성능 특성을 가집니다. 프리필 단계는 컴퓨팅 바운디드하며, 수천 개의 토큰에 대한 병렬 행렬 연산을 다수 수행하여 상당한 양의 FLOPS를 소모합니다. 반면 디코딩 단계는 메모리 I/O 바운디드하며, 각 토큰 생성을 위해 대규모 KV 캐시를 읽고 새 값을 쓰며 메모리 대역폭에 부담을 줍니다. 간단히 말해 프리필은 높은 처리량의 병렬 작업 부하인 반면, 디코딩은 순차적이며 지연 시간에 민감한 작업 부하입니다.

초기 LLM 서비스 시스템은 두 단계를 동일한 하드웨어 상의 단일 통합 파이프라인으로 처리했습니다. 따라서 일반적으로 요청 배칭을 통해 처리량을 우선시함으로써 프리필 단계를 중시했습니다. 그러나 대화형 애플리케이션이 증가함에 따라, 첫 번째 토큰까지의 시간(TTFT, 모든 토큰에 대한 프리필 지연) 및 출력 토큰당 시간(TPOT, 토큰당 디코드 지연)과 같은 실시간 성능( ) 지표가 순수 처리량만큼 중요해졌습니다. 단일 GPU 기반 추론 엔진이 두 단계를 동시에 처리할 때 TTFT와 TPOT를 동시에 최적화하는 것은 어렵습니다.

많은 요청을 배치하면 처리량은 향상되지만, 모든 요청이 가장 느린 프리필을 기다려야 하므로 TTFT는 악화됩니다. 또한 디코딩 단계가 새 prompt 프리필 뒤에 백로그가 쌓이게 되어 TPOT에도 영향을 미칩니다.

단일형 추론 시스템은 첫 번째 토큰까지의 시간(TTFT)을 개선(단축)하는 대신 후속 토큰 생성이 느려지는 대가를 치르거나, 토큰당 처리량(TPOT)을 개선(증가)하는 대신 새 요청이 높은 초기 지연 시간을 겪게 되는 선택을 해야 합니다. 극단적인 경우, 하나의 긴 prompt가 GPU를 완전히 점유하여 다른 사용자의 모든 프롬프트 프리필 작업을 차단할 수 있습니다. 또한 디코딩이 시작되면, 한 번에 하나의 토큰만 처리하는 방식은 각 토큰 생성 사이에 GPU 코어를 유휴 상태로 남겨둡니다.

이러한 문제를 해결하기 위해 연구진과 엔지니어 는 두 단계를 분리하는 방법을 모색했습니다. 핵심 통찰은 프리필과 디코딩이 반드시 동일한 하드웨어, 심지어 동일한 유형의 하드웨어에서 실행될 필요가 없다는 점입니다.

프리필과 디코딩 단계를 분리한다는 것은 각 단계의 요구사항에 특화된 서로 다른 리소스에 할당하는 것을 의미합니다. 이 아이디어는 DistServe에 관한 논문에서 시스템에 의해 최초로 제안되었으며, 단계 간 간섭을 제거함으로써 TTFT와 TPOT 모두에 대한 엄격한 지연 시간 요구사항을 동시에 충족할 수 있음을 입증했습니다.

DistServe 평가 결과, 프리필/디코딩 분리 기술이 적용되지 않은 최신 기준 대비 엄격한 지연 시간 서비스 수준 목표(SLO) 내에서 7.4배 더 많은 요청을 처리할 수 있는 잠재력이 확인되었습니다. 이에 따라 업계 프레임워크들은 프리필 서버와 디코딩 서버를 분리하는 방식을 실험하기 시작했습니다.

오픈소스 vLLM 라이브러리는 LMCache 및 기타 구성 요소와 연계하여 분산 운영 방식을 도입했습니다. NVIDIA의 Dynamo는 동적 라우팅 및 자동 확장 기능을 갖춘 분산 프리필 및 디코딩을 구현하며 운영 세부 사항을 공개적으로 문서화합니다. 많은 공급업체와 오픈 프레임워크가 분리를 구현하거나 평가 중입니다. 예를 들어, 엄격한 지연 시간 SLO를 충족하기 위해 OpenAI, Meta, xAI의 산업 규모 서비스 시스템이 이 분리된 접근 방식을 채택한 것으로 알려졌습니다. 따라서 대규모 LLM 추론에서 분리된 프리필 및 디코딩은 표준 관행입니다.

초대규모 환경에서는 대규모 추론 배포가 수십만에서 수백만 개의 GPU를 활용해 수십억 건의 요청을 처리하기도 합니다. 이러한 환경에서 분리의 비용 및 성능 이점은 막대합니다.

워크로드를 분할함으로써 각 단계를 독립적으로 최적화하고 한 단계가 다른 단계의 병목 현상이 되는 것을 방지할 수 있습니다. 본 장의 나머지 부분에서는 극한 규모의 분산 프리필/디코드 추론 시스템을 설계하고 운영하는 방법을 탐구합니다.

본 장에서는 프리필과 디코딩 작업자 간 요청을 라우팅하는 스케줄링 알고리즘, 고부하 환경에서 서비스 품질(QoS)을 유지하는 기술, 그리고 이러한 분리를 효율적으로 만드는 메커니즘을 살펴볼 것입니다. 고속 상호 연결부터 특수화된 디코딩 커널에 이르기까지 모든 것을 탐구할 것입니다. 또한 각 단계에 서로 다른 GPU 유형을 사용하는 이종 하드웨어 전략에 대해서도 논의할 것입니다.

왜 프리필-디코드 분리를 하는가?

현대적인 대화형 LLM 서비스는 종종 TTFT 지연 시간 p99(요청의 99%)를 200~300ms 미만으로 목표로 합니다. LLM 서비스에 일괄적인 접근 방식을 적용하면 상당한 성능이 저하되므로 프리필 작업을 분리하지 않고서는 이를 보장하기가 거의 불가능합니다.

참고로 MLPerf v5.0(2025)의 Llama2 70B(700억 매개변수) 추론 벤치마크는 p99(99번째 백분위수) SLO로 TTFT 약 450ms, TPOT 지연 40ms를 목표로 했습니다. Llama 3.1 405B(4050억 매개변수)의 경우, 벤치마크는 약 6초 TTFT와 175ms TPOT를 목표로 했습니다. 구체적으로, 이 SLO는 Llama 2 Chat 70B와 Llama 3.1 405B Instruct의 p99 TTFT 및 p99 TPOT 목표를 반영합니다.

한 사용자의 요청이 수천 토큰 규모의 극히 긴 prompt를 포함하고 다른 사용자의 요청이 매우 짧은 prompt를 포함하는 시나리오를 고려해 보십시오. 분리되지 않은 프리필 및 디코딩이 없는 경우, 이러한 요청이 거의 동시에 도착하면 긴 prompt의 프리필 계산이 GPU를 장시간 차단하게 됩니다.

분리 처리 없이 짧은 prompt를 가진 두 번째 요청은 디코딩을 시작하기 전에 불필요하게 오랜 시간을 기다려야 합니다. 이는 한 요청의 프리필 작업이 다른 요청의 디코딩 작업을 지연시키기 때문에 *간섭이라고* 합니다. 프리필과 디코딩 간의 간섭은 연속 배치(continuous batching) 맥락에서 그림 17-1에 표시되어 있습니다.

단순한 선입선출(FIFO) 스케줄링 전략 하에서는 긴 prompt가 모든 작업의 꼬리 지연 시간을 증폭시킬 수 있습니다. 일반적으로 큐 앞부분에 위치한 길거나 연산 집약적인 프리필 작업은 뒤에 있는 짧고 가벼운 요청들을 차단합니다. 이를 *헤드 오브라인 차단(head-of-line blocking)*이라 하며, 이는 낮은 활용도, 지연 시간 이상치, 불만족스러운 최종 사용자로 이어집니다.

유연한 분산형 아키텍처에서는 대규모 prompt 프리필을 컴퓨팅 최적화 프리필 작업자 전용 풀로 전송하는 동시에, 경량 prompt 프리필은 프리필 작업자를 우회하여 디코드 작업자에게 직접 전송할 수 있습니다. 이러한 유연성 덕분에 짧은 토큰은 헤드 오브 라인 블로킹의 영향을 받지 않습니다. 이는 전체 처리량을 극대화하고 지연 꼬리 효과를 최소화합니다.

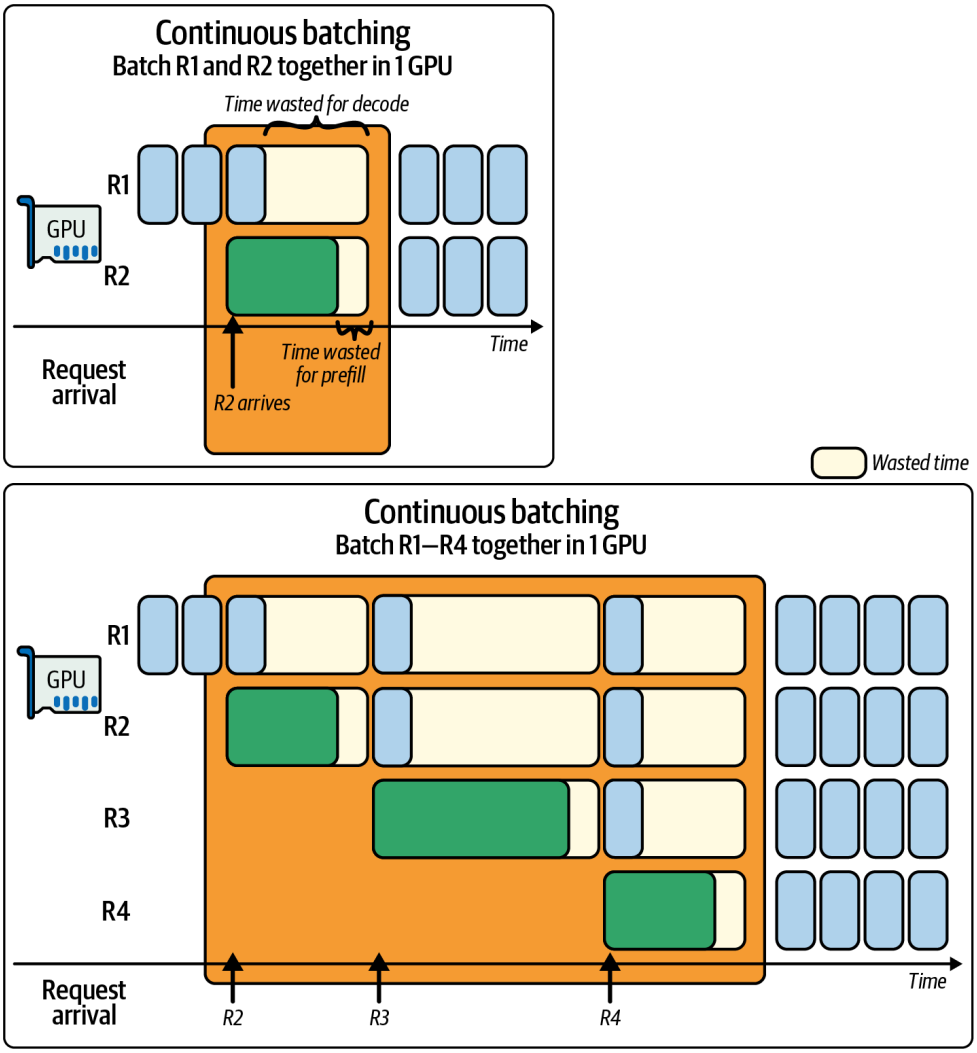

그림 17-1. 동일한 GPU에서 함께 실행되는 프리필과 디코딩으로 인한간섭 (출처: https://oreil.ly/GRkHs)

분리(Disaggregation)의 장점

분리에는 두 가지 주요 이점이 있습니다: 간섭 감소와 단계별 최적화입니다. 각각에 대해 살펴보겠습니다.

간섭 감소

분리(disaggregation)를 통해 프리필 작업은 더 이상 동일한 장치에서 디코드 작업과 경합하지 않습니다. 많은 토큰을 생성하는 바쁜 디코드 작업자가 다른 사용자의 prompt 처리를 방해하지 않으며, 그 반대의 경우도 마찬가지입니다.

각 단계에 전용 리소스를 할당함으로써 긴 prompt의 계산이 다른 사용자의 토큰 생성을 차단하지 않습니다. 실제로 이는 더 예측 가능한 지연 시간을 생성합니다. 그림 17-2는 동일 위치 배치 방식과 분산 방식의 프리필 및 디코딩 비교를 보여줍니다. 이 실험은 DistServe 논문과 저자들의 후속 블로그 게시물에서 더 자세히 설명됩니다.

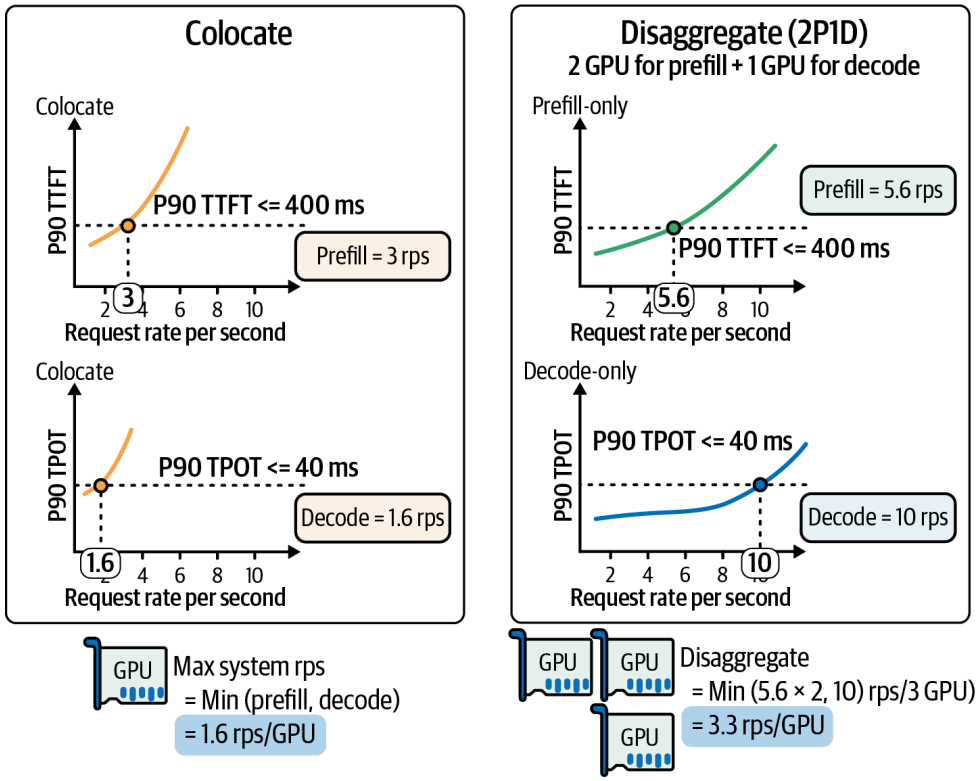

그림 17-2. 동일 위치 배치 방식과 분리 배치 방식의 프리필 및 디코드비교 (출처: https://oreil.ly/GRkHs)

여기서 SLO는 P90 TTFT에 대해 0.4초, P90 TPOT에 대해 0.04초로 설정되었습니다(예: 그림 17-2의 수평선). 콜로케이션 시스템은 주어진 TTFT 지연 시간 범위 내에서 초당 약 3건(RPS)의 유효 처리량만 지원할 수 있습니다. 또한 주어진 TPOT 지연 시간 바운디드 내에서만 1.6 RPS를 유지할 수 있습니다. 따라서 TTFT와 TPOT 지연 시간 SLO를 모두 충족해야 하므로, 동일 위치 구성의 유효 처리량은 1.6 RPS에 불과합니다.

두 단계를 분리하고 프리필 작업자 2개(GPU 2개)와 디코드 작업자 1개(GPU 1개)를 할당하는 *2P1D 구성에서*는 프리필 및 디코드 작업자 모두 단일 GPU를 사용하는 공동 배치 구성보다 더 나은 전체 RPS를 달성합니다. 구체적으로 프리필 작업자는 약 5.6 RPS에 도달하고 디코드 작업자는 3개의 GPU에 분산되어 약 10 RPS를 달성합니다. 따라서 2P1D 구성의 굿풋은 GPU당 3.3 RPS입니다.

