제5장. GPU 기반 스토리지 I/O 최적화

AI 워크로드에서 GPU에 데이터를 공급하는 것은 컴퓨팅 자체만큼 중요합니다. 수천 개의 GPU에서 100조 개의 매개변수를 가진 모델을 훈련하는 시나리오를 생각해 보십시오. 이러한 모델은 토큰, 이미지, 오디오, 비디오 등을 포함한 수십억 개의 훈련 샘플을 처리할 수 있습니다.

이는 방대한 양의 데이터를 스토리지에서 읽어내어 가능한 한 빠르게 GPU에 공급해야 함을 의미합니다. 스토리지 파이프라인이 느리다면 GPU는 자원이 부족해져 유휴 상태에 빠지게 됩니다. 이로 인해 앞서 논의한 정교한 통신 최적화에도 불구하고 낮은 활용률만 초래됩니다.

본 장에서는 스토리지 및 입력 파이프라인 최적화를 다룹니다. 구체적으로 디스크나 원격 스토리지에서 데이터를 효율적으로 읽는 방법, 이를 전처리하는 방법, 그리고 GPU 연산과 I/O 작업을 중첩시키는 방법을 보여줍니다.

빠른 스토리지와 데이터 로컬리티

대규모 모델 훈련 작업은 일반적으로 방대한 데이터셋을 읽어야 합니다. 대규모 언어 모델의 경우 수십억에서 수조 개의 훈련 샘플을 처리하는 것이 일반적입니다. 이는 언어 모델의 경우 테라바이트 단위의 텍스트 데이터, 비전 모델의 경우 페타바이트 단위의 이미지 데이터에 해당합니다.

초대규모 환경에서는 수천에서 수백만 개의 GPU가 수개월 동안 동시에 가동될 수 있으므로, 스토리지 시스템은 지속적으로 막대한 처리량을 제공해야 합니다. 랙 내부에 NVMe SSD를 배치하거나 랙 로컬 스위치 토폴로지와 함께 NVMe over Fabrics(NVMe-oF)를 사용하면 네트워크 홉을 최소화하고 성능 일관성을 향상시킬 수 있습니다.

데이터가 NFS 서버나 클라우드 객체 저장소(예: Amazon S3)와 같은 네트워크 연결 저장소(NAS)에 존재하는 경우, 모든 컴퓨팅 노드에서 합산된 읽기 대역폭이 충분한지 확인해야 합니다. 모델과 배치 크기에 따라 각 GPU가 200MB/s의 훈련 데이터를 처리해야 하는 시나리오를 가정해 보겠습니다. 총 8개의 GPU가 있다면 약 1.6GB/s의 총 대역폭이 필요합니다. Blackwell 및 Rubin과 같은 최신 하이엔드 GPU는 포화 상태를 유지하기 위해 더 많은 대역폭을 요구합니다.

NVIDIA Grace Blackwell GB200/GB300 NVL72 랙은 72개의 Blackwell GPU를 하나의 NVLink 도메인으로 연결합니다. 각 GPU가 200MB/s의 훈련 데이터를 처리해야 할 경우, 72개 GPU 모두를 가동시키려면 총 14~20GB/s의 스토리지 처리량이 필요합니다. 이러한 초고성능 워크로드에는 스토리지 솔루션도 이에 맞춰 확장되어야 합니다.

워크로드가 더 무거운 미디어나 다중 모달 샘플을 스트리밍하는 경우, 측정된 샘플당 바이트 수와 초당 샘플 수를 기준으로 보정하십시오. 이러한 경우 총 수요가 훨씬 더 높을 수 있습니다.

한 가지 해결책은 동일한 랙 내 NVMe SSD와 같은 고속 로컬 스토리지 또는 NVMe-oF 네트워크 토폴로지를 사용하는 것입니다. 다른 해결책은 Lustre나 GPFS(General Parallel File System) 등과 같은 병렬 파일 시스템을 활용해 로컬 SSD에 데이터를 캐싱하는 것입니다. 스토리지 시스템이 이를 따라잡을 수 있다고 가정할 때, 파이프라인을 포화 상태로 유지하기 위해 다중 데이터 로딩 스레드를 프로비저닝하는 것이 중요합니다. Python GIL에 주의하세요!

가능한 경우 데이터를 컴퓨팅 노드에 물리적으로 최대한 가깝게 배치하십시오. "가까움"은 로컬 NVMe SSD 드라이브처럼 동일한 물리적 노드에 위치하거나, 최소한 NVMe over Fabric(NVME-oF)이나 고급 스토리지 가속기와 같은 고속 상호 연결을 통해 동일한 랙 내에 위치함을 의미할 수 있습니다.

분산 다중 노드 모델 훈련의 경우, 데이터셋을 노드 간에 분할하여 각 노드가 주로 로컬 디스크의 데이터 하위 집합을 읽도록 하는 것이 일반적인 접근법입니다. 예를 들어, 100TB의 데이터와 10개의 노드가 있다면 각 노드의 로컬 스토리지에 10TB씩 미리 분할할 수 있습니다. 그러면 각 노드의 데이터 로더는 해당 로컬 10TB에서만 읽습니다. 이는 특히 데이터셋 크기가 RAM에 쉽게 들어가지 않을 때 중복 읽기로 네트워크가 포화되는 것을 방지합니다.

와 같은 프레임워크나 PyTorch의 `DistributedSampler` 는 각 프로세스가 에포크마다 고유한 데이터 조각을 할당받도록 작업자를 조정합니다. 이는 데이

터를 여러 클러스터 노드에 분할하는 목표와 잘 부합합니다.

순차적 대 무작위 읽기 패턴

GPU는 데이터 처리 속도가 매우 빠르지만, 효율성을 위해 데이터를 큰 연속 블록으로 읽는 것을 선호합니다. 마찬가지로 저장 장치도 작은 무작위 읽기보다 큰 순차적 읽기에서 훨씬 높은 처리량을 보입니다. 따라서 데이터셋이나 저장 레이아웃을 준비할 때는 가능한 한 순차적 접근이 이루어지도록 구성하세요.

예를 들어, 이미지 훈련 시 수백만 개의 개별 이미지 파일을 저장하는 것은 디스크 전체에 걸쳐 무수한 무작위 탐색을 유발하므로 피해야 합니다. 대신, 몇 개의 대형 바이너리 파일(예: Arrow, TFRecord, Parquet), 데이터베이스 파일, WebDataset tar 파일 또는 이에 상응하는 형식으로 저장하는 것을 고려하십시오. 이러한 경우 각 파일에는 많은 샘플이 연결되어 포함되므로 이상적입니다.

오늘날의 고속 GPU 환경에서는 작은 파일을 의 대규모 샤드로 통합하는 것이 더욱 중요합니다. 과도한 소규모 무작위 읽기 작업이 더 빠르게 병목 현상을 유발하기 때문입니다. 대부분의 현대적 병렬 파일 시스템과 객체 저장소가 어느 정도 소규모 무작위 읽기를 처리할 수 있지만, 성능을 명시적으로 검증하는 것이 최선입니다.

대용량 파일에서 데이터 청크를 읽으면 자연스럽게 한 번에 많은 샘플을 처리할 수 있습니다. Amazon S3 같은 객체 저장소를 사용할 경우, 바로 이 이유로 사전에 작은 객체를 큰 객체로 결합하는 것이 일반적입니다.

