제16장. 대규모 추론 자질, 디버깅 및 튜닝

대규모 LLM 추론 클러스터를 운영하려면 모든 것이 예상대로 실행되고 있는지 확인하는 모니터링 및 디버깅 도구가 필요합니다. 또한 성능이 목표치에서 벗어날 때 병목 현상을 신속하게 식별하는 데도 도움이 됩니다.

이 장에서는 자질을 위한 NVIDIA Nsight Systems, 클러스터 전체 텔레메트리 관리를 위한 Prometheus/Grafana와 같은 도구를 활용해 이러한 복잡한 시스템을 모니터링하고 디버깅하는 방법을 보여줍니다. 또한 GPU 사용률, 메모리 압박, 테일 지연 시간 백분위수, 캐시 적중률, 토큰별 타이밍 등 핵심 메트릭을 수집하고 해석하는 방법도 소개합니다. 이러한 메트릭은 추론 엔진 성능 최적화를 위한 지침이 됩니다.

다음으로 운영 성능 튜닝을 논의합니다. 여기에는 대규모 클러스터에서 GPU 활용도 최적화, 추론 지연 시간 감소, 처리량 증대를 위한 생산 환경 검증 방법을 포함합니다. 계산과 통신 중첩, 요청 스케줄링 및 배치 처리, NVLink, NVSwitch, InfiniBand 같은 고속 상호 연결 기술의 효과적 활용 기법이 포함됩니다.

또한 추론을 위한 실시간 양자화 기법을 비교합니다. 여기에는 일반화된 사후 훈련 양자화(GPTQ) 및 활성화 인식 가중치 양자화(AWQ)와 같은 구현을 사용하여 모델을 8비트 및 4비트 정밀도로 압축하는 방법이 포함됩니다. 이 과정에서 가중치만 양자화하는 것과 가중치와 활성화를 모두 양자화하는 것 사이의 장단점을 논의합니다. 모델 정확도를 유지하면서 메모리 사용량을 줄이고 처리량을 높이기 위해 서비스 파이프라인에 양자화를 적용하는 실용적인 지침을 제공합니다.

마지막으로, 저수준 성능 튜닝을 보완하는 애플리케이션 수준 최적화 기법을 고려합니다. 여기에는 prompt 압축, 프리픽스 캐싱, 중복 제거, 쿼리 라우팅(예: 폴백 모델), 부분 출력 스트리밍 등의 전략이 포함됩니다.

추론 성능 자질, 디버깅 및 튜닝

현대 LLM 추론 엔진 에는 특히 분리된 프리필 및 디코딩에서 많은 움직이는 부분이 있습니다. 일반적인 요청의 라이프사이클은 그림 16-1과 같이 여러 구성 요소를 포함합니다.

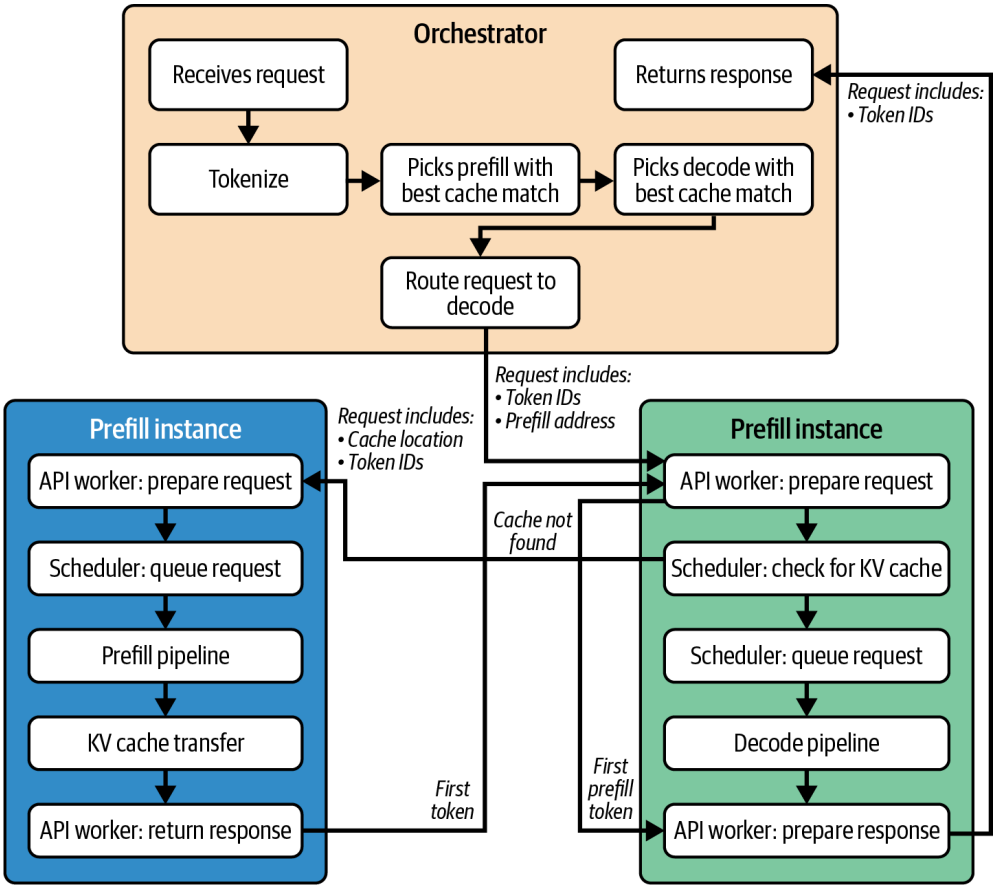

그림 16-1. 분산 프리필 및 디코드 LLM 추론 시스템에서 일반적인 요청의라이프사이클

이러한 복잡성으로 인해 추론 성능 튜닝 워크플로는 매우 반복적입니다. 신중한 튜닝과 지속적인 검증이 필요합니다.

먼저 메트릭을 관찰하여 현재 병목 현상(예: GPU 활용도 미달 또는 예상보다 높은 지연 시간)을 파악해야 합니다. 다음으로 "배치 크기 증가" 또는 "작업 X에 대한 통신-계산 중첩 증가"와 같은 개선 가설을 수립합니다. 그런 다음 수정 사항을 구현하고 가설을 테스트합니다.

이상적으로는 스테이징 환경에서 대표적 워크로드를 사용해 자질 도구를 통해 수정 사항을 테스트하여 변경 사항이 예상대로 동작하는지 검증해야 합니다. 예를 들어, 특정 연산이 적절한 메모리 및 컴퓨팅 오버랩을 보여주는지 확인할 수 있습니다.

마지막으로 수정 사항을 프로덕션에 배포하고 Grafana 및 로그를 모니터링하여 실제 워크로드에서 처리량과 지연 시간이 개선되었는지 검증해야 합니다. 새로운 병목 현상이 발생할 때마다 이 워크플로를 반복합니다.

이 관찰-가설-조정 루프는 지속적이어야 합니다. 현대적 배포 환경에서는 이러한 단계를 자동화하는 경우가 많습니다. 예를 들어, 스케줄된 부하 테스트와 주요 지표에 대한 후속 이상 탐지를 통해 조정 워크플로를 트리거할 수 있습니다.

최적화 사항(업데이트된 추론 런타임 및 모델 변형 포함)을 프로덕션에 배포할 때는 카나리아 롤아웃을 수행하는 것이 좋습니다. 소수의 서버에서 실행되는 트래픽의 작은 부분집합에 최적화를 배포함으로써, 모든 최종 사용자에게 완전한 프로덕션 배포를 진행하기 전에 최적화를 검증할 수 있습니다. 이러한 점진적 접근 방식은 모든 사용자에게 영향을 주지 않으면서 예상치 못한 부작용을 조기에 포착함으로써 그 "파급 범위"를 줄이는 데 도움이 됩니다.

과도한 토큰화 또는 추론 데이터 전처리로 인해 호스트 측 CPU 사용률이 100%로 급증하는 시나리오를 고려해 보십시오. 이는 추론 엔진이 처리할 수 있는 동시 스트림 수를 제한합니다. 한 가지 해결책은 GPU 가속 토큰화 라이브러리나 CUDA 또는 OpenAI의 Triton 언어로 작성된 맞춤형 GPU 커널을 사용하여 전처리를 GPU로 이동하는 것입니다.

새 라이브러리나 커널 배포 후 CPU 사용률을 전후로 모니터링해야 합니다. CPU 사용률이 감소하고 전체 처리량이 증가하면 시스템이 더 이상 CPU 기반 입력 전처리로 인한 병목 현상을 겪지 않는다는 의미입니다.

또한 접두사 캐시, prompt 임베딩 캐시, KV 캐시 등 모든 유형의 캐시 에 대한 캐시 적중률에 주목해야 합니다. "캐시 적중"과 "캐시 미스"에 대한 메트릭을 확보해야 합니다. 높은 캐시 적중률은 시스템이 데이터를 효과적으로 재사용하고 있음을 의미합니다. 반대로 높은 캐시 미스율을 확인한다면, 캐시 적중률을 극대화하기 위해 캐시 크기, 제거 정책 또는 캐싱 전략을 조정해야 할 가능성이 높습니다.

vLLM의 LMCache 구성 요소는 GPU 대 CPU 캐시 비율 조정 을 지원합니다. GPU 메모리 한계로 인해 미스가 높다면, 페이지 캐시 오프로드 기능을 활성화하여 CPU가 보조하도록 할 수 있습니다. 캐시 제거 정책(최근에 사용하지 않은 항목 제거[LRU], 가장 자주 사용하지 않은 항목 제거[LFU] 등)이 액세스 패턴과 항상 일치하도록 하십시오.

또 다른 시나리오는 배치된 요청 간 동일한 입력 시퀀스 접두사에 대해 데이터를 재사용하기 위해 KV 캐시를 사용하는 것으로, 접두사에 대한 KV 재계산을 피할 수 있습니다. 이 경우 요청이 접두사를 공유하는 빈도를 측정해야 합니다. 이는 *접두사 병합* 이벤트로 이어지며, vLLM의 접두사 캐시 히트 메트릭(`vllm:gpu_prefix_cache_queries` 및 `vllm:gpu_prefix_cache_hits` 포함)을 증가시킵니다. 이를 통해 예를 들어 쿼리당 히트 수로 히트율을 계산할 수 있습니다.

접두사 병합률 측정은 실제 캐시 적중률과 상관관계를 분석하여 캐싱 계층의 실질적 이점을 평가하는 데 도움이 됩니다. 이를 통해 공유 접두사를 극대화하도록 배치 및 스케줄링 정책을 조정하고, 다양한 워크로드 하에서 종단 간 처리량 및 지연 시간 개선 효과를 예측할 수 있습니다.

추론 엔진에서 합성 생성 데이터를 실행하여 반복되는 접두사가 많은 prompt를 테스트할 수 있습니다. 접두사 병합으로 인해 프리필 계산량이 감소하는 것을 확

인할 수 있을 것입니다.

vLLM 및 SGLang과 같은 최신 LLM 추론 엔진은 기본적으로 접두사 병합 메트릭을 노출합니다. 그러나 사용 중인 추론 엔진이 접두사 병합을 기본 메트릭으로 제공하지 않는다면, 효과성을 모니터링하기 위해 "중복 제거된 접두사 토큰"에 대한 사용자 정의 카운터를 구현해야 합니다.

접두사 병합이 예상대로 수행되지 않는 경우 접두사 일치 로직이 실패하는지 확인해야 합니다. 토큰화기 차이 여부를 확인하여 디버깅 프로세스를 시작하세요. 이는 대부분의 접두사 일치 문제의 주요 원인입니다.

성능 외에도 모니터링은 용량 계획 수립에 도움이 됩니다. 부하 증가에 따른 활용률과 지연 시간의 변화를 추적함으로써, 시스템이 특정 한계점(예: p95(95번째 백분위수) 지연 시간이 기하급수적으로 상승하기 시작하는 지점)에 도달할 시점을 예측할 수 있습니다. 이 경우 동적 배치 크기가 증가해도 반환하는 지점에 도달할 수 있습니다.

NVMe 기반 KV 캐시 확장을 포함한 계층형 캐싱 전략을 사용하는 경우, 해당 장치의 I/O 지연 시간을 반드시 모니터링하십시오. 높은 I/O 지연 시간은 캐시 성능을 크게 저하시킵니다.

GPU당 동시 처리 능력이 한계에 도달하고 배치 크기 증가가 더 이상 처리량을 향상시키지 않을 때는 GPU 추가, 모델 복제본 증설, 작업 분산을 위한 전문가 수 증가 등 스케일 아웃을 고려해야 합니다.

확장하기 전에 모델 압축 또는 낮은 정밀도(FP8/FP4)로의 전환을 고려하여 GPU당 더 효과적인 처리량을 확보해야 합니다. 그러나 하드웨어가 포화 상태(예: SM 100% 가동, 메모리 대역폭 최대 활용)에 도달하면, 더 많은 GPU 추가 또는 텐서/파이프라인 병렬 처리 활용이 처리량 향상의 유일한 방법이 될 가능성이 높습니다.

새로운 하드웨어 비용과 효율성 향상을 항상 비교 평가해야 합니다. 메모리와 FLOPS가 더 많은 최신 GPU로 업그레이드하는 것이 구형 GPU 군을 확장하는 것보다 비용 효율적일 때가 있습니다.

전문가 수를 늘리면 처리량 상한선을 높일 수 있지만, 추가적인 전수 통신을 관리하기 위해 전문가 라우팅 및 스케줄링도 개선해야만 가능합니다. 그렇지 않으면 단순한 확장만으로는 병목 현상이 네트워크로 옮겨갈 뿐입니다. 다음으로 모니터링과 최적화 노력이 실제로 성과를 내고 있는지 확인하는 방법을 논의해 보겠습니다.

시스템 메트릭 및 카운터 모니터링

전통적인 마이크로서비스 호출( )은 실행 시간이 비교적 균일하고 예측 가능한 반면, LLM 요청은 비균일하며 지연 시간 측면에서 크게 달라질 수 있습니다. 이 차이는 그림 16-2에 표시되어 있습니다.

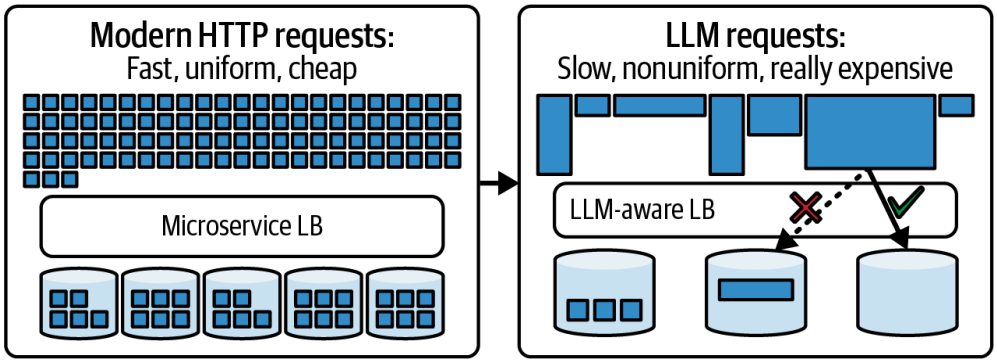

그림 16-2. 기존 마이크로서비스 호출과 LLM 호출의 차이

프로덕션 환경에서 지속적인 모니터링을 위해 각 GPU 컴퓨팅 노드에서 메트릭을 수집하는 데 Prometheus를 사용하는 것이 일반적이며, 이를 시각화하기 위해 Grafana 대시보드를 활용합니다. 추적해야 할 주요 GPU 메트릭에는 GPU 사용률(SM이 바쁜 시간의 백분율), GPU 메모리 사용량, 복사 엔진 사용률, PCIe 및 NVLink 처리량, GPU 온도 및 전력(예: 스로틀링)이 포함됩니다. 참고: L1 및 L2 활동, 점유율, 명령어 처리량과 같은 저수준 카운터는 DCGM 및 Prometheus 대신 Nsight Compute 또는 CUPTI로 수집할 수 있습니다.

메모리 조각화를 모니터링할 때는 DCGM(`cudaMemPool`) 메트릭과 비동기적 DCGM 할당기 통계가 유용합니다. 이를 모니터링에 통합하면 운영 환경에서 시스템 성능 문제 디버깅이 크게 용이해집니다.

또한 NVLink, NVSwitch 대역폭, NIC 처리량 등 상호 연결 활용도를 모니터링하는 것도 중요합니다. 이를 통해 다중 GPU 및 다중 노드 클러스터 구성에서 통신 병목 현상을 포착할 수 있습니다.

NVIDIA의 데이터 센터 GPU 관리자(DCGM)는 많은 GPU 메트릭을 노출하며, Prometheus가 이를 스크래핑하여 수집할 수 있습니다. 예를 들어 DCGM은 SM 활용률 %에 대한 `DCGM_FI_DEV_GPU_UTIL`, 메모리 복사 엔진 활용률에 대한 `DCGM_FI_DEV_MEM_COPY_UTIL`, 사용된 프레임버퍼 메모리에 대한 `DCGM_FI_DEV_FB_USED` 등을 제공합니다.

DCGM은 NVLink 오류 카운터를 노출하며, 일부 플랫폼 및 드라이버 버전에서는 처리량 카운터도 노출할 수 있습니다. 지속적인 링크 사용률 모니터링을 위해서는 `nvidia-smi nvlink` 및 Nsight 도구도 활용하세요. 이러한 메트릭을 대시보드에 통합하고 경보를 설정하여 GPU 간 및 노드 간 통신 트래픽으로 네트워크가 포화 상태일 때 식별할 수 있도록 해야 합니다. DCGM은 Xid 카운터와 함께 중요한 GPU 오류도 추적합니다.

DCGM은 NVLink 카운터를 노출하지만, 현재 시점에서 `dcgm-exporter` 은 기본적으로 모든 플랫폼에서 링크별 대역폭을 노출하지 않습니다. 따라서 링크 수준 처리량이 필요한 경우 DCGM을 직접 쿼리하거나 익스포터를 확장해야 할 수 있습니다.

초당 쿼리/요청 수, 평균 지연 시간 및 p95/p99 지연 시간, 활성 컨텍스트 수, 초당 토큰 처리량과 같은 상위 애플리케이션 메트릭 수집도 권장됩니다. KV 캐시 사용률 및 크기(전체 및 노드별)에 대한 메트릭 모니터링 역시 매우 중요합니다.

Prometheus 노드 익스포터 를 설정하여 각 노드에서 이러한 모든 메트릭을 수집하고, 데이터를 한 곳에 모으며, 중요 임계값에 대한 경보까지 설정할 수 있습니다. Grafana를 사용하면 이러한 메트릭을 실시간 대시보드로 시각화하여 팀과 공유할 수 있습니다. 예를 들어 그림 16-3은 쿠버네티스 클러스터 내 각 GPU에서 메트릭을 수집하여 Prometheus로 내보낸 후 Grafana로 시각화하는 방식을 보여줍니다.

이렇게 하면, 예를 들어 배치 처리량을 늘리기 위한 새로운 최적화를 배포할 때 각 GPU의 GPU 사용률이 증가하는지 Grafana가 즉시 보여줍니다. 또한 p95/p99 지연 시간이 목표 범위 내에 유지되는지 모니터링할 수도 있습니다.

카운터 역시 측정 시 매우 유용합니다—특히 동적이며 적응형 시스템에서 그렇습니다. 예를 들어, 추론 엔진이 현재 조건에 따라 배치 크기를 동적으로 조정한다면, "배치 크기 변경" 카운터를 증가시키고 싶을 수 있습니다.

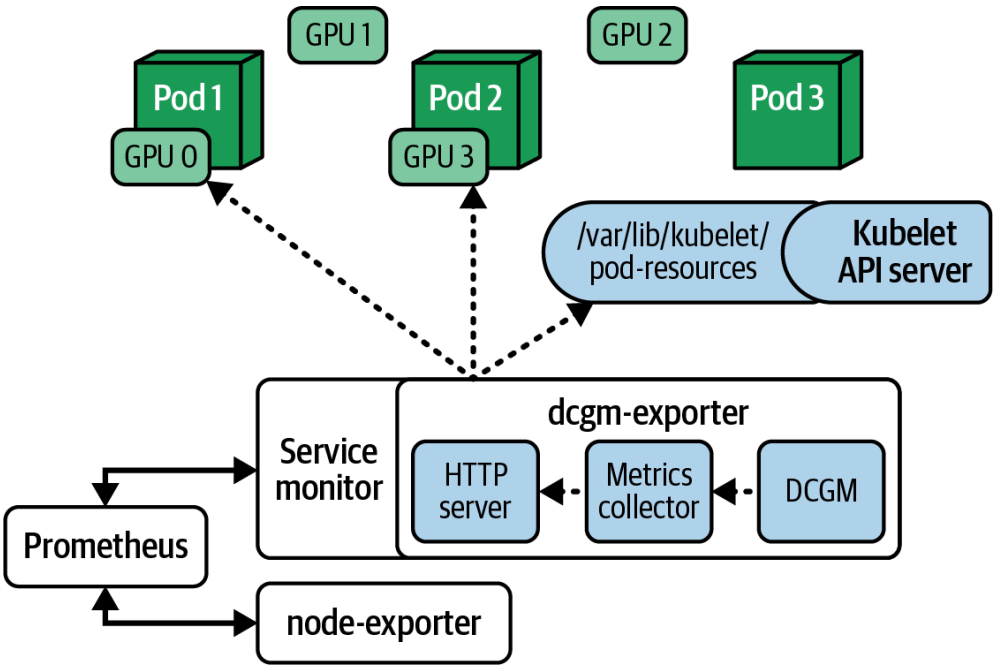

그림 16-3. DCGM은 쿠버네티스 GPU 노드에서 메트릭을 수집하여 프로메테우스에 전송합니다

다른 방법은 변경 사항을 로그 파일에 기록하는 것이지만, 이 경우 오프라인에서 Apache Spark 등을 사용해 로그 파일을 분석하려면 느린 텍스트 기반 검색/집계가 필요합니다. 이후 로그 파일 분석 결과를 Prometheus 메트릭과 수동으로 연관시켜야 합니다.

관심 있는 애플리케이션 수준 이벤트(오류 포함)에 대해 간단한 카운터를 증가시키면 데이터가 Prometheus로 전송되어 Grafana 대시보드에서 다른 모든 메트릭과 함께 실시간으로 즉시 확인할 수 있습니다. 또한 중요한 이벤트에 대해서는 구조화된 로깅과 분산 추적을 사용하는 것을 고려하십시오.

현대적인 애플리케이션 성능 관리(APM) 도구와 OpenTelemetry는 이러한 로그/추적을 수집하여 메트릭과 연관시킬 수 있습니다. 이를 통해 시스템 전체에 걸친 일관된 타임라인 뷰를 제공합니다. 이 타임라인에 대한 통찰력은 성능 문제 디버깅 시간을 단축하는 데 도움이 됩니다.

이러한 메트릭을 지속적으로 모니터링하면 다음에 조정해야 할 부분을 파악할 수 있습니다. 예를 들어 GPU 사용률이 예상보다 낮다면 GPU 메모리가 최대치에 도달했는지 확인할 수 있습니다. 완전히 활용되지 않는다면 배치 크기나 동시 요청 최대 수를 늘려볼 수 있습니다. 다만 지연 시간 서비스 수준 목표(SLO)는 반드시 주시해야 합니다. 이를 초과해서는 안 됩니다.

현대적인 추론 서버는 동적 배치 처리를 위한 "최대 지연 시간" 설정을 노출합니다. SLO(서비스 수준 목표)를 충족하도록 이 설정을 조정하세요. 값을 높이면 배치 크기(처리량)가 증가합니다. 지나치게 높이면 p99 지연 시간이 악화됩니다. 지연 시간 목표에 따라 지속적으로 조정하세요.

반면 GPU 메모리가 최대 용량에 가까울 경우, 추론 엔진이 비활성 KV 캐시 데이터를 호스트 CPU 메모리나 NVMe 스토리지로 스왑아웃하기 시작할 수 있습니다. 이로 인해 GPU 활용도가 저하되는데, GPU가 느린 CPU 메모리나 디스크로부터 추가 데이터 전송을 기다려야 하기 때문입니다.