GPU당 3.3 RPS는 프리필 작업자(5.6 RPS × 2 = 11.2 RPS)와 디코드 작업자(10 RPS)의 RPS 중 최소값을 취하여 계산됩니다. 이는 세 개의 GPU 전체에 걸쳐 총 10 RPS입니다. 따라서 RPS를 GPU 수(이 경우 3)로 나누어야 합니다. 구성된 SLO 기준, 이 시스템의 유효 처리량 결과는 10 RPS ÷ 3 GPU = GPU당 3.3 RPS입니다.

이 비교에서 디코드 측 개선은 주로 토큰당 지연 시간에 영향을 미칩니다. 반면 프리필 격리는 주로 첫 번째 토큰까지의 시간을 개선합니다. 양측 SLO(서비스 수준 목표)를 모두 충족해야 굿풋(goodput)으로 간주됩니다.

이러한 분리는 테일 지연 시간도 개선할 수 있습니다. 경험적으로, 분산 처리하는 시스템은 더 좁은 지연 시간 분포를 보이며 모놀리식 시스템에서 나타나는 긴 테

일을 피합니다. 크로스 페이즈 간섭을 제거함으로써 각 페이즈는 SLO를 더 안정적으로 충족하며 예측 가능한 일관성을 확보합니다.

이제 여러분은 "비용이 3배 증가하는 대신 성능이 2배 향상되는 것이 가치 있는가?"라고 질문해야 합니다. 이는 타당한 질문입니다. 이 솔루션의 비용 효율성을 높이기 위해서는 추가적인 튜닝이 필요하지만, 지연 시간 분포를 좁히고 굿풋을 개선하는 올바른 방향을 제시합니다. 워크로드에 따라 결정해야 합니다. 디어그리게이션은 굿풋 RPS 요구 사항을 충족하기 위한 인기 있는 옵션입니다.

단계별 최적화

단계별 최적화는 각 단계가 가장 적합한 하드웨어와 병렬성을 활용하도록 합니다. 예를 들어 프리필 단계는 컴퓨팅이 바운디드합니다. 따라서 일반적으로 텐서 병렬성을 높여 고성능 GPU에서 최대 FLOPS를 끌어냅니다. 또한 최신 GPU는 낮은 정밀도 모드(FP8 및 FP4)를 제공하여 컴퓨팅 집약적인 프리필 단계의 처리량을 증가시킬 수 있습니다.

가중치에는 FP4를, 검증된 경우 활성화에도 FP4를 선호해야 합니다. 많은 스택이 가중치에는 FP4를, 활성화에는 FP8을 사용합니다. 이러한 정밀도 감소는 정확도 손실을 최소화하면서 처리량을 극대화하고 HBM 사용량을 최소화하는 데 도움이 됩니다. 이러한 정밀도는 NVIDIA 텐서 코어 및 트랜스포머 엔진을 포함한 최신 하드웨어 및 소프트웨어 스택에서 지원됩니다.

반면 디코드 단계는 메모리 대역폭이 바운디드하며 GPU 간 동기화 오버헤드가 발생합니다. 따라서 연산 집약도를 높이기 위해 융합 커널에 더 의존하며 높은 메모리 처리량 GPU를 활용하므로 텐서 병렬성을 최소화하거나 제거하는 것이 가장 효율적입니다(`TP=1`).

단일 구조에서는 두 단계 모두에 동일한 GPU 유형과 병렬화 전략을 선택해야 하므로 최소한 한 단계에는 비최적화됩니다. 반면 분할 구조는 각 단계를 독립적으로 최적화하여 최대 효율을 달성할 수 있습니다.

단계를 분리하면 이기종 클러스터 구성도 가능해집니다. 즉, 최적의 비용 대비 성능을 위해 프리필과 디코딩 역할에 서로 다른 GPU 유형을 할당할 수 있습니다. 예를 들어, 컴퓨팅 최적화 GPU를 prompt 프리필에 사용하고 메모리 최적화 GPU를 토큰 생성에 활용하면 동종 구성보다 달러당 더 나은 처리량을 얻을 수 있습니다.

실제 환경에서 최신 GPU는 일반적으로 더 높은 FLOPS와 더 많은 GPU 메모리를 갖습니다. 따라서 프리필과 디코딩 모두에 최신 GPU 세대를 사용하는 것이 유혹적이며 더 흔한 선택입니다. 다만 이질성 활용이 비용 절감의 실행 가능한 옵션임을 인지해야 합니다.

이 장 후반부에서 이종 클러스터 개념을 자세히 살펴보겠습니다. prompt에는 고성능 GPU를, 생성에는 저가 GPU를 활용함으로써 대규모 환경에서 상당한 비용 절감 효과를 얻을 수 있음을 보여드리겠습니다.

요약하면, 분산 처리는 상호 간섭을 제거하고 각 단계를 특화하여 처리할 수 있게 합니다. 예상되는 결과로는 prompt 크기 불일치로 인한 긴 꼬리 현상이 사라져 지연 시간 분포가 더 좁아지고, 지연 시간 제약 하에서 처리량(굿풋)이 향상되며, 전반적인 자원 활용도가 개선되는 것입니다.

자질 도구(예: NVIDIA Nsight Systems)를 사용하여 프리필 및 디코딩 단계의 병목 현상을 식별해야 합니다. 이러한 도구는 다양한 작업자 노드 간 GPU 커널 및 RDMA 전송을 추적할 수 있습니다. 이를 통해 디코딩 커널이 통신을 완전히 중첩하는지 등을 검증하는 데 도움이 됩니다.

다음으로, 분산 처리 클러스터를 최대한 활용하는 방법에 대한 시스템 아키텍처, 통신 및 스케줄링 정책을 포함하여 분산 처리 서비스 시스템을 실제로 구현하는 방법을 논의해 보겠습니다.

분리된 프리필 및 디코드 클러스터 풀

분산 배치 환경에서는 두 개(또는 그 이상)의 작업자 풀을 유지하여, 한 세트의 GPU는 프릴(prefill) prompt 처리에 전념하고 다른 세트는 토큰 생성에 전념하도록 합니다. 이러한 작업자들은 데이터 센터 내 별도의 노드나 랙에 위치할 수 있으며, 상호 연결 속도가 충분히 빠를 경우 별도의 데이터 센터에 위치할 수도 있습니다. (참고: 현실적인 서비스 수준 목표(SLO)를 달성하기 위한 실용적인 설계 선택은 프릴과 디코드를 동일한 데이터 센터 내에 유지하는 것입니다.)

작업자 풀은 네트워크를 통해 통신하여 프리필이 생성한 모델의 KV 캐시를 디코딩을 수행할 GPU로 전달합니다. 스케줄러 또는 라우터가 이 통신을 조정합니다.

모델 가중치가 두 그룹의 GPU 서버에 로드된 구성을 고려해 보십시오. 한 그룹인 프리필 작업자는 prompt를 처리하고 KV 캐시를 계산합니다. 다른 그룹인 디코드 작업자는 프리필 작업자가 생성한 KV 캐시를 사용하여 토큰 생성을 처리합니다.

두 작업자 그룹은 일반적으로 고속 상호 연결(예: NVLink/NVSwitch 및 InfiniBand)과 RDMA를 통한 제로 카피 GPU-to-GPU 전송을 사용하여 통신합니다. 실제로 이러한 전송은 GPUDirect RDMA 또는 UCX를 사용하며, 엔드투엔드 검증을 위해 Nsight Systems에서 CUDA 커널, NVLink 활동, 스토리지 메트릭 및 InfiniBand 스위치 메트릭과 함께 상관관계를 분석할 수 있습니다.

슈퍼칩 기반 NVL 패브릭(예: Grace-Blackwell, Vera-Rubin 등)의 경우 NVIDIA Multi-Node NVLink(MNNVL)를 사용하고, TP 디코딩을 위해 NVLink 우선 콜렉티브를 활성화한 상태로 유지하며, 가능한 경우 AllGather 및 ReduceScatter 콜렉티브에 SHARP를 활성화하십시오.

시스템이 새 요청을 수신할 때 일반적으로 디코드 워커에서 이를 처리합니다. 이를 *디코드 중심 설계라고* 합니다. 프리필 워커는 이미 KV 계산으로 컴퓨팅 리소스가 바운디드되어 있으므로 이 방식이 선호됩니다.

디코드 워커가 클라이언트 I/O, 라우팅 및 세션 상태 관리를 처리하도록 함으로써, 추론 시스템은 프리필 워커의 과부하를 방지합니다. 또한 디코드 노드에 요청 인그레스를 중앙 집중화함으로써 네트워크 관리, 자동 확장 및 정책 적용이 간소화됩니다.

이는 디코드 작업자가 모든 요청의 중앙 집중식 인그레스로 기능하는 프리필-디코드 시스템 아키텍처의 한 유형입니다. 이 아키텍처는 NVIDIA Dynamo 추론 시스템에서 사용됩니다. 또 다른 일반적인 아키텍처는 전용 중앙 집중식 API 라우터를 사용하여 요청을 프리필 또는 디코드 작업자 중 하나로 라우팅하는 방식입니다. 그러나 이 방식은 시스템에 추가적인 이동 부품이 필요하며, 라우터와 프리필/디코드 작업자 간의 추가적인 조정, 그리고 확장성과 지연 시간에 대한 추가적인 고려 사항이 필요합니다.

디코드 워커가 요청을 수신한 후 자체적으로 프리필을 수행할지, 아니면 프리필 워커 풀로 "오프로드"할지 결정합니다. 프리필 워커 풀로 오프로드하기로 결정하면 디코드 워커는 나중에 키-값 결과를 다시 수신한 후 디코딩을 계속하고 다음 토큰을 생성합니다.

다음은 간소화된 NVIDIA Dynamo 클러스터 구성의 일부로, NVIDIA의 Inference Xfer Library(NIXL)를 사용하여 GPUDirect RDMA 기반 KV 캐시 전송을 수행하는 두 가지 역할을 정의합니다. 이를 통해 한 GPU가 네트워크를 통해 다른 GPU의 메모리에 직접 쓰기를 수행할 수 있습니다:

```
roles:
  - name: prefill_worker     # Prefill worker role
    model_path: models/llm-70b
    instance_count: 4        # 4 prefill workers
    gpu_type: B200           # B200 Blackwell compute-bound prefill

  - name: decode_worker      # Decode worker role
    model_path: models/llm-70b
    instance_count: 8        # 8 decode workers
    gpu_type: B300           # B300 Blackwell Ultra high-memory decode
```

프리필 작업자를 위한 또 다른 옵션은 NVIDIA의 루빈 CPX 가속기입니다. 루빈 CPX(CP는 "컨텍스트 처리"를 의미)는 프리필과 같은 컴퓨팅 집약적 워크로드를 위해 특별히 설계되었습니다. 루빈 CPX는 NVIDIA가 "일반 가속 컴퓨팅" GPU에서 벗어나 추론과 같은 광범위한 AI 워크로드 내 특정 단계(예: 프리필)에 최적화된 전용 칩으로 전환한 것을 의미합니다.

이 구성에서는 컴퓨팅 집약적인 프리필 작업에 적합한 B200 GPU를 사용하는 프리필 작업자 4개와, 메모리 집약적인 디코딩 작업에 필요한 높은 HBM 용량을 제공하는 B300 GPU를 사용하는 디코딩 작업자 8개를 배치했습니다. B200과 B300을 혼합 배치함으로써 FLOPS 성능과 HBM 용량 특성을 최적화하면서 비용을 최소화할 수 있습니다. 두 역할 모두 NIXL과 GPUDirect RDMA를 사용하여 KV 캐시 블록을 전송합니다. NIXL은 NVLink 및 RDMA NIC을 통한 GPU 간 데이터 이동을 추상화합니다. 또한 GPUDirect Storage용 커넥터를 제공하여 KV 캐시 페이지를 서로 다른 스토리지 계층에서 읽거나 쓸 수 있게 합니다.

시스템 실행 시 내부적으로 각 디코드 작업자는 GPU 메모리 영역을 등록하여 프리필 작업자가 RDMA를 통해 직접 쓰기할 수 있도록 합니다. 일반적으로 NIXL 디스크립터 같은 메모리 등록 메타데이터는 시작 시 또는 첫 접촉 시 교환됩니다. 이렇게 하면 각 원격 프리필 작업마다 전체 메모리 주소 구조체 대신 작은 식별자만 전송하면 됩니다.

예를 들어, Dynamo는 작업자 검색 및 임대를 위해 etcd를 사용합니다. 작업자는 필요한 메모리 핸들을 라우터 또는 제어 평면에 등록하여 피어들이 필요할 때 디스크립터를 획득할 수 있도록 합니다. 프리필 작업자는 처음 사용할 때 이를 검색합니다. 이렇게 하면 프리필 요청에 대상 KV 버퍼의 ID만 포함될 수 있어 제어 메시지를 가볍게 만듭니다.

또한 NVIDIA Dynamo의 NIXL 구현은 추론 데이터 이동을 위한 고처리량 RDMA 및 스토리지 추상화를 제공하며, NVLink, UCX 기반 패브릭, GPUDirect Storage용 플러그인을 포함합니다. 따라서 프리필 워커는 디코드 GPU 메모리에 KV 블록을 직접 쓸 수 있습니다.

프리필과 디코드가 서로 다른 TP 레이아웃을 사용하는 혼합 병렬 처리 배포 환경에서는 NIXL 읽기 직후 디코드 측에서 레이아웃 변환을 수행해야 합니다. 이렇게 하면 KV 페이지가 디코드 커널이 예상하는 레이아웃과 일치합니다. 이 변환은 네트워크 전송에 비해 지연 시간이 미미하며 재프리필을 방지합니다.

이 아키텍처는 각 단계의 확장성을 분리합니다. 예를 들어, 많은 동시 장문 prompt로 인해 프리필이 처리량 병목 현상이 발생한다면, 프리필 작업자를 추가하여 prompt 처리 용량을 늘릴 수 있습니다.

예를 들어, 많은 사용자가 긴 출력을 생성하여 디코딩이 병목이 되는 경우 디코딩 작업자를 확장하면 됩니다. 디코딩과 프리필이 분리되어 있으므로 한 쪽을 확장해도 다른 쪽에 직접적인 간섭이 발생하지 않습니다.

NVIDIA Dynamo와 같은 시스템은 동적 런타임 구성 가능한 분리를 지원하므로 클러스터를 중지하지 않고도 실시간으로 프리필 작업자를 추가하거나 제거할 수 있습니다. 새로운 프리필 작업자는 단순히 등록 후 큐에서 작업을 가져오기 시작합니다. 프리필 작업자가 어떤 이유로든 클러스터를 이탈할 경우(충돌, 재시작, 자동 확장 이벤트, 네트워크 분할 등), 디코딩 작업자가 일시적으로 더 많은 로컬 프리필을 수행하여 이를 보완합니다.

NVIDIA Dynamo의 분산 런타임은 워커 검색 및 임대 관리에 분산 서비스 계정(`etcd`)을 사용합니다. 플래너(Planner) 구성 요소는 임대를 취소하거나 자동 검색된 새 워커를 시작하여 워커를 확장할 수 있습니다. 이러한 동적 유연성은 부하가 자주 변동하는 초대형 규모 환경에서 매우 중요합니다. 이러한 상황이 발생하면 필요에 따라 워커를 역할 간에 교체해야 합니다.

프리필 작업자 설계

프리필 워커(또는 *prompt 서버*)는 요청의 초기 prompt 프롬프트 프리필 처리 단계를 전담하는 컴퓨팅 노드 입니다. 본 섹션에서는 프리필 노드가 중량급 계산을 효율적으로 처리하기 위한 아키텍처 설계와, 부하 상태에서 KV 캐시 채우기를 위한 지연 시간과 처리량 간의 균형 조정 방식을 논의합니다.

프리필 작업 부하가 계산 집약적이므로 프리필 노드는 높은 FLOPS 성능의 GPU를 사용하고 대규모 행렬 곱셈에 최적화되어야 합니다. 각 프리필 작업은 *n개의* 입력 토큰을 모든 모델 레이어를 통해 처리합니다.