또한 읽기 크기( )를 조정하는 것이 중요합니다. 1MB 단위로 읽으면 4KB 단위보다 읽기당 오버헤드가 낮아 처리량이 향상되기 때문입니다. 많은 데이터 로더 라이브러리는 버퍼 크기 및 프리페치 청크 크기를 조정할 수 있습니다. 예를 들어 Python의 `open()` 는 순차 스캔 가속화를 위해 OS의 미리 읽기 버퍼를 사용하지만, 무작위 읽기는 더 큰 버퍼나 버퍼링된 I/O 라이브러리로부터 큰 이점을 얻지 못합니다.

대신, 읽기를 더 큰 연속적인 청크로 배치하거나 고수준 데이터셋 API(예: `TFRecordDataset` 또는 PyTorch의 `IterableDataset` 및 `DataLoader`, 프리페치 크기 설정 가능)를 사용해야 합니다. 이러한 프레임워크와 라이브러리 대부분은 내부적으로 대용량 순차 읽기에 최적화되어 있지만, 버퍼 및 프리페치 매개변수 조정은 여전히 중요합니다.

액세스 패턴이 여전히 무작위여야 한다면, `pread()` 를 호출하는 스레드나 `io_uring` 같은 Linux 비동기 I/O 인터페이스를 사용하여 병렬로 다중 읽기를 수행하십시오. 사전 등록된 버퍼 및 폴링과 같은 기능을 통해 `io_uring` 는 커널 오버헤드를 최소화하면서 I/O 요청 배치를 제출할 수 있습니다. 이는 시스템 호출당 오버헤드를 줄여 무작위 읽기 처리량을 더욱 향상시킬 수 있습니다. 이는 지연 시간을 숨기고 높은 IOPS를 달성하는 데 도움이 됩니다.

대규모 동시 I/O에는 최적화된 파일 시스템 을 사용해야 합니다. Linux NVMe 서버에서는 XFS가 흔히 사용됩니다. 각 읽기 시마다 발생하는 비용이 큰 접근 시간 업데이트를 제거하려면 `noatime` 옵션으로 마운트해야 합니다. Amazon EFS와 같은 네트워크 스토리지 서비스의 경우, 최고 수준의 집계 처리량을 위해 EFS 파일시스템이 최대 I/O 성능 모드 (Max I/O performance mode)로 설정되어 있는지 확인하십시오. 일관된 대역폭이 필요한 경우 기본 버스팅 처리량 모드 (Bursting throughput mode)에서 프로비저닝 처리량 모드(Provisioned throughput)로 전환할 수 있습니다. 이러한 설정은 I/O 계층이 대규모 병렬 AI 워크로드를 처리할 수 있도록 보장합니다.

처리량 향상을 위한 NVMe 및 파일 시스템 조정

현대적인 Linux는 다중큐 블록 I/O 스케줄러인 를 사용하며, 이는 I/O 작업을 CPU 코어에 분산합니다. `blk-mq` 빠른 NVMe SSD의 경우 큐 깊이와 제출 큐 수를 조정해야 할 수 있습니다. 일반적으로 기본값으로도 충분하지만, 작업 부하가 순차적 작업이 많다고 판단되면 "none" I/O 스케줄러를 사용할 수 있습니다.

레거시 완전 공정 큐잉(CFQ) 스케줄러 는 더 이상 사용되지 않습니다. 최신 커널은 NVMe에 대해 기본적으로 `none` 또는 `mq-deadline` 멀티큐 스케줄러를 사용합니다. 이 설정은 `/sys/block/<device>/queue/scheduler` 를 통해 확인할 수 있습니다. "none" 스케줄러는 저지연 워크로드에 표준으로 사용됩니다. 일부 저장 장치에서는 예산 공정 큐잉(BFQ) 스케줄러를 접할 수도 있습니다.

고성능 NVMe의 경우, 처리량을 극대화하기 위해 여전히 `none` 또는 `mq-deadline` 멀티큐 스케줄러를 사용하는 것이 권장됩니다.
`/sys/block/nvme*/queue/scheduler` 를 사용하여 스케줄러를 확인하고 설정할 수 있습니다. 기본적으로 거의 항상 올바르게 구성되어 있지만, 간단한 확인을 통해 검증하는 것이 좋습니다.

또 다른 튜닝 요소는 미리 읽기(read ahead)입니다. 커널은 순차 읽기를 감지하면 자동으로 추가 데이터를 미리 읽습니다. 미리 읽기 설정은 `/sys/block/<device>/queue/read_ahead_kb` 에서 확인할 수 있습니다. 예를 들어 기본값은 128KB로 설정되어 있을 가능성이 높습니다. 대용량 파일을 스트리밍하는 경우 이 값을 몇 MB로 늘리십시오. 이는 시스템 호출 오버헤드를 줄이고 읽기 작업을 파이프라인 처리하여 처리량을 향상시킵니다. 이는 장치에 대해 `blockdev --setra` 를 사용하여 설정할 수 있습니다.

NVMe SSD 디스크를 사용하는 경우, 시스템에서 사용 가능한 가장 빠른 인터페이스에 설정되었는지 확인하십시오. 또한 병목 현상이 발생하지 않도록 충분한 레인(예: PCIe)이 확보되었는지 확인하십시오. 때로는 여러 SSD를 RAID 0 등으로 스트라이핑하여 이러한 장치를 완전히 활용하고 처리량을 극대화할 수 있습니다. 특히 단일 디스크로는 GPU를 완전히 활용하지 못하는 경우에 유용합니다.

Linux 페이지 캐시는 최근에 읽은 데이터를 디스크에서 RAM으로 자동 캐싱합니다. 대규모 데이터셋의 경우 사용 가능한 RAM을 초과하여 캐시 스래싱이 발생할 수 있습니다. 그러나 중간 규모의 데이터셋에서는 워밍된 캐시가 훈련 속도를 크게 향상시킬 수 있습니다.

데이터 전체 또는 상당 부분이 RAM(예: Grace Blackwell Superchip의 CPU + GPU 통합 메모리 포함)에 수용될 수 있다면, 시작 시 메모리에 완전히 사전 로드하는 것을 고려해야 합니다. 이는 GPU를 위한 초고속 인메모리 캐시를 효과적으로 생성합니다. 이를 통해 훈련 중 디스크 I/O를 크게 줄일 수 있습니다. 그러나 페타바이트 규모의 방대한 데이터 세트의 경우 일반적으로 실현 불가능합니다. 이러한 경우 최적화된 I/O로 데이터를 스트리밍하는 것이 해결책입니다.

데이터 로딩 시 반드시 다중 작업자 를 사용하세요(예: PyTorch의 `DataLoader(num_workers=N)`). 이렇게 분리된 CPU 스레드/프로세스가 데이터를 병렬로 가져와 전처리하여 훈련 작업의 다수 GPU에 공급합니다. 적절한 작업자 수는 경험적으로 결정해야 합니다.