GPU 메모리 복사 엔진 사용률이 급증하거나, SM 사용률이 낮은 상태에서 비정상적인 NVLink 사용률이 관찰된다면, 추론 엔진이 KV 캐시 데이터를 GPU 메모리 안팎으로 스왑하고 있을 가능성이 높습니다. 이는 과도한 데이터 전송 지연으로 인해 시스템 병목 현상을 유발합니다.

스왑이 발생한다면 추론 엔진의 페이징 매개변수를 조정하여 스래싱을 줄이고, FP8 또는 FP4 양자화를 적용하며, 캐시용 GPU 메모리 할당량을 늘리고, 스왑 전략을 변경할 수 있습니다. 이렇게 하면 복사 활용도는 낮아지고 컴퓨팅 활용도는 높아져 원하는 결과를 얻을 수 있습니다.

Grafana는 지연 시간 추적에도 활용됩니다. 엔드투엔드 요청 지연 시간 분포를 플롯할 수 있으며, 프리필 지연 시간과 토큰당 지연 시간도 함께 측정하는 경우가 많습니다. 특정 시간대에 p99 지연 시간이 급증한다면, 이를 GPU 메트릭 및 기타 로그와 상관관계를 분석해야 합니다.

예를 들어, p99 지연 시간 급증은 GPU 사용률이 떨어지는 시기와 연관될 수 있습니다. 혹은 지연 시간 급증이 트래픽 급증으로 인해 동적 배치 크기가 커진 것과

관련될 수도 있습니다. 이로 인해 해당 기간 동안 지연 시간이 증가할 수 있습니다. 확인을 위해 Grafana 대시보드의 지연 시간 그래프에 RPS(초당 요청 수)를 중첩하여 두 차트가 상관관계가 있는지 확인할 수 있습니다.

가설대로 동적 배치 크기 증가로 인한 예상된 스파이크라면, 서비스 수준 계약(SLA)을 초과하지 않는지 확인하세요. 초과할 경우 최대 요청 배치 대기열 지연 시간을 줄이거나 최대 배치 크기를 축소하여 지연 시간에 제한을 두는 방법을 시도할 수 있습니다.

문제 진단 시 로그 역시 매우 중요합니다. 배치 형성 시점, 통신 시작/종료 시점 등 핵심 이벤트를 기록하도록 코드를 구성해야 합니다. 필요 시 활성화/비활성화가 가능하고 요청-응답 지연 시간에 영향을 주지 않도록 `DEBUG` 수준을 사용하는 것이 가장 좋습니다.

디버그 로깅을 활성화하면 텍스트 형식의 단계별 타임라인을 확인할 수 있습니다. 디버깅 세션에서는 로깅 타임라인과 Prometheus/Grafana 메트릭을 함께 사용할 가능성이 높습니다. 예를 들어, 모든 대 모든 통신이 5밀리초 이상 소요되는 빈도를 확인할 수 있습니다.

로그 기반 타임라인과 메트릭을 결합하면 네트워크 문제와 같은 이상값을 확인할 수 있습니다. 이러한 문제가 지속될 경우 전문가 용량 계수를 높여 초과 토큰이 자동으로 보조 전문가 복제본(이상적으로는 더 안정적인 네트워크 경로를 가진 GPU에 호스팅됨)으로 스필오버되도록 할 수 있습니다. 이렇게 하면 부하가 균형 잡히고 지연 시간이 최소화됩니다.

실무에서는 용량 계수를 1.2~1.5로 설정하는 것이 일반적입니다. 이렇게 하면 주 전문가(primary expert)가 과부하 상태일 때 20%~50%의 추가 토큰을 재할당할 수 있기 때문입니다. 이는 MoE 추론에서 꼬리 지연 시간을 상당히 완화시킬 수 있습니다. 연결 상태가 저하된 GPU에서 약간 지연된 전문가 뒤에 요청을 대기열에 넣는 것보다 두 번째 전문가로 스필링하는 것이 더 낫습니다. 이는 네트워크가 어떤 이유로든 계속 문제를 겪을 경우 이상치에 대한 민감도를 줄여줍니다.

Nsight Systems 및 Nsight Compute를 사용한 자질

추론 코드 개발 및 튜닝 시( ) Nsight Systems를 활용하여 CPU와 GPU 전반의 워크로드 추적을 캡처할 수 있습니다. Nsight Systems는 마이크로초 단위의 해상도로 CPU 스레드, GPU 커널, CUDA 이벤트, NCCL 통신 등을 보여주는 타임라인 뷰를 제공합니다.

NVTX 어노테이션으로 코드를 계측하면 타임라인에 "프리필 단계", "디코드 단계", "전역 통신" 등의 영역을 명확하게 표시할 수 있습니다. 다음 코드는 NVTX v3 C API를 사용하여 명시적인 범위 푸시 및 팝을 통해 예시 프리필 및 디코드 단계 주변에 NVTX 범위 마커를 적용한 예시입니다:

```cpp
// Example C++ snippet with NVTX annotations using the C API.

#include <nvtx3/nvToolsExt.h>   // or <nvToolsExt.h>
#include "my_model.hpp"         // Your model's C++ interface
#include <vector>

// Small helpers to keep callsites tidy.
#define NVTX_PUSH(name, argb)                                  \
  do {                                                         \
    nvtxEventAttributes_t a{};                                 \
    a.version = NVTX_VERSION;                                  \
    a.size = NVTX_EVENT_ATTRIB_STRUCT_SIZE;                    \
    a.colorType = NVTX_COLOR_ARGB;                             \
    a.color = (unsigned int)(argb);                            \
    a.messageType = NVTX_MESSAGE_TYPE_ASCII;                   \
    a.message.ascii = (name);                                  \
    nvtxRangePushEx(&a);                                       \
  } while (0)

#define NVTX_POP() do { nvtxRangePop(); } while (0)

struct Token { int id; };

void run_inference(
    const std::vector<Token>& prompt_tokens,
    Model& model,
    int num_generate_steps) {
  // Prefill
  NVTX_PUSH("Prefill", 0xFF4F86F7);
  model.encode(prompt_tokens);
  NVTX_POP();

  // Decode one token at a time
  for (int t = 0; t < num_generate_steps; ++t) {
    NVTX_PUSH("Decode", 0xFFFF8C00);
    Token next_token = model.decode_next();
    // ... (sampling / streaming to client)
    NVTX_POP();
  }
}
```

여기서는 `nvtxRangePushEx`/`nvtxRangePop`로 영역을 명시적으로 표시합니다. `model.encode(...)` 바로 앞에 `"Prefill"` 범위를 푸시하고, 바로 뒤에 팝합니다. `decode` 루프 내부에서는 각 반복 시작 시 `"Decode"`를 푸시하고, `model.decode_next()` 후 팝합니다. 작은 `NVTX_PUSH`/`NVTX_POP` 헬퍼 함수는 색상(16진수 값)과 텍스트도 추가합니다. 이는 타임라인 시각화에서 불일치를 줄이면서 호출 위치를 간결하게 유지하는 데 도움이 됩니다. 명시적인 푸시/팝 쌍은 코드에서 명확하게 확인 가능하여 감사하기 쉽습니다.

색상 블록 주석은 Nsight Systems GPU 활동 타임라인에 '`"Prefill"`' 및 '`"Decode"`'으로 표시됩니다. 이를 통해 각 단계 소요 시간과 통신 작업과의 중첩 정도를 쉽게 파악할 수 있습니다. 이는 GPU 유휴 간격이나 예상치 못한 동기화 같은 문제점을 식별하는 데 도움이 됩니다.

참고: PyTorch의 `record_function()` 대신 NVTX C API(`nvToolsExt`)를 직접 사용합니다. 이를 통해 순수 C++ 런타임에서 핫 패스를 주석 처리할 수 있으며, Python이나 다른 언어에서 작업을 시작할 때도 마커가 일관되게 유지됩니다.

`model.encode(prompt_tokens)` 주변에 필요한 최소 영역으로 범위를 좁힘으로써, 자질 마커는 프리필 작업만을 정확히 커버하고 다른 코드는 포함하지 않습니다. 이는 추적 명확성과 성능 진단을 개선합니다.

여러 CUDA 스트림(예: H2D/D2H 복사를 위한 전용 "전송" 스트림 및 커널을 위한 "계산" 스트림)에 작업을 큐에 넣을 때는 스트림별 범위를 사용해야 합니다. 이를 위해 각 스트림의 호스트 코드를 고유한 NVTX 범위로 감쌀 수 있습니다.

예를 들어, 스트림 이름을 `nvtxNameCudaStreamA(transfer_stream, "transfer_stream")` 및 `nvtxNameCudaStreamA(compute_stream, "compute_stream")` 와 같이 지정할 수 있습니다. 그런 다음 메모리 복사/전송 작업에는 `nvtxRangePushA("transfer_stream")` 및 `nvtxRangePop()` 를, 커널 실행에는 `nvtxRangePushA("compute_stream")` 및 `nvtxRangePop()` 를 사용합니다.

NVTX로 명명된 스트림을 사용하면 Nsight Systems 타임라인에서 오버랩(또는 그 부재)이 명확해집니다. 다음은 이 모든 요소가 어떻게 결합되는지 보여주는 코드 예시입니다:

```
// One-time after creating the streams
nvtxNameCudaStreamA(transfer_stream, "transfer_stream");
nvtxNameCudaStreamA(compute_stream,  "compute_stream");

// Around H2D/D2H copies (transfer stream)
nvtxRangePushA("transfer_stream");
cudaMemcpyAsync(h_logits, d_logits, bytes, cudaMemcpyDeviceToHost,
                transfer_stream);
nvtxRangePop();

// Around kernel enqueues (compute stream)
nvtxRangePushA("compute_stream");
my_kernel<<<grid, block, 0, compute_stream>>>(...);
nvtxRangePop();
```

여기서는 스트림에 이름을 지정하고 각 스트림 범위로 enqueue 위치를 감싸 Nsight 타임라인의 가독성을 유지합니다. NVTX 범위가 호스트 스레드 타임라인에 주석 처리된다는 점을 유의해야 합니다. GPU 레인은 스트림별 커널/`memcpy`을 표시합니다. 스트림에 이름을 지정하면 분석 시 호스트 범위를 올바른 GPU 레인과 연결하는 데 도움이 됩니다.

Nsight Compute를 사용하면 개별 커널 의 자질을 파악하여 비효율성을 정확히 확인할 수 있습니다. Nsight Compute의 섹션 기반 프로파일링 기능을 활용하면 메모리 트랜잭션과 같은 커널의 특정 부분에 집중할 수 있습니다.

에서 잘 알려지지 않았지만 매우 유용한 또 다른 도구는 Nsight Compute의 CUDA 프로그램 카운터(PC) 샘플링 기능입니다. 이 기능은 프로그램 카운터를 샘플링하여 전체적인 무거운 계측 없이도 핫스팟을 식별합니다(그림 16-4 참조).

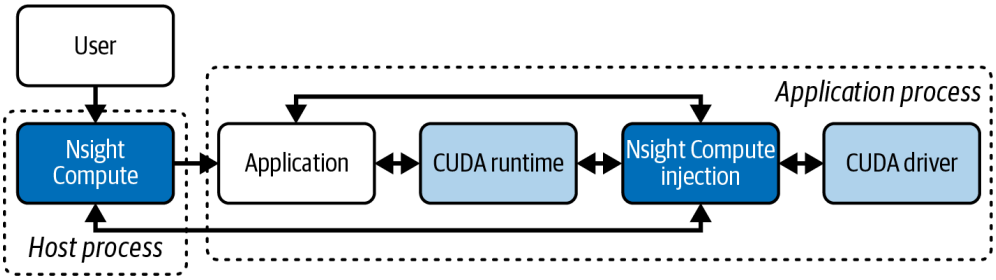

그림 16-4. Nsight Compute의 CUDA 프로그램 카운터(PC) 샘플링 기능은 낮은 오버헤드로 핫스팟을 식별하는 데 도움이 됩니다(출처: https://oreil.ly/DyKWR)

특히 이 기능을 활용하면 실행 중인 추론 서버의 자질을 파악하고 정확히 어떤 커널 명령어가 가장 많은 시간을 소모하는지 파악할 수 있습니다. 또한 낮은 오버헤드로 이를 수행할 수 있습니다. Nsight Systems 및 Nsight Compute를 통한 프로파일링을 살펴보았으니, 이제 추론에 대한 일반적인 문제 해결 레시피를 논의해 보겠습니다.

실제 서비스 운영 환경에서 문제 조사를 할 때는 최소한의 오버헤드로 핫스팟을 국소화하기 위해 먼저 프로그램 카운터 샘플링을 사용하세요. 샘플이 특정 커널이나 단계를 가리킬 때만 전체 추적으로 전환하십시오.

추론 문제 해결 방법

실제 운영 환경에서는 무거운 자질을 지속적으로 실행하는 것이 비현실적입니다. 따라서 GPU SM 사용률, KV 캐시 경고, 테일 지연 백분위수, 캐시 적중률, OOM 경고 등 경량 메트릭 기반 모니터링에 의존하여 이상 현상을 탐지하고 표적화된 수정 방안을 도출해야 합니다. 지표가 특정 임계값을 초과하면 소규모 배치 크기, KV 캐시 부족, 라우팅 핫스팟, 불균형한 샤딩, 메모리 오버커밋 등 근본 원인에 대한 가설을 세울 수 있습니다.

그런 다음 배치 크기 조정, 메모리 사용량 제한 상향(가능한 경우), 라우터 임계값 조정, CPU 오프로드 활성화 등의 해결책을 적용합니다. 수정 사항을 적용한 후에는 그 영향을 검증하고 메트릭이 임계값 아래로 안정화되었는지 확인해야 합니

다. 표 16-1은 일반적인 프로덕션 문제에 대한 주요 메트릭, 증상, 가능한 원인 및 권장 해결책을 보여줍니다.

표 16-1. 일반적인 문제 해결 증상, 원인 및 권장 조치

지표/증상	가능한 원인	권장 조치
SM 사용률 < 50%	작은 배치 크기 또는 융합 커널 부족	배치 크기 증가, 융합 커널 활성화 (FlashAttention 또는 cuDNN-융합 SDPA(`scaled_dot_product_attention`) PyTorch 백엔드 사용), 또는 사용자 정의 융합 커널 추가(예: Triton 사용); 이후 `nsys --trace=cuda` 로 자질.
KV 캐시 선점 경고	KV 캐시 공간 부족 (vLLM)	GPU 메모리 사용량 임계값 증가, 배치된 토큰 최대 개수 감소, 동적 KV 할당을 위한 PagedAttention 고려.
높은 테일 지연 시간 (p95 > 200 ms)	디코딩 노드 핫스팟 또는 헤드 오브라인 차단	라우팅 패턴 확인을 위해 라우터 로그를 검토하십시오. 프리페치 임계값을 조정하십시오. 추측 디코딩 경로를 활성화하십시오.
부하 시 캐시 적중률 < 60%	샤드 배치 불균형 또는 접두사 캐시 누락	접두사 캐싱 커넥터 구성(예: vLLM용 LMCache의 NIXL, NVIDIA Dynamo의 NIXL 커넥터)을 검증하고, 필요한 경우 접두사 캐시 TTL 또는 복제본 수를 증가시킵니다.
멀티테넌트 GPU에서 예기치 않은 OOM 발생	GPU 메모리 과다 할당	인스턴스별 GPU 메모리 사용률 낮추기, CPU/NVMe 오프로드 활성화, 프로세스를 CPU 소켓에 고정하여 소켓 간 트래픽 감소.
비정상적인 성능 이상치	클럭 불일치 또는 열 제한	모든 클럭이 동기화되었는지 확인하고, 열 및 전력 스로틀링을 모니터링하십시오.

참고: 모든 메트릭스 테이블의 수치 값은 개념 설명을 위한 예시입니다. 다양한 GPU 아키텍처에 대한 실제 벤치마크 결과는 GitHub 저장소를 참조하십시오.

일부 정보는 로그 파일에 묻혀 있을 수도 있습니다. AWS와 같은 클라우드 공급자는 로그 파일에서 정규 표현식(RegEx) 필터를 지원하여 로그 라인에서 숫자 값을 추출하고 이를 직접 메트릭으로 내보낼 수 있습니다. 예를 들어 AWS CloudWatch는 이 유용한 기능을 지원합니다. 다음 코드 블록에는 모니터링에 유용한 로그 라인 예시가 있습니다.

다음은 KV 캐시 공간 부족으로 인한 KV 캐시 선점(preemption)을 나타내는 vLLM 로그 스니펫 입니다. 이로 인해 KV 재계산이 트리거되어 GPU 컴퓨팅 리소스 사용량이 증가하고 지연 시간이 늘어납니다:

```
WARNING 2025-05-03 14:22:07 scheduler.py:1057 Sequence group 0 is preempte
PreemptionMode.RECOMPUTE because not enough KV cache space.
total_cumulative_preemption_cnt=1
```

다음은 NVIDIA Dynamo 라우팅 로그 샘플입니다. 첫 번째 줄은 로컬 디코드 워커에서 90%의 프리픽스 캐시 히트를 보여줍니다. 이로 인해 프리필이 로컬에서 계속 실행되었습니다. 다음 줄은 로컬 캐시 미스를 나타냅니다. 이후 라우터는 프리필을 원격 `GPU-node-03` 워커 노드로 전달합니다:

```
[Router] 2025-05-03T14:23:11Z INFO KVRouter: prefix-cache hit (90%) for
model=DeepSeek-R1; routing to local vLLM worker
[Router] 2025-05-03T14:23:12Z INFO KVRouter: cache miss; dispatching remot
prefill to GPU-node-03
```

전체 스택 추론 최적화

고성능 LLM 추론은 스택의 모든 계층에 걸쳐 의 조정된 최적화를 요구합니다. 여기에는 모델 아키텍처와 커널 구현부터 런타임 엔진, 시스템 오케스트레이션, 배포 인프라에 이르기까지 모든 것이 포함됩니다.

프루닝, 디스틸레이션, 스파시티, MoE 라우팅, 효율적 어텐션(예: FlashAttention), 양자화 인식 훈련과 같은 모델 수준 기법은 컴퓨팅 및 메모리 요구 사항을 줄일 수 있습니다. 커널 수준에서는 융합 연산, 맞춤형 어텐션 엔진(예: FlashInfer), 텐서 코어 활용, 블록 타일링, 비동기 메모리 전송이 GPU 처리량 극대화에 기여합니다.

동적 배치, 페이지형 KV 캐시, CUDA 그래프, 계산과 통신 중첩과 같은 런타임 전략은 가변 부하에서도 GPU를 포화 상태로 유지합니다. 시스템 오케스트레이션 계층은 프리필/디코딩 분산, 지능형 라우팅, 멀티테넌시 격리, 자동 확장(예비 장치 포함)을 통해 지연 시간을 균형 있게 조정하고 비용 효율성을 개선할 수 있습니다.

많은 프로덕션 시스템은 쿠버네티스 기반 오케스트레이션인 를 사용하여 프리필과 디코딩 배포를 별도로 실행합니다. 이들은 인그레스 컨트롤러를 사용하여 부하나 사용자 우선순위에 따라 요청을 라우팅합니다. 또한 트래픽이 급증할 때 즉시 가동할 수 있도록 대기 중인 GPU 포드를 준비해 둡니다.

마지막으로, 지리적으로 분산된 에지 서비스, 스마트 API 게이트웨이 배치 처리, 모델 변형에 대한 CI/CD, 실시간 자질과 같은 배포 패턴을 탐구해야 합니다. 이는 프로덕션 환경에서 최고의 안정성과 적응성을 제공할 것입니다. 표 16-2는 스택의 각 계층에 대한 일반적인 최적화 접근법을 설명합니다.

표 16-2. 스택 각 계층별일반적인 최적화 접근법

스택 레이어	주요 기술
모델	정확도 손실을 최소화하면서 모델 크기를 축소하기 위한 가지치기 및 지식 증류 스파시티(MoE)를 통한 계산 생략 메모리 사용량 및 중간 버퍼 감소 위한 효율적 어텐션(FlashAttention) 저정밀도에서의 강건성을 위한 FP16/BF16 또는 INT4/FP8 양자화 인식 훈련
커널	연산자 커널 융합(예: 선형 + GELU + 레이어 정규화)으로 런치 오버헤드 및 메모리 트래픽 감소 블록 스파시티 KV용 커스텀 어텐션 커널(FlashInfer) 및 JIT 컴파일 커널 행렬 연산을 위한 텐서 코어 및 특수 명령어 활용(cp.async, TMA)
런타임	vLLM, SGLang, NVIDIA Dynamo에 구현된 지연 제어 동적 배치(예: 연속 배치)를 통한 요청 통합 메모리 유연 할당 및 배치 병합을 위한 페이지형 KV 캐시 관리(vLLM의 PagedAttention) 추론당 오버헤드 감소를 위한 CUDA 그래프 및 버퍼 풀링 계산과 통신을 중첩시키기 위해 다중 CUDA 스트림 사용(데이터 전송용 스트림과 계산용 스트림 분리). 이벤트 기반 동기화 사용—필요할 때만.
시스템 오케스트레이션	헤드 오브 라인 차단 제거를 위한 프리필-디코딩 분리 지능형 라우팅 및 캐시 어피니티를 통한 부하 및 캐시 적중률 균형 조정 다중 테넌시 격리 및 사용자별 할당량으로 노이즈 이웃 방지 모델 로드 시간을 숨기고 트래픽 급증 시 지연 시간을 크게 개선하기 위해 약간의 비용 증가를 감수하는 웜 스페어 인스턴스를 통한 자동 확장
배포 및 인프라	지리적 분산 및 엣지 배포를 통한 네트워크 RTT 감소 서버 풀 간 요청 단위 배치 기능을 갖춘 스마트 API 게이트웨이 캐니리 모드에서 새로운 양자화 또는 커널 최적화 모델 변형을 롤아웃하기 위한 CI/CD 파이프라인 최적의 메모리 접근을 위한 고대역폭 상호 연결(NVLink/NVSwitch 및 InfiniBand)과 GPU와 CPU 간의 NUMA 어필리네이션
QoS 및 확장성	SLA 인식 동적 배치 및 테일 지연 제어 QoS 강화를 위한 MIG 또는 스트림 우선순위 기반 GPU 격리 TTFT, TPOT, 활용도 및 메모리 대역폭 활용도를 위한 실시간 자질 대시보드 워크로드 특성에 기반한 동적 병렬 처리 전환(TP, PP, DP)

최적화 시 레이어 간 시너지 효과를 고려하는 것이 중요합니다. 예를 들어 양자화(모델)는 메모리 사용량을 줄여 OOM 오류 없이 더 큰 배치 크기(런타임)를 가능하게 하며, 이는 다시 오케스트레이션 구성 요소가 GPU 사이클당 더 많은 요청을 병합할 수 있게 합니다.

자질 기반 접근도 필수적입니다. 지속적인 프로파일링을 통해 다음 최적화 대상 레이어를 결정해야 합니다. 예를 들어, 융합 및 양자화 후 전처리/후처리 단계에서 CPU가 병목이 되면 더 빠른 토큰화기를 도입하거나 일부 작업을 GPU로 오프로드해야 합니다.

최적화 적용 시 항상 고려해야 할 상충 관계가 존재합니다. 레이어 수준 CPU 오프로드나 고급 디코딩 기법 같은 기술은 복잡성을 가중시킵니다. 예를 들어, 추측 디코딩은 드래프트 모델을 추가하고, 메두사(Medusa)는 멀티헤드 병렬 디코딩을 도입합니다. 이러한 기법은 일반적으로 초장문맥이나 불규칙한 지연 시간 변동 같은 극단적인 경우에 한정됩니다. 스파스성, 배치 처리, 분산 처리 같은 경량 기법이 실제 운영 환경에서 대부분의 이점을 제공합니다.

모델 아키텍처, 커널 설계, 런타임 동작, 시스템 오케스트레이션, 배포 전략을 조율하여 풀스택 최적화 접근법을 채택하는 것이 권장됩니다. 이는 CUDA, cuDNN, NCCL 등을 포함한 소프트웨어 스택을 최신 상태로 유지하는 것을 의미합니다. 최신 버전에는 종종 최신 최적화 및 버그 수정이 포함됩니다.