프리필 작업자는 수천 개의 GPU 스레드를 병렬로 사용하며, 가능한 경우 여러 GPU 노드에 걸쳐 작업합니다. 텐서 병렬 처리 및 파이프라인 병렬 처리 등 익숙한 병렬 처리 기법을 활용하여 TTFT(전체 처리 시간)를 단축합니다.

메모리 관리

메모리 측면에서 프리필 노드는 전체 모델 가중치를 로드하고 prompt용 KV 캐시를 할당해야 합니다. 이 KV 캐시는 잠시 후 살펴보겠지만 디코드 작업자들로 전송됩니다.

프리필은 GPU 메모리에 모델 매개변수와 prompt 입력을 사용한 모델 전방 전달의 작업 활성화 값을 채웁니다. KV 캐시가 생성되면 즉시 디코드 작업자 노드로 전송됩니다. KV 캐시는 프리필 노드의 메모리에 오래 유지되지 않습니다.

모델이 매우 크거나 prompt가 매우 긴 경우, 메모리 제한으로 인해 프리필이 텐서 분할 또는 GPU 간 병렬 분할을 필요로 할 수 있습니다. 프리필 서버는 지연 시간 목표를 충족하기 위해 병렬화 전략(데이터, 텐서, 파이프라인, 전문가(MoE), 컨텍스트)에 유연하게 대응해야 합니다.

일부 추론 프레임워크는 프리필 작업 공간으로 사용할 GPU 메모리의 큰 덩어리를 미리 할당합니다. 이는 전체 메모리 조각화와 버퍼 할당 시간을 줄여줍니다.

지연 시간 대 처리량 최적화

분산 프리필 클러스터를 튜닝할 때, 각 개별 prompt의 TTFT(첫 응답 시간) 최소화와 높은 부하 하에서 초당 요청 수(RPS) 극대화 또는 TPOT(전체 응답 시간) 감소 사이의 근본적인 상충 관계에 직면합니다.

분산 시스템은 지연 시간 우선 접근 방식과 처리량 우선 접근 방식에 대한 서로 다른 스케줄링 정책을 지원함으로써 이러한 상충 관계를 처리합니다. 다음으로 각 접근 방식을 설명하겠습니다:

지연 시간 우선 접근법

> TTFT를 줄이려면 프리필 노드는 prompt가 도착하자마자 처리해야 하며, 거의 또는 전혀 배치 처리하지 않아야 합니다. 이 모드에서는 다른 요청이 배치에 채워지기를 기다릴 필요가 없습니다. 따라서 클러스터에 사용 가능한 GPU가 있다고 가정할 때, 모든 prompt는 즉시 실행을 시작하고 가능한 한 빨리 완료됩니다.
>
> 이 지연 시간 우선 접근법의 단점은 작은 배치 또는 배치 없이 사용하기 때문에 GPU 활용도가 낮아진다는 점입니다. 따라서 GPU가 빈번히 유휴 상태가 되며, 주어진 클러스터 규모에서 시스템이 처리할 수 있는 동시 요청 수가 줄어듭니다. 이 경우 프리필 클러스터 용량을 과도하게 프로비저닝하거나, 요청에 대한 엄격한 지연 시간 SLO를 보장하기 위해 배치 크기를 1로 아주 작게 설정할 수 있습니다.

처리량 우선 접근법

> 피크 처리량(RPS)과 최소 TPOT이 우선순위라면, 각 GPU를 완전히 로드하기 위해 prompts를 더 큰 그룹으로 배치해야 합니다. 8~32개의 prompts를 단일 배치로 축적하면 산술 집약도가 높아지고 GPU 컴퓨트 유닛이 지속적으로 작동하게 됩니다. 이는 전체 처리량을 증가시킬 것입니다.
>
> 처리량 우선 접근법의 단점은 각 요청이 배치 수집에 소요되는 시간과 동일한 배치 지연을 발생시킨다는 점입니다. 배치 크기가 클수록 지연 시간도 길어집니다.
>
> 극한의 처리량 추론 시스템 구성에서는 데이터 병렬 처리 또는 파이프라인 병렬 처리를 사용하여 요청당 여러 GPU를 할당할 수 있습니다.

데이터 병렬 처리에서는 전체 모델이 각 GPU에 복제됩니다. 배치는 GPU 간에 미니배치로 분할됩니다. 각 GPU는 자체 모델 복사본을 통해 해당 데이터 하위 집합에 대해 전파(forward pass)를 수행합니다. 이후 모든 GPU의 출력을 집계하여 최종 출력을 생성합니다.

데이터 병렬화는 모든 GPU에 걸쳐 메모리 대역폭과 연산 능력을 집계하여 배치당 성능을 향상시킵니다. 그러나 최대 병렬도는 총 GPU 수 ÷ 요청당 GPU 수로 감소합니다. 이는 전체 동시 요청 처리 능력을 저하시킵니다. 단일 요청당 너무

많은 GPU를 사용하는 경우 자원이 유휴 상태로 남을 수 있습니다. 이는 처리량과 동시성 간 불균형을 초래합니다.

파이프라인 병렬화는 모델의 레이어를 GPU 0과 GPU 1과 같은 서로 다른 GPU에서 순차적으로 수행되는 단계로 분할합니다. GPU 0이 마이크로배치 0에 대한 단계를 완료하는 즉시, 활성화 값을 GPU 1로 전달하고 마이크로배치 1에 대한 1단계를 시작합니다. 이러한 조립 라인 패턴은 모든 GPU가 서로 다른 작업 블록을 처리하도록 유지합니다.

파이프라인 병렬화는 배치당 처리량을 증가시키지만, 마이크로배치 크기나 단계 분할이 신중하게 균형 잡히지 않으면 GPU 간 통신 오버헤드와 파이프라인 "버블"을 추가합니다.

결국, 고정된 크기의 클러스터를 가정할 때 추가로 할당하는 각 GPU는 처리량은 증가시키지만 동시에 처리할 수 있는 요청 수는 감소시킵니다. GPU 클러스터를 확장하는 것은 항상 가능하지만, 클러스터 크기가 고정된 경우 사용 사례에 있어 지연 시간 SLO(서비스 수준 목표)와 처리량 SLO 중 어느 것이 더 중요한지에 따라 구성을 선택해야 합니다.

지연 시간 인식 스케줄링 및 배치 처리

분산 시스템은 앞서 언급한 지연 시간 인식 스케줄링 정책을 통합하여 이러한 요소들을 균형 있게 조정합니다. 예를 들어, 부하가 충분히 높아 소량의 요청을 결합해도 TTFT 목표를 위반하지 않는 경우가 아니면, 단일 요청 실행을 보장하고 요청을 배치하지 않을 수 있습니다.

많은 클러스터 설계는 스케줄러에 SLO 제약 조건을 포함합니다. 예를 들어 p90 TTFT가 $\leq X$ ms여야 한다면, 시스템은 일반적인 prompt 크기에서 SLO를 충족하는 최대 배치 크기 또는 병렬 처리 전략을 선택합니다.

또 다른 전략은 적응형 배치 창입니다. 예를 들어, 부하가 낮을 때는 배치 크기 1로 요청을 즉시 실행할 수 있습니다. 부하가 높아지면 시스템은 2~10ms와 같은 짧은 시간 창 내에 도착하는 요청의 마이크로배치를 허용할 수 있습니다. 이렇게 하면 약간의 지연이 GPU 활용도를 크게 향상시킬 수 있지만, 이는 필요하고 허용 가능한 경우에만 해당됩니다.

많은 추론 엔진은 프리필 작업자의 지연 시간을 우선시합니다. 시스템은 종종 prompt 작업을 가능한 한 빨리 실행하며, GPU 활용도가 다소 낮아지는 것도 감수합니다. 첫 번째 토큰이 빠르게 처리되면 사용자 경험이 크게 개선되기 때문입니다.

평균 부하보다 더 많은 프리필 용량을 프로비저닝하는 것이 일반적입니다. 이렇게 하면 프리필 클러스터가 지연 시간 급증 없이 prompt 급증을 흡수할 수 있습니다. 다음 장에서는 시간이 지남에 따라 프리필 작업자나 디코드 작업자 중 어느 쪽도 병목 현상이 되지 않도록 자원을 실시간으로 재조정하는 적응형 메커니즘에 대해 논의하겠습니다.

쿠버네티스와 같은 현대적 오케스트레이터는 각 계층을 자동으로 확장할 수 있습니다. 예를 들어 프리필 GPU 활용도가 높고 디코딩 활용도가 낮을 경우, 오케스트레이터는 프리필 포드(또는 노드)를 추가하는 자동 확장 이벤트를 트리거할 수 있으며, 심지어 일부 디코딩 포드/노드를 제거할 수도 있습니다.

이러한 적응형 스케일링은 종종 사전 채우기 대기열 길이 같은 지표를 활용해 의사 결정을 유도하는 방식으로 구현됩니다.

또 다른 방법은 우선순위 큐를 구현하여 짧은 prompt는 배치 처리량을 줄인 별도의 고속 레인으로 스케줄링하고, 길고 배치 가능한 prompt는 처리량 최적화 큐로 보내는 것입니다. NVIDIA Dynamo는 스케줄링 시 지연 시간 클래스를 지원합니다. 요청에 태그를 지정하고 클래스별로 다른 배치 처리 창을 설정하여 이를 구현할 수 있습니다.

핵심 요점은 프리필 작업자가 빠른 처리 시간을 우선시한다는 점입니다. 디스어그리게이션을 통해 디코딩이 별도의 작업자 집합에서 실행되므로 디코딩 성능을 저해하지 않고 이를 구현할 수 있습니다. 트래픽이 적은 시간대에는 프리필 GPU 사이클을 일부 "낭비"할 수 있지만, 트래픽이 정점인 시간대에는 낮은 TTFT(처리 시간)를 유지합니다. 이는 대화형 서비스에 있어 가치 있는 절충안입니다.

디코드 워커 설계

디코드 워커(또는 *생성 서버*)는 자동 회귀 디코드 단계에 전념합니다. prompt의 KV 캐시가 준비되면 디코드 워커로 전송되며, 디코드 워커는 KV 캐시를 활용하여 TPOT 지연 시간을 낮게 유지하기 위해 가능한 한 빠르게 나머지 출력 토큰을 생성합니다.

그림 17-3과 같이 요청이 처음 디코드 워커로 라우팅되면, 분산 라우터를 사용하여 프리필을 로컬에서 수행할지 원격에서 수행할지 먼저 결정해야 합니다. 원격 프리필을 선택하면 프리필 요청을 프리필 큐에 푸시하여 프리필 워커가 가져가도록 합니다.

프리필 작업자는 프리필 큐에서 지속적으로 토큰을 가져오고, 프리픽스 캐시에 캐시된 KV 블록을 읽은 후 프리필 작업을 수행합니다. 이후 해당 KV 블록을 디코드 작업자로 다시 쓰면 디코딩이 완료됩니다.

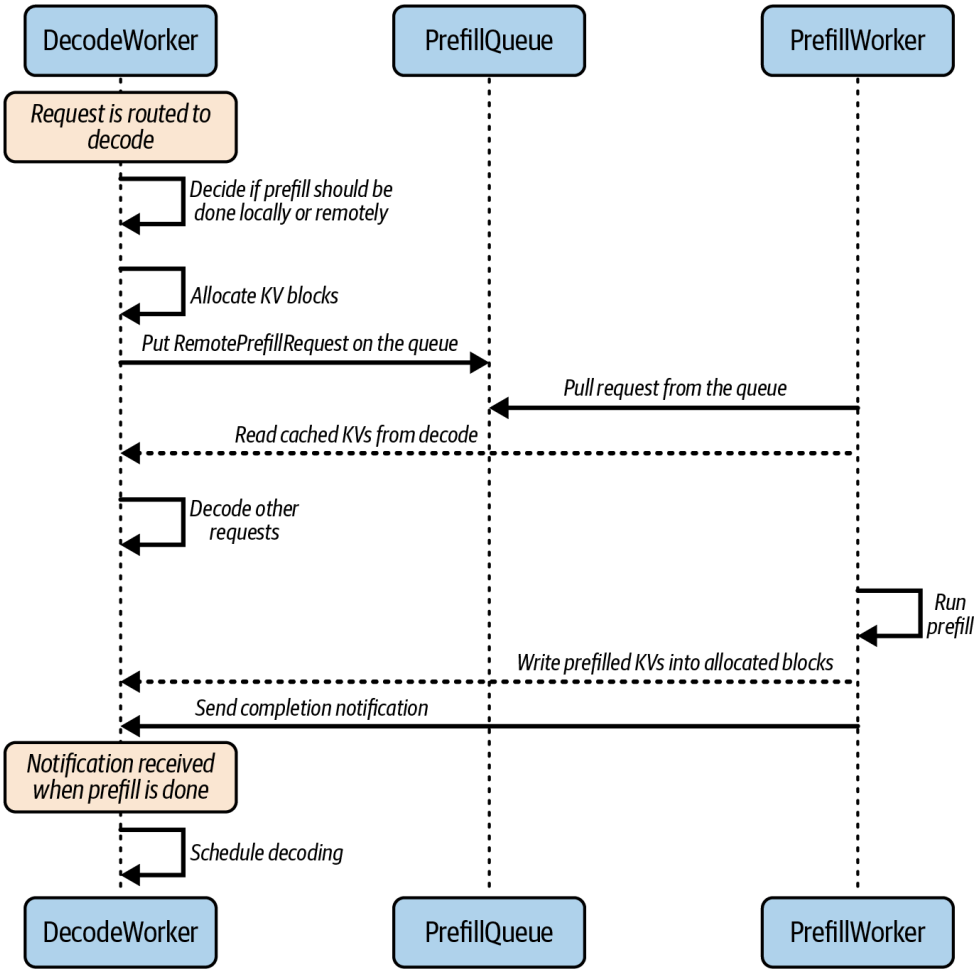

그림 17-3. 프리필 작업자 설계: 프리픽스 캐시 읽기 → 프리필 계산 → 디코딩용 KV 캐시 쓰기

디코드 워커 설계는 다수의 동시 시퀀스 생성을 효율적으로 처리하는 동시에 KV 캐시의 메모리 사용량을 관리하는 데 중점을 둡니다. 본 섹션에서는 디코드 서버가 연속 배치 처리 및 지능적인 메모리 관리 기법 등을 활용해 높은 처리량을 달성하는 방식을 설명합니다. 이러한 기법은 특히 긴 시퀀스의 경우 TPOT 지연 시간을 줄이고 확장성을 높이는 데 기여합니다. 먼저 두 단계 간 KV 캐시 전송 방식을 상세히 살펴보겠습니다.

프리필과 디코딩 간 KV 캐시 전송

고성능 분산 처리를 위해서는 프리필 작업자와 디코드 작업자 간에 KV 캐시 데이터를 최대한 효율적으로 이동시켜야 합니다. NIXL(4장에서 설명)과 같은 라이브러리를 사용하여 GPU 간 직접 전송을 수행하면 CPU 개입을 피하고 비차단 작업을 활용할 수 있습니다. 이렇게 하면 한 GPU가 KV 데이터를 전송하는 동안에도 전송 완료를 기다리지 않고 다른 포워드 패스 요청을 처리할 수 있습니다.

디코드 작업자에 도착한 사용자 요청을 고려해 보자. 이 경우 디코드 작업자의 스케줄러는 필요한 KV 블록을 할당하고 프리필 큐에 원격 프리필 요청을 추가한다. 이 프리필 요청에는 해당 KV 블록들의 식별자가 포함된다. 이 상호작용은 그림 17-4에 표시되어 있다.