PyTorch 성능 튜닝은 13장과 14장에서 자세히 다루겠지만, 여기서 `pin_memory=True` 를 활성화하고 `non_blocking=True` 를 사용해 호스트-장치 간 복사 작업이 중첩되도록 설정해야 한다는 점을 언급할 가치가 있습니다. 또한 `persistent_workers=True` 를 설정하면 에포크 간 작업자 재생성 오버헤드를 피할 수 있습니다. 작업 부하별로 `prefetch_factor` 를 조정하는 것도 유용합니다. 기본값 `prefetch_factor` 는 `num_workers` 가 0보다 클 때 2입니다.

워커 수가 너무 적으면 GPU가 유휴 상태가 됩니다. 워커 수가 너무 많으면 스레드들이 사용 가능한 CPU 코어와 I/O 대역폭을 놓고 경쟁하기 시작합니다. CPU 사용률과 디스크 처리량을 모니터링하세요. 이상적으로는 디스크 처리량 활용률이 100%에 가까우면서 CPU에는 어느 정도 여유가 있는 상태를 유지하는 것이 좋습니다.

GB200/GB300 슈퍼칩에 사용된 72코어 NVIDIA Grace CPU와 같이 코어 수가 매우 많은 CPU의 경우, 더 많은 데이터 로더 작업자를 활용할 수 있습니다. 다만 과도한 I/O 경합으로 인한 수익 감소 효과에 주의해야 합니다.

NVIDIA GDS 사용

GDS는 GPU가 CPU 메모리에 추가 복사본을 생성하지 않고 저장 장치 또는 네트워크 스토리지 스택을 통해 데이터를 직접 읽을 수 있게 하는 기능입니다. 일반적으로 GPU가 NVMe SSD에서 데이터를 읽으려면 데이터가 먼저 SSD에서 CPU 메모리로 이동합니다. 그런 다음 CUDA 호출이 CPU 메모리에서 GPU 메모리로 데이터를 복사합니다.

GDS는 스토리지-GPU 간 DMA를 가속화하는 반면, GPUDirect RDMA는 네트워크-GPU 간 DMA를 가속화하므로 두 기술은 상호 보완적입니다. 둘 다 CPU 오케스트레이션을 제거하지는 않지만, 호스트 메모리 바운스 버퍼를 제거합니다.

GDS를 사용하면 GPU가 SSD 또는 NIC에 대해 직접 메모리 액세스(DMA)를 시작하여 데이터를 자체 HBM 메모리로 이동시킬 수 있습니다. 이는 CPU 경로를 통한 추가 복사 작업을 우회합니다. GDS는 로컬 NVMe 장치와 NVMe-oF를 사용하는 원격 스토리지를 지원합니다.

실제 적용 시 GDS는 스토리지와 GPU 메모리 간 호스트 메모리 바운스 버퍼를 우회하는 직접 DMA 경로를 생성합니다. 이로 인해 GDS의 적용 범위가 클러스터 파일 시스템 및 일부 객체 스토리지 시스템까지 확대됩니다. (참고: CPU는 여전히 I/O를 구성하고 조정합니다.)

GDS를 활성화하려면 최신 NVIDIA GPU와 직접 메모리 액세스를 지원하는 스토리지 스택, 그리고 올바른 NVIDIA 드라이버 및 CUDA 툴킷이 필요합니다. 일반적으로 로컬 NVMe SSD 또는 RAID 볼륨이 사용됩니다. GDS 지원은 파일 시스템과 RDMA 지원 스택에 따라 달라집니다. 현재 지원되는 스택은 다음과 같습니

다: - 로컬 NVMe 및 NVMe-oF 기반 XFS/EXT4 파일시스템(`O_DIRECT` 사용) - RDMA 기반 NFS - BeeGFS, WekaFS, VAST, IBM Storage Scale 등 `nvidia-fs` 와 통합되는 특정 병렬 파일시스템

애플리케이션은 올바른 API를 사용해야 합니다. GDS를 통해 파일을 읽으려면 CUDA의 `cuFile` 라이브러리를 사용할 수 있습니다. `cuFile` 는 자동 버퍼 정렬 및 일반적인 파일 시스템과의 통합과 같은 기능을 지원합니다.

실질적으로 GDS가 설정되어 있고 읽기 경로가 `cuFileRead` 를 사용하는 경우, 데이터가 디스크에서 GPU 메모리로 직접 이동할 수 있습니다. 이는 CPU 사용률을 낮추고(CPU가 다른 전처리를 수행할 수 있게 함) 특히 CPU가 병목 현상일 때 처리량을 향상시킬 수 있습니다. `cuFileRead` 는 Linux 파일 시스템과 직접 통합됩니다. 또한 `cuFileReadAsync` 및 `cuFileWriteAsync` 과 같은 `cuFile's` 비동기 API를 사용하여 CUDA 스트림(11장에서 설명)에 스토리지 I/O를 통합하여 중첩 및 파이프라인 처리를 구현할 수 있습니다.

`O_DIRECT` 가능한 경우 를 사용하여 직접 DMA를 활성화하고 OS 페이지 캐시를 우회하십시오. 최신 GDS 릴리스에서는 `cuFile` 가 비 `O_DIRECT` 파일 디스크립터에서도 작동할 수 있지만, 정렬 불일치로 인해 추가 복사 작업이 발생하거나 성능이 저하될 수 있습니다.

WekaIO, DDN, VAST, Cloudian 등 많은 스토리지 벤더들이 GDS 인식 솔루션이나 플러그인을 출시하여 자사 시스템이 RDMA를 통해 데이터를 GPU 메모리로 직접 전달할 수 있도록 했습니다. 이러한 생태계 지원 덕분에 GDS는 기업용 네트워크 연결 스토리지(NAS) 및 병렬 파일 시스템에서 바로 사용할 수 있습니다.

VAST Data의 보고서에 따르면 특정 AI 워크로드에서 GDS 사용 시 읽기 처리량이 20% 향상되었습니다. 해당 사례에서 단일 A100 GPU에 GDS를 적용했을 때 순차 읽기 처리량이 20% 증가했으며, 이는 해당 NIC당 100Gb/s 링크 용량에 상당히 근접한 수준이었습니다. 그림 5-1은 GDS 적용 여부에 따른 아키텍처를 보여줍니다.