전체 스택 접근법은 각 스택 계층이 병목 현상이 될 가능성을 줄입니다. 이를 통해 팀은 체계적으로 병목 현상을 제거하고, 일관된 저지연성을 달성하며, 대규모 LLM 추론을 위한 하드웨어 활용도를 극대화할 수 있습니다.

정확성 문제 디버깅

모니터링은 버그로 인한 이상 현상 을 포착하는 데도 도움이 됩니다. 예를 들어, 메모리 사용량이 시간이 지남에 따라 계속 증가한다면 CUDA 커널의 메모리 누수로 인한 것일 수 있습니다. 테스트 중 컴퓨트 샌티나이저(`compute-sanitizer`)를 사용하여 장치 메모리 오류, 경합 조건 및 범위를 벗어난 액세스를 포착하는 것이 좋습니다. 예시는 다음과 같습니다:

```
compute-sanitizer --tool memcheck your_binary
```

한 GPU의 사용률이 다른 GPU보다 현저히 낮다면, 이는 NCCL 통신 그룹에서 조용한 NCCL 실패나 잡히지 않은 오류로 인해 이탈했을 수 있습니다. 로그에서 `WARN NCCL_COMM_FAILURE` 를 검색하여 NCCL 오류 코드를 확인할 수 있습니다. 매우 상세한 오류 로그를 제공합니다.

환경 변수 `NCCL_DEBUG=WARN`를 설정하여 NCCL 디버깅을 활성화하십시오. 이는 그렇지 않으면 감지되지 않을 수 있는 오류를 표면화하는 데 도움이 됩니다. 그러나 NCCL 로그는 매우 상세하다는 점을 유의하십시오!

NCCL 테스트 스위트( )를 사용하여 all-reduce 및 all-to-all 성능과 정확성 문제를 디버깅하십시오. 비동기 통신 오류를 감지하고 처리하기 위해 `ncclCommGetAsyncError`와 `ncclCommAbort` 도 함께 사용할 수 있습니다. NCCL의 `IB GID` 추적을 활성화하고 NVSwitch 시스템 텔레메트리도 사용하여 상호 연결 수준에서 발생하는 문제도 감지하는 것을 고려하십시오.

Prometheus의 알림 관리자( )에서 "GPU 사용률 10% 미만 상태가 60초 이상 지속", "메모리 사용량 W% 이상", "NVLink 오류율 X 초과", "PCIe 재전송 Y 이상", "온도 Z도 이상" 등과 같은 비정상 패턴을 감지하도록 알림을 설정해야 합니다.

실제 운영에서는 표 16-3과 같이 Prometheus Alertmanager 규칙을 구성할 수 있습니다. 이를 통해 문제를 사전에 조사할 수 있습니다.

표 16-3. GPU 기반 시스템에 대한 일반적인 Prometheus 알림예시 세트

메트릭	조건	중요도	참고
GPU 사용률	60초 이상 동안 10% 미만	유휴	저활용
GPU 활용도	> 90%	병목 현상	포화 가능성
메모리 사용량	80% 초과	경고	OOM에 접근 중
메모리 사용량	> 95%	위험	메모리 부족 오류 발생 위험이 매우 높음
온도	85 °C 이상	경고	열 스로틀링에 근접함
온도	> 95 °C	위험	시스템 종료 또는 하드웨어 손상 위험
NVLink 재생/복구 오류	≥ 1	중요	링크 재시도 또는 복구를 나타냄
NVLink CRC 오류	초당 100개 이상의 오류	중요	링크에서 높은 CRC 실패율
PCIe 재생 오류	≥ 1	중요	PCIe 버스에서 패킷 재시도
수정 불가능한 ECC 오류	≥ 1	중요	리셋이 필요한 데이터 손상

하드웨어 오류 카운터와 경보도 함께 설정해야 합니다. 예를 들어 ECC 오류나 NVLink 재시도가 보고되면 즉시 경보를 발령하세요. 이러한 오류는 성능을 급격

히 저하시키거나 드롭아웃을 유발할 수 있습니다. 드롭아웃은 GPU 또는 그 상호 연결이 무음으로 연결이 끊어질 때 발생합니다. 예를 들어 NVLink가 끊어지거나 치명적 오류 후 GPU가 "버스에서 이탈"할 수 있습니다.

링크별 NVLink 처리량 및 총합을 측정하려면 DCGM을 사용하십시오. 여기에는 `DCGM_FI_DEV_NVLINK_TX_BANDWIDTH_L*`, `DCGM_FI_DEV_NVLINK_RX_BANDWIDTH_L*`, `*_TOTAL` 등이 포함됩니다. 필요한 경우 `nvidia-smi nvlink`, Nsight Systems/Compute 또는 NVSwitch 카운터로 대체하십시오.

NCCL 장애 사례를 고려해 보십시오. 한 노드의 GPU 사용률이 0%에 가까워지는 반면 다른 노드는 90%인 경고를 받을 수 있습니다. 노드 로그를 확인하고 NCCL 오류를 발생시키는 노드를 찾아 문제를 디버깅할 수 있습니다.

이러한 능동적 모니터링 및 알림을 통해 문제를 더 빠르게 포착하고, 장애 노드를 찾아 정상 상태로 복구할 수 있습니다. 이 경우 NCCL 커뮤니케이터를 재초기화하거나 노드를 완전히 재부팅할 수 있습니다(단, 재시작/재부팅 후 노드가 NCCL 그룹에 재가입하도록 해야 합니다).

NCCL 오류에 대한 "NCCL Error" 카운터를 증가시켜 이러한 문제를 더욱 신속하게 포착할 수 있습니다. 또한 추론 서버가 NCCL 오류를 로깅하면 Fluentd 또는 AWS CloudWatch가 이를 자동으로 수집하여 카운터로 변환합니다.

그런 다음 Grafana에서 노드별 GPU 사용률 차트 위에 오류 카운터 차트를 중첩 표시할 수 있습니다. 이렇게 하면 NCCL 오류와 GPU 사용률 하락을 연관 지어 실패한 노드를 훨씬 빠르게 식별하고 복구할 수 있습니다.

애플리케이션 수준 카운터는 특히 시스템 메트릭과 결합할 때 운영 환경에서 매우 유용합니다.

최적화는 처리량 증가, 지연 시간 감소, 활용도 개선 등을 입증하는 실제 메트릭으로 검증되기 전까지는 성공적이라고 간주해서는 안 됩니다. 현대 AI 추론 시스템의 복잡성을 고려할 때, 시스템 성능 튜닝에 대한 엄격한 측정 중심 접근 방식이 필수적입니다.

요약하면, 애플리케이션 수준 카운터, Fluentd 또는 AWS CloudWatch에서 자동 수집된 로그 카운터, 저수준 시스템 메트릭을 결합해야 합니다. 이러한 풀스택 텔레메트리 접근 방식은 프로덕션 워크로드에서 최고 성능으로 운영되는 다중 노드 LLM 추론 시스템을 운영하고 최적화하는 데 필요한 가시성을 제공합니다. 메트릭, 카운터, 로그를 시스템 동작의 진실된 기준으로 삼아야 합니다.

직관과 감정은 종종 잘못된 디버깅 방향으로 이끌 수 있습니다. 그러나 메트릭은 거짓말을 하지 않습니다. 코드를 사전에 적절히 계측하고, 문제가 발생했을 때는 메트릭을 신뢰하십시오.

동적 배치, 스케줄링 및 라우팅

모델을 클러스터 전반에 걸쳐 최적화하여 분할 및 병렬화한 후에도, 다중 노드 추론 클러스터 배포 환경에서는 애플리케이션 수준 최적화를 위한 추가 기회가 여전히 존재합니다. 본 섹션에서는 요청을 동적으로 배치하고 최적화된 스케줄링 및 라우팅 전략을 활용하여 GPU 활용도를 극대화하고, 지연 시간을 최소화하며, 처리량을 향상시키는 기술에 집중합니다.

동적 배치 처리

추론 서비스 시스템 에서 가장 강력한 성능 기법 중 하나는 배칭입니다. 배칭은 여러 입력 데이터를 하나의 배치로 결합하여 모델이 함께 처리하도록 함으로써 커널 실행 및 메모리 로드와 같은 고정 비용을 여러 입력에 분산시켜 처리량을 향상시킵니다. 그러나 이는 개별 요청 지연 시간을 희생하는 대가로 이루어집니다. 일부 개별 요청은 배치에 합류하여 처리되기까지 일정 시간(예: 2ms)을 기다려야 할 수 있습니다.

동적 배칭은 동적 크기의 추론 요청을 실시간으로 배치하는 요청 배칭의 특수한 형태입니다. 일정 시간 동안 요청을 버퍼링하거나 지정된 배치 크기에 도달할 때까지 요청을 대기하도록 구성할 수 있습니다. vLLM, SGLang, NVIDIA Dynamo를 포함한 모든 현대적 LLM 추론 엔진은 동적 배칭을 지원합니다.

동적 배칭은 *정적* 배칭 과 대조됩니다. *정적 배칭*은 고정된 배치 크기를 고정하거나(또는 모든 시퀀스를 가장 긴 시퀀스 길이로 채움) 해당 배치의 모든 요청이 완료될 때까지 기다린 후 결과를 반환합니다. 이는 일찍 도착한 요청에 대해 무제한 대기 지연을 초래할 수 있으며, 패딩 작업에 GPU 사이클이 낭비되게 합니다.

동적 배치에서는 시스템이 들어오는 요청을 누적한 후, 목표 배치 크기에 도달하거나 짧은 시간 제한(예: 2ms)이 경과하면 "도착한 모든 요청"을 즉시 처리합니다. 이로 인해 최대 지연 시간은 사용자가 지정한 시간 제한 값으로 바운디드됩니다.

동적 배칭은 실시간 크기 조정을 통해 커널 실행 오버헤드를 여러 시퀀스에 분산시키면서 정적 배칭의 최악의 지연 상황을 피할 수 있습니다. 이는 가변 부하 환경에서 GPU 활용도와 예측 가능한 지연 시간을 모두 개선합니다. 그림 16-5는 정적 배칭과 동적 배칭의 차이를 보여줍니다.

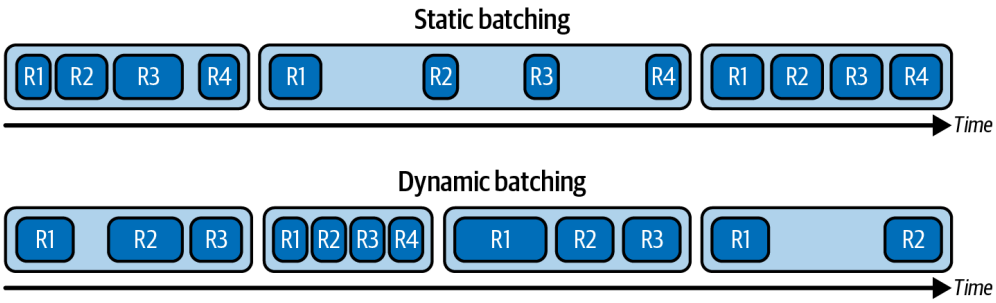

그림 16-5. 정적 배치와 동적 배치의차이

동적 배치 처리 방식은 실제 요청 도착 패턴과 지연 시간 목표(예: 최대 지연)에 따라 배치 크기를 자동으로 확장하거나 축소합니다. 핵심은 허용 가능한 바운디드 내에서 지연 시간을 유지하면서 전체 처리량을 증가시키는 방식으로 지능적으로 배치하는 것입니다. 현대적인 추론 엔진은 마이크로배칭을 구현하여 GPU로 배치를 전송하기 전 단 몇 밀리초 동안만 요청을 누적합니다. 일반적으로 2~10ms의 지연 시간이 사용되지만, 이는 지연 시간 SLO를 충족하도록 조정해야 합니다.

예를 들어, 2ms 배치 지연을 설정하면 서버가 추가 요청을 기다린 후 배치를 전송하기까지의 시간을 결정할 수 있습니다. 해당 간격 내에 세 개의 요청이 도착하면 배치기가 즉시 이를 그룹화하여 GPU로 전송합니다. 2ms 타이머가 만료된 후 도착하는 추가 요청은 다음 배치에 수집됩니다. 이 타임아웃 기반 트리거는 요청별 대기열 지연 시간을 바운디드(2ms 초과 불가)하면서 여러 시퀀스를 그룹화하여 처리량을 향상시킵니다.

이러한 마이크로배치는 추가되는 지연을 줄이고 GPU가 한 번에 하나의 요청이 아닌 여러 요청을 동시에 처리할 수 있게 합니다. 대규모 LLM 모델은 작은 배치 크기에서 메모리 대역폭이 바운디드되기 때문에 배치 크기를 늘리면 연산 집약도가 향상되고 전체 하드웨어 활용도가 높아집니다.

실제 LLM 서비스 시스템은 지연 시간을 크게 증가시키지 않으면서 우수한 처리량을 제공하는 균형 잡힌 배치 크기를 선택합니다. 높은 부하 환경에서 동적 배치 처리는 GPU가 요청 사이에서 유휴 상태가 되는 것을 방지하므로 처리량과 지연 시간 모두를 개선할 수 있습니다. 실제로 요청 도착률이 높을 경우, 동일한 시간 내에 더 많은 작업을 처리하므로 배치 처리가 전체 대기열 대기 시간을 줄일 수 있습니다. 이는 고부하 트래픽 패턴에서 종단 간 지연 시간과 꼬리 지연 시간 모두에 이점을 제공합니다.

동적 배치 처리( )는 낮은 초당 요청 수(RPS)에서 다른 요청이 배치에 합류할 때까지 대기하면서 약간의 지연을 추가합니다. 이는 낮은 RPS에서 지연 시간을 약간 증가시킵니다. 그러나 중간에서 높은 RPS에서는 이 지연이 분산되어 모든 요청을 일일이 처리할 때 발생하는 대기열 지연에 비해 무시할 수 있는 수준이 됩니다. 따라서 배치 처리는 꼬리 지연 시간을 포함한 전체 지연 시간을 감소시킵니다.

개선 효과를 검증하려면 Grafana 같은 도구를 사용해 부하 대비 지연 시간 백분위수를 그래프로 표시해야 합니다. 배치 처리를 적용하면 처리량이 증가함에 따라 전체 p50 지연 시간이 평탄하게 유지되거나 오히려 감소하는 경우가 많습니다. 이는 변곡점까지 발생합니다. 이 변곡점을 기록하고 해당 값 이하로 유지하세요.

지연 시간 SLO(서비스 수준 목표)에 맞춰 배칭을 구성하는 것이 중요합니다. 예를 들어 특정 길이의 요청에 대해 p99 지연 시간을 2초로 보장한다면, 배칭 대기열에서 한 요청이 500밀리초 지연되는 것을 허용할 수 없습니다. 기본적으로 동적 배칭 지연 시간은 과도한 배칭 지연을 피하기 위해 초기 설정 시 p99 지연 시간 요구 사항보다 훨씬 낮은 수준(1~2밀리초 정도)으로 설정해야 합니다.

적응형 배치 지연을 사용하면 배치 지연 값이 낮은 RPS에서는 동적으로 0ms에 가깝게 떨어지고, 피크 부하 시에는 필요에 따라 5~10ms까지 증가할 수 있습니다. 이 적응형 접근 방식은 vLLM 등 다양한 트래픽 패턴에서 SLO 준수를 유지하기 위해 사용됩니다.

배칭은 주로 트래픽이 많은 시나리오에서 이점을 제공합니다. 트래픽이 낮을 경우 시스템은 단일 요청만 처리하므로 지연 시간이 매우 낮습니다. 그러나 고부하 시 시스템은 공격적인 배칭을 적용하여 높은 처리량을 달성할 수 있으며, 대기열 축적을 방지하고 여러 요청에 걸쳐 지연 시간을 분산시켜 전반적인 지연 시간을 개선합니다.

지속적 배치(Continuous Batching)

에서 *비행 중 배치* (*in-flight batching*) 또는 *반복 수준 스케줄링(iteration-level scheduling*)으로도 알려진 연속 배치는, 완전한 시퀀스가 완료될 때까지 기다리지 않고 토큰 생성 반복마다 배치를 재충전함으로써 높은 GPU 활용도를 유지합니다. 완료된 요청을 제거하고 GPU 준비 상태에 전적으로 기반하여 즉시 새로운 요청을 불러옵니다. 이 기술은 채팅 어시스턴트와 같은 저지연 사용 사례에 특히 중요합니다.

동적 배치와 같은 타임아웃 기반 접근법과 달리, 이벤트 기반 연속 배치 전략은 유휴 컴퓨팅 슬롯과 다른 접근법들의 패딩 오버헤드를 제거합니다. 고정된 "최대 지연" 타이머에 의존하지 않음으로써, 연속 배치는 새로운 요청이 진행 중인 배치 중간에 생성 과정에 합류할 수 있게 하며, 그림 16-6에서 보듯이 가장 긴 시퀀스에 의해 차단되지 않습니다.

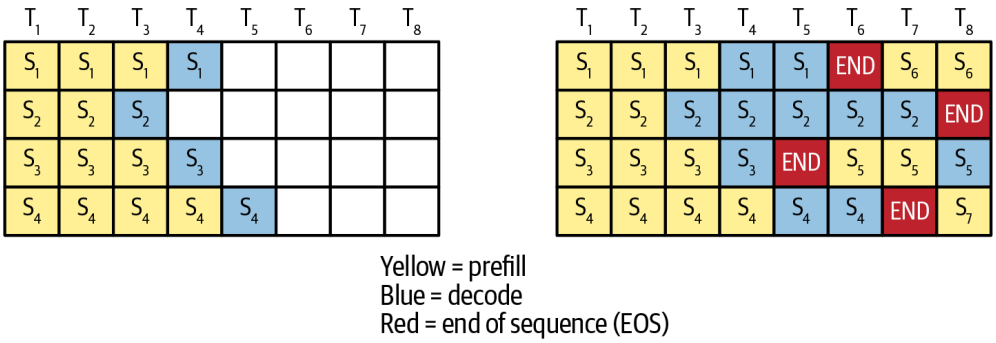

그림 16-6. 연속 배치 처리로 생성 중인 배치 중간에 새로운 시퀀스(요청)가 합류할 수 있음

여기서 연속 배치 방식은 추론 컴퓨팅 파이프라인에서 낭비되는 슬롯을 최소화합니다. 각 배치에서 가장 긴 응답이 완료될 때까지 기다리는 것으로 인한 유휴 시간을 제거합니다. 배치 내 모든 시퀀스 완료를 기다리는 방식(가변 출력 길이로 인해 비효율적이며 GPU 활용도 저하를 초래함) 대신, 연속 배치 처리에서는 각 반복마다 완료된 시퀀스를 새로운 시퀀스로 대체합니다. 이 접근법은 새로운 요청이 GPU 슬롯을 즉시 채울 수 있게 하여 더 높은 처리량, 감소된 지연 시간, 그리고 더 효율적인 GPU 활용도를 제공합니다.

대규모 모델에서 4~8개 요청을 배치하면 1~2개 요청 배치 대비 처리량이 2~3배 증가하는 경우가 많습니다. 이는 GPU의 연산 유닛과 메모리 파이프라인 활용도가 향상되기 때문입니다. 또한 요청당 추가 지연 시간은 수 밀리초 수준에 불과하여 저지연 사용 사례에 이상적인 전략입니다. 표 16-4는 지금까지 논의된 다양한 배치 전략(단순 정적 배치 전략 포함)을 요약합니다.

표 16-4. 정적, 동적, 연속 배치 방식비교

측면	정적 배치	동적 배치	지속적 배치
트리거	고정 배치 크기 또는 시퀀스 길이	배치 크기 목표 또는 타임아웃	토큰 생성 완료 이벤트
지연 시간 바운디드	제한 없음; 전체 배치 완료까지 대기	바운디드 `max_batch_delay_ms`	최소; 배치 중간에 제거 및 재충전
패딩 오버헤드	높은 오버헤드: 가장 긴 시퀀스까지 패딩	중간 오버헤드: 각 배치 내에서 패딩	낮음: 패딩 대기 없이 슬롯 재충전
GPU 유휴 시간 완화	불량: 특히 가변 크기 로드 시	개선됨: 단, 소규모 배치에서 타이머가 작동할 경우 GPU 유휴 발생 가능	우수: 모든 반복자에서 GPU를 포화 상태로 유지
구현	간단함	중간: 타이머 및 큐 관리 필요	복잡: 토큰별 조정 및 상태 추적이 필요

지속적 스케줄링

요청 스케줄링의 또 다른 차원은 모델의 동시 실행과 관련됩니다. vLLM 커뮤니티에서는 이를 *연속 스케줄링이라고* 부릅니다. 핵심 아이디어는 모델이 작거나 지연 시간 목표에 의해 배치 크기가 제한될 경우, 단일 모델 인스턴스만으로는 GPU를 완전히 포화시키지 못할 수 있다는 점입니다.

지속적 스케줄링은 GPU를 OS 스케줄러가 CPU를 다루는 방식과 유사하게 처리합니다. 독립적인 소규모 커널을 별도의 CUDA 스트림에 실행합니다. 이를 통해 하드웨어 워프 스케줄러는 명시적 컨텍스트 전환 없이 작업 간 워프를 교차 실행할 수 있습니다.

예를 들어, 추론 엔진이 GPU를 완전히 포화시키지 못할 경우 동일한 GPU에서 여러 모델 추론 스트림을 동시에 실행할 수 있습니다. 이렇게 하면 한 인스턴스가 데이터 전송이나 비-GPU 작업을 기다리는 동안 다른 인스턴스가 실행될 수 있습니다.

특히 디코딩 단계에서 한 번에 하나의 토큰만 생성하면 GPU 활용도가 떨어지는 경우가 많습니다. 단일 텍스트 시퀀스에서 수행할 때 특히 그렇습니다. 이는 작업 부하가 상대적으로 작으면서도 계산량 대비 메모리 이동이 많이 필요하기 때문입니다.

이러한 비효율성을 해결하기 위해 추론 엔진은 서로 다른 사용자 요청에서 발생한 여러 디코딩 작업을 교차 실행할 수 있습니다. 예를 들어, 10명의 사용자가 동시에 서로 다른 시퀀스를 디코딩하는 경우, 단순히 하나의 큰 행렬 곱셈으로 배치할 수 없습니다. 각 시퀀스의 길이가 다를 수 있기 때문입니다. 이 경우 GPU가 효율적인 행렬 연산을 수행하려면 많은 패딩이 필요해집니다.

대신 연속 스케줄러는 각 시퀀스의 다음 토큰에 대해 10개의 작은 디코드 커널을 별도의 CUDA 스트림에서 빠르게 연속 실행할 수 있습니다. 이는 GPU의 세분화된 워프 스케줄러를 활용하여 시퀀스 간 진정한 동시성을 가능하게 합니다. 따라서 커널은 SM 간 실행을 교차 실행합니다. 이는 라운드 로빈 처리 패턴을 근사화하여 유휴 사이클을 방지합니다.

각 디코딩 작업을 별도의 스트림에 큐잉함으로써 GPU는 시퀀스 간 메모리 전송과 계산을 중첩할 수 있습니다. 이는 토큰당 지연 시간을 줄이고 전체 처리량을 향상시킵니다.

GPU는 워프 수준에서 커널 간 전환을 수행합니다. CPU의 비효율적인 제어 흐름이나 네트워크 I/O 대기 때문에 하나의 디코드 커널이 정지하더라도, 다른 활성 커널은 즉시 진행할 수 있습니다. 이는 GPU를 완전히 활용하도록 유지합니다.

지속적 스케줄링은 실행 준비 완료 토큰 작업의 대기열을 유지함으로써 토큰 단위 세분화에서도 높은 활용도를 달성합니다. 최종 결과는 배치 처리와 유사합니다. GPU는 항상 토큰 생성에 작업 중이지만, 이를 하나의 거대한 행렬 곱셈으로 결합할 필요는 없습니다. 이는 단순히 토큰 계산 사이에 GPU가 유휴 상태로 방치

되지 않도록 한다는 의미입니다. 이러한 토큰 단위 세분화 스케줄링은 수백만 사용자에 걸친 처리량 극대화에 필수적입니다.