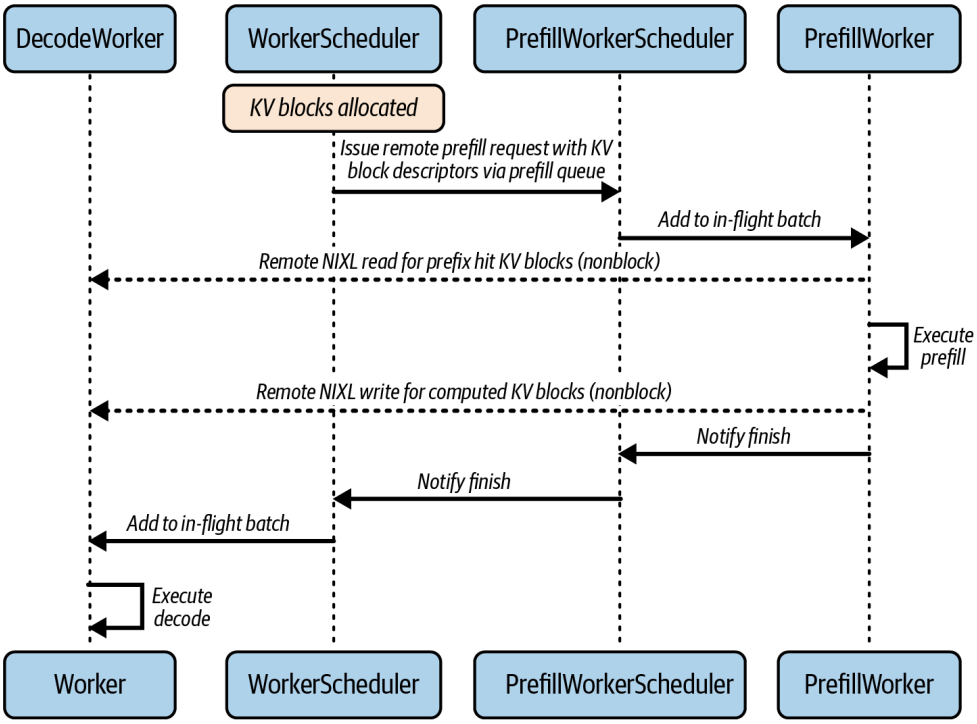

그림 17-4. NIXL을 사용한 프리필 및 디코드 작업자 간 KV 캐시 데이터전송 ; RDMA 전 여러 PagedAttention 블록을 ~128-토큰 페이로드로 병합 (참고: CUDA에서 vLLM은 블록당 16토큰으로 기본 설정됨)

프리필 작업자는 선택된 전송 프로토콜을 통해 NIXL을 사용하여 원격 GPU 메모리에 대한 직접 읽기/쓰기 작업을 수행합니다. 이는 CPU 복사 작업을 피하고 비차단 진행을 가능하게 합니다. 프리필 작업자가 프리필 요청을 완료하는 즉시, 디코딩 작업자의 스케줄러는 해당 디코딩 요청을 자체 디코딩 파이프라인에 추가합니다. 이를 통해 컴퓨팅 작업과 데이터 이동이 원활하게 중첩될 수 있습니다. 재등록 빈도를 최소화하려면 큰 고정 창 크기를 가진 사전 등록된 피어 메모리를 사용하십시오. Nsight Systems 타임라인을 통해 제로 카피 전송 중첩을 확인할 수 있습니다.

종단 간 데이터 이동 및 중첩을 검증할 때는 추적 플래그를 사용하여 Nsight Systems로 자질을 부여하는 것이 좋습니다. InfiniBand 링크 텔레메트리에는 HCA/NIC 카운터용 `--nic-metrics=true` 와 스위치 카운터용 `--ib-switch-metrics-device=<GUIDs>` 를 추가하십시오. 이를 통해 스위치 메트릭을 캡처하고 호스트/장치 활동을 샘플링할 수 있습니다. 이를 통해 CUDA 커널, UCX 활동, 스토리지 메트릭 및 네트워크 동작이 상호 연관된 결과를 얻을 수 있습니다. 다음은 CUDA/UCX 추적을 활성화하고 CPU 활동, GPU 메트릭, 스토리지 메트릭, InfiniBand 스위치 텔레메트리 데이터를 수집하는 통합 명령어입니다:

```
nsys profile --trace=cuda-hw,osrt,nvtx,ucx,gds \
  --trace-fork-before-exec=true \
  --cuda-event-trace=true \
  --cuda-graph-trace=node \
  --cuda-memory-usage=true \
  --sample=cpu \
  --gpu-metrics-device=all \
  --nic-metrics=true \
  --ib-switch-metrics-device=<GUIDs> \
  --storage-metrics --storage-devices=all \
```

```
    --gds-metrics=driver \
    -o nsys_reports/prefill_decode \
    <your_launch_here>
```

프리필 엔진과 디코드 엔진이 서로 다른 병렬 처리(예: 텐서 병렬) 레이아웃을 사용할 수 있습니다. 이 경우 시스템은 수신 측에서 NIXL 읽기 후(데이터 사용 전) 레이아웃 변환 커널을 삽입하여 각 KV 블록을 디코드 작업자가 기대하는 레이아웃에 재정렬할 수 있습니다.

지속적 배치

디코딩 서버는 *반복 수준 배치*(*iteration-level batching*)라고도 하는 연속 배치에 크게 의존합니다. 대규모 행렬 연산을 수행하는 프리필 단계와 달리, 디코딩 단계는 각 새 토큰 생성이 상대적으로 작은 벡터-행렬 연산이기 때문에 많은 소규모 연산을 수행합니다. 개별 토큰이 벡터로 표현되기 때문입니다.

작은 토큰 단위 작업 부하로 인한 낮은 GPU 활용도를 피하기 위해, 디코딩 작업자는 여러 입력 시퀀스를 한 번에 처리하여 각 반복 단계에서 더 큰 행렬 연산(예: 여러 토큰 생성)을 생성할 수 있습니다. 이는 각 디코딩 작업의 연산 집약도를 높입니다.

예를 들어, 32개의 서로 다른 텍스트 생성 요청이 중간 단계에 있으며 다음 토큰을 생성할 준비가 되어 있다고 가정합니다. 32개의 개별적인 단일 토큰 전방 전달을 계산하는 대신, 연속 배치 스케줄러는 요청을 결합하여 32개의 토큰(시퀀스당 하나씩)을 병렬로 생성하는 하나의 전방 전달을 수행합니다.

이렇게 하면 행렬 곱셈의 효과적인 배치 크기가 32가 되어 GPU 연산 유닛을 지속적으로 활용할 수 있습니다. 문제는 모든 시퀀스가 정확히 동시에 토큰을 요청하지 않는다는 점입니다. 일부는 일찍 완료되고 다른 일부는 늦게 시작될 수 있습니다.

서로 다른 요청과 사용자 간에도 배칭이 가능하다는 점을 기억하십시오. 대부분의 현대적 추론 서버는 시퀀스의 컨텍스트 길이가 동일한 경우 서로 다른 사용자의 요청에서 디코딩 단계를 자동으로 그룹화합니다. 이는 실시간으로 배칭 디코딩을 효과적으로 수행하는 것입니다. 디코딩 처리량을 극대화하기 위해 이러한 솔루션을 고려하십시오.

---

vLLM에서 CUDA 그래프 캡처 범위는 `--max-seq-len-to-capture` 플래그로 제어됩니다. 캡처 크기는 일반적으로 최대 시퀀스 수에 맞춰 정렬됩니다. 시퀀스가 이 길이를 초과하면 vLLM은 이거 모드로 전환됩니다. 이 플래그가 런타임 시 배치되는 시퀀스 수를 제어하지 않는다는 점에 유의해야 합니다. 동시 디코드 슬롯 수는 `--max-num-seqs`(반복 내 시퀀스의 바운디드)로 제어됩니다. 예측 가능한 메모리 사용을 위해 이 값을 `--max-num-batched-tokens`.

지속적 배치(Continuous Batching)는 각 단계에서 배치 크기를 동적으로 업데이트하여 시퀀스가 서로 다른 시점에 토큰을 요청하는 문제를 해결합니다. 구체적으로, 지속적 배치 전략은 각 단계에서 해당 시점에 다음 토큰을 요청할 준비가 된 모든 시퀀스를 모아 단일 디코딩 단계로 배치합니다.

디코딩이 진행 중일 때 새 요청이 프리필을 완료하고 준비 상태가 되면, 임의로 긴 대기 시간 없이 다음 배치에 합류합니다. 이는 전체 배치를 모을 때까지 기다린 후 디코딩하는 정적 배치 방식과 대비됩니다.

시퀀스가 텍스트 끝 토큰에 도달하거나 시퀀스 길이 제한에 의해 완료되면, 해당 시퀀스는 후속 배치에서 즉시 제거됩니다. 시퀀스가 아직 준비되지 않은 경우(예: 프리필에서 처리 중인 긴 prompt)에는 준비될 때까지 포함되지 않습니다.

실질적으로 연속 배치에서는 각 반복의 배치 크기가 변동될 수 있습니다. 그러나 서버는 항상 제한 범위 내에서 당시 이용 가능한 시퀀스를 모두 포함하도록 배치 크기를 극대화하려 합니다. 이는 토큰당 대기 시간을 최소화하면서 높은 활용도를 달성합니다.

지속적 배치 처리 방식은 디코딩 GPU 작업자가 절대 유휴 상태에 머물지 않도록 보장합니다. 작업자는 항상 사용 가능한 요청을 처리하며, 개별 시퀀스가 새 토큰을 기다리는 동안에도 GPU를 지속적으로 활용함으로써 지연 시간 제약 하에서 처리량을 극대화합니다.

마찬가지로 Microsoft의 DeepSpeed와 NVIDIA의 TensorRT-LLM 추론 엔진은 디코딩 중 GPU 활용도를 높이기 위해 페이지형 KV 캐시를 활용한 연속 또는 인플라이트 배치 방식을 구현합니다. 구체적으로 DeepSpeed는 여러 생성 요청을 결합하고, TensorRT-LLM은 스케줄러를 사용하여 스트림 간 디코딩 작업을 그룹화합니다.

분산된 디코드 클러스터에서는 연속 배치의 효과가 더욱 강력해집니다. 디코드 GPU는 생성 작업만 처리하므로, 대규모 맞춤형 prompt 작업에 의해 중단되지 않고 이 연속 루프에 100% 사이클을 할당할 수 있습니다. 이는 특히 부하 상태에서 더 안정적인 처리량 지표를 제공합니다.

고부하 상태에서는 디코딩 노드에 수십에서 수백 개의 시퀀스가 동시에 활성화될 수 있습니다. 각 반복마다 대량의 시퀀스를 배치 처리함으로써 하드웨어 활용도를 극대화할 수 있습니다.

반면 저부하 상태에서는 단 하나의 시퀀스만 활성화되어 있어도 디코드 작업자가 즉시 토큰을 생성할 수 있습니다. 배치 채우기를 기다릴 필요가 없습니다. 이 경우 해당 순간 GPU 활용도는 낮아지지만, 단일 시퀀스의 지연 시간은 낮게 유지됩니다. 따라서 연속 배치 방식은 두 극단 상황을 모두 처리합니다: 높은 동시성에서는 효율적이고, 낮은 동시성에서는 응답성이 뛰어납니다. 이는 높은 처리량과 낮은 지연 시간의 좋은 균형점입니다.

가변 길이 시퀀스 그룹화

LLM 추론에서 가변 길이 시퀀스를 처리하려면 계산 및 메모리 낭비를 방지하기 위한 세심한 스케줄링과 배치 작업이 필요합니다. 한 배치에 짧은 prompt와 긴 prompt를 혼합하면 패딩 오버헤드가 발생합니다. 토큰 수 대비 최대 50%에 달하는 패딩이 필요할 때도 있습니다. 이는 제한된 GPU 및 네트워크 자원을 낭비합니다.

길이가 다른 prompts를 함께 배치할 경우, 모든 짧은 시퀀스는 가장 긴 시퀀스와 일치하도록 패딩 처리되어야 합니다. 이 패딩은 GPU 사이클, 메모리 대역폭, GPU 간 또는 네트워크 전송을 여전히 소모하는 "무작위" 토큰을 도입합니다. 일부 경우 패딩은 일반적인 생성형 AI 워크로드에서 전체 토큰의 최대 절반을 차지할 수 있습니다. 이는 추론 효율성을 크게 저하시킵니다.

간단한 해결책은 시퀀스 길이에 따라 요청을 버킷으로 그룹화하는 것입니다. 이렇게 하면 각 배치에 유사한 크기의 시퀀스가 포함됩니다. 0–512 토큰, 513–1,024 토큰 등과 같은 정적 길이 버킷을 사용하면 배치 경계를 고정하고 패딩 오버헤드를 최소화할 수 있습니다.

vLLM의 디코드 스케줄러는 회전 풀 형태로 다중 작업 처리(`SequenceGroup`) 인스턴스를 유지합니다(각 prompt는 다중 작업 처리 디코드 작업(`SequenceGroup`)입니다). 스케줄러는 디코드 반복당 고정된 토큰 예산이 소진될 때마다 각 그룹을 진행시킵니다. 다중 작업 처리 디코드 작업(`SequenceGroup`)이 해당 청크 처리를 완료하면 풀에서 이탈하고, 새로운 다중 작업 처리 디코드 작업(`SequenceGroup`)이 풀에 합류합니다. 이를 통해 정적 패딩 버킷에 의존하거나 GPU를 저활용하지 않으면서도 파이프라인이 지속적으로 작업으로 가득 차게 유지됩니다.

이러한 배치 및 스케줄링 기법은 프리필과 디코딩 클러스터를 분리하는 분산형 프리필-디코딩 배포 환경과 잘 부합합니다. 이 배포 구성에서는 별도의 디코딩 노드가 연속 배치 같은 기법을 활용해 엄격한 SLO 하에서 TPOT 편차를 최소화할 수 있습니다. 한편 전용 프리필 노드는 입력 처리 처리량을 극대화하고 TPOT을 최소화하도록 독립적으로 튜닝할 수 있습니다.

NVIDIA의 프로그램 방식 종속 런치(PDL) 및 장치 시작 CUDA 그래프 런치(12장에서 논의)는 토큰당 런치 오버헤드를 줄이고 작업을 중첩하며 디코드 반복 간 버블을 제거하는 데 사용됩니다. 이러한 기능은 일반적으로 애플리케이션 코드에서 수동으로 설정하기보다 프레임워크를 통해 활성화됩니다.

장치에서 런칭된 그래프를 사용할 때는 ` `cudaGraphInstantiateFlagDeviceLaunch` `로 인스턴스화하고 노드를 단일 장치에 유지하십시오. PDL을 사용하여 단계(예: 디코드 반복)의 끝에서 종속 커널을 중첩 실행하십시오. 이는 토큰당 런칭 버블을 추가로 줄여줍니다.

길이 버킷팅, 연속 배치, 분산 처리, PDL, 장치 시작 CUDA 그래프를 결합함으로써 vLLM, SGLang, NVIDIA Dynamo와 같은 현대적 추론 시스템은 prompt 길

이가 극단적으로 달라지는 경우에도 높은 처리량과 낮은 지연 시간을 동시에 달성할 수 있습니다. 또한 이는 자원 효율성이나 확장성에 영향을 주지 않습니다.

---

vLLM에서 `--max-seq-len-to-capture` 는 CUDA 그래프가 커버하는 최대 시퀀스 길이를 제어합니다. 기본값은 8192입니다. 연속 배치 처리 시 vLLM은 가장 가까운 캡처 크기로 패딩할 수 있으므로 패딩 낭비를 최소화하려면 `--max-num-seqs` 와 `--max-num-batched-tokens` 를 정렬하십시오. CUDA 그래프는 일반적인 시퀀스 길이에 대한 반복적인 CUDA 그래프 재구축을 최소화하는 데 도움이 됩니다. 이는 런타임 배치 동작을 직접적으로 결정하지 않습니다. vLLM의 런타임 배치는 앞서 설명한 바와 같이 디코드 스케줄러의 동적 `SequenceGroup` 풀에 의해 관리됩니다. 프로덕션 환경에서는 HBM(KV) 사용량을 바운디드하고 연속 배치 시 패딩을 줄이기 위해 `--max-seq-len-to-capture` 과 함께 `--max-num-seqs` 및 `--max-num-batched-tokens` 을 조정하는 것이 권장됩니다.

KV 캐시의 메모리 관리

디코딩은 지금까지 처리된 전체 시퀀스(이전에 디코딩된 토큰 포함)에 대한 어텐션을 필요로 하므로, KV 캐시 메모리는 디코딩 작업자에게 핵심 자원입니다. 각 시퀀스는 트랜스포머 레이어별 키-값 텐서와 과거 토큰별 키-값 텐서를 저장합니다. 그림 17-5는 서로 다른 요청 간에 공유되는 KV 캐시 예시를 보여줍니다.