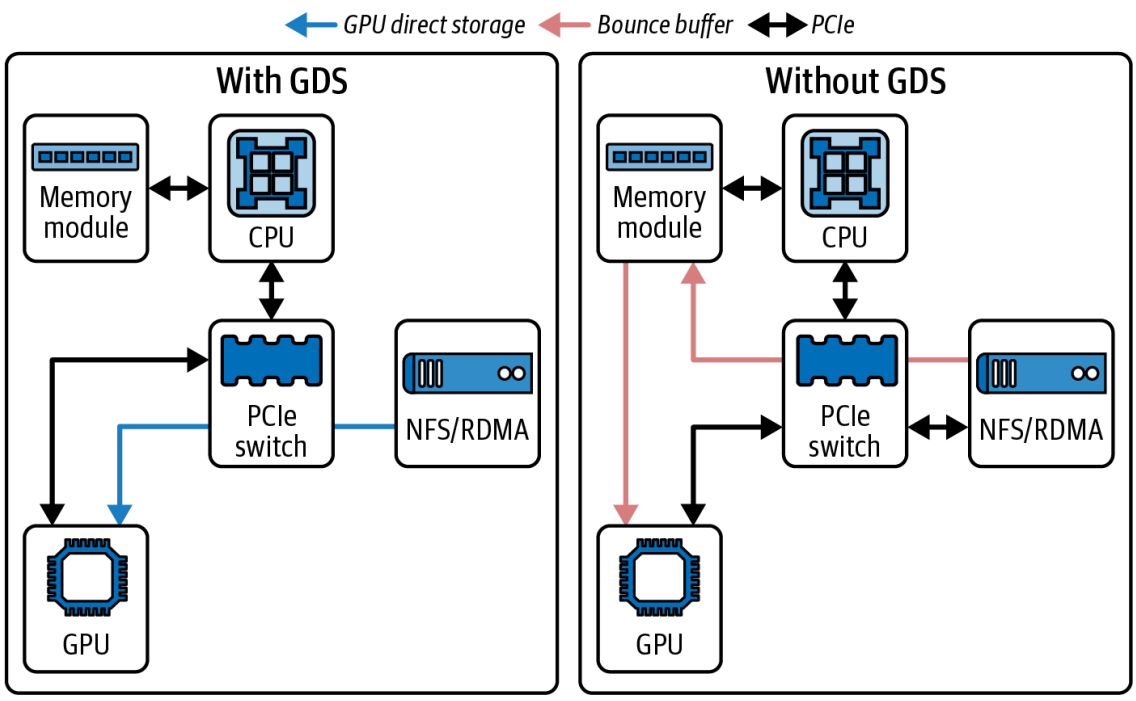

그림 5-1. GDS 적용 전후VAST Data의 네트워크 아키텍처 비교

왼쪽은 호스트 메모리를 경유하는 전통적인 단계별 DMA 방식입니다. 오른쪽은 호스트 메모리 복사 과정을 우회하고 CPU 사용률을 낮추는 GDS 기반의 직접 GPU 풀 방식입니다. VAST 보고서에 따르면 NVIDIA Ampere A100 GPU에서는 20%, NIC 대역폭이 더 넓고 CPU 부하가 더 큰 Hopper H100 GPU에서는 30% 이상의 읽기 처리량 향상이 측정되었습니다.

IO 크기, 큐 깊이, NIC 세대, 파일 시스템 구현 등에 따라 성능 향상이 달라지므로 워크로드와 패브릭에서 검증하십시오.

그러나 GDS는 튜닝이 필요할 수 있으며, 모든 워크로드에서 큰 향상을 보이는 것은 아닙니다. CPU가 데이터 전송을 쉽게 처리하고 있다면 GDS가 처리량에 큰 변화를 주지 않을 수 있습니다. 다만 CPU 사용률을 낮추어 데이터 처리 및 기타 작업 수행을 위한 CPU 자원을 확보해 줍니다. 반면 CPU가 다수의 스레드 간 전송(`memcpy`) 작업으로 포화 상태라면 GDS가 큰 도움이 될 것입니다.

GDS 사용 시 ' `O_DIRECT` ' 세마틱 과 정렬이 올바르게 적용되었는지 확인해야 합니다. 스토리지-GPU 데이터 경로에서는 호스트 고정 메모리가 사용되지 않습니다. `cuFile` 는 GPU 장치 버퍼를 등록하며, `nvidia-fs` 커널 드라이버는 스토리지 장치 또는 RDMA NIC과 GPU 메모리 간 DMA를 직접 조정합니다. POSIX 파일 디스크립터와 직접 통합되므로, RDMA를 지원하는 네트워크 파일 시스템을 포함한 일반 파일에서 `cuFile` 을 사용할 수 있습니다.

1MB의 작은 훈련 배치 크기를 사용하고 초당 1,000개의 배치를 GPU에 공급하려는 경우를 생각해 보십시오. 이는 대략 1,000MB/s에 해당합니다. CPU로 이 복

사 작업을 수행하면 쉽게 몇 개의 코어를 소모하게 됩니다. GDS를 사용하면 GPU가 디스크에서 직접 1,000MB/s를 가져와 CPU를 해방시킵니다. 더 높은 속도나 수천 개의 GPU를 사용할 경우 이 효과는 더욱 두드러집니다.

훈련 워크로드가 압도적으로 읽기 중심이기 때문에, 대부분의 GDS 성능 향상은 스토리지에서 데이터를 읽을 때 평가됩니다. 그러나 빠른 체크포인트 쓰기도 중요합니다. RDMA 가속 쓰기의 경우, 파일 시스템은 GDS용 RDMA 쓰기를 지원해야 합니다.

WekaFS는 초대형 AI 훈련 워크로드를 위한 잘 알려진 스토리지 공급자입니다. RDMA를 통한 읽기 및 쓰기 워크로드 모두를 위한 GDS 인식 플러그인이 포함된 병렬 파일 시스템을 제공합니다.

cuda-checkpoint를 이용한 GPU 상태 체크포인트

`cuda-checkpoint` Linux에서 NVIDIA의 cuda-checkpoint 유틸리티와 Userspace의 Checkpoint/Restore(CRIU) 같은 CPU 프로세스 체크포인트 도구를 함께 사용해 GPU 상태를 체크포인트할 수 있습니다. `cuda-checkpoint` 는 실행 중인 프로세스 내 CUDA를 일시 중지하고, 제출된 작업이 완료될 때까지 대기한 후, 장치 메모리를 드라이버가 관리하는 호스트 할당 영역으로 복사하고 GPU 리소스를 해제합니다. 이렇게 하면 CPU 측 체크포인트 도구가 프로세스 스냅샷을 생성할 수 있습니다.

정지 경로는 CUDA 드라이버 진입점을 잠그고, 미처리 작업을 처리하며, 장치 메모리를 호스트로 복사한 후 GPU 리소스를 해제합니다. 정지 시간 추정 시 사용 중인 장치 메모리 양과 정지 중 사용 가능한 호스트 링크 대역폭을 고려해야 합니다.

드라이버가 중지 단계에서 장치 메모리를 호스트 할당 영역으로 복사하므로, 실제 중지 시간은 메모리 이미지 크기와 플랫폼 간 연결 성능에 의해 바운디드됩니다. 중지 단계에서 소요된 실제 시간을 확인하려면 잠금 및 체크포인트 호출 주변에 Nsight Systems 마커를 사용하여 자질해야 합니다.

프로세스 재개를 원할 경우, 드라이버는 GPU를 재획득하고, 장치 메모리를 원래 주소에 매핑하며, 스트림 및 컨텍스트와 같은 CUDA 객체를 복원한 후, 드라이버와 프로세스의 잠금을 해제하여 CUDA 호출이 진행되도록 합니다.

구체적으로 CUDA 드라이버 API는 다음 함수를 제공합니다: `cuCheckpointProcessLock` `cuCheckpointProcessCheckpoint` `cuCheckpointProcessRestore` `cuCheckpointProcessUnlock` 복원(Restore)은 지속성 모드 활성화(또는 `cuInit` 호출)가 필요하며, 동일한 칩 유형의 다른 물리적 GPU로 재매핑될 수 있습니다.

이 경로는 프레임워크 수준 모델 체크포인트(예: PyTorch 체크포인트)와는 별개임을 유의해야 합니다. CUDA 체크포인트는 장시간 실행되는 훈련 및 추론 작업의 결함, 선점, 마이그레이션에 유용합니다.