고정된 간격으로 또는 GPU가 여유 상태가 될 때마다, 연속 스케줄러는 대기 중인 다음 토큰 요청을 모두 구성 가능한 크기의 배치로 모은 후 단일 GPU 호출로 배치를 전송합니다. 이는 처리량과 지연 시간의 균형을 맞추는 데 도움이 됩니다.

사용자 정의 스케줄러를 설계할 경우 하이브리드 방식을 고려하십시오: 토큰 스케줄링에 짧은 타이머와 최대 배치 크기를 사용합니다.

---

vLLM 과 SGLang과 같은 최첨단 시스템은 동적 배치와 연속 스케줄링을 결합합니다. 이들의 맞춤형 연속 스케줄러는 GPU 시간의 '버블'을 최대한 활용하는 개념을 중심으로 구축되었습니다.

vLLM의 PagedAttention은 이 하이브리드 접근법의 구체적인 딥러닝 예시입니다. PagedAttention은 어텐션 키-값(KV) 캐시를 슬라이스(페이지)로 분할하고, 활성 시퀀스 간 페이지별 연산을 동적으로 그룹화합니다. 이는 대규모 프리필 GEMM 연산과 신속한 소규모 디코드 커널을 교차 실행함으로써 고부하 상태에서도 거의 100%에 가까운 GPU 활용률을 유지합니다.

vLLM은 블록 수준의 KV 캐시 구조를 효율적으로 활용하여 각 요청의 상태를 별도로 추적합니다. 이러한 세밀한 관리 방식은 실시간으로 워크로드를 지속적으로 압축 및 다중화함으로써 빠른 스트리밍 응답과 최적의 하드웨어 활용도를 제공합니다.

또 다른 예로 SGLang의 RadixAttention( )은 트리 기반 KV 캐시 그룹화를 사용합니다. 이는 유사한 거의 100%에 가까운 GPU 활용도를 달성하며, 사용되지 않은 캐시 페이지에 대한 지연 제거(lazy eviction)를 도입합니다. vLLM의 PagedAttention과 SGLang의 RadixAttention에 대해서는 잠시 후 더 자세히 다룰 예정입니다. 두 접근 방식 모두 오픈 소스이므로 코드에서 직접 스케줄링 구현을 검토할 수 있습니다.

스톨 프리 스케줄링(챕터 프리필)

prompt가 극도로 길 경우( ), 이를 청크로 분할하고 프리필 작업과 디코딩 작업을 교차 실행할 수 있습니다. 이를 *스톨 프리 스케줄링* 또는 *청크 프리필이라고* 합니다.

20K 토큰 prompt를 예로 들어 보겠습니다. prompt를 5K 토큰 챕터로 분할하면 반복당 최대 스톨을 줄이고 반복당 작업을 고정된 토큰 수로 제한할 수 있습니다. 이를 통해 반복당 지연 시간을 예측 가능하게 합니다. 다양한 챕터 크기에 따른 챕터별 프리필 비용은 표 16-5에 나와 있습니다.

표 16-5. 다양한 청크 크기의 청크별 프리필을 사용한 20K 토큰 prompt의비용

청크 크기	챕터 수	청크당 주의 작업 수	총 MLP 토큰
20K	1	20K × (20K + 1) ÷ 2 = 200M	1 × 20K = 20K
10K	2	50M + 150M = 200M	2 × 10K = 20K
5K	4	12.5M + 37.5M + 62.5M + 87.5M = 200M	4 × 5K = 20K

여기서 KV 캐싱으로 인해 전체 컨텍스트에 따라 챕터당 어텐션 비용이 증가합니다. 각 새 챕터는 이전 챕터의 모든 토큰에 대해 자체 토큰에 대한 어텐션을 계산합니다.

자기 주의(self-attention)에서 *길이 N인* 시퀀스에 대한 QK 내적 연산 횟수는 삼각형 분포를 보이며, 대략 N(N + 1) ÷ 2입니다. 이는 레이어당 헤드당 수행되는 Q × K 내적 연산 횟수입니다. 이 비용은 *N에 대해* 이차적(O(N^2))입니다. 따라서 N = 20,000일 때 약 2억 개의 연산이 수행됩니다. 청크화는 이 총량을 줄이지 않습니다. 청크화는 디코더에 작업이 제공되는 시점만 변경할 뿐입니다. 청크형 프리필의 이점은 지연 시간 평활화와 향상된 중첩성이지, 총 어텐션 내적 연산 횟수 감소가 아닙니다.

프리필 자기 주의는 레이어당 헤드당 N(N + 1) ÷ 2 Q × K 개의 내적을 수행합니다. 이 비용은 *N에* 대해 이차적이며 O(N^2)입니다. 청크화 및 파이프라인 프리필은 오버랩과 지연 시간만 변경할 뿐, 주의 작업의 총량은 변경하지 않습니다. 전체 어텐션 비용을 줄이는 유일한 방법은 유효 컨텍스트를 축소하거나 로컬/스파스 어텐션을 사용하는 것입니다. 예를 들어 고정 어텐션 윈도우 *W*를 사용하면 비용을 O(NW)로 줄일 수 있습니다.

요약하면, 청크 프리필 기법은 prompt 프리필 처리와 토큰 생성을 밀접하게 교차 배치하여 스케줄링합니다. 이는 대량 배치 프리필 연산의 높은 효율성과 스트리밍 디코딩의 응답성을 통합합니다. 이를 통해 많은 워크로드에서 높은 처리량을 유지하면서 꼬리 지연 시간을 낮출 수 있습니다.

지연 시간 인식 스케줄링 및 동적 라우팅

지연 시간 인식 스케줄링은 동시 다발적( )으로 들어오는 요청을 분석하고, 동적 배치(dynamic batches)를 생성하며, 배치들을 지연 시간을 최소화하도록 라우팅할 수 있습니다. 두 개의 GPU를 가진 추론 시스템에 동시에 도착하는 여섯 개의 prompt를 생각해 보십시오. prompt 길이는 순서대로 6K, 2K, 6K, 2K, 2K, 2K입니다. 순진한 선입선출(FIFO) 스케줄러와 지연 시간 인식 스케줄러를 비교해 보겠습니다.

먼저 FIFO 스케줄러는 도착 순서에 따라 두 개의 배치를 생성합니다. 배치 1은 [6K, 2K, 6K]로 GPU 1에 전송됩니다. 배치 2는 [2K, 2K, 2K]로 GPU 2에 전송됩니다. 결과는 표 16-6에 요약되어 있습니다.

표 16-6. 길이 [6K, 2K, 6K, 2K, 2K, 2K]의 입력 prompt 시퀀스에 대한FIFO 대 지연 시간 인식 스케줄링

GPU	prompt 배치	셀프 어텐션 QK 연산 T(N) = N(N + 1) ÷ 2	토큰 *N*
GPU 1	6K, 2K, 6K	T(6K) + T(2K) + T(6K) = 18,003,000 + 2,001,000 + 18,003,000 = 38,007,000	14K
GPU 2	2K, 2K, 2K	3 × T(2K) = 3 × 2,001,000 = 6,003,000	6K

여기서 FIFO 전략은 처음 세 prompt [6K, 2K, 6K]를 GPU 1로 전송하여 약 38,007,000회의 자기 주의 Q × K 내적 연산과 14,000 토큰의 MLP를 수행하는 반면, GPU 2는 약 6,003,000회의 자기 주의 Q × K 연산과 6,000 토큰의 MLP를 처리합니다. 크리티컬 패스는 GPU 1에서 약 38,007,000개의 자기 주의 Q × K 내적 연산으로 결정됩니다.

대조적으로, 지연 시간 인식 스케줄러는 6개의 요청에 대한 예상 지연 시간을 분석하고 이를 [2K, 2K, 6K]의 두 배치로 재구성합니다. 이 전략에서 GPU당 자기 주의 비용은 표 16-7에 표시된 바와 같이 22,005,000개의 내적과 10,000개의 토큰 MLP입니다.

표 16-7. 길이 [6K, 2K, 6K, 2K, 2K, 2K] prompt에 대한지연 시간 인식 스케줄링 (해당 순서)

GPU	prompt 배치	셀프 어텐션 QK 연산 T(N) = N(N + 1) ÷ 2	토큰 *N*
GPU 1	2K, 2K, 6K	T(2K) + T(2K) + T(6K) = 2,001,000 + 2,001,000 + 18,003,000 = 22,005,000	10K
GPU 2	2K, 2K, 6K	T(2K) + T(2K) + T(6K) = 2,001,000 + 2,001,000 + 18,003,000 = 22,005,000	10K

이를 통해 크리티컬 패스 셀프 어텐션이 38,007,000에서 22,005,000으로 약 42% 감소합니다. 따라서 지연 시간 인식 스케줄러는 TTFT를 크게 줄일 수 있습니다. 지연 시간 인식 스케줄러는 prompt 길이, 셀프 어텐션의 삼각형 N(N + 1) ÷ 2 비용, MLP의 O(N) 비용을 인식하므로 컴퓨팅 가중치를 균형 있게 조정하고 전체 지연 시간을 최소화할 수 있습니다. 예를 들어, 이 예시에서 지연 시간 인식 스케줄러는 2K 토큰 요청을 더 앞당겨 처리합니다. 이를 통해 가벼운 요청들은 프리필을 더 빨리 완료하고, 무거운 6K 토큰 프리필이 완료되기를 기다리지 않고 디코딩을 시작할 수 있습니다.

이 계산은 유효 토큰에 대해서만 연산하는 가변 길이 융합 어텐션 커널(예: FlashAttention 또는 cuDNN 백엔드를 사용한 PyTorch 스케일드 도트 프로덕트 어텐션(SDPA))을 가정합니다. 커널이 배치 내 최대 시퀀스 길이로 패딩하는 경우, 어텐션 연산 비용은 배치 크기 × 해당 배치 내 최장 시퀀스의 T개 토큰에 근접합니다. 이 경우 FIFO와 지연 시간 인식 전략 간의 차이는 줄어듭니다.

간단히 말해, 배치 내 도착 순서가 전역적으로 이상적이지 않을 경우 단순한 선입선출(FIFO) 스케줄러는 GPU 중 하나에 과부하를 줄 수 있습니다. 지연 시간 인식 스케줄러는 들어오는 요청을 분석하고 이를 더 균형 잡힌 배치로 재배열하여 지연 시간을 최소화할 수 있습니다.

일부 추론 서비스 프레임워크는 스케줄링 정책을 온라인으로 조정하기 위해 강화 학습을 통합했습니다. 이는 변화하는 트래픽 패턴에 자동으로 적응함으로써 여기서 설명한 지연 시간 인식 전략을 확장합니다.

시스템 수준 최적화

시스템 수준 최적화 기법 에는 계산과 통신 중첩, GPU 활용도 극대화, 전력 및 열 관리, 오류 처리, 메모리 액세스 최적화 등이 포함됩니다. 전반적으로 GPU 하드웨어가 거의 100%의 시간 동안 유용한 작업(goodput)을 수행하도록 보장합니다. 또한 처리량과 지연 시간 간의 상충 관계와 프로덕션 SLO에 적합한 균형을 찾는 방법에 대해서도 논의합니다.

통신과 연산의 중첩

이 책에서 반복적으로 살펴본 바와 같이, 대규모 모델을 다수의 GPU에 분산하여 추론 성능을 구현하려면 계산과 통신을 중첩시키는 것 이 매우 중요합니다. GB200/GB300 NVL72와 같은 시스템의 엄청난 대역폭(랙 내 GPU 간 NVLink 총 대역폭 최대 ~130TB/s)은 데이터 전송이 충분히 빨라 계산이 대기 상태에 빠질 가능성이 낮아지므로 중첩이 더욱 효과적임을 의미합니다. 그러나 NVL72를 사용하더라도 다중 CUDA 스트림과 비차단 집단 연산을 활용하는 것이 중요합니다. 이를 통해 통신-계산 중첩을 수행하여 지연 시간을 숨기고 ,이러한 속도에서도 72개의 GPU 모두를 바쁘게유지할 수 있습니다.

실제 적용 시 NCCL의 GPUDirect RDMA 지원을활성화 하고,NCCL의 그룹 호출을 활용하여 여러 개의 작은 올-리듀스(all-reduce) 연산을 중첩 실행하는 것이 좋습니다. 또한 SHARP(적합한 스위치가 있는 InfiniBand에서 사용 가능)를 사용하여 축소 연산을 네트워크 패브릭으로 오프로드하는 것을 고려하십시오. 이 최적화는 일부 패브릭 및 토폴로지에서 처리량을 향상시킬 수 있습니다. 일부 테스트에서 대규모 전수 통신에 대해 약 10%~20%의 처리량 개선을 보였습니다.

GPU 간 데이터 전송이 필요한 추론 단계(예: MoE의 전수-전수 전송)나, 단순히 prompt 데이터를 CPU에서 GPU로, 응답을 GPU에서 CPU로 전송하여 최종 사용자에게 반환하는 경우를 생각해 보십시오. 이러한 모든 시나리오에서 가능한 한 GPU가 다른 작업을 수행하는 동안 이러한 전송을 비동기적으로 수행하는 것이 바람직합니다.

기본적인 예로 별도의 CUDA 스트림을 사용해 연산과 데이터 전송을 중첩시키는 방법이 있습니다. 예를 들어, 전용 전송 스트림(고정된 호스트 메모리 포함)에서 `cudaMemcpyAsync` 을 실행하는 동시에 기본 스트림에서 연산 커널을 실행합니다.

PCIe 또는 NIC를 통한 전송 시 페이지 잠금 호스트 버퍼를 사용해야 합니다. 이렇게 하면 CUDA 드라이버가 직접 DMA를 수행하여 컴퓨팅과 병렬 처리가 가능합니다.

그런 다음 데이터가 필요할 때만 CUDA 이벤트로 동기화합니다. 이렇게 하면 GPU가 I/O 대기 상태에 빠지는 것을 방지할 수 있습니다. 이 방식으로 데이터 전송은 비차단 CUDA 호출(예: 고정 메모리)과 CUDA 스트림을 사용하므로 GPU가 이러한 작업 대기 상태에 걸리지 않습니다. 풀 듀플렉스 NVLink 대역폭을 지원하는 최신 GPU에서는 이러한 작업 중첩으로 통신 지연 시간을 크게 숨길 수 있습니다. 다만 특정 워크로드에 대한 자질을 통해 이를 반드시 검증해야 합니다.

이러한 맥락에서 데이터 전송에는 CPU에서 GPU로 새로운 prompt 임베딩을 보내는 작업이나 GPU에서 CPU로 마지막 토큰의 로짓을 이동하는 작업이 포함될 수 있습니다. 예를 들어, GPU가 새로운 토큰 배치에 대한 로짓을 계산한 직후, 해당 로짓을 후처리(예: 샘플링)를 위해 CPU로 비동기 복사하거나 스트리밍 응답으로 클라이언트에 다시 전송하는 작업을 시작할 수 있습니다. 동시에 GPU에 남아 있는 다른 토큰 배치를 위한 다음 컴퓨트 커널을 즉시 실행할 수 있습니다.

다중 노드 환경에서는 전-전 통신과 같은 통신 콜렉티브 작업을 쪼개어 병렬 처리할 수 있습니다. 일부 MoE 런타임에서 사용하는 기법은 토큰 배치를 두 부분으로 분할한 후, 전문가들이 첫 번째 부분의 토큰을 처리하는 동안 두 번째 부분의 토큰 전송을 시작하는 것입니다.

전문가들이 첫 번째 절반을 처리하는 동안 두 번째 절반의 데이터는 대상 GPU에 도착하게 되므로, 대기 없이 작업을 진행할 수 있습니다. 이를 위해서는 컴퓨트 커널에 대한 NCCL 호출의 신중한 스케줄링이 필요합니다.

예를 들어, CUDA 이벤트를 사용하여 배치 절반이 완료되었음을 알리는 방식으로 이 중첩을 구현할 수 있습니다. 이때 해당 데이터 부분에 대해 NCCL 올투올(all-to-all)을 실행할 수 있으며, 동시에 다음 부분의 계산이 완료됩니다. CUDA 그래프를 사용하면 이러한 비동기 패턴을 포착하고 런치 오버헤드를 줄일 수 있습니다. 그 결과 GPU가 통신을 기다리며 유휴 상태로 있는 시간이 줄어들어 GPU 활용도가 향상됩니다.

또 다른 중첩 영역은 CPU와 GPU 간입니다. 예를 들어 GPU가 다음 토큰 생성에 바쁠 때 CPU는 동시에 다음 입력을 준비하거나 이전 생성된 토큰의 결과 처리를 수행할 수 있습니다. 고도로 최적화된 추론 엔진은 CPU 측 전처리(예: 입력 텍스트 토큰화)를 다른 요청의 GPU 계산과 병렬로 중첩합니다. 예를 들어 엔진은 현재 배치의 GPU 계산과 중첩하면서 다음 배치를 준비할 수 있습니다.

이는 당연해 보이지만, 하나의 스레드가 네트워킹과 큐잉을 처리하는 동안 다른 스레드가 GPU 작업을 시작하는 멀티스레드 아키텍처가 필요합니다. 또 다른 스레드는 응답 후처리 등을 처리할 수 있습니다. Python 또는 C++ 추론 루프는 병렬로 수행될 수 있는 작업을 기다리는 이유로 GPU가 새 작업을 시작하는 것을 절대 차단해서는 안 됩니다.

파이프라인 병렬 시나리오에서는 시스템이 (파이프라인 단계 간) 통신과 각 단계 내 계산을 중첩시켜야 합니다. 예를 들어, GPU 0은 토큰 t에 대한 작업을 완료한 후, 다른 시퀀스의 토큰 t+1 처리를 시작하는 동시에 GPU 1로 활성화 값 전송을 시작할 수 있습니다.

현대적인 상호 연결 기술과 프레임워크는 서로 다른 리소스를 사용하는 한 컴퓨팅 커널과 데이터 전송이 중첩되도록 허용합니다. 추론 파이프라인에서는 각 파이프라인 단계가 독립적인 스트림을 가질 수 있도록 여러 CUDA 스트림을 사용해야 합니다. 이렇게 하면 활성화 값의 송수신이 서로 다른 마이크로배치를 처리하는 다른 스트림과 병렬로 수행될 수 있습니다. 그 결과 파이프라인 버블이 감소합니다.

네트워킹 측면에서는 NVIDIA GPUDirect RDMA와 같은 기술을 통해 InfiniBand NIC 같은 네트워크 어댑터가 CPU 개입 없이, 호스트 메모리를 경유하지 않고 GPU 메모리를 직접 읽고 쓸 수 있습니다. KV 캐시나 전문가 활성화 값의 노드 간 전송에 GPUDirect를 활용함으로써 추론 엔진은 지연 시간과 CPU 오버헤드를 줄입니다.

GPUDirect RDMA는 호스트 메모리를 통한 스테이징을 제거하고 NIC DMA 엔진이 GPU 메모리에 직접 접근할 수 있게 합니다. CPU는 작업 요청을 게시하지만 데이터 경로는 CPU를 우회합니다. 이로 인해 CPU 코어는 다른 작업에 활용될 수 있습니다.

두 NVL72 랙 간 MoE 레이어에서 모든-모든 통신에 InfiniBand를 사용하는 실제 사례를 고려해 보십시오. 먼저 모든-모든 통신을 수행한 후 연산을 동기식으로 처리하면 GPU 활용도가 중첩 없이 낮게 유지될 수 있습니다. 그러나 모든-모든 통신 배치와 연산을 중첩하면 GPU 활용도를 크게 높일 수 있습니다.

MoE 계층에서 전-전 통신을 중첩하려면 토큰 배치를 반으로 분할하고, 첫 번째 반의 전문가들이 아직 계산 중인 동안 두 번째 반의 교환을 시작합니다. 이렇게 하면 통신 지연 시간을 계산 작업과 교차 배치하여 숨깁니다.

본질적으로 각 GPU는 이미 보유한 토큰에 대한 로컬 전문가 출력 계산을 시작하는 동시에 다른 노드로부터 남은 토큰을 수신합니다. 첫 번째 배치를 완료할 때쯤

이면 두 번째 배치가 도착하여 즉시 계산할 수 있습니다.

이러한 최적화는 상당히 저수준이며 NCCL 그룹과 CUDA 스트림 간의 CUDA 이벤트 동기화를 포함하지만, 상당한 처리량 향상과 더 부드러운 지연 시간을 제공합니다. 실제로는 먼저 NCCL 그룹의 일부로 NCCL 전수 통신을 완료를 기다리지 않고 실행합니다. 그런 다음 즉시 다음 컴퓨트 커널을 실행합니다. CUDA 이벤트를 사용하여 완료 여부를 반드시 확인하세요.

스트리밍 출력을 위해 I/O와 컴퓨팅을 중첩할 수도 있습니다. 예를 들어, 몇 개의 토큰이 생성되는 즉시 소켓을 통해 최종 사용자에게 전송합니다. 이때 모델은 이미 다음 토큰을 처리 중입니다. 이렇게 하면 네트워크 지연 시간(예: 최종 사용자에게 응답을 다시 전송하는 시간)이 컴퓨팅 작업(예: 후속 토큰 처리) 뒤에 숨겨집니다. 따라서 최종 사용자는 중단 없이 안정적인 스트림을 경험합니다.

이러한 클라이언트에 대한 비차단 스트리밍 패턴은 모든 현대적 LLM 추론 엔진에서 채택됩니다. 이들은 SSE/웹소켓을 사용하여 클라이언트에 토큰 업데이트를 전송하기 위해 별도의 스레드를 사용합니다.

스트리밍 출력을 직접 구현하는 경우, 생성된 토큰을 네트워킹 스레드에 전달할 때 스레드 안전 큐나 잠금 장치를 반드시 사용하십시오. 이 전달 과정에서 동기화 문제가 발생하지 않도록 해야 합니다. 이러한 문제는 식별 및 디버깅이 매우 어려울 수 있습니다.

중첩 처리하지 않았다면 소량의 토큰만 생성한 후, 해당 토큰을 최종 사용자에게 전송하는 동안 GPU가 유휴 상태로 남아 있었을 것입니다. 대신, 네트워크 전송은 응답 버퍼의 출력을 가져와 사용자에게 스트리밍하는 비동기 스레드에서 처리됩니다. 이를 통해 메인 추론 스레드(예: CUDA 스트림)는 즉시 더 많은 토큰 생성을 계속할 수 있습니다. 이는 효과적으로 토큰 생성(컴퓨팅)과 출력 전송(I/O)을 파이프라인화합니다.

요약하자면, 통신과 연산을 중첩시키려면 파이프라인 개념으로 사고하고 가능한 한 비동기 작업을 사용해야 합니다. 현대 GPU와 네트워킹 하드웨어는 이를 구현하기 위한 기본 기능을 제공합니다. 여기에는 CUDA 스트림, 비차단 집단 연산(nonblocking collectives), RDMA 등이 포함됩니다. CUDA 파이프라인 헤더의 `CUDA_STREAM_INIT_PIPELINE_START`(`cuda::memcpy_async`)와 `CUDA_STREAM_INIT_PIPELINE_END`(`cuda::barrier`)와 같은 CUDA 장치 측 기본 기능은 커널 내 연산과 글로벌 메모리에서 공유 메모리로의 이동을 중첩시키는 데 도움이 됩니다. (참고: 호스트-디바이스 및 디바이스-호스트 전송은 여전히 명시적 CUDA 스트림과 고정된 호스트 메모리가 필요하며, 이를 통해 연산과 중첩됩니다.) 효율적인 LLM 추론 시스템은 이러한 기능을 최대한 활용합니다.

요약하자면, 가능한 한 항상 통신과 계산을 병렬 처리하십시오. 그렇지 않으면 NVLink/NVSwitch를 사용하더라도 다중 GPU로 확장 시 하드웨어 최고 성능에 미치지 못할 수 있습니다. 이러한 최적화는 모델 출력에 변화가 없으므로 최종 사

용자에게는 대부분 보이지 않습니다. 그러나 추론 클러스터의 성능을 극대화하고 사용자가 애플리케이션을 계속 이용하도록 하는 데 필수적입니다.