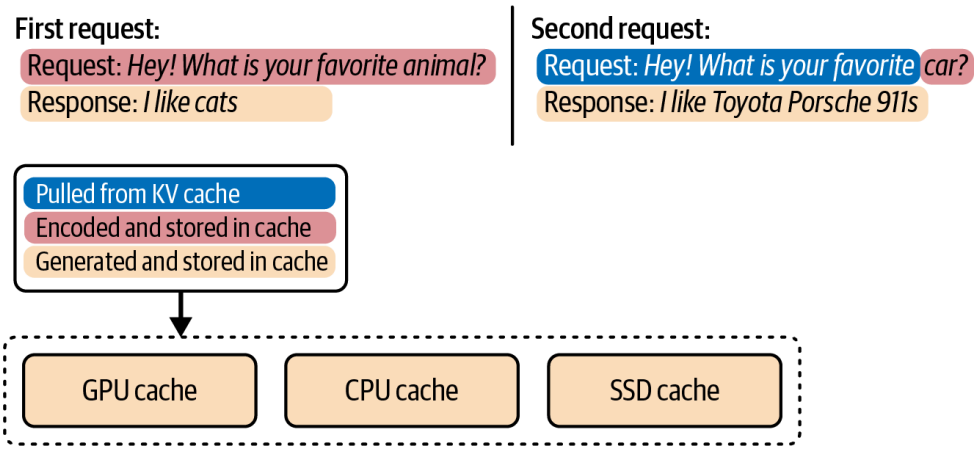

그림 17-5. 요청 간 KV 캐시 데이터관리 및 재사용

대규모 모델과 긴 시퀀스의 경우, KV 메모리는 토큰 수에 선형적으로 증가하며 어텐션 레이아웃(attention layout)과 레이어당 토큰 수(`dtype`)에 따라 달라집니다. 실용적인 추정치는 `bytes_per_token = 2 x n_layers x n_kv_heads x head_dim x bytes_per_element` 입니다. (참고: 레이어당 토큰 수(`2 x`)는 레이어당 토큰별 키와 값을 모두 포함합니다.)

40개 레이어, 40개 어텐션 헤드(헤드 차원 128), FP16을 사용하는 Llama급 13B 모델을 고려해 보겠습니다. 표준 다중 헤드 어텐션(MHA)을 사용하는 4,096토큰 컨텍스트의 경우 KV 크기는 ~0.819 MB/토큰 또는 총 ~3.36 GB입니다. FP8 키-값을 사용하면 약 1.68GB가 됩니다.

8개의 쿼리 그룹을 사용하는 그룹화 쿼리 어텐션(GQA)(`n_kv_heads = 8`)의 경우, 4,096토큰 KV는 FP16에서 약 0.671GB, FP8에서 약 0.336GB입니다. 또한

1개의 KV 헤드를 사용하는 다중 쿼리 어텐션(MQA)의 경우 FP16에서 약 0.084GB입니다.

모델의 실제 픽셀 정밀도(`n_layers`), 픽셀 이동 정밀도(`n_kv_heads`), 픽셀 이동 정밀도(`head_dim`), 키-값(KV) 정밀도를 항상 사용하여 계산해야 합니다. FP8 및 FP4는 키-값 정밀도(`bytes_per_element`)를 변경하기 때문입니다.

GPU가 다수의 동시 시퀀스를 처리할 경우, 모델 가중치 외에도 순전히 KV 캐시 저장 공간으로 인해 메모리 한계에 빠르게 도달할 수 있습니다. 따라서 다수의 긴 시퀀스가 처리 중인 경우 GPU 메모리 고갈을 방지하기 위해 디코드 작업자가 최적화된 메모리 할당기를 사용하도록 해야 합니다. 다음은 디코드 작업자가 KV 메모리를 효율적으로 관리하기 위해 사용하는 전략입니다:

페이지형 GPU 메모리 할당기

vLLM의 PagedAttention 메커니즘 이 대표적인 예시입니다. KV 캐시를 고정 크기 페이지로 분할하고 비활성 페이지를 CPU 메모리로 스왑할 수 있습니다. vLLM 외에도 SGLang과 NVIDIA TensorRT-LLM에 페이지형 KV 메모리 관리자가 구현되어 있습니다. NVIDIA Dynamo는 이러한 기술을 기반으로 합니다. 해당 시스템들은 DRAM과 NVMe를 활용한 외부 KV 계층도 추가로 구성합니다. 또한 대역폭 균형을 위해 재계산과 I/O를 스케줄링할 수 있습니다. 이는 LMCache 및 기타 유사한 라이브러리나 런타임 프로젝트에서 흔히 볼 수 있는 방식입니다.

고메모리 GPU 및 사용자 정의 할당기

디코드 서버는 종종 대용량 HBM을 탑재한 GPU(예: Blackwell B200의 180GB HBM, Blackwell B300의 288GB HBM)를 사용하여 KV 캐시를 저장합니다. 또한 vLLM 및 NVIDIA TensorRT-LLM과 같은 시스템은 최적화된 메모리 관리자를 사용하여 KV 메모리를 고정 크기 페이지로 할당함으로써 조각화를 줄이고, 요청 간 효율적인 접두사 재사용을 가능하게 하며, 길이가 다양한 수백 개의 시퀀스를 관리합니다. 이는 과도한 조각화와 낭비 없이 메모리를 효율적으로 공유합니다.

KV 캐시 오프로딩(예: 페이징 아웃)

GPU 메모리가 가득 차면 디코드 작업자는 오래된 KV 블록을 CPU RAM이나 NVMe 같은 콜더 스토리지로 오프로드할 수 있습니다. 예를 들어 시퀀스가 1,000개의 토큰을 생성하지만 현재 모두 즉시 필요하지 않은 경우, 일부 초기 토큰의 KV를 호스트 CPU 메모리로 이동시킬 수 있습니다.

필요할 때 요청에 따라 GPU 메모리로 다시 불러올 수 있습니다. 토큰이 어텐션에 필요할 때 오프로딩은 약간의 지연 페널티를 유발하므로 오프로딩 사용 시 주의해야 합니다. 디코드 서버는 KV 데이터 페이징이 생성에 미치는 영향을 최소화하기 위해 데이터 전송을 미리 가져오거나 중첩시키려 합니다.

컨텍스트 제한 및 압축

일부 배포 환경에서는 디코딩된 토큰 수나 출력 시퀀스 길이에 하드 리미트를 설정합니다. 이는 애플리케이션 수준의 절충안으로, KV 캐시 크기를 제한하고 무제한 확장을 방지합니다. KV 압축을 통해 토큰당 필요한 KV 메모리도 줄일 수 있습니다. 예를 들어, KV를 낮은 정밀도(FP16, FP8, INT8)로 저장하면 메모리 사용량을 크게 절감할 수 있습니다.

또 다른 예로는 멀티쿼리 어텐션(MQA) 사용이 있습니다. 여기서 헤드는 토큰당 하나의 KV 벡터를 공유합니다. 이는 헤드 수에 비례하여 KV 크기를 줄입니다. 이는 모델 아키텍처 변경으로 KV 사용량을 직접 감소시킵니다. 그룹화 쿼리 어텐션(GQA)과 DeepSeek의 다중 잠재 어텐션(MLA)도 KV 캐시 크기 축소에 도움이 됩니다.

분리된 메모리 계층 구조

분리형 설계의 또 다른 장점은 디코드 클러스터의 GPU 메모리가 모델 가중치 + KV 캐시 저장 전용으로 완전히 할당된다는 점입니다. 또한 대규모 prompt 프리필 계산을 처리하려 하지 않는데, 이는 모놀리식 서비스 시스템에서 프리필 중 일시적으로 많은 추가 메모리를 소모하게 됩니다.

각 디코드 GPU는 일반적으로 모델 병렬 처리를 사용하지 않는 한 전체 모델 가중치를 로드한 후 남은 메모리를 KV 저장소로 사용합니다. 예를 들어, 모델 가중치가 GPU 메모리의 70GB를 차지하고 GPU 총 용량이 180GB라면 약 122GB가 KV에 할당됩니다.

이는 해당 GPU에서 동시에 처리 가능한 토큰 × 시퀀스 수에 직접적인 영향을 미칩니다. 분산 처리 자체가 KV 메모리 문제를 해결하지는 않지만, 역할을 분리함으로써 메모리 용량과 메모리 대역폭을 최적화하는 디코드 노드 유형을 선택할 수 있습니다.

클러스터를 이렇게 구성한 후에는 프리필 작업을 프리필 작업자 풀로 오프로드할 시점, 또는 디코드 작업자에서 로컬로 처리할 시점을 결정해야 합니다. 오프로드에는 대기열 지연, 네트워크 전송 등 오버헤드가 발생하므로 실제로 지연 시간 개선에 도움이 될 때만 사용해야 합니다. 이 결정은 다음에 설명할 라우팅 정책에 의해 이루어집니다.

분리된 라우팅 및 스케줄링 정책

모든 요청이 를 통해 프리필 작업자로 오프로드될 필요는 없습니다. 사실 불필요한 경우 오버헤드만 증가시키고 큰 이점은 없습니다. 따라서 분산 추론 시스템은 라우팅 정책을 사용하여 조건부로 분산 처리하고, 도움이 될 가능성이 있을 때만 원격 프리필 경로를 사용합니다. 표 17-1은 KV 인식 및 접두사 인식 라우팅을 포함한 라우팅 전략의 개요를 보여줍니다.

표 17-1. 접두사 인식 및 KV 인식 라우팅을 포함한상위 수준 라우팅 전략

라우팅 전략	설명
라운드 로빈	각 노드에 대한 경로를 하나씩 순차적으로
최소 요청	활성 요청이 가장 적은 작업자로 라우팅
접두사 인식	요청의 접두사를 사용하여 작업자 선택
KV 인식	요청과 가장 잘 일치하는 KV 캐시를 가진 작업자로 라우팅

분리형 라우터는 요청을 처음 수신하는 디코드 작업자에서 각 새 요청마다 실행됩니다. 그리고 로컬 프리필을 수행할지, 아니면 프리필 작업자 풀에서 원격 프리필을 수행할지 신속하게 결정합니다.

라우팅 요소

프리필 작업자 풀( )로 프리필 작업을 오프로드할지 여부는 요청 및 시스템 상태와 관련된 여러 요소에 따라 결정될 수 있습니다. 일반적인 라우팅 요소로는 현재 대기열 길이, GPU 메모리 가용성, 특정 GPU가 특정 모델이나 prompt 유형에 더 적합하다는 점을 고려한 전문화 등이 있습니다.

vLLM의 KV 캐시 인식 라우터와 같은 고급 라우터는 캐시 로컬리티도 고려합니다. 이들은 요청의 일부 접두사가 이미 캐시에 존재하는 디코드 작업자로 요청을 라우팅합니다. 그림 17-6은 작업자들이 발행한 KV 캐시 이벤트로부터 수신된 데이터를 기반으로 시스템 내에서 요청을 이동시키는 KV 캐시 인식 라우터의 예시를 보여줍니다.

목표는 캐시 적중률을 극대화하고 부하를 균형 있게 분산시키는 방식으로 라우팅하는 것입니다. 표 17-2는 일반적인 분산 설계에서 라우팅 결정에 영향을 미치는 주요 요소들을 요약합니다.

표 17-2. 라우터가 프리필 오프로드 결정을 내리는 데 영향을 미치는요소

요소	설명	라우팅 결정에 미치는 영향
prompt 길이	입력 prompt의 토큰 수(모든 접두사 캐싱 후).	긴 prompt ⇒ 더 많은 컴퓨팅 → 길이가 임계값을 초과하면 프리필을 오프로드. 짧은 prompt ⇒ 로컬에서 수행.
접두사 캐시 적중률	디코드 작업자의 KV 캐시에 이미 존재하는 prompt의 범위(이전 요청에서).	대형 접두사 캐시 적중(prompt 대부분 캐시됨) ⇒ 프리필이 효과적으로 짧아지고 메모리 바운디드가 더 커짐 → 로컬에서 수행. 캐시 적중 없음(모든 토큰이 새 토큰) ⇒ 컴퓨팅 부하가 크므로 → 오프로드하는 것이 유리할 가능성이 높음.
프리필 대기열 길이	글로벌 프리필 큐에 대기 중인 작업 수(프리필 작업자의 부하 정도).	큐가 길 경우(프리필 작업자 지연) ⇒ 새 요청 오프로딩 피하기(로컬에서 수행). 큐가 비어 있거나 적을 경우 ⇒ 프리필 작업자 여유 있음 → 다른 조건 충족 시 오프로딩.
디코딩 작업자 부하	로컬 디코드 작업자의 현재 부하(진행 중인 디코드 작업 등).	디코딩 GPU가 많은 디코딩 스트림으로 바쁜 경우, 오프로딩은 병렬 처리에 도움이 됩니다(GPU의 무거운 연산 부담을 덜어줌). 디코딩이 대부분 유휴 상태이고 프리필 큐가 백업된 경우, 사용 가능한 용량을 활용하기 위해 로컬 프리필을 수행합니다.
지연 시간 SLO 긴급도	요청의 지연 시간 요구 사항의 우선 순위 또는 엄격함.	긴급한 저지연 요구 사항 ⇒ prompt가 최대한 빨리 계산되도록 오프로드할 수 있음(특히 로컬 디코드가 바쁜 경우). 완화된 요구 사항은 리소스 절약을 위해 로컬에서 실행될 수 있음(“QoS 및 조기 거부 정책” 참조).

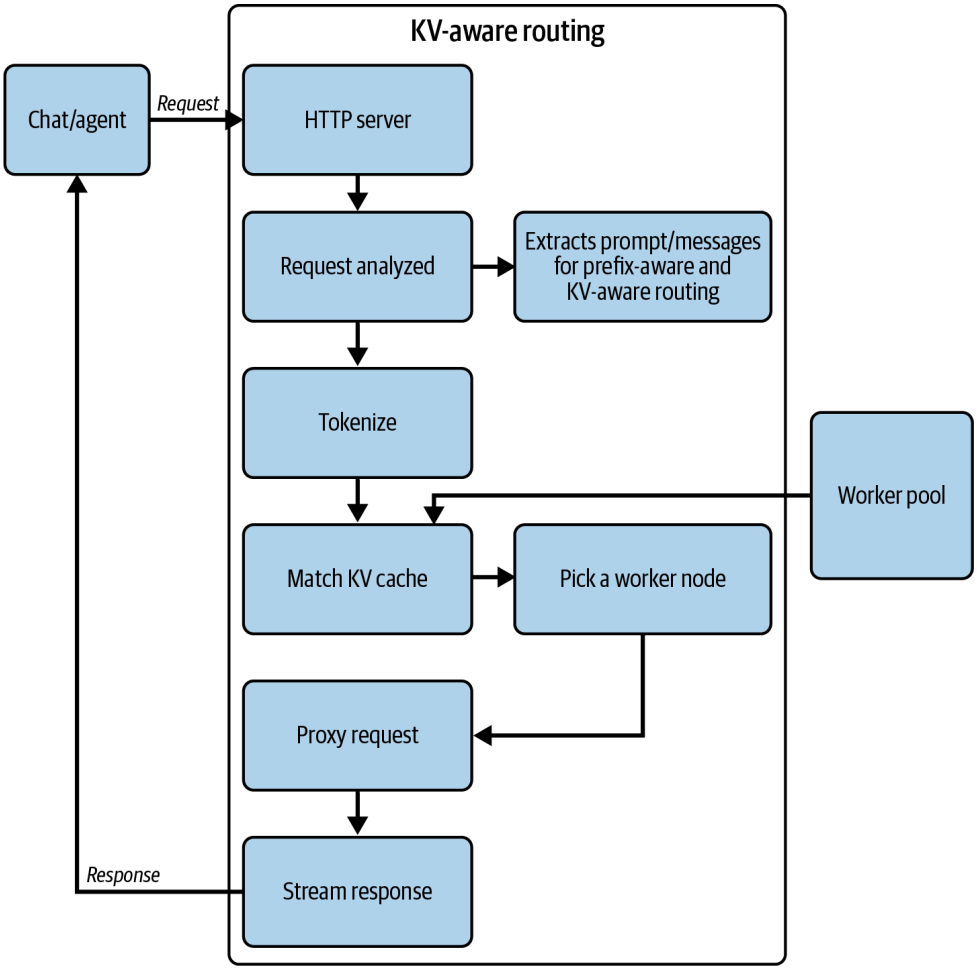

그림 17-6. 작업자가 발행한 KV 캐시 이벤트에서 수신한 데이터를 기반으로 한KV 캐시 인식 요청 라우팅

이러한 요소들은 원격 실행으로 이점을 얻을 수 있는 요청만 오프로드하는 것을 선호합니다. 여기에는 길고 연산 집약적인 prompt가 포함됩니다. 한편, 짧고 캐시 히팅이 발생하는 prompt는 오버헤드를 최소화하기 위해 로컬에서 처리됩니다. 임계값은 조정 가능합니다. 표 17-2의 각 요소는 다음과 같이 설명된 특정 절충점을 다룹니다:

prompt 길이

사소한 prompt를 오프로드하는 데 시간을 낭비해서는 안 됩니다. 예를 들어 5토큰 prompt의 경우, 디코드 GPU가 자체적으로 빠르게 처리할 수 있으므로 원격 실행의 오버헤드(비록 작더라도)를 감수할 가치가 없습니다. 오프로드는 디코드 GPU가 장시간 묶여 있을 수 있는 무거운 prompt에 한정됩니다.