GDS를 통한 데이터 인제스트와 달리, 체크포인트 경로는 GPU 메모리에서 스토리지로 직접 DMA하지 않습니다. 대신, 드라이버가 일시 중지 중에 장치 메모리 이미지를 먼저 호스트 메모리로 가져옵니다. 그런 다음 CRIU가 해당 프로세스 메모리를 체크포인트 이미지로 영구 저장합니다. 프레임워크의 상태 딕셔너리(state-dict)나 분할된 체크포인트 파일을 대체하지 않고 보완하는 용도로 사용하십시오.

gdsio를 통한 GDS 측정

NVIDIA는 디스크와 GPU 간 GDS 처리량 벤치마킹을 위한 도구인 `gdsio`(기본 설치 경로: `/usr/local/cuda/gds/tools`)를 제공합니다. 이 도구는 매우 유용합니다.

GDS 사용 시, 특히 CPU 제약 시나리오에서 처리량이 10~20% 이상 향상되는 경우가 흔합니다. NVIDIA의 `gdsio` 도구를 사용하여 순수 CPU 매개 읽기("before")와 직접 GDS 읽기("after")를 비교한 예시를 살펴보겠습니다. 다음은 CLI 명령어와 처리량/지연 시간결과입니다:

```
# Before (Storage → CPU Memory only)

# CPU path, host memory, async copies (-x 2)
$ /usr/local/cuda/gds/tools/gdsio \
    -f /mnt/data/large_file \
    -d 0 -w 4 -s 10G -i 1M -I 0 -x 2

Total Throughput: 8.0 GB/s
Average Latency: 1.25 ms
```

첫 번째 호출은 호스트 메모리 고정 및 비동기 복사(-x 2)를 적용한 CPU 경로(읽기 모드 -I 0)로 기준값을 수집합니다. 두 번째 호출은 동일한 구성에서 GDS 경로(-x 0)를 읽기 모드(-I 0)로 활성화합니다. 경로 비교 시 전송 선택기를 일관되게 사용해야 합니다. `gdsio` 의 경우 `-x 2` 는 CPU 매개 전송을 측정하고, `-x 0` 는 GDS 경로를 측정합니다:

```
# After (Storage → GPU Memory using GPUDirect Storage)
#   - same config, GDS path (-x 0)
$ /usr/local/cuda/gds/tools/gdsio \
    -f /mnt/data/large_file \
    -d 0 -w 4 -s 10G -i 1M -I 0 -x 0

Total Throughput: 9.6 GB/s
Average Latency: 1.00 ms
```

GDS를 사용하여 디스크에서 GPU 메모리로 직접 데이터 경로를 생성하면, 표 5-1에 표시된 바와 같이 처리량이 20% 증가하고 평균 I/O 지연 시간이 감소합니다. 이는 호스트 버퍼를 통해 데이터를 이동하는 데 사용되던 CPU 사이클을 확보하면서 이루어집니다. 이 간단한 벤치마크는 시스템에서 GDS의 이점을 검증하는 방법을 보여줍니다.

표 5-1. GDS 적용 전후의처리량 및 지연 시간 비교

경로	처리량	지연 시간
스토리지 → CPU (GDS 제외)	8.0 GB/s	1.25 밀리초
스토리지 → GPU (GDS 포함)	9.6 GB/s (+20%)	1.00 밀리초 (–20%)

이 예시에서 GDS(저장소 → GPU) 사용 시 읽기 처리량은 8.0 GB/s에서 9.6 GB/s로 증가했으며, 지연 시간은 1.25 ms에서 1.00 ms로 감소했습니다. 이는 처리량(증가)과 지연 시간(감소) 모두 약 20% 개선된 것을 의미합니다.

DeepSeek의 Fire-Flyer 파일 시스템

DeepSeek는 AI 워크로드가 방대한 양의 무작위 읽기 작업을 수행한다는 관찰에서 출발해, 완전히 새로 설계된 커스텀 오픈소스 파일 시스템인 Fire-Flyer File System(3FS) 을 개발했습니다.

이러한 무작위 읽기 작업은 기존 읽기 데이터 캐싱 방식을 LLM 훈련 및 추론 워크로드에 비효율적일 뿐만 아니라 오히려 역효과를 내게 합니다. 캐싱을 제거하

고 직접 파일 I/O를 활용함으로써, 3FS는 모든 요청이 NVMe SSD 장치로 직접 전달되도록 보장하며 비효율적인 캐시 관리를 피합니다. 이 접근 방식은 직접 스토리지 접근을 우선시하는 현대적 HPC 파일 시스템과 유사합니다. 따라서 3FS는 읽기 작업 시 커널 페이지 캐시 개입과 호스트 메모리 복사 작업을 최소화합니다.

3FS는 AI 전용으로 설계된 스토리지 코드 설계 트렌드를 반영합니다. 이는 고성능 병렬 파일시스템과 연동하여 유사한 직접 GPU 처리량을 달성하도록 설계된 NVIDIA의 GDS와 유사합니다.

3FS는 클러스터 관리자, 메타데이터 서비스, 스토리지 서비스, 클라이언트라는 네 가지 핵심 구성 요소로 이루어집니다. 이 구성 요소들은 InfiniBand나 RoCE와 같은 RDMA 지원 패브릭을 통해 상호 연결되어 CPU 개입과 호스트 측 복사 작업을 최소화합니다. 이러한 구성 요소와 연결 구조는 그림 5-2에 표시되어 있습니다.

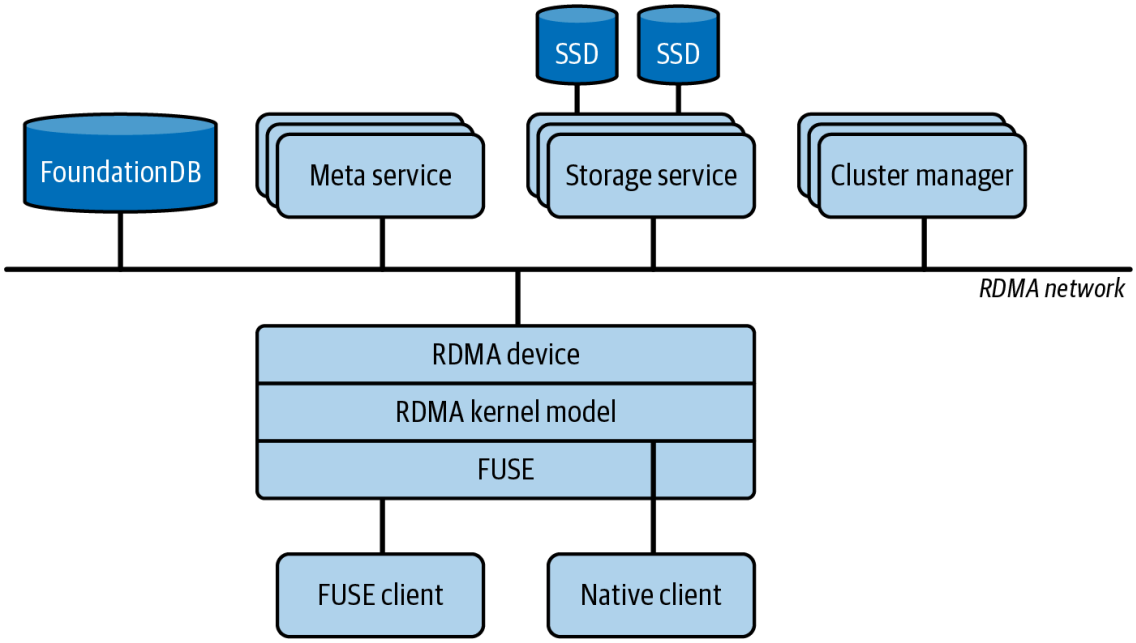

그림 5-2. DeepSeek의 Fire-Flyer 파일 시스템(3FS)구성 요소 (출처: *https://oreil.ly/xD3id*)

3FS는 Linux 기반 파일 시스템으로, 기존 애플리케이션과의 호환성을 유지하면서 RDMA 읽기를 활용해 GPU가 직접 접근 가능한 데이터 전송을 수행합니다. 메타데이터는 확장성 확보를 위해 여러 노드에 분산 및 복제됩니다. 데이터 경로는 최적의 처리량을 유지하기 위해 OS 페이지 캐시를 완전히 우회합니다.