GPU 활용도 극대화와 지연 시간과의 상충 관계

성능 튜닝의 궁극적 목표는 GPU가 행렬 곱셈과 같은 유용한 작업을 최대한 바쁘게 수행하도록 하면서, 유휴 상태나 활용도가 낮은 GPU 자원을 최소화하는 것입니다. 앞서 설명한 배치 처리, 병렬화, 중첩 처리 등 많은 기법들은 GPU 활용도를 거의 100%에 가깝게 달성하기 위해 고안되었습니다.

GPU 활용률, 특히 유용한 활용도(goodput)는 지속적으로 모니터링해야 할 중요한 성능 지표입니다. 유용한 활용도를 100%로 끌어올리려는 동시에 지연 시간과 같은 서비스 수준 목표(SLO)를 위반하지 않도록 주의해야 합니다. 어느 시점부터는 수익 감소의 한계점에 도달할 수 있습니다.

현재 단순 구현으로 GPU 활용도 60%를 보이는 추론 서버를 고려해 보십시오. 조사 결과 디코딩 단계가 병목 현상인 것으로 나타났는데, 이는 단일 스레드가 모든 시퀀스를 순차적으로 처리하기를 기다리기 때문입니다. 여러 스트림을 사용한 동시 디코딩을 도입해 보겠습니다. 앞서 설명한 대로 디코딩을 인터리빙함으로써 GPU 활용도를 95%로 높이고 처리량을 두 배로 증가시킵니다. 이는 동시 디코딩이 작동함을 확인시켜 줍니다.

대규모 배치에 더 많은 요청을 넣어 100%에 도달하려 할 수 있지만, 이는 개별 쿼리 속도를 저하시킵니다. 다양한 배치 크기를 테스트하여 처리량 대 지연 시간 곡선을 작성하는 것이 종종 유용합니다. 이 차트에서는 처리량 증가가 지나친 지연 비용을 초래하기 시작하는 지점에서 급격한 '무릎' 현상이 나타납니다.

처리량-지연 시간 곡선의 최적점을 찾으려면 먼저 완전한 동시성을 활성화하고 가능한 한 많은 작업을 중첩시켜 하드웨어가 지속적으로 가동될 수 있도록 합니다. 그런 다음 배치 크기를 점진적으로 늘려 최대 자원 활용도에 도달할 때까지 진행합니다. 이 시점에서 단일 쿼리 지연 시간을 측정하고 응답 시간 목표를 충족할 만큼만 (예: 16에서 8로) 축소합니다.

p50 중앙값 지연 시간과 함께 p95 및 p99 꼬리 지연 시간을 모니터링하는 것이 권장됩니다. 이는 작은 지터와 불균일한 배치 채우기가 p95/p99에서 눈에 띄는 장꼬리 이상치를 생성할 수 있기 때문입니다. 많은 요청을 처리하는 대규모 클러스터에서는 0.1%의 아웃라이어도 빈번하게 발생할 수 있습니다. 따라서 사용자 경험 측정에 있어 p50보다 p99 또는 심지어 p99.9가 더 중요할 수 있습니다. 또한 롱테일 지연 시간은 공격적인 SLO(서비스 수준 목표)를 충족시키기 위해 과도한 프로비저닝을 강요하는 경우가 많습니다. 따라서 테일 지연 시간을 줄이는 것은 직접적인 비용 절감 효과를 가져옵니다.

경험상 좋은 규칙은 최대 처리량에 도달하는 배치 크기보다 약간 낮은 수준으로 제한하는 것입니다. 팀들은 일반적으로 최대 피크 처리량의 90%를 목표로 하며, *이를 헤드룸 버퍼라고 부릅니다*. 이는 최대 속도로 실행할 경우 예측 불가능한 지연 시간 급증이 발생할 수 있기 때문입니다. 예를 들어, 배치 크기를 16개 요청으로 늘렸을 때 일부 요청에 소량의 지연이 발생하기 시작하면 배치 한도를 8개로 줄여야 합니다. 이렇게 하면 더 일관된 지연 시간을 제공하면서 처리량은 약간만 감소합니다.

일부 추론 시스템은 이러한 지표를 기반으로 배치 크기를 동적으로 조정할 수 있습니다. 이 기술과 다양한 적응형 추론 전략은17~ 19장에서다룰 예정입니다.

GPU를 지속적으로 100%로 가동하면 전력 및 열 한계에 도달하여 스로틀링이 발생할 수 있다는 점을 기억하는 것이 중요합니다. 효율적인 커널로 90%로 가동하는 것이 스로틀링이 발생하는 100% 가동보다 성능이 더 우수할 수 있습니다. 다음으로, CPU 기반 애플리케이션 개발자와 시스템 엔지니어가 종종 고려하지 않는 특성인 GPU 전력 및 열 제약에 대해 논의해 보겠습니다.

전력 및 열 제약 조건

고성능 컴퓨팅( )의 또 다른 고려 사항은 전력 및 열 제약입니다. 냉각이 불충분한 상태에서 GPU를 지속적으로 100%로 가동하면 열 스로틀링이 발생합니다. 현대 GPU 시스템은 열 스로틀링을 줄이고 성능을 유지하기 위해 수냉식입니다. 구형 공랭식 시스템을 사용 중이라면 클럭 다운 현상에 주의하세요.

GPU 사용률이 높을 때 `nvidia-smi` 또는 DCGM으로 다운클럭을 모니터링할 수 있습니다. 특히 DCGM은 `DCGM_FI_DEV_CLOCK_THROTTLE_REASONS` 를 통해 XID 스로틀링 원인을 노출합니다. 또한 `nvidia-smi` 는 GPU가 전력 스로틀링될 때 "Pwr Throttle" 플래그를 표시합니다.

보드의 전력 한계 에 도달할 수도 있습니다—특히 부스트 클럭 시에 그렇습니다. 최신 GPU는 부스트 상태에서 훨씬 더 많은 전력을 소모하므로, 전력 한계 도달로 인해 GPU가 다운클럭되는지 모니터링해야 합니다. 이런 상황이 발생하면 GPU 성능이 저하됩니다. 항상 DCGM의 `DCGM_FI_DEV_POWER_USAGE` 메트릭을 모니터링하고 정상 범위를 초과할 때마다 경보를 발령하십시오.

GPU가 지속적으로 전력 제한에 도달한다면, 지연 시간 급증을 유발할 수 있는 열 제한을 피하기 위해 다이내믹 부스트 모드 활성화 또는 약간의 언더클럭킹을 고려하십시오.

소프트웨어적으로 이러한 제약을 우회하려면, 배치 간 지연을 약간 추가하여 활용도를 완화할 수 있습니다. 동시 실행 수를 제한하는 방법도 있습니다. 현대 GPU는 클럭 제한 기능도 지원합니다. 따라서 지연을 추가하기보다 `nvidia-smi -lgc` 등을 사용하여 GPU 클럭을 약간 제한함으로써 열 제한에 도달하는 것을 방지할 수 있습니다. 이는 처리량을 약간만 감소시키면서 더 일관된 성능을 제공합니다.

하드웨어 측면에서는 냉각 성능을 개선하거나, 냉각이 이미 충분한 경우 전력 한도를 높일 수 있습니다. 전력 한도를 높이려면 `nvidia-smi -pl` 를 사용하거나 GPU 부스트 설정을 조정하세요.

이 모든 것은 실제 시스템을 한계까지 밀어붙일 경우 성능에 큰 영향을 미치는 예상치 못한 부작용이 발생할 수 있음을 상기시켜 줍니다. 시스템이 전력 또는 열 제약으로 인한 스로틀링으로 성능 저하를 감지할 때 소프트웨어 해결책을 즉시 적용할 수 있도록 추론 엔진에 "부스트 오프(boost-off)" 플래그를 포함하는 것이 권장됩니다. 이렇게 하면 완전한 안정성과 성능이 회복될 때까지 시스템을 약간 저활용 상태로 더 낮은 온도에서 운영할 수 있습니다.

오류 처리

일반적으로 성능과 연관되지는 않지만, 오류 처리를 효율적으로 수행하는 것이 중요합니다. 완전히 활용된 추론 시스템은 과도한 오류 급증을 처리할 여유가 적습니다. 특히 오류 처리는 시스템 내에서 가장 최적화된 코드 경로가 아닐 가능성이 높기 때문입니다.

실패 시나리오에서는 빠른 실패(failing fast)가 핵심입니다. 요청이 오류가 발생할 경우, 느린 실패로 GPU 시간을 낭비하기보다는 즉시 오류를 반환하는 것이 최선이기 때문입니다. 인퍼런스 호출 주변에 적절한 예외 처리 및 타임아웃을 구현하여 중단이나 충돌을 포착하도록 하십시오.

또한 백프레셔(backpressure) 구현을 권장합니다. 이는 오류가 급증할 경우 새로운 요청을 거부하거나 배치 크기를 줄여 시스템이 회복할 여유를 확보할 수 있음을 의미합니다.

대규모 환경에서는 여유 공간을 확보하기 위해 대부분 유휴 상태로 유지되는 추가 복제본을 구축하는 것이 좋습니다. 이는 오류 급증이나 기타 예상치 못한 상황에 대한 완충 역할을 할 수 있습니다. 비용 절감을 위해 자동 확장(autoscaling)을 먼저 고려할 수 있지만, 자동 확장은 새 리소스를 프로비저닝하는 데 시간이 소요된다는 점을 기억하세요.

유휴 용량에도 비용이 발생하지만, 고객 손실이나 SLA 위반으로 인한 비용은 종종 더 큽니다. 정상 상태에서는 최소 10~15%의 완충 용량을 확보하는 것이 권장됩니다. 가동 중단을 감당할 수 없는 더 중요한 서비스의 경우, 100% 완충 용량(또는 전체 클러스터 복제본)을 프로비저닝하는 것이 좋습니다.

요청 즉시 추가 부하를 처리할 수 있는 사전 예열된 유휴 노드를 대체할 수 있는 것은 없습니다. 최종 사용자를 잃는 비용은 예상치 못한 추가 부하를 처리할 준비가 된 몇 개의 복제본을 유휴 상태로 유지하는 비용보다 훨씬 높을 가능성이 큽니다.

메모리

자원 활용도 최적화는 메모리 측면에서도 적용됩니다( ). GPU 메모리와 메모리 대역폭은 점진적으로 증가하는 반면, 모델 규모와 컨텍스트 길이는 기하급수적으로 증가하고 있습니다. 따라서 메모리는 최적화가 필요한 귀중한 자원이며, 가까운 미래에도 그 중요성이 지속될 것입니다.

따라서 GPU 메모리와 메모리 대역폭을 효과적으로 활용해야 합니다. 메모리는 일반적으로 모델 가중치와 KV 캐시로 채워집니다. 추론 중에는 모델 가중치를 항상 GPU HBM에 유지하는 것이 최선입니다. CPU DRAM이나 NVMe 스토리지로 메모리를 페이지 인/아웃하면 추가적인 페이지 결함과 전송 지연이 발생합니다.

이러한 추가 전송 지연은 Grace Blackwell이나 Vera Rubin 같은 최신 CPU-GPU 슈퍼칩에서도 마찬가지입니다. 따라서 고성능 LLM 추론 엔진은 주문형 페이징에 의존하기보다 메모리를 명시적으로 관리합니다.

압축은 추론 시스템의 메모리 사용량을 줄이는 효과적인 기법입니다. 특히 시스템으로 들어오는 모든 쿼리에 대해 프리필 단계에서 생성되는 KV 캐시에 적용됩니다.

KV 캐시의 크기는 쿼리 수와 입력 크기에 비례하여 확장되므로, KV 캐시 압축 및 양자화를 고려해야 합니다. KV 캐시 압축/양자화란 모델이 정밀도 손실을 허용할 수 있을 경우 정밀도를 낮춘 키와 값을 저장하는 것을 의미합니다. 또한 이상적이진 않지만, KV 캐시 오프로딩은 거의 사용되지 않는 KV 캐시 데이터에 대한 옵션입니다.

KV 캐시 오프로딩 및 메모리 풀 할당

추론 엔진은 드물게 사용되는 분산형 언어 모델( )의 키-값(KV) 캐시 항목을 CPU 메모리나 디스크로 오프로드(페이지 아웃)함으로써 더 활발히 사용되는 데이터를 위한 공간을 확보합니다. 이는 가상 메모리 처리 방식과 유사합니다.

예를 들어, vLLM의 PagedAttention은 관리형 메모리 풀을 활용해 키-값( ) 캐시를 CPU 메모리와 NVMe 스토리지로 오프로드합니다. 마찬가지로 SGLang의 RadixAttention은 트리 구조 캐시를 사용해 가장 적게 사용되는 접두사를 지연 제거(lazy eviction)할 수 있습니다. NVIDIA Dynamo 역시 키-값 캐시 오프로드 및 메모리 관리에 유사한 메커니즘을 적용합니다.

또한 우수한 KV 캐시 할당기가 없다면, 다양한 길이의 요청이 시스템을 통과할 때 과도한 메모리 조각화(예: ~20%–30%)가 발생할 수 있습니다. 이는 OOM(Out Of Memory) 오류가 발생하기 전에 처리할 수 있는 요청 수를 제한합니다. 이전 장에서 논의한 대로 큰 풀 크기를 가진 할당기를 사용하는 것을 잊지 마십시오.

적절한 메모리 페이징 전략을 채택하면 조각화를 허용 가능한 몇 퍼센트 수준으로 줄일 수 있습니다. 이는 GPU에 더 많은 컨텍스트를 압축하여 시스템 충돌 없이 높은 활용도를 유지할 수 있음을 의미합니다.

적절한 KV 캐시 메모리 관리를 통해 현대적인 추론 엔진은 GPU 메모리 부족 없이 훨씬 더 많은 동시 사용자를 처리합니다. 자체 KV 캐시 메모리 풀을 관리하고 데이터를 CPU 및 NVMe 스토리지로 오프로드함으로써 이를 달성합니다. 따라서 최소한의 조각화로 거의 완전한 메모리 활용률을 구현합니다.

단점은 컨텍스트가 다시 활성화되어 페이징 인이 필요할 때 발생하는 약간의 추가 데이터 전송 지연입니다. 그러나 이 오버헤드는 전체 요청 지연 시간에 비해 일반적으로 미미합니다.

메모리 활용도와 컴퓨팅 활용도는 서로 밀접하게 연결됩니다. 메모리 관리가 부실하면 메모리를 낭비하게 되며, 더 많은 동시 작업을 처리할 때 GPU를 완전히 활용할 수 없습니다. 가능한 한 많은 관련 데이터를 GPU에 유지함으로써 시스템은 방대한 수의 동시 요청을 처리할 수 있습니다.

부적절한 메모리 관리는 피크 부하 시 반복적인 OOM(Out of Memory) 충돌을 유발할 수 있습니다. 이로 인해 GPU가 풀에서 제외되고 연쇄적인 지연 문제가 발생합니다. 적절한 메모리 관리를 통해 OOM을 방지하십시오. 이는 활용도를 극대화하고 클러스터 안정성을 유지합니다.

요약하면, 효율적인 메모리 관리는 효과적인 처리량을 향상시키고, 메모리 조각화를 줄이며, 예기치 않은 OOM 오류를 방지하고, 특히 고처리량 시나리오에서 더 많은 동시 요청 처리를 가능하게 합니다.

실시간 추론을 위한 양자화 접근법

추론 성능을 향상시키는 가장 효과적인 방법 중 하나는 정밀도를 낮추는 것입니다. 이는 메모리 사용량과 메모리 대역폭 활용도를 즉시 감소시키고 계산 속도를 높입니다.

양자화는 모델의 가중치(때로는 활성화 값)를 더 적은 비트 수로 표현하는 기술입니다. 최신 NVIDIA GPU는 FP16, FP8, FP4 등 다양한 형식을 지원하는 저정밀도 텐서 코어를 통해 저정밀도 연산을 기본적으로 지원합니다.

이 섹션에서는 추론을 위한 양자화 기법을 구체적으로 논의합니다. 여기에는 GPTQ, AWQ, SpQR 및 기타 구조적 스파스성을 고려한 방법과같은 가중치 전용 양자화 기법과 가중치 및 활성화 양자화를 위한 완전 정밀도 감소 기법이 포함됩니다. 특히 GPTQ와 AWQ는 실제 적용에서 매우 효과적인 것으로 입증되었습니다.

많은 대규모 모델의 경우, GPTQ를 활용한 4비트 가중치 전용 양자화는 FP16 모델의 정확도를 99% 이상 유지할 수 있습니다. 또한 약 2배의 추론 가속화와 약 4배 작은 모델 크기를 제공합니다. AWQ는 3-4비트 양자화에서 정확도를 더욱 향상시킵니다. 이러한 기술은 Hugging Face Transformers, PyTorch, vLLM 등 다수의 AI 프레임워크에 통합되어 있으며, GPTQ 및 AWQ 양자화 모델을 직접 로드할 수 있도록 지원합니다. 다음으로, 정밀도 감소 형식을 사용한 정확도와 성능

의 상충 관계, 그리고 양자화를 효과적이고 안전하게 서비스 워크플로에 통합하는 방법을 살펴보겠습니다.

정밀도 감소: FP16에서 FP8 및 FP4로

초기 LLM 추론은 FP32에서 TF32, FP16, BF16과 같은 정밀도 감소 형식으로 전환함으로써 상당한 성능 향상 을 보였습니다. 예를 들어 NVIDIA 텐서 코어는 FP32 대비 FP16 사용 시 처리량이 2배 증가합니다. 이는 반정밀도 곱셈-덧셈 연산을 특수화된 텐서 코어 하드웨어 파이프라인에 융합하여 수학 성능을 두 배로 높이는 방식으로 이루어지며, 눈에 띄는 정확도 손실 없이 가능합니다.

FP8은 정밀도를 8비트 부동 소수점으로 더욱 낮춥니다. 이는 FP16/BF16 대비 메모리 사용량을 절반으로 줄입니다. 또한 GPU가 16비트 연산 대비 사이클당 두 배 많은 8비트 곱셈-덧셈을 실행하므로 텐서 코어 연산 처리량이 다시 두 배로 증가합니다.

`torch.set_float32_matmul_precision("high")` 로 TF32 연산을 활성화하기만 해도 PyTorch에서 적당한 속도 향상을 얻을 수 있지만, NVIDIA의 Transformer Engine(TE)이 제공하는 8비트 및 4비트 정밀도 지원을 최대한 활용하는 것이 바람직합니다. TE는 FP8 및 FP4 커널을 라이브러리 형태로 제공하여 기존 코드가 최소한의 변경만으로 이러한 저정밀도를 사용할 수 있게 합니다.

NVIDIA의 TE는 이러한 축소된 정밀도에서 텐서별 스케일링 계수를 자동으로 관리합니다. 추론 시에는 추론 서버가 FP16으로 모델을 로드하더라도 FP8 행렬 곱셈을 사용할 수 있습니다.

TE는 수치적 안정성을 유지하기 위해 각 텐서에 스케일링을 적용하며, 일반적으로 두 가지 방식 중 하나로 스케일링 계수를 선택합니다: 훈련 중 대표 데이터를 사용한 고정식 사전 보정 단계(*정적 보정)* 또는텐서의 최대 절대값을 추적하는 동적 계산값(*amax 기반 동적 스케일링*) *입니다.* 그림 16-7은 정밀도 변환을 위한 TE의 범위 분석, 스케일링 계수 및 대상 형식 사용을 보여줍니다.

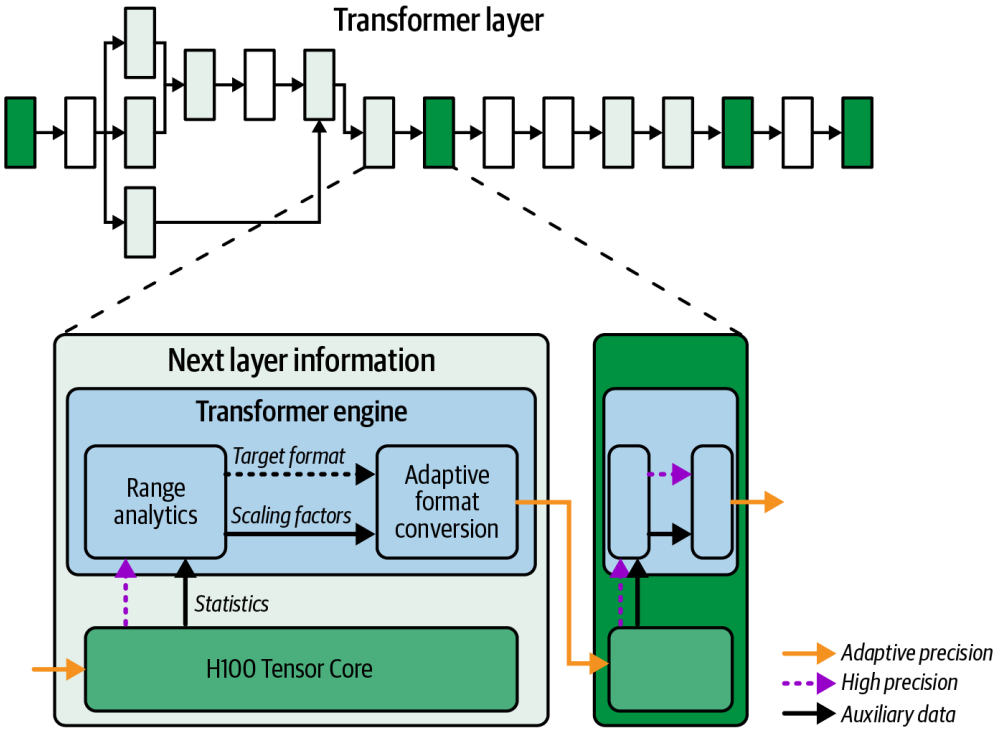

그림 16-7. NVIDIA Transformer Engine(TE)이 트랜스포머 레이어의 정밀도 변환을 위해 범위 분석, 스케일링 계수 및 대상 형식을 사용하는 모습

더 많은 압축을 위해 FP4 형식은 FP8보다 모델 가중치 저장 공간과 전송량을 더 효과적으로 줄입니다. 스케일링 메타데이터 및 패킹을 고려할 때, FP8 대비 효과적 감소율은 일반적으로 약 1.8배, FP16 대비 약 3.5배입니다. 그러나 FP4의 동적 범위가 매우 제한적이기 때문에, FP4에서 신뢰할 수 있는 추론을 위해서는 채널별 스케일링 또는 GPU의 TE에서 지원하는 NVIDIA의 블록별 *마이크로스케일링*과 같은 다른 보정이 필요합니다. 이러한 기법들은 정확도 손실을 최소화하여 FP4를 대규모 네트워크에 활용 가능하게 만드는 데 필수적입니다.

가중치 전용 양자화(GPTQ, AWQ)

현대적인 LLM 서비스 스택에서 가중치 전용 양자화는 일반적으로 GPTQ 또는 AWQ와 같은 방법을 사용하여 가중치를 4비트 정수로 압축하는 동시에 활성화 값은 FP8, FP16 또는 INT8과 같은 더 높은 정밀도로 유지합니다. 이는 FP16 대비 가중치 메모리 사용량을 약 4배 줄이고 가중치 대역폭을 절반 이하로 감소시키며, 일반적으로 적절히 보정될 경우 정확도 손실을 최소화합니다.

NVIDIA의 FP4 구현(공식 명칭은 *NVFP4*이지만 본문에서는 *FP4*로 표기)은 하드웨어에서 블록 단위 마이크로스케일링을 사용합니다. NVIDIA TE는 블록 단위 NVFP4 마이크로스케일링을 위한 하드웨어 지원을 제공합니다.

커널에 따라 텐서별 또는 채널별 스케일링을 사용하십시오. 활성화에는 텐서별 스케일링을, 가중치에는 채널별 스케일링을 적용하는 것이 권장됩니다. 예를 들어, KV 캐시 양자화에 FP8 E4M3를 사용할 때는 일반적으로 텐서별 스케일링을 적용합니다.

블록별 미세 스케일링이란 텐서 전체에 단일 스케일링 계수를 적용하는 대신, 텐서 내 고정 크기 블록(일반적으로 32개 요소)마다 별도의 스케일을 유지하는 것

을 의미합니다. 이러한 개별 스케일은 지역적 값 분포에 적응하여 범위를 보존하고 단일 스케일 양자화에 비해 양자화 오류를 줄입니다.