접두사 캐시

현대적 추론 시스템은 대개 이전에 처리된 컨텍스트(예: 대화의 이전 턴이나 반복되는 시스템 prompt)에 대한 KV 쌍을 저장할 수 있는 KV 캐시를 구현합니다. 새 요청의 prompt가 이미 처리된 prompt와 중복되는 긴 접두사를 가질 경우, 디코드 작업자는 해당 접두사의 KV 데이터를 이미 메모리에 보유할 수 있습니다. 이 경우 캐시 미스 부분에 해당하는 prompt의 나머지 부분만 계산하면 됩니다.

남은 부분이 짧으면 오프로딩의 이점이 줄어듭니다. 또한 전체 prompt가 이미 캐시된 경우, 프리필 계산 자체가 필요하지 않습니다. 디코딩은 캐시

된 상태를 즉시 사용해 진행할 수 있습니다. 라우터는 이를 고려하여 효과적인 prompt 길이를 (`prompt_length` – `prefix_cached_length`)로 간주합니다.

큰 접두사 히트는 필요한 연산량을 줄일 뿐만 아니라, 많은 KV 데이터를 프리필 작업자(prefill worker)로 전송했다가 다시 가져와야 함을 의미하며 이는 무의미합니다. 따라서 이러한 요청은 로컬에 유지되며 캐시를 활용합니다.

프리필 큐 길이

이는 본질적으로 프리필 워커 클러스터의 부하를 측정합니다. 프리필 워커가 대기 중인 작업으로 과부하 상태라면, 추가 작업을 전송하는 것은 요청이 대기열에 머물게 되어 TTFT(전체 처리 시간)에 도움이 되기보다 해가 될 수 있습니다. 이러한 경우 디코드 워커는 일시적으로 더 많은 작업을 직접 수행하도록 지시받습니다.

이는 프리필 클러스터에 자연스러운 부하 분산 메커니즘을 제공합니다. 전용 프리필 계층이 포화 상태일 때 시스템이 로컬 컴퓨팅으로 우아하게 회귀하기 때문입니다. 프리필 대기열이 다시 짧아지고 작업자가 작업을 처리하면 긴 prompt에 대한 오프로딩이 재개됩니다. 이러한 동적 균형은 디스어그리게이션이 다양한 조건에서 우수한 성능을 발휘하는 이유 중 하나입니다.

디코딩 작업자 부하

명시적으로 그렇게 코딩되지는 않더라도 라우터는 본질적으로 작업 부하 분배를 돕습니다. 디코딩 작업자가 이미 여러 스트림 디코딩으로 바쁘더라도 새로 들어오는 prompt는 여전히 오프로드할 수 있습니다. 이는 바람직한데, 그렇지 않으면 해당 GPU가 무거운 컴퓨팅과 디코딩을 동시에 처리해야 하므로 양쪽 모두 속도가 느려질 수 있기 때문입니다.

반대로 시스템 부하가 매우 낮아 디코딩 GPU가 여유로운 경우, 로컬에서 더 긴 프리필을 처리할 수 있습니다. 실제로 유휴 상태의 디코딩 GPU는 전체 부하가 낮아 프리필 대기열도 비어 있을 가능성이 높습니다. 이러한 조건은 이미 간접적으로 해당 사례를 포함합니다. 다만 일부 구현에서는 로컬 GPU 활용도를 직접 확인하여 프리필 오프로드 필요 여부를 결정할 수도 있습니다.

지연 시간 SLO 및 우선순위

혼합된 SLA 또는 우선순위 클래스가 존재하는 시스템에서는 QoS 향상을 위해 라우팅을 수정할 수 있습니다. 가장 빠른 TTFT가 필수인 고우선순위 요청의 경우, 큐 확인을 우회하고 즉시 오프로드하여 디코딩 GPU가 비어 있더라도 즉시 계산을 시작하도록 할 수 있습니다. 이 경우 시스템은 해당 디코딩 작업자를 다른 고우선순위 디코딩 작업에 할당할 수 있습니다.

또는 요청의 우선순위가 낮을 경우, 시스템은 프리필 클러스터 리소스를 전혀 사용하지 않을 수도 있습니다. 대신 디코드 작업자에서 로컬로 처리

되도록 하거나 심지어 지연시킬 수도 있습니다. 우선순위 요청 처리에 대해서는 "QoS 및 조기 거부 정책"에서 다시 살펴보겠지만, 기본 라우터가 이러한 유형의 고려 사항으로 확장될 수 있다는 점만 알아두시기 바랍니다.

라우팅 정책에 사용되는 구체적인 임계값과 가중치는 경험적으로 결정해야 합니다. 예를 들어, 특정 유형의 GPU(예: Blackwell B200)에서는 50 토큰 미만의 prompt가 로컬에서 더 빠르게 계산되는 반면, 더 큰 prompt는 오프로드에서 이점을 얻을 수 있습니다.

신규 하드웨어 출시 시, 최신 GPU는 더 높은 FLOPS와 메모리 대역폭을 갖기 때문에 임계값이 변경될 수 있습니다. 예를 들어, 단일 B200이 디코드 작업자에서 더 많은 토큰을 완전히 처리할 수 있으므로, 오프로드에 대한 손익분기점 prompt 길이가 다소 높아질 수 있습니다.

하드웨어 업그레이드 시 컴퓨팅 FLOPS 및 메모리 대역폭 개선에 따라 `PREFILL_LENGTH_THRESHOLD` 과 같은 임계값을 경험적으로 조정해야 합니다.

효과적인 라우팅의 결과는 시스템이 양쪽 장점을 모두 확보한다는 점입니다. 짧은 prompt는 로컬 실행으로 인해 TTFT(총 처리 시간)가 더 빠르며 추가 오버헤드도 발생하지 않습니다. 긴 prompt는 프리필과 디코딩이 서로 다른 GPU에서 병렬로 수행되므로 병렬 처리의 이점을 얻습니다. 또한 프리필 작업자 풀은 필요할 때만 활용되며, 처리 속도를 따라잡지 못할 때 과부하가 걸리는 상황을 피합니다.

이러한 조건부 전략은 DistServe 프로토타입과 vLLM, Dynamo 같은 현대적 추론 서버에서 핵심적이었다. 이를 통해 순수 처리량(raw throughput)이 아닌 지연 시간 제약 하에서 유용한 처리량(goodput)을 향상시킬 수 있다.

코드 내 동적 라우팅 정책 예시

실제 적용 시 라우팅 정책은 간단한 조건부 검사로 구현될 수 있습니다. 다음 코드는 이 라우팅 논리의 단순화된 버전을 보여줍니다:

```
# Offload prefill decision running on a decode worker
# (B200/B300 tuned)
def should_offload_prefill(prompt_length: int,
                           prefix_cached_length: int,
                           prefill_queue_size: int,
                           decode_active_reqs: int,
                           ttft_slo_ms: int = 500) -> bool:
    # Effective prefill after prefix KV hits
    eff_len = max(0, prompt_length - prefix_cached_length)

    # Tunables (kept in config; shown here for clarity)
    PREFILL_LENGTH_THRESHOLD = 256  # see split_policy.prompt_length_thresh
```

```
    PREFILL_QUEUE_MAX         = 10    # see autoscale/prefill queue-length gu
    DECODE_LOAD_THRESHOLD     = 8     # active decode streams

    long_prefill = (eff_len >= PREFILL_LENGTH_THRESHOLD)
    prefill_available = (prefill_queue_size < PREFILL_QUEUE_MAX)

    # Prefer remote when prefill is compute-heavy
    # and the pool has capacity.
    if long_prefill and prefill_available:
        return True

    # If local decode is busy and prefill is moderately long,
    # free decode via offload.
    if decode_active_reqs >= DECODE_LOAD_THRESHOLD and eff_len >= 64:
        return True

    # Otherwise keep prefill local
    # (lower overhead / better cache locality).
    return False
```

이 의사 코드에서 `PREFILL_LENGTH_THRESHOLD`는 시스템 조정 매개변수(예: 50 또는 100 토큰)로, "긴" prompt를 정의합니다. `PREFILL_QUEUE_MAX`는 프리필 작업자 풀이 포화 상태로 간주되는 임계값으로, 특히 처리 중인 작업이 너무 많을 때 적용됩니다.

디코드 작업자는 새 요청을 수신하자마자 `should_offload_prefill()`를 호출합니다. 함수가 `True`를 반환하면 디코드 작업자는 prompt를 메시지로 패키징하여 글로벌 프리필 작업 큐에 푸시합니다. 이후 KV 캐시 결과가 반환될 때까지 대기하며 다른 작업을 수행합니다.

`should_offload_prefill()`가 `False`를 반환하면 디코딩 작업자는 즉시 자체적으로 프리필 계산을 수행합니다. 이렇게 하면 프리필 작업자가 지연되기 시작할 경우 새 요청이 로컬 계산으로 전환되어 대기열 지연을 피할 수 있습니다. 이는 디코딩 풀과 프리필 풀 간의 부하를 균형 있게 조정하는 적응형 라우팅 방식입니다.

동적 라우팅 정책 구성 예시

실제 배포 환경에서는 라우팅 정책을 하드코딩하지 말고 파일이나 UI를 통해 로구성해야 합니다. 예를 들어 NVIDIA의 Dynamo는 JSON 또는 YAML 구성에서 복잡한 라우팅 및 자동 확장 규칙을 지정할 수 있습니다. 다음은 일부 정책 로직을 캡슐화한 Dynamo Planner JSON의 단순화된 예시입니다:

```
model: ...
split_policy:
  prompt_length_threshold: 256
  prefix_cache_weight: 10.0
  queue_length_weight: 1.5
  decode_load_weight: 0.5
```

```
    enable_hotspot_prevention: true
  cache:
    reuse_prefix: true
    min_cache_hit_ratio: 0.75
  autoscale:
    prefill:
      min_replicas: 4
      max_replicas: 12
      scale_up:   { queue_length: 8,  gpu_utilization: 80 }
      scale_down: { queue_length: 2,  gpu_utilization: 40 }
    decode:
      min_replicas: 8
      max_replicas: 24
      scale_up:   { queue_length: 16, kv_cache_usage: 75 }
      scale_down: { queue_length: 4,  kv_cache_usage: 30 }
  qos:
    enable_early_rejection: true
    low_priority_threshold_ms: 500
    reject_on_slo_violation: true
```

여기서 구성은 256개의 토큰으로 구성된 `prompt_length_threshold` 을 가진 `split_policy` 를 정의합니다. 또한 캐시 히트, 큐 길이, 디코딩 부하 등의 요소에 대한 가중치를 지정합니다. 프리필 및 디코딩 역할 모두에 대한 `autoscale` 동작도 구성하며, 여기에는 큐 길이, GPU 사용률, KV 캐시 사용량에 기반한 확장/축소 방식이 포함됩니다.

또한 조기 거부(early rejection)와 같은 QoS 규칙을 적용하고 요청을 "느린" 것으로 간주할 임계값을 조정할 수 있습니다. 실제로 Dynamo 라우터는 시작 시 이 JSON을 읽거나 동적으로 가져와 전체 클러스터에 분산된 요청에 대한 각 라우팅 결정을 지시합니다.

용량 인식 라우팅

이전 섹션에서 언급한 바와 같이, NVIDIA Dynamo( )는 동적 라우팅 정책을 지원합니다. 이 동적 라우팅 기능의 구현체는 Dynamo GPU Planner입니다. 플래너는 TTFT, TPOT 및 KV 캐시 전송 예상 비용과 같은 메트릭을 사용하여 라우팅을 수정하거나, 심지어 단계별 GPU를 재할당/확장하여 병목 현상을 줄이고 워크로드 변화에 적응할지 결정합니다. 이를 통해 시스템은 그림 17-7과 같이 수요가 급증하는 상황에서도 높은 성능을 유지할 수 있습니다.

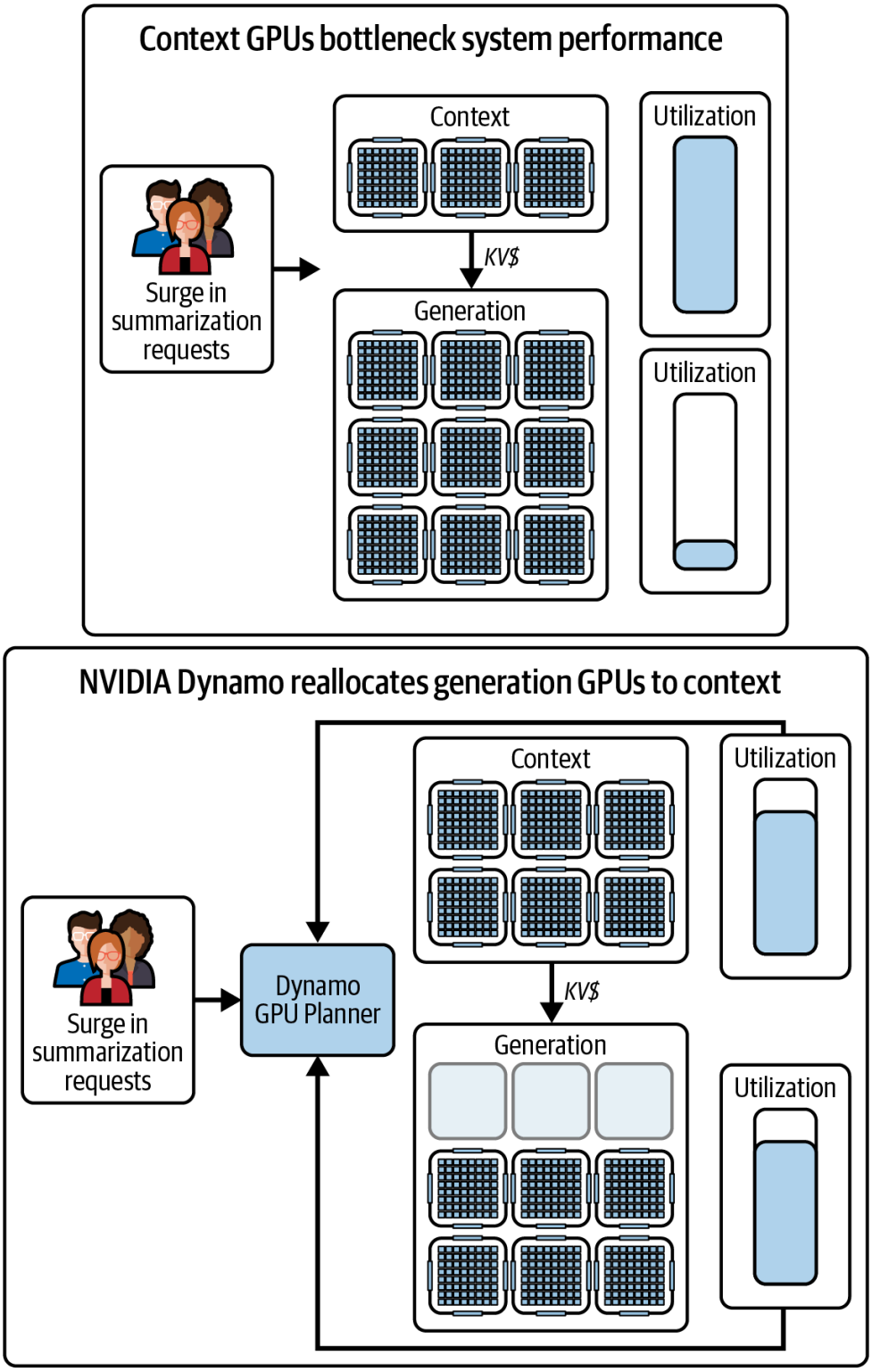

그림 17-7. NVIDIA의 Dynamo GPU 플래너는 GPU 활용도 지표를 기반으로 수신 요청 처리 방식과 프리필 및 디코딩 작업자 할당을 결정합니다

여기서 Dynamo의 플래너는 디코드(작은 요약 출력)에 비해 프리필(컨텍스트)에 많은 양의 프리필(큰 입력)이 필요한 대규모 요약 prompt가 유입됨에 따라 더 많은 GPU를 프리필(컨텍스트) 단계로 전환하기로 선택합니다.