사용자 공간에서 FUSE를 사용하여 구현된 파일 시스템 은 커널 수준의 파일 시스템 통합과 커널 간(`O_DIRECT`) 의미론이 필요한 GDS 경로를 제공할 수 없습니다. GDS 지원 커널 클라이언트 또는 특별히 통합된 병렬 파일 시스템만이 GPU 메모리로의 직접 전송을 제공할 수 있습니다.

GPU 파이프라인에 데이터를 직접 공급하기 위해 DeepSeek는 3FS에 RDMA 기반 전송을 통합합니다. 진정한 GDS 경로가 필요한 경우 NVMe, NVMe-oF, BeeGFS, WekaFS, IBM Storage Scale 또는 VAST와 같은 GDS 지원 커널 파일 시스템 클라이언트를 사용하십시오. 이를 통해 최소한의 오버헤드로 GPU 장치 메모리에 직접 비동기식 제로 카피 데이터 이동이 가능합니다.

3FS는 데이터 프리페치 및 전송이 GPU 커널과 동시에 실행되도록 지원함으로써, 본 장에서 소개한 I/O와 계산을 중첩하는 기법을 보완합니다. 3FS는 캐스케이딩 파이프라인/웨이브 개념(4장에서 논의됨)을 스토리지 계층으로 효과적으로 확장합니다.

DeepSeek는 대규모 클러스터에서 3FS의 초당 멀티 테라바이트(TB/s) 수준의 집계 읽기 처리량을 공개적으로 보고했으며, 해당 환경에서 최대 7.3 TB/s의 결과를 달성했습니다. 다른 벤치마크에서는 68노드 AI-HPC 클러스터(10×16TB NVMe SSD 및 듀얼 100Gb/s)를 사용한 대규모 3FS 클러스터가 6.6TB/s 수준의 집계 읽기 처리량을 달성했습니다. 이는 배경 워크로드를 추가로 1.4TB/s 처리하면서 동시에 이루어졌습니다. 보고된 3FS 처리량인 6.6TB/s는 유사한 하드웨어에서 Ceph의 약 1.1TB/s를 훨씬 뛰어넘습니다.

3FS는 노드 간 I/O를 조정함으로써 이러한 성능을 달성합니다. 이러한 수준의 지속적 대역폭은 데이터 스테이징 단계가 병목 현상이 되는 것을 방지하고 , 훈련 및 추론 워크로드 전반에 걸쳐 GPU 활용도를 높게 유지하는 데 도움이 됩니다.

DeepSeek는 랜덤 읽기에 최적화된 자체 파일 시스템을 구축하고 RDMA 우선 데이터 경로와 통합함으로써, 대규모 AI 시스템의 성능 잠재력을 최대한 활용하기 위해서는 스토리지 설계를 포함한 엔드투엔드 풀스택 성능 엔지니어링이 필수적임을 입증합니다.

3FS는 스토리지 계층을 재고함으로써 I/O 병목 현상의 마지막 잔여 부분을 제거할 수 있음을 보여줍니다. 자체 파일 시스템을 구축하는 것은 상당한 초기 투자와 지속적인 유지 관리가 필요한 고급 기술입니다. 대신 기존 분산 파일 시스템이나 객체 저장소를 기반으로 시작하는 것이 더 현실적입니다. 다음으로 이들에 대해 논의해 보겠습니다.

분산형 병렬 파일 시스템 및 객체 저장소

여러 노드에서 훈련할 때, 일반적으로 NFS 서버와 같은 공유 파일 시스템이나 Lustre, GPFS, Ceph 등과 같은 병렬 파일 시스템을 사용하는 설정이 일반적입니다. 이러한 시스템을 사용하면 모든 노드가 동일한 데이터 세트에 접근할 수 있습니다. 편리하지만, 이러한 파일 시스템은 제대로 구성되지 않으면 병목 현상이 발생할 수 있습니다.

설정은 간단하지만, 많은 노드가 동시에 읽기를 수행할 경우 단일 NFS 서버는 쉽게 처리량 병목 현상을 유발할 수 있습니다. NFS 사용이 불가피하다면 서버에 고속 NIC를 다수 장착해야 합니다. 또한 데이터셋을 분할하여 각 서버가 데이터의 일부를 처리하도록 여러 NFS 서버를 사용하는 것도 고려해야 합니다.

다중 GPU 클러스터의 경우, 소규모(몇 개의 노드 등)에만 NFS를 고려해야 합니다. 대규모 훈련 클러스터에서는 단일 NFS 서버(고성능 구현체라도)가 병목 현상을 일으킬 가능성이 높습니다. 이 때문에 현대적인 AI 훈련 클러스터에는 병렬 파일 시스템이나 Amazon FSx for Lustre와 같은 클라우드 스토리지 캐시가 선호됩니다.

Amazon FSx for Lustre와 같은 클라우드 스토리지 캐시 의 경우 성능 향상 효과를 검증하고 캐시 추가 비용을 정당화하는 것이 중요합니다. 기대하는 성능을 얻지 못한다면 클라우드 공급자와 직접 협력하여 아키텍처를 검증하고 구성 설정을 확인하십시오.

NFS에는 rsize/wsize(읽기/쓰기 요청 크기)와 같은 조정 매개변수( )도 있습니다. 처리량 향상을 위해 최대값(예: 1MB) 사용을 권장합니다. 기본 NFS 스토리지가 NVMe SSD를 사용해 충분히 빠른지 확인하고, 필요 시 RAID 0 구성으로 설정하세요. NFS 클라이언트 마운트 옵션도 반드시 점검해야 합니다. 이 역시 조정해야 합니다.

예를 들어 `rsize=1048576,wsize=1048576,noatime,async` 옵션을 사용하여 NFS 클라이언트를 마운트하면 1MiB 블록을 사용하고 접근 시간 업데이트를 제거할 수 있습니다(`noatime`). 또한 `actimeo=60,lookupcache=pos` 옵션을 추가하면 파일 속성과 디렉터리 항목을 60초 동안 캐시할 수 있습니다. 이러한 간단한 조정만으로도 요청당 오버헤드를 크게 줄이고 대규모 공유 데이터셋에서 병렬 읽기 처리량을 향상시킬 수 있습니다.