추론 워크로드를 반영하는 보정 데이터로 항상 양자화하십시오. 훈련 데이터의 일부에 대한 일회성 보정은 런타임 사용 패턴을 포착하지 못할 수 있습니다.

실제 적용에서 GPTQ(훈련 후 양자화)와 AWQ는 LLMs의 4비트 가중치에 흔히 사용되며, 정확도 손실은 무시할 수 있을 정도로 적습니다. Hugging Face 등의 오픈소스 도구를 통해 이러한 기법을 자동으로 적용할 수 있습니다.

MoE 전문가 구조는 정밀도 감소 가중치를 사용해 메모리에 더 많은 전문가를 수용할 수 있으므로 가중치 양자화를 더욱 매력적으로 만듭니다. 따라서 CPU 메모리와 교환해야 하는 전문가 수가 줄어듭니다. 또한 필요 시 더 많은 활성 전문가와 더 많은 전문가 복제본을 가진 대규모 모델을 사용할 수 있습니다.

GPTQ와 같은 훈련 후 양자화(PTQ) 도구는 근사 2차 알고리즘을 적용하여 레이어별로 가중치를 3~4비트까지 양자화합니다. 이는 몇 시간의 GPU 작업으로 거의 정확도 저하 없이 수행됩니다. 최신 GPTQ 변형들은 비대칭 보정과 병렬 연산을 통해 이 알고리즘을 더욱 정교화합니다. 이는 양자화 오류를 줄이고 효율적인 저비트 지원을 더 큰 모델까지 확장합니다.

AWQ는 소수의 "두드러진" 가중치 채널을 식별합니다. 두드러진 채널은 모델 출력에 비례하지 않게 큰 영향을 미치는 큰 활성화 크기를 생성합니다.

AWQ는 모든 가중치를 4비트 정밀도(예: INT4)로 변환하기 전에 채널별 스케일링을 적용하여 이러한 채널을 보존합니다. NVIDIA의 TE는 보존된 채널에 채널별 스케일링 계수를 적용하여 모델 정확도를 유지하는 방법을 알고 있습니다.

PyTorch와 같은 대부분의 AI 프레임워크와 vLLM, TensorRT-LLM과 같은 추론 엔진은 GPTQ 양자화 및 AWQ 양자화 모델 체크포인트 로딩을 기본적으로 지원합니다.

활성화 양자화

와 함께 활성화 값을 양자화하면 GEMM 입력(및 잠재적으로 누산기)에 낮은 정밀도 값을 사용하여 성능을 향상시킬 수 있습니다. 이는 MLP 레이어의 어텐션 기반 KV 캐시와 중간 활성화 값 모두에 대한 메모리 트래픽을 줄여줍니다.

그러나 활성화 분포는 입력값 변화에 따라 크게 달라질 수 있습니다. 따라서 적절한 미세 조정이나 보정 없이는 활성화 양자화가 어려울 수 있습니다. 중간 방안으로 보정된 INT8 활성화를 사용할 수 있습니다. 이는 텐서별 보정을 사용하는 NVIDIA의 INT8 모드에서 비롯되었으며, TensorRT에서 대표 보정 데이터 세트로 생성된 활성화 히스토그램으로부터 스케일링 계수를 선택하는 데 활용됩니다.

8비트 활성화 양자화를 위한 훈련/보정 불필요한 PTQ 방법인 SmoothQuant 는 간단한 행/열 스케일링 알고리즘을 사용하여 양자화 오류를 활성화에서 가중치로 일부 이동시킬 수 있습니다. 이를 통해 최소한의 미세 조정으로 가중치와 활성화 모두에 INT8을 사용할 수 있으며, 정확도 손실(예: < 1%)이 낮은 완전한 INT8 추론을 가능하게 합니다.

가중치에 GPTQ/AWQ를 적용하기 전에 SmoothQuant 활성화 양자화를 사용하면 저정밀도에서 정확도를 더 잘 보존하는 것으로 나타났습니다.

훈련 후 양자화 워크플로

앞서 설명한 양자화 기법 은 모델 훈련 중 수행되는 양자화 인식 훈련(*모델 프리트레이닝*)과 달리 훈련 후 적용됩니다. 일반적인 워크플로는 FP16/FP32로 모델을 훈련 또는 미세 조정한 후, 대표 데이터셋에 GPTQ나 AWQ 같은 훈련 후 보정 스크립트를 실행하여 양자화 매개변수를 결정하는 것입니다. 이후 양자화된 모델 가중치를 추론 엔진에 로드하여 서비스를 제공합니다.

필요 시 소규모 미세 조정 작업을 실행하여 손실된 정확도를 회복할 수도 있으나, GPTQ 및 AWQ 기법에서는 일반적으로 불필요합니다. 완전한 QAT가 필요하다면, 평가 시 저정밀도 연산을 모방하기 위해 모델 그래프에 "가짜 양자화" 연산을 적용한 상태로 몇 에포크만 훈련하면 됩니다. 이를 통해 해당 정밀도에서의 예상 정확도를 측정할 수 있습니다.

실제 적용 시 LLMs에 대한 양자화 인식 미세 조정은 계산 비용이 높고 항상 가능한 것은 아닙니다. 그러나 예를 들어 1,000개의 prompt로 구성된 소규모 보정 데이터셋을 백분위수 클리핑이나 LMS(손실 인식 양자화) 같은 기법과 함께 사용하면 완전한 재훈련 없이도 양자화 스케일을 미세 조정할 수 있습니다.

압축과 모델 견고성 사이의 균형을 맞추는 데 각별히 주의해야 한다는 점을 유의하십시오. 양자화는 특정 오류를 증폭시킬 수 있습니다. 예를 들어, 모델이 사실적 지식의 한계선에 간신히 머물고 있었다면 양자화가 오류를 유발할 수 있습니다.

PTQ는 미묘한 분포 변화를 놓칠 수 있는 가정을 하므로 항상 다운스트림 작업에서 검증해야 합니다. 따라서 정밀도가 낮아진 양자화 모델을 사용할 때는 응답에 대한 광범위한 A/B 테스트를 수행하여 품질이나 안전성 측면에서 퇴보가 없는지 평가해야 합니다. 퇴보가 관찰되면 모델을 더 높은 정밀도로 유지하거나 양자화 형태로 가벼운 미세 조정을 수행하여 성능을 복원해야 합니다.

가중치 및 활성화 양자화 결합

활성화 양자화를 4비트로 수행하는 것은 여전히 어려운 과제입니다. 저정밀도 가중치 양자화와 고정밀도 활성화 양자화를 결합하면 메모리 절감, 연산 효율성, 정확도 사이에서 최적의 균형을 이루는 경우가 많습니다. 따라서 많은 생산 시스템에서는 가중치 전용 4비트 양자화(예: GPTQ/AWQ)와 8비트 활성화 양자화를 함께 사용합니다.

구체적으로, 한 W4A8(8비트 활성화) 변형에서는 런타임이 INT4 가중치를 언패킹하고 학습 또는 보정 스케일을 사용하여 FP8로 양자화 해제하며 FP8 텐서 코어에서 행렬 곱셈을 실행합니다. 이 하이브리드 경로는 TensorRT-LLM과 같은 추론 엔진에서 제공되며 적절히 보정될 경우 거의 무손실 수준의 정확도를 달성합니다. 이는 FP16에 비해 가중치 저장 공간을 4배 줄이면서도 활성화의 전체 동적 범위를 보존합니다.

반면, 기존의 INT4 및 INT8 W4A8 방식은 4비트 정수(INT4) 가중치와 8비트 정수(INT8) 활성화 값을 페어링하고 INT8 텐서 코어에서 연산을 수행합니다. 이 접근 방식은 히스토그램 기반 보정을 통해 활성화 범위를 품질 손실 없이 INT8로 매핑합니다.

INT4/INT8 커널은 최신 정수 텐서 코어 파이프라인에서 약간 더 높은 원시 처리량을 제공할 수 있지만, 신중한 활성화 보정이 필요하며 FP8만큼의 동적 범위를 가지지 않습니다.

하이브리드 INT4 및 FP8 접근 방식은 두 방식의 장점을 결합합니다. 하이브리드 방식에서는 4비트 정수(INT4) 가중치를 언패킹하여 FP8 입력으로 재해석합니다. 이후 FP8 텐서 코어에서 계산을 실행합니다. 이 INT4 및 FP8 하이브리드 W4A8 변형은 현대 GPU에서 거의 무손실 정확도, 대폭적인 메모리 대역폭 감소, 우수한 처리량을 제공합니다.

현대 GPU에서는 두 가지 저정밀도 경로가 생산 환경에서 흔히 사용됩니다. 첫째, NVFP4 워크플로는 Transformer Engine 또는 TensorRT로 관리되는 마이크로 스케일링을 적용한 FP4 블록을 사용합니다. 둘째, W4A8 워크플로는 TensorRT-LLM의 융합 디퀀타이제이션을 통해 실행되는 FP8 또는 INT8 활성화 함수와 함께 INT4 가중치를 사용합니다. 모델 보정 및 정확도 목표에 따라 경로를 선택하십시오.

요약하면, 양자화는 추론 비용을 줄이는 최상의 방법 중 하나입니다. 모델 크기를 2배 이상 줄임으로써 GPU당 처리량을 효과적으로 두 배로 늘릴 수 있습니다. 앞서 언급한 GPTQ, AWQ, SmoothQuant 등의 기법은 정확도 손실을 최소화하면서(예: 종종 1% 미만 감소) 이러한 이점을 달성하는 데 도움이 됩니다.

추가 성능 향상을 위해 8비트 가중치로 시작한 후 4비트 가중치 전용 양자화를 평가하는 것이 권장됩니다. 이후 최대 최적화가 필요하고 캘리브레이션에 시간을 투자할 수 있을 때만 W4A8(가중치 전용 양자화)로 진행하세요.

다음 단계는 변환 오버헤드를 제거하는 것입니다. 양자화-역양자화 작업을 컴퓨트 커널에 직접 융합함으로써 양자화 사용으로 얻은 이점을 유지할 수 있습니다.

양자화-역양자화 단계의 실행 그래프 내 융합

현대적 추론 엔진은 가변 정밀도( ) TE를 활용해 가중치 패킹을 수행하고, INT8 사용 시 발생하는 명시적 보정 오버헤드 없이 고효율 혼합 정밀도 연산을 제공합니다. 이러한 추론 엔진은 일반적으로 CUDA/Triton 커널을 구현하여 필요 시 "양자화-역양자화" 단계를 실행 그래프에 수동으로 융합합니다.

별도의 양자화/역양자화 커널이 너무 많은 추가 런치를 유발하여 정밀도 감소 연산의 지연 시간 및 대역폭 이점을 상쇄할 경우, 이러한 양자화-역양자화 단계를 그래프에 융합해야 합니다. 이는 특히 기본 융합 INT8 지원이 부족한 추론 백엔드에 유용합니다.

양자화-역양자화를 주요 연산 커널에 융합하는 것은 고처리량 추론 파이프라인에서 변환으로 인한 병목 현상을 제거하여 성능을 복원하는 데 특히 가치가 있습니다. 다음으로, 현대 AI 추론 서비스 엔진이 지원하는 다양한 애플리케이션 수준 최적화를 살펴보겠습니다.

애플리케이션 수준 최적화

핵심 모델 및 시스템 최적화 외에도, LLM 서비스의 성능과 사용자 경험을 크게 향상시킬 수 있는 몇 가지 상위 수준의 기법이 있습니다. 이러한 방법은 애플리케이션 또는 추론 서비스 계층에서 작동하며 prompt 구성 및 캐싱 방식, 대화 기록 보존 방식, 다양한 모델로의 요청 라우팅 방식 등을 개선합니다.

이러한 최적화는 모델 가중치 수정이나 신규 하드웨어 배포를 수반하지 않습니다. 애플리케이션 계층에서의 알고리즘적·시스템적 개선 사항으로, 비용이 거의 발생하지 않아 사실상 '무료'로 효율성과 사용성을 크게 향상시킬 수 있습니다.

본 섹션에서는 prompt 압축, 접두사 캐싱 및 중복 제거, 대체 모델 라우팅, 스트리밍 출력 등 몇 가지 최적화 기법을 논의합니다. 이러한 전략은 입력 크기 축소, 중복 계산 방지, 다양한 요청 유형에 대한 유연한 처리, 최종 사용자의 지연 시간 인식 개선을 통해 성능을 향상시킵니다.

prompt 압축

사용자는 종종 매우 긴 prompt 나 대화 기록을 LLM에 전송합니다. 그러나 적절한 응답을 생성하는 데 이 모든 컨텍스트가 반드시 필요한 것은 아닙니다. prompt 압축은 관련 정보를 잃지 않으면서 입력 prompt를 줄이거나 단순화하는 일련의 기법을 의미합니다.

일부 시스템 prompt나 시스템이 주입하는 지시문은 상당히 장황합니다("당신은 사용자를 돕기 위해 설계된 친근한 어시스턴트 ChatGPT입니다..."). 큰 시스템

prompt는 입력 컨텍스트에서 많은 공간을 차지하며, 이는 모든 요청마다 반복됩니다.

prompt 압축은 모델이 수행해야 할 작업량을 줄입니다. 입력 길이가 짧아지면 GPU 연산량이 감소하므로 비용 절감으로 직접 연결됩니다.

트랜스포머 기반 LLM 내 어텐션 메커니즘은 $O(n^2)$ 시간 복잡도를 가집니다. 여기서 n은 토큰 수로 측정된 입력 크기입니다.

---

prompt 압축의 간단한 형태는 중복되거나 관련 없는 텍스트를 제거하는 것입니다. 사용자의 prompt에 질의 응답과 무관한 대량의 텍스트가 포함된 경우를 생각해 보십시오. 질문이 텍스트 중 한 단락에만 관련된 복사 붙여넣기 기사 등이 해당됩니다. 이 경우 상류 구성 요소가 LLM에 입력하기 전에 관련 부분을 요약하거나 추출할 수 있습니다.

prompt 압축의 또 다른 형태는 대화 요약 또는 생략입니다. 예를 들어 긴 대화 기록의 경우 초기 대화 내용이 더 이상 관련성이 없을 수 있습니다. 시스템은 오래된 부분을 압축된 형태로 요약하여 대화를 지능적으로 정리할 수 있습니다.

ChatGPT와 같은 많은 생산 환경 채팅봇은 컨텍스트 길이 제한을 준수하고 전체 처리 속도를 높이기 위해 prompt 압축을 자동으로 수행합니다. 이는 장시간 대화 시 필수 기능입니다.

이를 위해 소규모 LLM 또는 기존 규칙 기반 시스템을 활용해 대화의 가장 오래된 부분을 간결한 요약으로 정리할 수 있습니다. 일반적으로 다음과 같은 정책을 적용합니다: 대화 길이가 최대 컨텍스트 길이의 75%를 초과할 경우, 가장 오래된 25%의 메시지를 요약합니다. 이 요약본은 이후 대화의 최근 부분 앞에 추가되며, 이후 진행될 새로운 prompt가 됩니다.

prompt 압축을 구현할 때는 중요한 사실이 누락되지 않도록 주의해야 합니다. 한 가지 접근법은 모델이 prompt를 압축(예: 요약 생성)한 후, 압축된 prompt에 대해 모델에게 질문하여 검증하는 것입니다. 이렇게 하면 알고리즘이 압축된 prompt 버전에서 핵심 정보가 유지되었는지 확인할 수 있습니다.

이는 매우 긴 대화에서 과도한 지연을 방지하고 모델이 대화의 가장 최근 부분에 집중하도록 합니다. 이전 턴을 잘라내면 유효 prompt 길이 N이 줄어들어, 프리필 자기 주의 작업량이 $N(N + 1) \div 2$ (대략 $O(N^2)$)에서 더 짧은 창에 대한 삼각형 형태의 더 작은 비용으로 감소합니다. 따라서 비용이 창 크기에 대해 2차 함수에 가까워지더라도, 매우 긴 대화에서는 더 작은 N이 상당한 차이를 만듭니다.

좋은 요약은 관련 없는 세부 사항을 걸러내 응답 품질을 향상시킬 수 있습니다. 그러나 나쁜 요약은 사용자가 진정으로 중요하게 생각하는 입력의 핵심 부분을

생략할 수 있습니다. 요약은 시스템이 확신을 가질 때, 또는 대화가 명백히 본론에서 벗어났을 때 가장 효과적입니다.

prompt 클렌징

또 다른 기법은 prompt 클렌징입니다. 이는 입력 포맷팅과 토큰화를 개선하는 데 사용됩니다. 모델로 전송되는 불필요한 공백이나 마크업의 양을 줄이는 데 도움이 됩니다. 토큰화기는 공백과 줄바꿈을 포함한 모든 문자를 처리합니다. 불필요한 토큰을 적게 보낼수록 좋습니다.

OpenAI의 tiktoken 같은 토큰화 도구는 공백 처리 효율이 매우 높지만, 마크다운이나 HTML이 많은 긴 prompt는 토큰 수를 불필요하게 늘릴 수 있습니다. HTML 태그 제거나 장식용 따옴표를 일반 텍스트로 변환하는 간단한 전처리만으로도 이상한 토큰화를 방지하고 토큰 수를 줄일 수 있습니다.

예를 들어, 입력의 의미에 영향을 주지 않는 빈 줄이나 반복되는 구두점을 전송하지 않음으로써 추론 엔진의 계산량을 잠재적으로 줄일 수 있습니다. 이는 여기저기서 몇 개의 토큰만 절약할 수 있지만, 특히 긴 입력 prompt의 경우 수천 건의 요청에 걸쳐 누적됩니다.

경우에 따라 전체 콘텐츠 대신 참조를 사용해 prompts를 압축할 수 있습니다. 예를 들어 사용자의 prompt에 시스템이 이전에 본 긴 텍스트가 포함된 경우(예: 이전에 저장한 문서를 참조하는 경우), 이를 "file0"과 같은 참조로 대체할 수 있습니다.

모델을 해당 참조에 대한 실제 콘텐츠를 검색하도록 미세 조정하거나 검색 시스템을 활용할 수 있습니다. 이는 검색 강화 생성(RAG) 영역에 속하며, 여기서 더 깊이 다루지는 않겠지만 핵심은 대체 방법이 존재할 경우 원본 콘텐츠를 항상 모델에 입력할 필요가 없다는 점입니다.

일부 추론 엔진은 매번 prompt를 전송하는 대신 세션당 한 번만 시스템 prompt를 설정할 수 있도록 지원합니다. 이는 요청마다 시스템 prompt를 재압축하는 것보다 더 나은 해결책입니다.

다음 섹션에서는 접두사 캐싱을 통해 대규모 시스템 prompt의 효율성을 개선하는 방법을 논의할 것입니다. 하지만 지금은 prompt 압축을 활용해 기능적으로 동등한 더 짧은 지시문 세트를 생성해 보겠습니다. 예를 들어, 모델이 긴 시스템 prompt를 나타내기 위해 특수 토큰이나 메타데이터를 사용하도록 훈련시킬 수 있습니다. 이렇게 하면 항상 긴 자연어 버전을 처리할 필요가 없습니다.

텍스트 기반 규칙으로 가득 찬 200토큰 시스템 prompt를 10개의 특수 토큰으로 대체할 수 있다고 가정해 보십시오. 이 토큰들은 200토큰의 자연어로 표현된 동일한 텍스트 기반 규칙을 트리거합니다. 이를 위해서는 모델이 메타데이터를 파싱하고 규칙을 도출하며 이를 따르도록 훈련되어야 합니다.

주어진 구성 토큰에 대해 미리 구성된 특정 명령어 세트를 로드하도록 모델에 지시하는 "구성(config)" 토큰에 대한 연구가 진행 중입니다. 이는 특정 prompt 접두사(예: 시스템 prompt)에 고유 ID를 할당하는 것과 유사합니다. 예를 들어, 500단어에 달하는 긴 정책을 대체하는 토큰으로 `<POLICY_A>` 을 인식하도록 모델을 미세 조정하는 것이 일반적입니다.

Hugging Face Transformers 라이브러리는 널리 사용되는 CTRL 방식을 구현합니다. prompt 트리밍을 위한 구성 토큰 작업을 시작하기에 좋은 출발점입니다.

긴 시스템 prompt를 대체하기 위해 특수 토큰과 메타데이터를 사용하는 것은 주로 훈련 단계에서의 고려사항입니다. 그러나 prompt 크기를 줄이고 사전 채우기 오버헤드를 감소시킨다면 추론 속도를 크게 향상시킬 수 있습니다. 대규모 환경에서는 이는 쿼리당 필요한 컴퓨팅 자원을 직접적으로 줄여 더 많은 비용 절감으로 이어집니다.

접두사 캐싱

LLM에 대한 여러 요청은 입력에서 공통 *접두사*를 공유하는 경우가 많습니다. 많은 쿼리가 동일한 시스템 prompt로 시작할 수 있기 때문입니다. 동일한 접두사에 대해 매번 모델 출력을 재계산하는 대신, 한 번만 계산한 후 후속 요청에 동일한 키-값 캐시를 재사용할 수 있습니다.

이 기술은 *접두사 캐싱*(*prefix caching*)으로 알려져 있으며, 때로는 접두사 메모이제이션(prefix memoization)이라고도 합니다. 접두사 캐싱을 사용하면, 어텐션 레이어의 키와 값을 포함한 트랜스포머의 상태가 저장되어 동일한 접두사를 가진 요청이 다시 나타날 때 재사용됩니다.

접두사 캐싱은 반복되는 부분에 대해 캐시된 접두사 계산 결과를 재사용함으로써, 일반적으로 O(N × L) 총 작업량(길이 L의 N개 요청)을 O(N + L) 작업량으로 전환할 수 있습니다.

---

vLLM은 재계산을 피하기 위해 접두사 캐싱(`enable_prefix_caching=True`)을 구현합니다. 먼저 입력 prompt의 처음 *N개* 토큰이 이전 요청(또는 동일 세션 내 이전 요청)에서 캐시에 이미 저장된 접두사와 일치하는지 확인합니다. 일치할 경우, vLLM은 해당 *N개* 토큰에 대한 비용이 많이 드는 어텐션 재계산을 피하고, 캐시된 키-값 쌍(KV)의 데이터를 새 컨텍스트로 복사하기만 합니다. 접두사 캐싱이 활성화되었는지, 그리고 충분한 메모리 할당이 이루어졌는지 확인하십시오.

vLLM과 같은 추론 엔진은 들어오는 쿼리를 마이크로배치로 자동 그룹화하여 높은 GPU 활용도를 달성하고, 여러 요청에 걸쳐 오버헤드를 분산시키며, 예를 들어 연속 배치 방식을 통해 지연 시간을 낮출 수 있습니다. vLLM과 같은 추론 엔진을

직접 사용하지 않는 경우, 스택 내에서 접두사 공유를 활용할 방법을 모색하십시오.

접두사 캐싱은 반복되는 접두사가 있는 워크로드를 가속화할 수 있습니다. 긴 문서에서 10개의 별도 질문을 묻고자 하는 시나리오를 고려해 보십시오. 각 질문은 prompt에 문서 내용을 포함시킨 후 하나의 질문을 추가하는 형식입니다. 예를 들어, "[긴 문서 텍스트] 질문 1: ...", "[긴 문서 텍스트] 질문 2: ..." 등입니다.

문서 텍스트는 10개 prompt 모두 동일합니다. 일반적으로 모델은 각 질문마다 문서 전체에 대한 키-값(KV) 항목을 재처리해야 합니다. 이는 모델이 긴 문서를 10번 재인코딩해야 하므로 10배의 중복 작업이 발생합니다. 접두사 캐싱을 사용하면 한 번만 인코딩하고, 이후 질문마다 더 작은 질문 접미사에 대한 계산만 수행합니다.

구체적으로, 접두사 캐싱을 사용하면 첫 번째 쿼리는 문서 부분에 대한 트랜스포머 상태를 계산하고, 이후 쿼리에서는 문서 내용이 KV 캐시에 있는 접두사와 일치하므로 모델이 바로 "질문 1: ...", "질문 2: ..." 부분 처리로 넘어갈 수 있습니다.