반대로 추론 요청이 급증할 경우, 플래너는 입력 토큰 수 대비 많은 출력 토큰을 생성하는 추론 요청 특성상 디코드 단계로 GPU를 재배정할 수 있습니다. 다른 경우에는 프리필과 디코드가 동일한 GPU 작업자 노드에서 수행되는 전통적인 모놀리식 방식으로 요청을 처리하도록 선택할 수도 있습니다.

요약하면, NVIDIA의 Dynamo Planner와 같은 구성 요소는 실시간 메트릭을 지속적으로 모니터링하고 이를 TTFT 및 TPOT 지연 시간과 같은 애플리케이션 수준 SLO와 비교함으로써 이러한 의사 결정 과정을 자동화할 수 있습니다. 이러한

정보를 활용하여 Dynamo Planner는 각 프리필/디코드 단계에 할당되는 GPU 리소스의 양을 늘리거나 줄이는 방식으로, 완전한 분산 처리, 분산 처리 없음 또는 그 중간 단계로 요청을 처리할지 동적으로 결정할 수 있습니다. 그 결과 생성된 적응형 시스템은 프리필 및 디코딩 워크로드 전반에 걸쳐 리소스 활용도를 최적화하며, 공격적인 성능 목표를 달성합니다.

지연 시간 인식 라우팅

추론 라우터는 단순한 임계값 규칙( )을 넘어 더 정교한 지연 시간 점수 모델을 사용하여 각 요청에 가장 적합한 작업자를 선택할 수 있습니다. 예를 들어, 작업자의 현재 부하 상태, 사용 중인 메모리 양, 관련 캐시 보유 여부 등과 같은 실시간 메트릭을 기반으로 각 잠재적 작업자에 대한 지연 시간 비용을 지속적으로 계산할 수 있습니다. 그런 다음 지연 시간 비용이 가장 낮은 작업자에게 요청을 전송하는 식입니다.

가장 낮은 지연 비용을 가진 작업자가 가장 여유롭고 선호되는 작업자라고 가정해 보겠습니다. 간단한 지연 비용 함수는 다음과 같을 수 있습니다:

```
# Lower cost is preferable
latency_cost = 0.7 * (occupancy_percent)
  + 0.3 * (active_req_count)
```

이 특정 지연 비용 함수는 GPU 점유율을 더 중시합니다. 이는 GPU가 현재 수행 중인 작업량과 상관관계가 있습니다. 부차적으로, 해당 엔진에서 현재 처리 중인 요청 수와도 연관됩니다. 새로운 요청은 지연 비용이 가장 낮은 엔진으로 전송됩니다.

이 예시에서 0.7과 0.3으로 설정된 가중치는 경험적 데이터에 기반해 조정될 수 있습니다. 예를 들어, KV 메모리 사용량이 많은 데이터 스왑이나 높은 메모리 대역폭 사용으로 인한 속도 저하의 주요 예측 변수로 확인된다면, 해당 가중치를 더 높게 설정하는 것이 바람직합니다.

보다 정교한 라우팅 정책에는 추가 요소를 포함할 수 있습니다. 예를 들어, 지연 시간 비용 함수에 다음 요소들을 통합할 수 있습니다:

정확한 캐시 매치 가용성

작업자가 이미 필요한 접두사 키-값 쌍(KV)을 캐시에 보유한 경우(접두사 히트) 요청을 훨씬 빠르게 처리할 수 있습니다. 이 경우 시스템은 지연 시간 비용을 줄이기 위해 큰 음수 가중치를 할당하여 해당 작업자를 선호할 수 있습니다. 이 시나리오에서는 값이 낮을수록 유리하기 때문입니다.

KV 점유율

메모리 사용량이 높을수록 GPU가 더 바쁘다는 의미입니다. 이 경우 시스템은 양의 가중치를 할당하여 지연 시간 비용을 증가시키고 라우터가 이

작업자를 피하도록 유도합니다. 이 시나리오에서는 낮을수록 더 바람직하기 때문입니다.

활성 요청

병렬 요청이 많을수록 잠재적인 컨텍스트 스위칭 오버헤드가 발생하므로 지연 시간 비용이 증가하여 해당 작업자를 피하게 됩니다.

메모리 대역폭 활용도

예를 들어, GPU가 많은 긴 시퀀스를 처리하며 메모리 대역폭을 많이 사용하고 있다면, 작업을 추가하면 속도가 느려질 뿐입니다. 이는 지연 비용을 증가시켜 시스템이 이 작업자를 선택하지 않도록 합니다.

최근 키-값 사용량

최근에 작업자 엔진에서 동일한 접두사가 사용되었고 캐시가 "따뜻한" 상태라면, 키-값 쌍이 여전히 L2 캐시에 있거나 프리필 작업자로부터 전송 중인 경우가 많아 성능이 향상될 가능성이 높습니다. 이 경우, 최근에 접두사를 확인했기 때문에 지연 비용을 줄이고 이 작업자를 선호하도록 작은 음수 가중치를 포함할 수 있습니다.

표 17-3은 이러한 고급 라우팅 정책 구성을 요약합니다. 일부 요소와 예시 비용을 나열합니다.

표 17-3. 라우팅 요소 및 상대적 비용

요소	해석	지연 비용에 미치는 영향
KV 점유율 (%) (`occupancy_percent`)	높을수록 메모리 압박 증가	+3
활성 요청 수 (`active_req_count`)	비행 중인 요청이 많을수록 = 잠재적 대기열 발생	+1
KV 캐시 일치 (`cache_match_flag`)	엔진이 필요한 접두사 KV를 이미 보유함	–10 (큰 음수)
메모리 대역폭 % (`mem_bw_percent`)	높음 = 메모리 버스가 바쁨	+0.5
최근 KV 사용량 (`recent_prefix_flag`)	이 엔진에서 최근 접두사가 사용됨	–1

이러한 추가 지표를 활용하여 지연 시간 비용 계산을 수정해 보겠습니다. 이제 라우터는 다음과 같이 복합 지연 시간 비용을 계산합니다:

```
# Lower cost is preferable
def latency_cost(occupancy_percent: float,
                 active_reqs: int,
                 cache_match_flag: bool,
                 mem_bw_percent: float,
                 recent_prefix_flag: bool) -> float:
    return (
        3.0 * occupancy_percent
      + 1.0 * active_reqs
      - 10.0 * int(cache_match_flag)
      + 0.5 * mem_bw_percent
      - 1.0 * int(recent_prefix_flag)
    )
```

프리필 및 디코딩 클러스터는 약간 다른 공식을 사용할 수도 있습니다. 프리필의 경우 캐시 히트(예: 미리 계산된 접두사)를 고려하는 것이 더 가치 있을 수 있는 반면, 디코딩의 경우 메모리 대역폭이나 사용 가능한 KV 공간이 결정에 더 큰 영향을 미칠 수 있습니다.

라우터는 각 워커로부터 텔레메트리 데이터를 수신해야 합니다. 여기에는 대기 큐 길이, KV 캐시 사용량, 메모리 활용도 등과 같은 메트릭이 포함되어 지연 시간 비용을 지속적으로 업데이트할 수 있습니다.

결과적으로 시스템은 지연 시간에 가장 적은 영향을 미치는 곳으로 트래픽을 동적으로 라우팅할 수 있습니다. 저부하 상태에서는 모든 작업자의 점수가 낮아 어느 작업자를 선택하든 상관없을 수 있습니다. 모든 작업자가 요청을 빠르게 처리할 수 있기 때문입니다.

그러나 고부하 상태에서는 라우터가 가장 여유로운 워커에게 새 작업을 보내 효과적인 부하 분산을 수행합니다. 이는 데이터 지역성(캐시 히트)을 활용해 작업량을 줄이는 효과도 있습니다. 다른 엔진이 여유로울 때 바쁜 엔진 뒤에 작업이 쌓일 가능성이 낮아져 평균 및 꼬리 지연 시간이 모두 감소합니다.

앞서 캐시 적중률을 고려하면 라우터가 관련 데이터를 캐시한 상태라면 약간 더 바쁜 서버를 선택할 수 있음을 살펴보았습니다. 이는 해당 요청에 대한 서비스 속도 향상을 가져옵니다. 점수 함수는 이러한 상충 관계를 정량적으로 포착합니다.

NVIDIA Dynamo는 이와 유사하지만 반대 방향(높을수록 좋음)의 계산을 수행합니다. 구체적으로 원격 프리필 점수를 계산하며, 점수가 임계값을 초과하면 프리필 작업자 풀로 작업을 오프로드합니다. 다음은 계산된 점수가 구성 가능한 '`remote_prefill_min_score`' 값을 초과할 경우 조건부 분산을 사용하도록 시스템을 구성하는 예시 YAML 스니펫입니다:

```
disaggregated_router:
  enable: true
  policies:
    - metric: prefill_length   # Prompt length after prefix cache hit
      threshold: 256
```

```
      action: prefer_remote    # if prompt > 256 tokens, offload prefill
      weight: 0.7
    - metric: prefill_queue_depth  # requests queued for prefill workers
      threshold: 10
      action: prefer_local     # if queue > 10, lean toward prefill locally
      weight: 0.3
  remote_prefill_min_score: 0.5  # overall threshold decide remote prefill
```

여기서 라우터는 prompt 길이(`prefill_length`)와 prompt 토큰 수(`prefill_queue_depth`)를 기반으로 점수를 계산합니다. 이 경우 prompt가 256 토큰보다 길면 자동 분할(`prefill_remote`)을 선택하고 프리필 작업을 오프로드합니다. 점수 계산에서 이 부분의 가중치는 여기서 설정된 대로 0.7입니다.

`prefill_local` 그러나 사전 채우기 대기열이 매우 깊어 여기에서 설정된 대로 대기 작업이 10개 이상인 경우, 여기에서 설정된 대로 가중치 0.3으로 다중 경로 추론(경합)에 투표합니다. 결합된 점수가 0.5 `remote_prefill_min_score` 보다 크면, 이 경우 0.5보다 크면, Dynamo는 사전 채우기를 오프로드합니다. 그렇지 않으면 사전 채우기를 로컬에 유지하고 오프로드하지 않습니다.

다중 경로 추론(레이싱)

다중 경로 추론(Multipath inference)은 동일한 요청을 두 개의 서로 다른 모델 크기나 두 개의 서로 다른 경로로 보내는 방식으로, 높은 신뢰성을 위해 사용됩니다. 이는 본질적으로 가장 빠른 결과를 얻기 위한 경쟁(racing)이며, 흔히 *레이싱(racing)*이라고 불립니다. Google과 Meta의 프로덕션 시스템은 꼬리 지연 시간(tail latency)을 줄이기 위해 모델 간 레이싱을 사용하는 것으로 알려져 있습니다. 비용은 많이 들지만 효과적인 기술입니다.

멀티패스 추론을 직접 구현할 경우, 요청이 멱등성을 가지며 두 경로가 모두 실행되어도 문제가 발생하지 않도록 해야 합니다. 또한 GPU 사이클을 절약하기 위해 느린 경로는 prompt에 취소해야 합니다.

멀티브랜치, 작업자 간 병렬 추측적 디코딩

1 5장에서 논의한 바와 같이, 일부 고급 벡터 인퍼런스 서버는 추측적 디코딩을 지원합니다. 추측적 디코딩은 여러 토큰 분기를 병렬로 생성한 후 가장 이상적이지 않은 분기를 제거한다는 점을 기억하십시오.

분리(disaggregation)의 직접적인 부분은 아니지만, 추측적 디코딩은 라우터의 의사 결정 과정 위에 계층화될 수 있습니다. 예를 들어, 시스템은 불확실한 추측적 디코딩 분기를 감지하고 여러 추측적 요청을 서로 다른 디코드 작업자에게 병렬로 보낼 수 있습니다. 이는 여러 추측적 다음 토큰 분기를 생성하고 토큰 분기

생성 시 예측 불가능성을 가릴 것입니다. 이는 예측 불가능한 분기가 발생하는 세대에 대해 추가 컴퓨팅을 대가로 더 낮은 지연 시간을 제공합니다.

구현 시 라우터는 이러한 추측적 노력을 조정하고 결과를 병합합니다. 클러스터 과부하를 방지하기 위해 리소스 사용량이 바운디드해야 하지만, 추가적이고 잠재적으로 낭비될 수 있는 컴퓨팅 및 메모리 대역폭 클러스터 리소스를 희생하는 대가로 초저지연 인식 추론을 위한 잠재적 최적화 방안으로서 다중 경로 추론은 주목할 가치가 있습니다.

요약하면, 지연 시간 인식 라우터는 현재 부하와 캐시 재사용과 같은 잠재적 가속 효과를 모두 고려하여 각 요청이 최적의 위치로 전송되도록 보장합니다. 이는 분산 서비스 시스템의 "두뇌" 역할을 하며, 각 GPU의 하위 수준 스케줄링과 협력하여 작동합니다. 글로벌 스케일링 및 KV 캐싱 전략과 함께, 이는 굿풋을 극대화하고 지연 시간을 최소화하기 위한 포괄적인 접근 방식을 형성합니다.

QoS 및 조기 거부 정책

QoS 정책은 지연 시간 목표를 충족시키기 위해 특정 요청에 우선순위를 부여하거나 속도를 조절합니다. 조기 거부(*입장 제어라고도 함*) 는 시스템이 포화 상태일 때 우선순위가 낮은 쿼리를 거부할 수 있습니다. 이를 통해 리소스와 다른 요청에 대한 지연 시간과 같은 SLO를 보존합니다.

현대 시스템은 *대기 시간 임계값*을 기준으로 이를 수행합니다. 예를 들어, 요청이 대기열에 *X밀리초* 이상 머문 경우 SLO를 심각하게 위반하게 두기보다는 거부하거나 하위 계층 서비스로 오프로드하는 것이 더 나을 수 있습니다. 이러한 종류의 QoS 가드는 미션 크리티컬 추론 시스템에서 점점 더 보편화되고 있습니다.

초대규모 추론 시스템에서는 고부하 상태에서도 꼬리 지연 시간 보장을 유지하기 위해 QoS 메커니즘이 필요합니다. 최적화된 분산 클러스터 구성이라 하더라도 부하가 용량을 초과하면 요청이 대기열에 쌓이기 시작하고 지연 시간이 증가합니다.

모든 요청에 대해 지연 시간 SLO가 위반되는 것을 허용하기보다는, 잘 설계된 시스템은 우아하게 부하를 줄이는 것을 선호합니다. 특히 낮은 우선순위 요청의 경우, 이를 처리하면 다른 요청의 지연 시간 보장이 깨질 것이 분명할 때 일부 요청을 거부하거나 연기함으로써 이를 수행할 수 있습니다.

이는 웹 서버가 극심한 부하 시 HTTP 503 "오버로드" 오류를 반환하는 방식과 유사합니다. LLM 서비스의 경우, 요청을 제시간에 처리할 수 없을 때 선제적으로 거부하거나 다운샘플링할 수 있습니다. 프리필/디코딩 분산 환경에서 QoS를 구성하는 몇 가지 요소는 다음과 같습니다:

지연 시간 SLO 추적

시스템은 목표 TTFT(첫 토큰 도착 시간) 및 TPOT(전체 처리 시간) 목표를 인지해야 합니다. 예를 들어 첫 토큰은 99% 확률로 200ms 이내에 반환되어야 합니다. 내부 텔레메트리 데이터를 활용하면 각 디코드 작업자는 현

재 대기열 길이 등을 기반으로 즉시 수락될 경우 신규 요청의 현재 TTFT를 추정할 수 있습니다. 마찬가지로 다른 디코드 스트림이 추가될 경우 TPOT도 추정할 수 있습니다.