Amazon S3( )와 같은 객체 저장소는 일반적인 파일 시스템은 아니지만 AI 워크로드에서 매우 흔히 사용됩니다. 훈련 중 객체 저장소에 접근하는 것은 무분별하

게 수행할 경우 느릴 수 있습니다. 해결책은 종종 로컬 NVMe SSD 저장소에 데이터를 스테이징하거나 객체 저장소 위에 캐싱 계층을 사용하는 것입니다(예: S3 위에 구축된 Amazon FSx for Lustre). `s5cmd` 및 `aws s3 cp` 과 같은 도구를 사용하면 훈련 시작 전에 데이터를 다운로드할 수 있습니다.

최상의 성능을 얻으려면 AWS S3 C++ SDK와 같은 고도로 병렬화된 최적화 데이터 전송 도구와 `s5cmd` 같은 멀티스레드 유틸리티를 사용하십시오.

또한 범위 요청으로 Amazon S3에서 객체를 읽고 캐싱을 수행하는 스트리밍 라이브러리를 사용할 수도 있습니다. Amazon S3에서 직접 읽을 경우 가능한 한 큰 요청을 사용하고, 멀티스레드 범위 `Get` 작업을 활용하세요.

Lustre 및 GPFS와 같은 병렬 파일 시스템은 높은 동시성과 처리량을 위해 설계되었습니다. 예를 들어 Lustre 설정에는 데이터를 제공하는 여러 객체 저장소 대상(OST)이 있습니다. 파일을 여러 OST에 스트라이핑하면 처리량을 배가할 수 있습니다. 이러한 병렬 파일 시스템을 사용하는 경우 대용량 데이터 파일을 여러 OST에 스트라이핑해야 합니다.

*파일을 여러 OST에 스트라이핑한다는 것*은 파일의 조각들이 서로 다른 서버에 존재함을 의미합니다. 이는 병렬 읽기를 가능하게 합니다. 예를 들어, Arrow, TFRecord 또는 Parquet 파일을 4개의 OST에 스트라이핑할 수 있습니다. 각 OST가 500MB/s를 제공한다면, 이론상 최대 읽기 처리량 2GB/s를 달성할 수 있습니다.

데이터 튜닝, 복제 및 압축

이러한 파일 시스템을 조정하려면 문서를 반드시 확인하십시오. 예를 들어, Lustre에서는 `lfs setstripe` 명령어를 사용하여 대규모 데이터셋에 대해 4개 또는 8개 OST에 스트라이핑을 설정해 OST 대역폭을 통합합니다.

훈련 중 파일 시스템의 I/O를 모니터링하세요. Lustre의 경우 `lmt` 같은 도구나 벤더별 모니터링 도구를 사용하세요. 스토리지 클러스터 내 개별 노드가 과부하 상태인지 확인해야 합니다. 해당된다면 원인을 파악해야 합니다. 대부분의 경우 샤딩 문제로 인해 더 많은 읽기/쓰기 작업이 소수의 노드에 집중되기 때문입니다.

네트워크 읽기를 완전히 제거하려면, 경우에 따라 컴퓨팅 클러스터의 각 노드에 데이터셋을 복제하는 방법을 선택할 수 있습니다. 이는 각 노드에 충분한 저장 공

간이 있다는 전제 하에 가능합니다. 이는 확실히 무차별적인 방법이지만, 네트워크 읽기를 완전히 제거하는 데 상대적으로 흔하고 매우 효과적인 해결책입니다. 추가 저장 공간 비용을 감수하면 즉각적인 성능 향상을 확인할 수 있습니다.

성능 향상을 위한 또 다른 옵션은 파일 시스템이나 객체 저장소에 압축된 데이터를 저장하고 실시간으로 압축을 해제하는 것입니다. 이미지(JPEG)나 압축된 텍스트(Arrow 및 Parquet)가 대표적인 예입니다. 이는 일부 추가 CPU 또는 GPU 사이클을 소모하는 대신 I/O 대역폭을 절약할 수 있습니다. 그러나 I/O가 병목 현상이고 CPU 및 GPU가 유휴 상태라면 이는 합리적인 타협점입니다.

많은 데이터 파이프라인이 이미 압축( )을 수행하고 있으므로, 압축률을 확인하고 파이프라인의 모든 단계에서 이를 사용하고 있는지 확인하기만 하면 됩니다. 핵심은 압축 해제 단계가 병목 현상이 되지 않도록 균형을 찾는 것입니다.

nvJPEG 같은 라이브러리는 GPU에서 이미지를 디코딩할 수 있습니다. 현대 GPU는 LZ4, Snappy, Deflate 같은 형식을 지원하는 온다이 디컴프레션 엔진을 추가하여 GPU 메모리로 데이터를 이동하고 압축 해제하는 속도를 높입니다. 압축된 배치 데이터를 디스크에 저장할 경우, Blackwell GPU는 디컴프레션 엔진을 활용해 파이프라인 내에서 이를 압축 해제할 수 있습니다. 이를 통해 SM(스케일러블 메모리)은 컴퓨트 커널과 같은 고부가가치 작업 실행에 집중할 수 있습니다. I/O에 바운디드된 워크로드에는 이러한 압축 형식을 우선적으로 고려해야 합니다.

이는 CPU에서 GPU로 산술 연산을 오프로드하는 또 다른 방법이며, 예를 들어 훈련 중 데이터 디코딩과 기울기 계산을 중첩할 수도 있습니다. 또한 CPU-GPU 간 NVLink-C2C 인터커넥트(최대 900GB/s 양방향 대역폭)의 높은 대역폭 덕분에 CPU 지원 단계가 병목 현상이 되는 것을 방지할 수 있습니다.

이러한 지능적인 소프트웨어 및 하드웨어 GPU 오프로드 기능을 활용하면 CPU에서 작업 부하를 더욱 전환하고 입력 데이터 파이프라인의 균형을 유지할 수 있습니다. 핵심은 여전히 압축 해제 시간이 I/O를 대체하여 병목 현상이 되지 않도록 하는 것이며, 그렇지 않으면 추가 압축 계산이 가치가 없을 수 있습니다.

스토리지 I/O 모니터링

모든 성능 엔지니어링 작업과 마찬가지로, 실시간( ) 측정이 핵심입니다. 네트워크 통신 모니터링과 마찬가지로, 사용 가능한 모든 도구를 활용하여 스토리지 파이프라인 통신을 모니터링하는 것이 중요합니다.

이러한 도구에는 Linux `iostat`, `iotop`, `nvme-cli`, `perf`, `eBPF` 등이 포함됩니다. 또한 벤더별 유틸리티와 대시보드를 활용하여 대기열, 지연 시간, 미리 읽기 효과, 캐시 적중률을 모니터링할 수 있습니다. 이를 통해 로컬 NVMe 장치 사용량을 확인하고 NAS 또는 객체 저장소에서 데이터를 읽을 때 네트워크 링크가 포화 상태인지 판단할 수 있습니다.