접두사 캐싱을 사용하면 추론 엔진이 거의 선형적인 속도 향상을 달성할 수 있습니다. 예를 들어, 하나의 문서에 대해 10개의 별도 질문을 할 경우, 긴 문서의 어텐션이 10번이 아닌 단 한 번만 계산되기 때문에 접두사 캐싱을 사용하면 사용하지 않을 때보다 약 10배 빠르게 처리됩니다.

접두사 캐싱의 또 다른 적용 분야는 채팅 세션입니다. "다중 회화" 채팅 세션에서 대화 기록은 각 새로운 "턴"에 대한 공통 접두사 역할을 하기 때문입니다. 최종 사용자가 새 메시지를 보낼 때, 이전 메시지들은 계속 성장하는 접두사를 형성합니다.

최적화된 추론 시스템은 마지막 턴의 KV 캐시를 유지하고 새 사용자 메시지에 대해서만 KV를 계산합니다. 턴 사이에 대화를 재설정하지 않는다면 ChatGPT와 같은 대부분의 채팅 모델이 바로 이 방식을 사용합니다.

접두사 캐싱의 이점은 대화형 환경에서 가장 두드러집니다. 특히 사용자가 동일한 컨텍스트/세션에서 신속하게 후속 질문을 할 때 그렇습니다. 이 경우 접두사(대화 기록)가 이미 캐시되어 있으므로 두 번째 답변이 첫 번째보다 훨씬 빠르게 제공됩니다.

OpenAI는 ChatGPT 소비자 제품에서 상태를 지원하는 것 외에도, API에서도 상태 유지 대화 기능을 지원합니다. 이 기능은 여러 최적화 중 하나로 프리픽스 캐싱 방식을 백엔드에서 활용합니다. 상태 비저장형 API 호출의 경우, 클라이언트가 대화 이력을 전달하면 접두사 캐싱으로 유사한 효과를 얻을 수 있습니다. 이때 서버는 접두사 캐시에서 마지막 턴까지 미리 계산된 숨겨진 상태를 검색한 후, 새로운 사용자 메시지로 이어서 처리합니다.

이를 직접 구현하려면 대화 기록을 해시 처리하세요. 이렇게 하면 캐시에서 조회할 일관된 키를 얻을 수 있습니다. 다만, 다수의 대화 전체에 대한 키-값(KV) 캐시를 저장하면 GPU 메모리를 매우 빠르게 소모할 수 있으므로 메모리 관리를 신중히 해야 합니다. vLLM의 페이징 기능은 비활성 캐시를 CPU 메모리나 디스크로 페이징한 후 캐시 히트 시 GPU 메모리로 다시 페이징하여 이 문제를 해결합니다.

여러 요청 간 또는 단일 요청 내에서도 prompt 일부를 중복 제거할 수 있습니다. 여기서 중복 제거란 동일한 하위 prompt를 결합하여 트랜스포머 상태를 한 번만 계산하는 것을 의미합니다. 예를 들어 두 사용자가 거의 동시에 완전히 동일한 prompt 접두사를 보낸 경우, 시스템은 두 prompt를 병합하여 한 번만 처리할 수 있습니다.

단일 요청 내에서도 입력에 반복되는 시퀀스가 존재할 경우 동일 현상이 발생할 수 있습니다. 이는 반복 텍스트를 중복 제거하기 위해 메모이제이션 디코딩 같은 기법을 사용하는 훈련 단계에서 더 흔합니다. 추론 단계에서는 동일한 요청 내에서 길고 반복적인 입력 시퀀스가 발생하는 경우가 드물지만 가능성은 있습니다.

동일한 접두사에 대한 반복 쿼리가 많은 워크로드라면, 히트율을 극대화하기 위해 캐시에 더 많은 메모리를 할당해야 합니다. 반대로 접두사가 거의 재사용되지 않는다면 더 작은 캐시를 사용하거나 아예 접두사 캐싱을 비활성화해야 합니다. 항상 프로덕션 환경에서 접두사 히트율을 측정하고 그에 따라 조정하세요.

접두사 캐시는 일반적으로 토큰 시퀀스 *트라이* (*trie*, "트라이"로 발음)로 구현됩니다. 트라이(*종종 접두사 트리라고도 함*)는 트리 기반 데이터 구조로, 각 에지는 단일 토큰을 나타내고 각 노드는 루트부터 해당 지점까지의 토큰 시퀀스를 인코딩합니다.

토큰 시퀀스 트라이에서는 관측된 모든 prompt 접두사가 자체 토큰 경로로 저장됩니다. 이를 통해 공유 접두사를 빠르게 조회할 수 있습니다. 새로운 요청이 도착하면 추론 엔진은 트리를 루트부터 다음 토큰과 더 이상 일치하지 않을 때까지 토큰 단위로 탐색합니다. 일치하는 토큰 시퀀스는 그림 16-8의 예시 시스템 prompt "You are ChatGPT, a friendly assistant designed to help users..."에서 보듯이 가장 길게 캐시된 접두사를 완성하는 노드에 도달합니다.

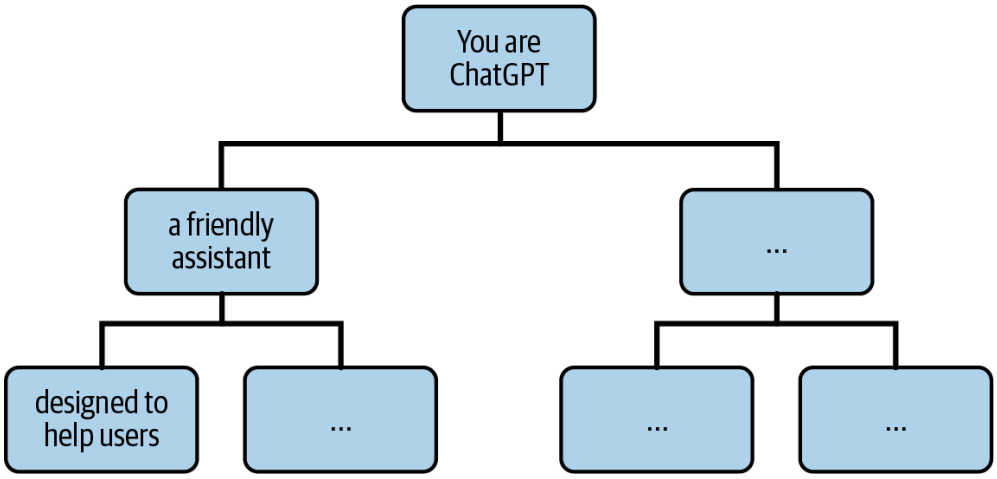

그림 16-8. 트라이(trie) 데이터 구조로 구현된접두사 캐시

이 시점에서 시스템은 공유된 KV 캐시 데이터(포인터 복제)를 재사용하고, 시퀀스 내 남은 토큰에 대해서만 디코딩을 재개합니다. 이는 이미 캐시된 접두사에 대한 어텐션 점수가 이미 계산되었으므로 중복된 자체 어텐션 계산을 피합니다.

SGLang의 RadixAttention은 전체 토큰 시퀀스에 대해 라딕스 트리( ) 압축 트리(또는 라딕스 트리)를 사용합니다. 이 유형의 접두사 캐시는 공간을 절약하기 위해 토큰 시퀀스를 트리 내 단일 에지로 압축합니다. 트리의 각 노드는 접두사의 KV 캐시를 보유하는 연속적인 GPU 페이지로 저장된 KV 캐시 텐서를 가리킵니다.

RadixTree 데이터 구조는 공간 효율적이며 빠른 접두사 검색, 효율적인 삽입, LRU 방식의 제거를 제공합니다. 다음은 토큰 생성 중 래딕스 트리를 사용한 KV 캐시 조회 과정을 보여주는 의사 코드입니다:

```
# Simplified RadixAttention KV cache example
# radix_attention_example.py

radix: RadixTree = RadixTree()  # holds edge labels + node.cache pointers

def generate_with_radix(prompt_tokens: List[int]):
    # 1) Find longest cached prefix
    node, prefix_len = radix.longest_prefix(prompt_tokens)
    # shallow-clone the KV cache for that prefix
    model_state = ModelState.from_cache(node.cache)  # refcount bump

    # 2) Process remaining prompt suffix
    for token in prompt_tokens[prefix_len:]:
        model_state = model.forward(token, state=model_state)

    # 3) As we go, insert or split edges in the radix tree
        matched = prompt_tokens[:prefix_len + 1]
        # insert returns the node for this full prefix
        node = radix.insert(matched, cache=model_state.kv_cache)

        prefix_len += 1
```

```
    # 4) Now generate new tokens autoregressively
    output_tokens = []
    while not model_state.is_finished():
        token, model_state = model.generate_next(model_state)
        output_tokens.append(token)

        # cache each generated prefix as well
        matched = prompt_tokens + output_tokens
        node = radix.insert(matched, cache=model_state.kv_cache)

    return output_tokens
```

생성이 시작되면 엔진은 `radix.longest_prefix(prompt_tokens)`를 호출하여 다중 토큰 에지를 따라 래디스 트리를 하향 탐색합니다. prompt의 가장 긴 캐시된 접두사와 일치하는 최하위 노드에 도달할 때까지 진행합니다. 그런 다음 `ModelState.from_cache(node.cache)`를 사용하여 해당 노드의 KV 캐시 페이지를 경량 복제합니다. 이는 캐시된 접두사에 대한 자체 주의(self-attention)를 재계산하지 않고 `model_state`를 초기화합니다.

다음으로 prompt의 미처리 접미사 부분만을 토큰 단위로 처리하며 래디스 트리를 실시간으로 업데이트합니다. 새 토큰마다 `radix.insert(...)`를 호출하여 필요에 따라 에지를 분할하거나 새로 생성합니다. 이후 각 새 노드에 중간 `model_state.kv_cache`를 저장합니다.

전체 prompt가 처리되면 루프는 자동회귀적 디코딩 단계로 전환되며, 여기서 `model.generate_next(model_state)`를 통해 새 토큰을 생성합니다. 마찬가지로 생성된 각 접두사를 라딕스 트리에 삽입합니다. 이 접근법은 중복 계산을 최소화하고, KV 페이지의 공간 효율적인 저장을 활용하며, 빠른 접두사 조회 성능을 제공하면서도 점진적 캐시 업데이트를 지원합니다.

SGLang의 KV 캐시 설계는 다중 회차 대화, 소량 예제, 분기 논리 등 모든 일반적인 재사용 패턴을 자동으로 포착합니다. 동시에 공유 접두사가 효율적인 GPU 접근을 위해 대형 통합 메모리 청크로 가져오도록 보장합니다.

캐시를 언제 무효화해야 할지 판단하는 것은 항상 어려운 과제입니다. 캐시 메모리가 다른 용도로 필요할 경우 일부 접두사를 제거해야 할 수 있습니다. 캐싱 시스템은 "최근에 가장 적게 사용된 항목"(LRU)과 같은 다양한 정책을 지원합니다. LRU에서는 가장 최근에 사용되지 않은 접두사가 먼저 제거됩니다. 예를 들어 SGLang의 RadixAttention은 GPU 메모리가 부족할 때 그림 16-9와 같이 최근에 가장 적게 사용된 라디스 트리 리프를 지연 제거합니다.

이것은 SGLang이 다중 요청 처리에 사용하는 접두사 캐시 트리 구조와 LRU 제거 정책입니다. 본 예시는 LMSys의 훌륭한 SGLang 블로그 포스트를 기반으로 합니다.

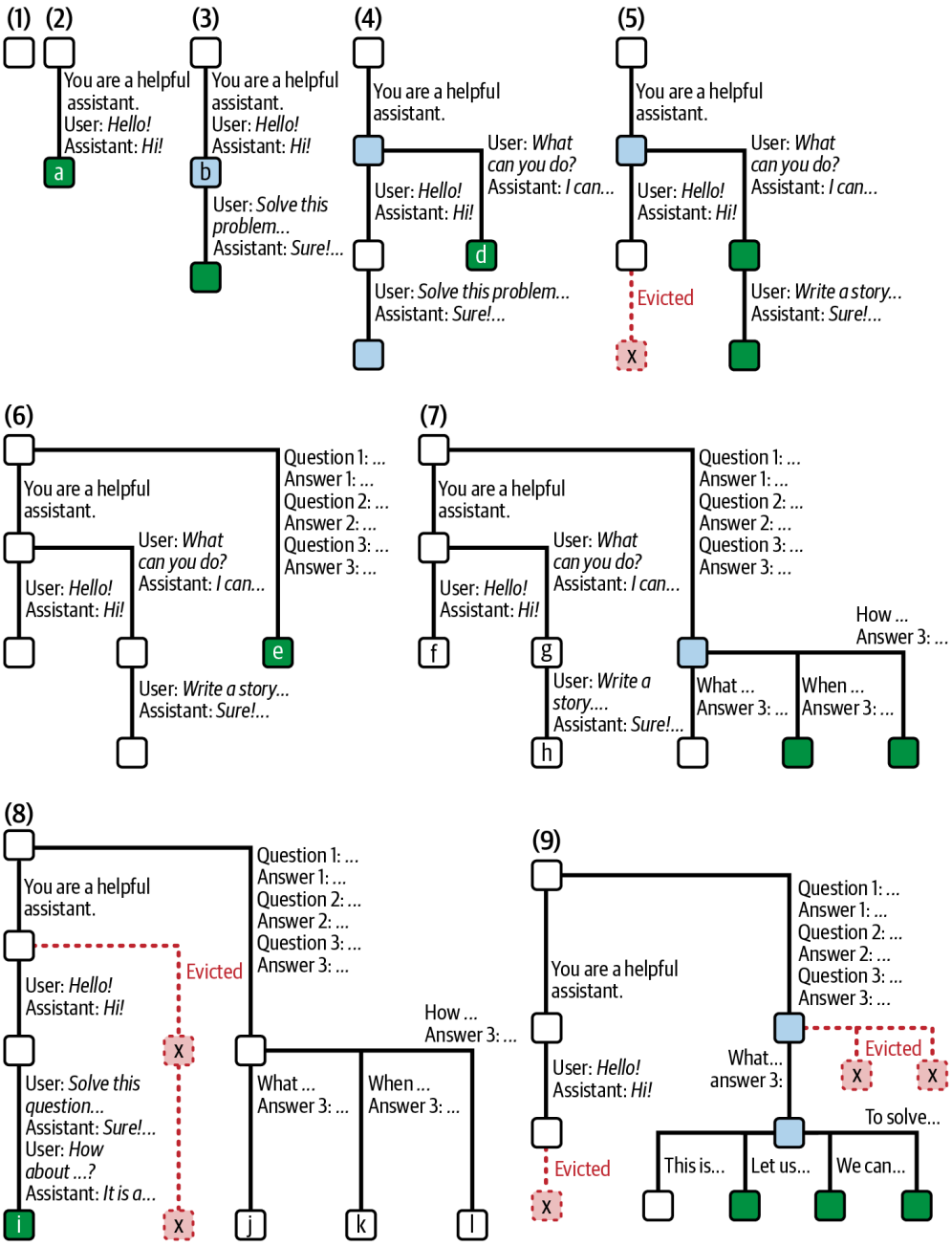

그림 16-9. 다중 요청에 대한접두사 캐시 진화 (출처: https://oreil.ly/7LBoC)

여기에는 두 개의 채팅 세션과 해당 세션에 걸친 여러 쿼리가 존재합니다. 각 라벨은 토큰 시퀀스(예: 부분 문자열)로 구성됩니다. 녹색 노드는 트리 내 신규 노드를 나타냅니다. 파란색 노드는 현재 접근 중인 캐시된 노드입니다. 빨간색 노드는 제거된 노드입니다. 각 단계별 설명은 다음과 같습니다:

1. 초기 빈 라딕스 트리는 비어 있습니다.
2. 서버는 수신된 prompt "Hello!"를 처리하고 LLM이 생성한 "Hi!"로 응답합니다. 이 간단한 응답으로 많은 토큰이 단일 에지로 트리에 추가됩니다. 이 에지는 녹색의 새 노드에 연결됩니다. 구체적으로 시스템 prompt "You are a helpful assistant", 사용자 메시지 "Hello!", LLM 응답 "Hi!"가 통합됩니다.
3. 서버가 새로운 prompt를 수신합니다. 이는 다중 회화(multiturn conversation)의 첫 번째 턴입니다. 서버는 트리에서 prompt 접두사를 성공적으로 조회하고 KV 캐시 데이터를 재사용합니다. 새로운 턴이 새로운 녹색 노드로 트리에 추가됩니다.
4. 새로운 채팅 세션이 시작되며, 3단계의 노드 *b*가 두 개의 별도 노드로 분할됩니다. 이를 통해 두 채팅 세션이 시스템 prompt를 공유할 수 있습니다.

5. 4단계의 두 번째 채팅 세션이 계속됩니다. 그러나 메모리가 제한되어 있어 4단계의 노드 c는 제거되어야 하며, 이는 빨간색으로 표시됩니다. 4단계의 노드 d 뒤에 새로운 턴이 추가됩니다.
6. 서버가 새로운 prompt(쿼리)를 수신합니다. 처리 후 서버는 이 prompt를 트리에 삽입합니다. 이 새로운 prompt는 기존 prompt/노드와 접두사를 공유하지 않으므로 루트 노드를 분할해야 합니다.
7. 서버가 더 많은 prompt(쿼리)가 포함된 배치를 수신합니다. 이 prompt들은 6단계의 prompt와 접두사를 공유합니다. 따라서 시스템은 6단계의 노드 e를 분할하고 접두사를 공유합니다.
8. 서버가 3단계(첫 번째 채팅 세션) 대화에서 새 메시지를 수신합니다. 이 경우 5단계에서 두 번째 채팅 세션의 모든 노드(예: g, h 노드)를 제거합니다. 이는 해당 시점에서 가장 최근에 사용되지 않은(LRU) 노드이기 때문입니다.
9. 서버는 8단계의 *노드 j에 대한* 쿼리에 대한 추가 답변을 요청하는 메시지를 수신합니다. 메모리 제한으로 인해 시스템은 8단계의 노드 i, k, l을 제거해야 합니다.

이 예시는 여러 요청 간 접두사 공유 방식을 보여줍니다. 또한 접두사 캐싱 환경에서 LRU 캐시 제거 정책이 작동하는 방식을 설명합니다. 다음으로 GPU 리소스 활용도를 높이기 위한 계단식 모델 배포 패턴 사용법을 살펴보겠습니다.

모델 캐스케이딩 및 계층형 모델 배포

모든 쿼리가 최신 대형 모델의 성능 과 비용을 필요로 하는 것은 아닙니다. 일부 사용자 요청은 훨씬 작고 빠르며 저렴한 모델로도 충분히 처리될 수 있습니다. 적절한 모델 선택을 *모델 캐스케이딩* 또는 *폴백 모델 라우팅이라고* 합니다.

모델 캐스케이딩을 구현하려면 계층형 모델 배포를 활용할 수 있습니다. 이는 서로 다른 규모와 성능을 가진 여러 모델을 유지하는 접근 방식입니다. 들어오는 각 요청에 대해 쿼리 복잡도, 요구되는 정밀도 또는 현재 시스템 부하를 기준으로 사용할 모델을 동적으로 선택합니다.

7,000억 매개변수의 대규모 최첨단 모델과 GPU 리소스 요구량이 적어 호스팅 비용이 저렴하고 속도가 빠른 700억 매개변수의 소형 모델을 모두 갖춘 질문응답(QA) 추론 배포 환경을 예로 들어 보겠습니다. 사용자가 매우 직관적이고 사실적인 질문(분류기나 휴리스틱으로 판단)을 할 경우, 700억 매개변수 모델이 잘 처리할 수 있습니다.

간단한 질문의 경우, 쿼리를 소형 모델로 라우팅하면 7000억 파라미터 모델의 50밀리초 대비 50밀리초 내에 응답할 수 있습니다. 소형 모델의 답변이 낮은 신뢰도(응답에 반환된 로짓 값으로 판단)로 인해 불만족스러울 경우, 시스템은 질문을 대형 모델로 라우팅할 수 있습니다.

이 2단계 접근법은 전체 평균 지연 시간과 컴퓨팅 사용량을 줄일 수 있습니다. 다만 소형 모델의 초기 응답에 대한 신뢰도가 부족해 두 모델 모두로 라우팅될 경우 개별 요청의 지연 시간이 길어질 수 있습니다. ChatGPT와 같은 많은 상용 AI 서

비스가 비용과 지연 시간을 줄이기 위해 단순한 질의를 더 작고 빠른 모델로 라우팅하는 이 방식을 사용한다고 널리 알려져 있습니다.

실제 적용 시, 예를 들어 쿼리의 60%를 10배 더 작은 모델로 라우팅하면 추론 컴퓨팅 비용을 크게 절감할 수 있습니다. 설계 및 튜닝이 적절히 이루어진다면 전체 비용을 최대 5배까지 줄일 수 있습니다. 이는 고비용 모델이 필요한 경우에만 사용되기 때문입니다.

여러 모델과 라우팅 시스템을 유지하는 것은 엔지니어링 오버헤드입니다. 하나의 모델 성능뿐만 아니라 두 번째 모델, 두 모델 간의 상호작용, 라우팅 시스템까지 모니터링해야 하기 때문입니다. 트래픽 양과 비용 구조가 이를 정당화할 때만 추구하십시오.

모델 캐스케이딩 구현에는 모델 라우팅 결정을 지원하는 쿼리 분류기 또는 휴리스틱 메커니즘이 필요합니다. 이 메커니즘은 지속적인 적응, 포괄적인 휴리스틱, 지속적인 라우터 튜닝을 요구하는 경우가 많아 올바르게 해결하기가 상대적으로 어려운 문제입니다. 따라서 많은 조직은 여전히 오프라인 분석을 통해 결정 기준을 업데이트하는 휴리스틱 및 규칙 기반 라우터에 의존하고 있습니다.

한 가지 접근법은 이를 앞서 설명한 추측적 디코딩과 유사한 추측적 문제로 간주하는 것입니다. 먼저 소규모 모델을 초안 모드로 실행하여 신뢰도 점수와 함께 답변을 생성하도록 합니다. 신뢰도가 높으면 해당 응답을 반환하고, 그렇지 않으면 대규모 모델을 호출합니다.

큰 모델이 필요한 경우, 작은 모델의 추가 지연으로 인해 사용자가 너무 오래 기다리지 않도록 해야 합니다. 실제 적용 시, 질문이 어렵다고 판단되면 소형 및 대형 모델을 병렬로 실행할 수 있습니다. 이렇게 하면 소형 모델의 지연 시간이 대형 모델 실행 시간과 겹칩니다. 이는 비효율적이고 직관적이지 않아 보이지만, 추가 컴퓨팅 비용을 감수하더라도 지연 시간이 중요한 일부 사용 사례에서는 지연 시간을 낮추는 데 도움이 될 수 있습니다.

또 다른 접근법은 사전에 잘 알려진 사용자 질의에 대한 별도의 분류기를 훈련시키는 것입니다. 예를 들어, 복잡성을 분류하고 "이 질문이 대형 모델의 방대한 지식이나 추론 능력을 필요로 하는가?"라고 판단할 수 있습니다. 그렇지 않다면 소형 모델을 사용하세요.

이 분류기 자체도 사용자 질의가 시간에 따라 진화함에 따라 주기적인 재훈련이 필요하다는 점을 유의해야 합니다. 또한 성능 저하 및 고장 가능성을 가진 또 다른 구성 요소를 도입하므로 모니터링이 필요합니다. 이 모델이 가볍고 빠르며 견고해야 합니다. 그렇지 않으면 추론 시스템의 새로운 병목 현상이 될 수 있습니다.

초기에는 간단한 휴리스틱을 사용하는 것이 일반적입니다. 예를 들어, "오늘 날씨는 어때요?" 또는 "X의 대통령은 누구인가요?"처럼 짧고 사실적인 사용자 질의의 경우 작은 모델만 사용하면 됩니다. *설명하다*, *분석하다*, *자세히 설명하다* 같은

단어가 포함된 더 긴 prompt(예: 50 토큰 이상)나 창의적인 질의의 경우 작은 모델을 거치지 않고 큰 LLM에 직접 전송하면 됩니다.