접수 제어(조기 거부)

요청이 완전히 수락되고 할당되기 전에, 시스템은 이 요청을 허용하면 시스템 과부하가 발생하거나 SLO를 위반하는지 확인할 수 있습니다. 해당될 경우, "서버가 바쁘니 나중에 다시 시도하세요" 유형의 응답으로 즉시 요청을 거부할 수 있습니다. 이를 *조기 거절이라고* 합니다. 실제로는 시스템 부하의 전체적인 관점이나 단순히 단일 노드의 휴리스틱을 활용하여 조기 거절을 트리거할 수 있습니다.

예를 들어 OpenAI의 공개 API는 대기열이 너무 높을 경우 오류를 반환합니다. 이는 지연 시간 약속을 위반하는 대신 수행됩니다. 일부 제공업체는 피크 부하 시 최대 생성 길이를 동적으로 낮춥니다. 이는 효과적으로 답변 품질과 지연 시간 사이의 절충을 의미합니다.

우선순위 지정

모든 요청이 동등한 것은 아닙니다. 예를 들어 유료 고객이나 핵심 서비스에서 오는 요청은 우선순위가 높을 수 있습니다. 무료 계층 사용자나 백그라운드 작업에서 오는 요청은 우선순위가 낮을 수 있습니다. 시스템은 스케줄링 결정에 우선순위를 반영할 수 있습니다. 예를 들어 디코드 작업자의 스케줄러는 우선순위가 높은 디코드 작업을 먼저 처리할 수 있습니다. 또는 프리필 작업 큐를 우선순위에 따라 정렬할 수 있습니다. 시스템이 바빠지면 우선순위가 낮은 작업은 더 오래 대기하거나 우선순위가 높은 작업을 위해 빠르게 실패 처리될 수 있습니다.

우아한 성능 저하

시스템이 과부하 상태에 접어들면 요청을 완전히 거부하기보다는 중요도가 낮은 요청에 대한 서비스를 저하할 수 있습니다. 예를 들어 중요도가 낮은 요청에는 더 작은 모델을 사용하거나 prompt를 잘라내거나 출력 토큰 수를 제한할 수 있습니다. 간단한 접근법으로는 부하가 높을 때 하위 계층 요청에 대해 허용되는 최대 prompt 길이 또는 출력 길이를 일시적으로 줄이는 것입니다.

분리 환경에서 우아한 성능 저하의 흥미로운 형태는 시스템에 부하가 걸렸을 때 우선순위가 낮은 요청에 대한 원격 프리필을 일시적으로 비활성화하는 것입니다. 원격 프리필은 지연 시간 최적화를 위해 추가 클러스터 리소스를 사용하므로, 우선순위가 낮은 쿼리는 로컬 프리필만 수행하고 디코드 작업자 리소스만 사용하도록 결정할 수 있습니다. 이로 인해 해당 요청에 대한 응답 속도는 느려지지만, 프리필 작업자를 우선순위가 높은 쿼리를 위해 확보할 수 있습니다.

예를 들어, 추론 서버는 각 요청에 우선순위 태그를 지정하고, 프리필 작업자가 고우선순위 작업으로 바쁠 때 라우터가 저우선순위 요청에 대해 `should_offload_prefill` 로직을 무시하도록 할 수 있습니다.

다음은 초기 거부 정책의 예시입니다. 이는 LLM 게이트웨이 상류에서 요청자를 처리하기 전에 각 디코드 작업자에서 실행될 수 있습니다:

```
# Early rejection based on estimated latency and priority
from dataclasses import dataclass

class QoSController:
    def __init__(self, ttft_slo_ms: int = 500):
        self.ttft_slo_ms = ttft_slo_ms

    def admit_request(self, priority: str) -> bool:
        # The queue length and per-request averages are fed by the metrics
        est_ttft = (get_current_prefill_queue_length()
                    * get_avg_prefill_time_per_req()
                    + get_current_decode_queue_length()
                    * get_avg_decode_time_per_req())

        if est_ttft > self.ttft_slo_ms and priority.lower() == "low":
            # reject low priority request to protect SLOs
            return False
        return True
```

여기서 TTFT는 각 작업이 지연을 유발하므로 대기 중인 프리필 작업 수와 앞으로 처리될 디코드 작업 수를 고려하여 추정합니다. 디코드 작업은 일반적으로 중첩되므로 이는 대략적인 추정치임을 유의하십시오. 추정된 TTFT가 허용 최대값(SLO)을 초과하면 저우선순위 요청은 거부하고 `should_offload_prefill` 함수에서 False를 반환합니다.

고우선순위 요청의 경우, 여전히 이를 수락하며 필요한 경우 다른 대기 중인 작업을 일부 희생할 수 있습니다. 보다 정교한 접근법은 새로운 고우선순위 요청을 위해 공간을 마련하기 위해 이미 대기 중인 저우선순위 요청을 선점하는 것입니다. 이는 종종 우선순위 수준별로 별도의 대기열을 유지함으로써 구현됩니다.

좀 더 자세히 살펴보면, 앞서 예시에서 `avg_prefill_time_per_req()` 및 `avg_decode_time_per_req()` 함수는 요청별 사전 채우기(prefill) 및 디코딩(decode) 소요 시간의 관측값에 대해 지수 이동 평균을 사용하여 실시간으로 값을 계산합니다. 이 값들은 프롬프트 토큰 수(사전 채우기) 및 생성된 토큰 수(디코딩)로 정규화됩니다. 또한 입력 길이가 다양한 경우, 엔진은 각 요청의 실제 토큰 수에 토큰당 평균값을 곱하여 외삽합니다.

`get_current_prefill_queue_length()` 및 `get_current_decode_queue_length()` 함수는 스케줄러 내부 큐에서 추적하는 보류 중인 프리필 및 디코드 작업 수를 가져옵니다. 이 값들은 스케줄러가 유지 관리합니다.

이러한 토큰별 시간 추정은 스케줄러 루프를 통해 실시간으로 갱신됩니다. 기본적으로 스케줄러 루프는 `/metrics` 엔드포인트를 사용하여 매초마다 업데이트됩니다. 이를 통해 동적인 워크로드 변화를 포착합니다.

조기 거부 및 우선순위 지정은 시스템이 포화 상태에 가까워질 때 붕괴되지 않고 통제된 방식으로 실패하도록 보장합니다. 가장 중요한 요청을 가진 사용자는 제공된 SLO 내에서 계속 서비스를 받습니다. 한편, 덜 중요한 트래픽은 제거됩니다.

많은 프로덕션 배포 환경에서 이를 구현하려면 LLM 게이트웨이 계층과의 협조가 필요합니다. 구체적으로 게이트웨이는 특정 오류 코드를 반환하거나 서버 과부하를 나타내는 코드를 클라이언트에 반환할 수 있습니다. 핵심은 시스템이 과도한 요청을 받아들여 모든 사용자의 SLO를 놓치는 대신, 수락한 작업에 대한 지연 시간 약속을 충족할 수 있는 상태를 유지하는 것입니다.

또 다른 QoS 고려 사항은 적응형 생성 제한입니다. 디코딩 단계가 지나치게 오래 실행되어 TPOT SLO 위반을 초래할 위험이 있을 경우, 시스템은 생성을 조기에 중단할 수 있습니다. 예를 들어, 사용자가 1,000개의 토큰을 요청했지만 시스템에 부하가 걸린 경우, 시스템은 200개의 토큰만 생성하도록 허용한 후 중단할 수 있습니다. 이렇게 하면 다른 요청을 위한 자원을 확보할 수 있습니다. QoS 정책은 지연 시간 목표를 충족하기 위해 특정 요청에 우선순위를 부여하거나 속도를 조절합니다. 시스템이 포화 상태일 때 조기 거부는 우선순위가 낮은 쿼리를 거부하여 다른 사용자의 SLA를 유지할 수 있습니다.

요약하면, 분산 아키텍처는 내재적 간섭을 줄이지만 고정 용량은 여전히 부하 급증에 압도될 수 있습니다. QoS 메커니즘은 지연 시간 약속을 보장함으로써 분산 아키텍처를 보완합니다. 조기 거부는 나머지 요청의 지연 시간을 보호하기 위해 일부 작업을 거부하거나 우선순위를 낮춥니다.

초대규모 시스템을 구축할 때 이러한 정책은 핵심 성능 최적화만큼 중요합니다. 이를 적용하지 않으면 요청 홍수가 거대한 대기열과 속도 저하를 유발하여 성능 개선 시도를 모두 무너뜨릴 수 있습니다.

분산 프리필 및 디코드의 확장성

확장성의 또 다른 측면은 노드 추가 시 디스어그리게이트 프리필( ) 성능이 어떻게 유지되는가입니다. 설계상 디스어그리게이트 방식은 여러 측면에서 상대적으로 선형적으로 확장됩니다. 더 많은 prompt 처리량을 처리하려면 프리필 노드를 추가할 수 있습니다. 더 많은 토큰 생성 처리량을 처리하려면 디코드 노드를 추가할 수 있습니다.

주요 과제는 이 둘의 균형을 맞추는 것입니다. 규모가 커질수록 적응형 스케줄러의 중요성은 더욱 커집니다. 대규모 시스템에서는 트래픽 패턴 변화와 같은 불균형 발생 가능성이 훨씬 높기 때문입니다. 동적 비율 구성을 통해 클러스터는 프리필과 디코드의 비율을 실시간으로 재조정할 수 있습니다.

또 다른 고려 사항은 다중 모델 서비스입니다. 디스어그리게이션을 통해 디코딩 특성이 유사한 모델 간에 디코딩 용량을 공유할 수 있습니다. 예를 들어 모델 A와 모델 B가 동일한 클러스터에 호스팅되는 경우, 일부 디코딩 워커를 두 모델 유형

모두 처리하도록 할당할 수 있습니다. 이는 동일한 기본 모델 아키텍처를 재사용하는 서로 다른 LoRA 어댑터를 호스팅할 때 특히 유용합니다.

이를 통해 초유연한 다중 테넌트, 다중 모델, 프리필-디코드 분산 추론 서버 시스템을 구축할 수 있습니다. 본 문서 범위를 벗어나지만, 분산 처리의 모듈성이 이러한 유연성을 제공한다는 점을 인지하시기 바랍니다. 각 모델별 프리필 프론트엔드를 유지하면서 다중 모델에 공통 디코드 풀을 할당할 수 있습니다.

마지막으로 초대형 시스템의 테일 레이턴시를 분석해 보겠습니다. 초대규모 추론 시스템이 수천, 수백만 개의 GPU 노드로 확장될수록 테일 레이턴시 제어는 점점 더 어려워집니다. 서버 수가 증가할수록 단일 서버 장애 발생 확률이 훨씬 높아지기 때문입니다.

분리 방식이 여기서도 도움이 됩니다. 프리필 작업을 한 노드에, 디코딩 작업을 다른 노드에 분리하면 한쪽의 느린 노드가 반대편 작업에 극심한 영향을 미치지 않습니다.

이 장에서 논의한 조기 거부, 2단계 스케줄링, KV 캐시 재사용 같은 기법은 스트래글러를 완화하는 데 도움이 됩니다. 예를 들어 요청 컨텍스트의 상당 부분을 캐싱하면, 하나의 느린 작업이 전체 생성 속도를 크게 늦추지 않습니다. 이미 많은 계산이 완료되었기 때문입니다.

요약하면, 분산 시스템은 적절히 구성 및 튜닝될 경우 대규모 환경에서 더 높은 내구성을 보입니다. 이러한 시스템은 동시 요청이 수백만에서 수십억으로 증가해도 일관된 지연 시간을 유지할 수 있습니다. 간섭 요소를 제거하고 적응성을 추가함으로써 꼬리 지연 시간 분포가 개선되거나, 적어도 부하 증가에 따른 악화 속도가 완화됩니다.

핵심 요약

KV 캐시의 고속 RDMA 전송부터 동적 라우팅 알고리즘 및 QoS 정책에 이르기까지 본 장에서 다룬 기술들은 대규모 생산 추론 워크로드에 필수적인 구성 요소임이 입증되었습니다. 주요 요점은 다음과 같습니다:

프리필-디코딩 간섭 제거로 지연 시간 개선

프리필과 디코딩 단계를 분리하면 간섭과 헤드오브라인 차단이 제거됩니다. 긴 prompt가 짧은 prompt를 지연시키지 않으므로 더 좁은 지연 시간 분포를 얻을 수 있습니다. 또한 각 단계가 지연 시간 SLO를 안정적으로 충족할 수 있게 합니다.

각 단계를 독립적으로 최적화

프리필(prompt 처리)은 컴퓨팅에 바운디드하며 최대 병렬 처리와 높은 FLOPS 성능의 혜택을 받습니다. TTFT(전체 처리 시간)를 최소화하기 위해 고성능 GPU와 FP8/FP4 정밀도 감소 같은 기법을 선호합니다. 디코드(토큰 생성)는 메모리에 바운디드하며 높은 메모리 대역폭 GPU와 연속 배

치 같은 기법을 통해 토큰당 처리량을 극대화하는 혜택을 받습니다. 분리는 각 단계에 서로 다른 하드웨어와 병렬 처리 설정을 허용합니다. 고정된 일률적인 구성은 이를 허용하지 않습니다.

KV 캐시 활용

prompt 사전 채우기 재계산을 피하기 위해 KV 캐시를 사용하는 것이 일반적입니다. 스마트 라우터는 새 요청을 사전 채우기 작업자로 오프로드할지, 아니면 디코드 작업자에서 직접 처리할지 결정할 수 있습니다. 라우팅 정책은 prompt 길이, 캐시 적중률, 클러스터 부하를 고려해야 합니다.

지능적인 라우팅

vLLM, SGLang, NVIDIA Dynamo와 같은 현대적 추론 서버는 다중 요소 점수 알고리즘을 사용하여 사전 채우기 및 디코딩 요청을 라우팅함으로써 처리량을 극대화하고 적절한 지연 시간을 유지합니다. 많은 프레임워크는 스케줄러와 대시보드에서 TPOT을 *토큰 간 지연 시간* (ITL)이라고도 부릅니다. 프리필 노드와 디코드 노드에 대해 TTFT(p50/p95/p99)와 ITL/TPOT(p50/p95/p99)을 별도로 추적하는 것이 권장됩니다. 이렇게 하면 업그레이드 시 성능 저하를 더 쉽게 디버깅할 수 있습니다. 예를 들어, 프리필 서버가 포화 상태이거나 prompt가 짧은 경우 원격 프리필을 건너뜁니다.

QoS를 사용하여 대규모 환경에서 SLA 유지

과부하 시에는 접속 제어와 우선순위 지정을 적용하십시오. 모든 사용자의 시스템 지연 시간이 급증하는 것보다 초과 요청이나 저우선순위 요청을 신속하게 실패 처리하는 것이 최선입니다. 실제 시스템은 "바쁨" 오류 반환 또는 응답 길이 저하를 통해 조기 거부를 구현합니다. 이는 99번째 백분위수 지연 시간이 지정된 임계값 아래 유지되도록 보장하기 위함입니다. 분산 처리와 QoS를 함께 적용하면 하루 수십억 건의 요청이 발생하는 시나리오에서도 과부하 연쇄 반응을 방지할 수 있습니다.

결론

분리형 프리필 및 디코딩은 현대 대규모 LLM 추론에서 보편화되었습니다. 메타, Amazon, 엔비디아, 구글, 오픈AI(일명 'MANGO') 등 거의 모든 주요 AI 공급업체가 대규모 LLM 배포에 분리형 아키텍처를 채택했습니다.

대부분의 기업이 완전한 생산 아키텍처를 공개하지는 않지만, 이 접근법과 그 이점을 보여주는 수많은 오픈소스 구현체(예: vLLM, SGLang, NVIDIA의 Dynamo)가 존재합니다. 분산형 PD는 단일형 설계보다 지연 시간 제약 하에서 더 높은 유효 처리량(goodput)과 우수한 비용 효율성을 제공합니다.

다음 장에서는 현대적인 GPU 및 네트워킹 하드웨어를 더욱 진보된 스케줄링 및 캐싱 최적화와 결합하여, 지연 시간이나 비용을 희생하지 않고도 수조 매개변수 규모의 LLMs 모델을 수십억 명의 사용자에게 서비스하는 방법을 계속 살펴보겠습니다.