모든 요청에 대해 어떤 모델이 처리했는지(예: "소형" 대 "대형")와 후퇴 처리 여부를 기록하고 태그해야 합니다. 이를 통해 모델별로 지연 시간, 정확도, 후퇴 횟수를 분석할 수 있습니다.

이 태깅 데이터는 디스패처의 결정도 모니터링할 수 있게 합니다. 소형 모델의 답변이 거부된 횟수를 집계하면, 소형 모델 재훈련이나 라우터의 휴리스틱 개선으로 이어질 수 있는 패턴을 식별할 수 있습니다.

반대로 소형 모델이 많은 쿼리를 잘 처리한다면, 이 정보를 활용해 신뢰도 임계값을 높일 수 있습니다. 이는 소형 모델 사용률을 높이고 응답 지연 시간을 줄이며 최종 사용자 경험을 개선할 것입니다.

용량 기반 라우팅도 가능합니다. 대형 모델 클러스터가 최대 부하 상태일 때, 요청을 대기열에 쌓아 지연을 증가시키기보다는 중요도가 낮은 요청을 일시적으로 소형 모델로 라우팅할 수 있습니다. 성능은 다소 떨어질 수 있으나 신속한 답변을 제공합니다. 이는 고부하 시 우아한 성능 저하(graceful degradation)의 한 형태입니다. 사용자는 품질 저하를 느낄 수 있으나, 타임아웃이나 과도한 대기보다 낫습니다.

많은 AI 서비스가 의도적으로 이를 적용합니다. 예를 들어 피크 시간대에는 무료 계층 사용자를 자동으로 소규모 모델로 라우팅하여 대규모 모델을 유료 계층 사용자에게 할당할 수 있습니다. 이는 자원이 제한된 상황에서 현실적인 절충안입니다.

17~19장에서는 보다 고급 동적 및 적응형 추론 서버 기능을 다룰 예정입니다. 이러한 기능은 사용 가능한 GPU 메모리, 메모리 대역폭 활용도, KV 캐시 활용도 등 현재 시스템 부하에 크게 의존합니다.

모델 캐스케이딩의 또 다른 형태는 콘텐츠 자체에 기반합니다. 예를 들어, 사용자가 안전성 문제로 LLM이 답변을 거부하는 질문을 할 경우, 일부 추론 시스템은 안전하지 않은 prompt를 처리하도록 특별히 미세 조정된 별도의 모델로 전환하거나 심지어 검색 시스템으로 전환하기도 합니다.

이는 성능보다는 콘텐츠 정책에 관한 문제이지만, 모델 라우팅 로직을 설계할 때 여전히 중요한 고려사항입니다. 또한 대체 모델은 대개 훨씬 작기 때문에, 실제로는 성능 향상을 가져옵니다. 위험한 요청을 신속하게 처리함으로써 주 모델이 안전한 쿼리를 처리할 수 있는 여유를 확보해주기 때문입니다.

배포 관점에서 볼 때, 다중 모델 설정은 각 모델이 담당 트래픽을 처리할 충분한 인스턴스를 필요로 하므로 신중한 확장 관리가 필요합니다. 예를 들어 단순한 쿼

리가 많이 유입될 경우, 소형 LLM은 포화 상태에 이르는데 대형 모델은 유휴 상태로 남아 병목 현상이 발생할 수 있습니다.

컨테이너 오케스트레이션을 활용하면 시간대별로 소형 모델 인스턴스를 자동 확장하거나 동적으로 조정할 수 있습니다. 필요 시 대형 모델 풀의 GPU를 소형 모델 풀로 전환하는 것도 가능합니다.

응답 스트리밍

추론 시스템의 "지각된" 성능 을 향상시키려면 출력 토큰이 생성되는 즉시 스트리밍해야 합니다. 이는 응답을 보기 위해 최종 사용자가 전체 완료를 기다리게 하는 것보다 훨씬 낫습니다. 이렇게 하면 사용자는 정보가 도착하는 즉시 읽기 및 처리 작업을 시작할 수 있습니다. 총 소요 시간이 동일하더라도 효과적인 상호작용 속도를 크게 높일 수 있습니다.

인간은 분당 200~300단어를 읽을 수 있으며, 이는 초당 약 4~7토큰(빠른 독자의 경우 초당 최대 13토큰)에 해당합니다. 이러한 속도로 응답을 스트리밍하는 것이 이상적입니다. 모든 성능 결정은 상충 관계로 귀결되므로, 최적화 선택, 용량 계획 등을 수립할 때 고려해야 할 중요한 지표입니다.

시스템의 토큰 처리량을 인간의 읽기 속도와 비교하여 모니터링하는 것이 중요합니다. 예를 들어, 모델 지연으로 인해 시스템이 초당 2토큰만 스트리밍하는 경우, 이는 추가 최적화가 필요하다는 신호입니다.

스트리밍은 vLLM, SGLang, NVIDIA Dynamo를 포함한 대부분의 현대적 추론 엔진에서 지원됩니다. 스트리밍을 활성화하면 서버는 WebSockets, 서버 전송 이벤트(SSE), HTTP 스트리밍 프로토콜 등을 통해 토큰을 클라이언트로 즉시 전송합니다. 이는 앞서 언급한 초당 4~13토큰이라는 인간 읽기 속도 기준을 충족하기 위해 토큰 생성 즉시 수행되어야 합니다.

모델은 기본적으로 한 번에 하나의 토큰을 생성합니다. 단, EAGLE이나 Medusa와 같은 추측형 또는 다중 토큰 디코딩 기법을 사용할 때는 예외입니다. 따라서 시스템은 생성된 토큰을 2~5개 단위로 배치화한 후 스트림을 플러시하고 배치된 토큰을 최종 사용자에게 전송해야 합니다.

플러싱 전에 배치에 너무 많은 토큰을 축적하지 않는 것이 중요합니다. 그렇지 않으면 스트리밍의 목적을 달성하지 못합니다. 반면, 패킷당 하나의 토큰을 전송하는 것은 오버헤드로 인해 비효율적일 수 있습니다. 프레임워크는 종종 새 줄 또는 문장 끝 토큰마다 플러싱합니다.

예를 들어, 답변이 100개의 토큰으로 구성되고 완전히 생성하는 데 5초가 걸린다면, 스트리밍을 사용할 경우 각 배치를 5개 토큰으로 설정할 때 첫 번째 토큰 배치는 0.25초 후에 도착합니다(5토큰 ÷ 초당 20토큰 = 0.25초). 이후 응답이 완료될

때까지 토큰 배치가 꾸준히 전송됩니다. 이렇게 하면 사용자는 단 0.25초 만에 읽기를 시작할 수 있습니다.

스트리밍이 없다면 사용자는 5초 동안 빈 화면을 바라보다 갑자기 전체 답변이 한꺼번에 표시되는 상황을 마주하게 되는데, 이는 바람직하지 않습니다. 이는 분노 클릭이나 기타 형태의 사용자 불만으로 이어질 수 있습니다.

성능 측면에서 스트리밍은 하나의 큰 메시지 대신 추가적인 소규모 네트워크 패킷을 전송하므로 약간의 오버헤드를 발생시킵니다. 그러나 이 오버헤드는 일반적으로 모델의 계산 시간과 비교할 때 무시할 수 있을 정도이며, 개선된 최종 사용자 경험과 비교해도 마찬가지입니다. HTTP/2와 지속적 연결을 사용하면 연결을 재설정할 필요 없이 토큰을 연속적인 스트림으로 전송함으로써 오버헤드를 줄일 수 있습니다.

Nagle 알고리즘 과 지연 확인(delayed acknowledgments)에 유의하십시오. 이 상호작용은 수십 밀리초의 지연을 추가할 수 있으며, 일부 스택에서는 최악의 경우 설정에서 최대 약 200ms까지 지연될 수 있습니다. 토큰 플러시 지연 시간을 최소화하려면 `TCP_NODELAY` 을 사용하고, 가능한 경우 빠른 확인(quick-ack) 또는 감소된 지연 확인 타이머(reduced delayed-ack timers)를 활용하십시오. 이는 실시간 스트리밍에 중요한 토큰 전송 지연 시간(token-send latency)을 줄이는 데 도움이 됩니다. 단점은 더 많은 소형 패킷이 전송 경로를 채워 대역폭 효율이 저하된다는 점입니다. 그러나 초저지연 튜닝 시 이 점을 유념하십시오.

네트워크 문제 등으로 클라이언트가 스트림을 느리게 소비할 경우 모델의 생성을 차단하지 않도록 흐름 제어를 적절히 관리하는 것이 중요합니다. 이상적으로는 클라이언트가 스트림 소비에 뒤처지더라도 추론 엔진이 전체 응답 생성을 계속해야 합니다.

시스템은 데이터를 최종 사용자에게 전송하기 위해 별도의 스레드와 CUDA 스트림을 사용해야 합니다. 이렇게 하면 응답 전송 중 발생하는 문제로 인해 메인 토큰 생성 루프가 중단되지 않습니다.

추론 엔진은 흐름 제어를 관리하고 무제한 메모리 증가를 방지하기 위해 바운디드 버퍼를 유지합니다. 클라이언트가 중단되거나 연결이 끊어질 경우 이러한 상황이 발생할 수 있습니다. 이러한 유형의 극단적인 시나리오는 실제 운영 환경에서 상당히 흔하므로 처리하는 것이 중요합니다. 실제로는 최대 50~100개의 토큰이 누적되도록 허용할 수 있습니다.

특정 한도를 초과하면 추론 엔진은 생성 작업을 일시 중지하거나 연결을 완전히 종료할 수 있습니다. 이렇게 하면 클라이언트가 생성 중간에 연결이 끊기거나 단순히 느린 연결로 인해 속도를 따라가지 못할 경우, 엔진이 토큰 생성을 중단하고 해당 자원을 다른 요청 처리에 할당할 수 있습니다.

버퍼 한도 설정은 메모리 사용량과 사용자 경험 사이의 균형을 맞추는 작업입니다. 짧은 응답에서는 거의 문제가 되지 않지만, 특히 느린 소비자 연결 환경에서

매우 큰 응답의 경우 다소 흔히 발생합니다.

흐름 제어를 개선하는 또 다른 방법은 토큰 풀링을 사용하는 것입니다. 모델이 스트리밍에 필요한 속도보다 빠르게 토큰을 생성할 경우, 시스템은 의도적으로 지연을 추가하여 생성 속도를 완화할 수 있습니다. 예를 들어, 응답의 단순한 부분 때문에 모델이 0.5초 만에 20개의 토큰을 급격히 생성하는 경우, 이들을 한 번에 모두 전송하면 큰 덩어리가 표시된 후 다음 부분을 위해 일시 중지될 수 있습니다.

애플리케이션 UI에 보다 안정적인 타자기 효과를 적용하고 싶을 수 있습니다. 이 경우 토큰 전송 사이에 50ms와 같은 인위적인 스트림 지연을 도입할 수 있습니다. 이는 실제 지연 시간에 큰 영향을 주지 않으면서 토큰이 갑작스럽게 쏟아지는 현상을 방지해 사용자 경험을 개선합니다.

토큰 풀링 지연을 구성 가능하게 설정하여 다양한 사용자 유형(예: 유료 사용자, 무료 사용자 등)에 맞춰 다른 UX를 제공할 수 있습니다. 유료 사용자는 더 빠르게 스트리밍할 수 있는 반면, 무료 사용자는 약간 속도가 제한됩니다. 이는 제한된 리소스로 부하를 관리하는 데 도움이 됩니다.

스트리밍 응답은 최종 사용자가 응답이 완료되기 전에 중간 결과를 평가하고 조치를 취할 수 있게 합니다. 예를 들어, 사용자가 답변의 첫 부분을 보고 원하는 방향이 아니라고 판단하면 중지 또는 중단 버튼을 사용하여 생성을 중단할 수 있습니다.

이는 불필요한 컴퓨팅 자원을 절약하고, 사용자가 피드백을 제공하기 위해 잘못된 완료를 기다릴 필요가 없어 사용자 경험을 개선합니다. 조기 중지 현상을 모니터링하거나 최소한 측정해야 합니다.

명시적 중지가 너무 많다면 모델의 관련성 문제일 수 있습니다. 특정 지점에서 많은 사용자가 중지하는 경우, 모델이 자주 궤도를 이탈하거나 해당 길이 이후로 지나치게 장황해질 수 있음을 시사합니다. 이는 모델을 더 간결하게 만들기 위한 추가적인 미세 조정이 필요하다는 신호일 수 있습니다. 또는 최종 사용자에게 생성 및 전송되는 최대 토큰 수를 조정할 수도 있습니다.

또는 조기 중단이 자주 마음을 바꾸는 좌절한 '분노 클릭' 유형의 사용자에 의해 발생할 수도 있습니다. 어느 쪽이든, 사용자가 토큰 생성 프로세스를 중단할 수 있도록 허용하는 것이 최선입니다. 이렇게 하면 추론 클러스터가 다른 요청을 처리하기 위해 자원을 회수할 수 있습니다.

요약하자면, 스트리밍은 반응형 LLM 서비스에 필수적입니다. 순수 처리량을 증가시키지는 않으며 오히려 약간의 오버헤드를 추가하지만, 사용자가 체감하는 시스템 속도를 향상시킵니다. 스트리밍 응답은 생성 과정에 방해가 되지 않도록 신중하게 구현해야 합니다. 응답을 최종 사용자에게 전송하는 작업은 별도의 스레드나 CUDA 스트림에서 처리해야 합니다. 그래야 불안정한 최종 사용자 네트워크 연결로 인해 메인 토큰 생성 루프가 중단되지 않습니다.

테스트 환경에서 스트리밍 활성화/비활성화 시 엔드투엔드 지연 시간을 지속적으로 자질( )할 것을 권장합니다. 토큰 발생 간격이 균일하게 유지되고, 잠금 경합이나 I/O 대기 같은 중대한 병목 현상이 발생하지 않도록 확인하세요. Locust 같은 도구는 Python 친화적이며 클라이언트를 시뮬레이션할 수 있습니다. 이를 통해 대규모 저지연 스트리밍 워크로드를 테스트할 수 있습니다.

디바운싱 및 요청 병합

많은 생산 시스템은 *디바운싱(debouncing)* 과 *요청 병합(request coalescing)*이라는 사용자 경험(UX) 보호 기능도 구현합니다. 응답 전에 일시 중지(디바운싱)함으로써 시스템은 사용자가 실수로 또는 분노 클릭으로 인해(그림 16-10 참조) 짧은 시간 내에 여러 요청을 보내는지 인식할 수 있습니다.

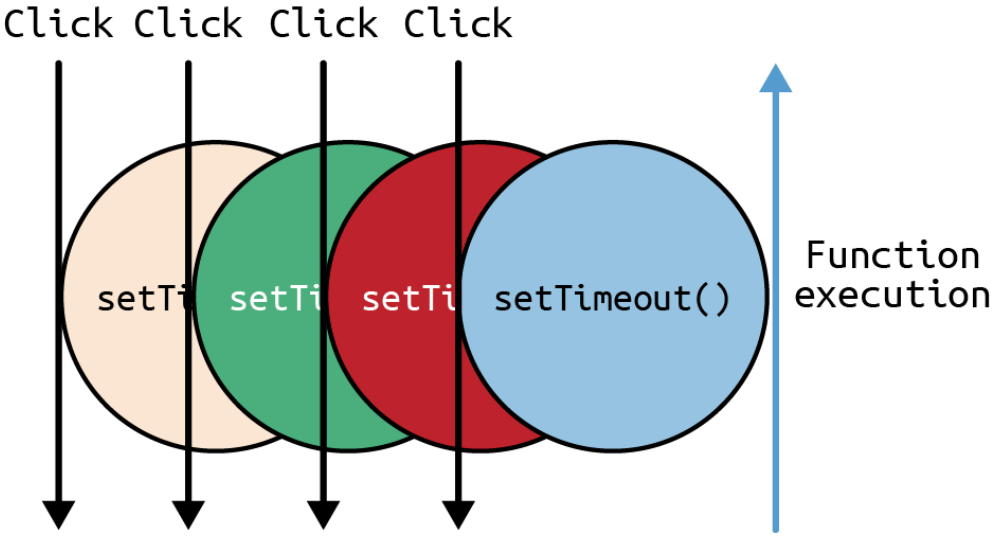

그림 16-10. 디바운싱은 동작을 수행하기 전에 잠시 대기합니다

이 경우 시스템은 여러 쿼리를 하나의 쿼리로 병합하거나 최신 쿼리만 남기고 나머지는 모두 삭제할 수 있습니다. 이러한 애플리케이션 수준의 안전장치는 백엔드에 가해지는 과도하고 반복적이며 낭비적인 부하를 줄이는 데 도움이 됩니다.

예를 들어 사용자가 제출 버튼을 두 번 클릭하거나 1초 이내에 매우 유사한 두 개의 쿼리를 보낼 경우, 시스템은 중복 쿼리를 삭제할 수 있습니다. 분노 클릭 사용자는 좌절감에 참지 못하고 여러 번 재제출하는데, 이는 종종 애플리케이션의 느린 응답 시간 때문입니다.

아이러니하게도, 디바운싱과 요청 통합은 오히려 더 많은 지연을 발생시켜 '분노' 사용자층을 더욱 좌절시키고 클릭을 더 많이 유발할 수 있습니다! 이 경우, 요청이 전송 중인 동안 UI가 입력을 비활성화하는 것이 도움이 됩니다. 이는 UI 수준에서 더블클릭을 방지할 수 있습니다. 하지만 서버 측 보호 장치도 함께 마련하는 것이 좋습니다.

현대적인 추론 로드 밸런서는 디바운싱 간격을 지원합니다. 적당한 간격(예: 2~5ms)을 사용하고, 배치 효율성과 추가 지연 시간 사이의 적절한 균형을 찾으세요. 그리고 다음에 ChatGPT에 무언가를 제출할 때 디바운싱 지연을 눈여겨보세요. 이제 알고 보니 좀 짜증 나지 않나요?!

토큰 출력 제한 및 타임아웃

응답 토큰 수가 지연 시간에 직접적인 영향을 미치기 때문에( ), 출력 길이 제한이나 타임아웃을 구현할 수 있습니다. 이러한 애플리케이션 계층 제약은 모델이 합리적인 토큰 수를 초과하여 계속 생성하는 무분별한 생성(runaway generation)을 방지할 수 있습니다.

토큰 출력 제한 및 타임아웃은 일관된 지연 시간 유지에 도움이 됩니다. 또한 악의적이거나 우발적인 prompt로 인해 GPU 자원을 점유할 수 있는 극도로 긴 생성 작업을 방지합니다. 많은 공개 API가 바로 이러한 이유로 엄격한 출력 제한을 설정합니다. 이는 남용 방지 메커니즘이자 성능 보호 장치 역할을 합니다.

서버 측 타임아웃도 반드시 설정하세요. 예를 들어 생성 시간이 30초를 초과하면 부분 결과나 사과 메시지를 반환합니다. 사용자는 응답이 멈춘 상태보다 빠른 실패를 기대합니다.

또한 모델이 장황하게 답변하는 경향이 있는지 모니터링하세요( ). 적당한 토큰 제한(예: 채팅 답변에 4,096토큰)을 적용하면 모델이 주제를 벗어나지 않고 긴 답변을 피하도록 유도하여 품질을 실제로 향상시킬 수 있습니다.

이러한 제한과 시간 초과 값은 사용자 요구에 기반하여 선택하는 것이 중요합니다. 이러한 제한은 지연 시간을 예측 가능하게 유지하고 최악의 경우 컴퓨팅 시나리오를 제한합니다. 이는 용량 계획 등에 도움이 됩니다.

요약하면, 입력 최적화(예: 압축 및 정제), 캐싱, 스마트 라우팅, UX 최적화(예: 스트리밍 데이터)의 조합은 작업 부하를 줄이고 추론 클러스터 규모를 축소하며 비용을 절감하고 사용자 만족도를 향상시킬 수 있습니다. 이러한 애플리케이션 수준 전략은 앞선 섹션의 저수준 최적화를 보완하여 LLM 서비스 성능 개선을 위한 종합적인 접근 방식을 완성합니다.

핵심 요약

본 장의 기법들은 효율적인 LLM 서비스가 현대적인 GPU 하드웨어, 혁신적인 소프트웨어 최적화, 포괄적인 모니터링을 결합한 종합적인 엔지니어링 노력임을 보여줍니다. 프로파일링 도구는 병목 현상을 식별합니다. 체계적인 디버깅 기법은 문제를 해결합니다. 본 장의 주요 요점은 다음과 같습니다:

포괄적인 자질

추론 스택 전반에 걸쳐 종단 간 자질을 수행하십시오. 프로파일러를 사용하여 각 단계의 지연 시간과 리소스 사용량을 측정함으로써 엔지니어는 속도 저하 및 비효율성을 정확히 파악할 수 있습니다. 이 데이터 기반 접근 방식은 병목 현상을 제거하기 위한 표적 최적화를 안내합니다.

모니터링 및 관측 가능성

배포된 추론 서비스에 대한 강력한 모니터링을 구현하십시오. 지연 시간 백분위수, 처리량, GPU 사용률, 메모리 사용량과 같은 핵심 지표를 실시간으로 추적하여 성능 저하나 자원 포화 상태를 조기에 감지하십시오. 로깅과 추적을 활용하여 요청별 처리 과정을 가시화하고 대규모 작업 부하에서 핫스팟이나 이상 현상을 식별하십시오.

디버깅 및 반복적 튜닝

체계적인 디버깅 워크플로를 채택하면 수만 개의 노드에 걸쳐서도 성능 및 정확성 문제를 신속하게 해결할 수 있습니다. 이렇게 하면 예상치 못한 급증(예: 처리량 감소 또는 지연 시간 증가)이 발생할 때 상위 수준의 증상에서 하위 수준의 문제까지 쉽게 파고들 수 있습니다.

메트릭을 통한 최적화 검증

GPU 디버거, 메모리 누수 탐지기, 성능 관련 유닛 테스트 같은 도구는 양자화나 커널 융합 같은 최적화가 성능 및 정확성 오류를 은밀히 유발하지 않는지 검증하는 데 도움이 됩니다. 이 반복적인 튜닝-테스트 사이클은 새로운 최적화를 프로덕션에 배포할 때마다 높은 성능과 신뢰성을 유지하는 데 필수적입니다.

효율성 및 비용 최적화

추론 비용 효율성을 높이는 개선에 집중하세요. 처리량과 활용도를 증가시키는 모든 최적화는 쿼리당 비용을 직접 개선합니다. 시스템 자질 및 개선을 통해 팀은 더 적은 GPU로 더 많은 요청을 처리할 수 있습니다. 이는 인프라 비용과 전력 효율성 측면에서 상당한 절감 효과를 가져옵니다.

결론

자질, 디버깅, 전체 스택 시스템 튜닝은 대규모 환경에서 효율적이고 신뢰할 수 있으며 비용 효율적인 LLM 추론을 유지하는 데 필수적입니다. 모델 규모가 수조 개의 매개변수로 성장함에 따라, 프로덕션 클러스터는 배포당 수십만 개, 심지어 수백만 개의 GPU로 확장되고 있습니다. 이러한 규모에서는 소프트웨어와 하드웨어의 지속적인 공동 설계가 여전히 중요합니다. 성능 향상을 위해 하드웨어에만 의존하는 것은 더 이상 충분하지 않습니다—특히 전력이 제한 요인으로 부상하고 있는 상황에서 더욱 그렇습니다. 소프트웨어와 하드웨어 모두 코디자인되어야 하며, 스택의 모든 계층에서 지속적으로 함께 튜닝되어야 합니다.

최적화된 추론 인프라는 최종 사용자에게 빠른 응답 시간, 예측 가능하고 안정적인 시스템 동작, 낮은 운영 비용을 제공합니다. 이를 통해서만 세계에서 가장 크고 강력한 모델을 지원하는 추론 시스템을 구현할 수 있습니다.

다음 장에서는 현대 LLM 추론 시스템에서 가장 많은 컴퓨팅, 메모리, 네트워크 자원을 소모하는 부분을 깊이 있게 이해하고 최적화하는 방법을 살펴보겠습니다: KV 캐시 계산(프리필링), 클러스터 내 모든 워커와의 공유, 그리고 이를 활용한 새 토큰 생성(디코딩) 과정입니다.