또한 Nsight Systems와 같은 도구를 활용하여 I/O 대기 시간을 추적하고 GPU 커널과의 중첩을 시각화할 수 있습니다. Nsight Systems 옵션 `--trace=gds` 을 사용하면 `cuFile` API 활동과 타임라인 추적을 캡처할 수 있습니다. `/etc/cufile.json` 를 통해 GDS `cuFile` 정적 추적점을 활성화하면 Nsight Systems에서 `cuFile` 이벤트를 확인할 수 있습니다. NVMe 피어 투 피어 DMA 경로에 대한 커널 모드 카운터는 Nsight Systems에 노출되지 않으며 모든 GDS 스택에서 사용할 수 있는 것은 아닙니다.

또 다른 도구는 NVIDIA의 데이터 센터 GPU 관리자(DCGM)로, 유용한 GPU I/O 통계를 보고합니다. 이러한 GPU 전용 도구들은 호스트 OS 도구를 보완하여 I/O로 인한 GPU 대기 현상에 대한 보다 완전한 그림을 제공합니다.

PyTorch에서 `next(data_iterator)` 를 호출하면 GPU가 다음 배치를 기다리며 유휴 상태로 있는 총 시간을 측정합니다. 이 시간에는 백그라운드 프리페칭과 호스트 → 디바이스 복사 작업이 포함되며, 단순히 Python 데이터 로딩 로직만을 반영하지 않습니다.

순수한 데이터 로딩 비용만 분리하려면, 임시로 `num_workers=0` 를 설정하여 프리페칭을 없앨 수 있습니다. 그러면 반복자 풀링 시간만 측정할 수 있습니다. 호스트 → 장치 복사 오버헤드를 포착하려면 `.to("cuda")` 또는 고정 메모리 스테이징을 별도의 타이머로 감싸거나 (또는 CUDA 이벤트 사용) 측정하세요.

병목 지점이 Python 파이프라인에 있는지, *아니면* GPU 메모리로의 `memcpy` 에 있는지 구분하려면 다음을 수행하고 전체 "GPU 유휴" 시간과 타이밍을 비교하세요:

DataLoader 대 Python 비용

`num_workers=0` 로 자질을 확인하여 Python 루프 및 변환 자체에 소요되는 시간을 확인하세요. 이렇게 하면 백그라운드 스레드 스케줄링이 제거됩니다.

호스트 → 디바이스 복사 비용

Nsight Systems의 "Copy" 레인을 검사하여 장치 전송 시간만 측정함으로써 GPU 버퍼로 데이터를 스테이징하는 과정이 실제로 GPU를 얼마나 오

래 지연시키는지 정량화하십시오. `.to("cuda")` 호출을 `torch.cuda.Event` 로 감싸는 방법도 있습니다.

이 두 타이밍을 전체 "GPU 유휴" 시간과 비교하면, Python 파이프라인 가속화(예: 작업자 추가, 변환 단순화)가 필요한지, 아니면 H2D 전송 경로 최적화(예: 고정 메모리 사용, 인터커넥트 대역폭 증대, GDS 전환)가 필요한지 판단할 수 있습니다.

저장 파이프라인을 모니터링하면, 예를 들어 GPU가 데이터 대기 시간으로 전체 시간의 30%를 소비한다는 사실을 발견할 수 있습니다. 이 경우 GPU의 전체 처리량은 I/O에 의해 제한되므로, 여기서 언급된 전략 중 일부를 구현하여 I/O 정체를 줄이고 컴퓨팅 처리량을 높여야 합니다. 튜닝 후에는 예를 들어 GPU가 데이터 대기 시간으로 소비하는 시간이 5%로 감소할 수 있습니다. 동시에 초당 수행되는 전체 훈련 단계 수도 비례하여 증가할 것입니다—이 경우 6배 증가합니다.

스토리지 및 I/O 최적화는 종종 누적되는 작은 비효율성 제거에 관한 것입니다. 예를 들어 여기서는 5ms 지연, 저기서는 10MB 부족한 버퍼 등이 해당됩니다. 그러나 대규모 환경에서는 이러한 비효율성을 해결하는 것이 큰 차이를 만듭니다. 핵심은 이전 섹션과 유사합니다: 파이프라인을 가득 채우세요. 이 경우 컴퓨팅 파이프라인뿐만 아니라 데이터 파이프라인도 포함됩니다. 디스크부터 GPU 메모리에 이르는 모든 구성 요소를 모니터링하고, 자질하며, 분석하고, 개선하여 데이터가 가능한 한 지속적으로 GPU로 스트리밍되도록 해야 합니다.

데이터 파이프라인 조정

원시 스토리지 I/O 외에도, CPU(또는 GPU)에서의 전처리 및 데이터 로딩 파이프라인 은 전체 AI 워크로드 성능의 핵심 요소입니다. 잘 조정된 데이터 파이프라인은 GPU가 새 데이터를 기다리며 유휴 상태가 되지 않도록 보장합니다. 또한 GPU라는 괴물을 먹여 살리기 위해 적절한 양의 CPU 작업이 병렬로 수행되는 것이 중요합니다.

현대적인 Deep Learning 프레임워크는 데이터 로딩 및 전처리를 위한 고수준 API를 제공합니다. 이러한 API는 성능을 위해 조정될 수 있으며, 반드시 조정되어야 합니다. 고급 데이터 파이프라인 관리를 위한 일반적인 전략과 NVIDIA의 DALI 및 NeMo 같은 도구에 대해 논의하겠습니다.

효율적인 데이터 로딩 및 전처리

훈련 시 일반적인 데이터 로딩 프로세스( )는 저장소에서 데이터 읽기, JSON 파싱이나 JPEG 디코딩과 같은 데이터 디코딩/디세리얼라이징, 텍스트 토큰화나 이미지 크롭과 같은 변환 적용, 데이터를 배치 단위로 정리하는 단계를 포함합니다. 이러한 단계는 CPU 집약적일 수 있지만, 연산 집약적이라면 GPU로 오프로드할 수도 있습니다. 높은 처리량을 유지하기 위해 여기 설명된 여러 기법을 활용할 수 있습니다:

여러 작업자 프로세스/스레드 사용

앞서 언급한 바와 같이 PyTorch `DataLoader` 와 같은 프레임워크에서는 `num_workers` 를 지정할 수 있습니다. 각 작업자는 데이터를 가져오고 전처리하기 위해 병렬로 실행됩니다. 일반적으로 Python GIL 문제를 피하기 위해 별도의 프로세스로 실행됩니다. 메인 프로세스는 큐를 사용하여 작업자 프로세스에서 배치 데이터를 비동기적으로 가져옵니다.

Python 병목 현상 방지

데이터 로딩 로직이 Python으로 작성된 경우, Python 수준에서 과도한 처리가 이루어지지 않도록 주의해야 합니다. 순수 Python 코드로 루프 내에서 개별 텍스트 라인을 토큰화하는 코드가 보인다면 이는 위험 신호입니다. 이러한 경우 가능하면 연산을 벡터화하도록 변경하거나, 성능 향상을 위해 C++/C 바인딩을 사용하세요. 이러한 일반적인 작업을 위한 다양한 라이브러리가 존재합니다. Hugging Face Tokenizers 라이브러리나 TorchText 등이 대표적입니다. 이들은 Python 바인딩을 제공하지만, 속도 향상을 위해 내부적으로는 빠른 Rust/C++로 작성되었습니다. 이 시점에서 Python은 C/C++ 코드 위에 구축된 사용하기 쉬운 인터페이스에 불과합니다.

CPU-GPU 작업 중첩

핵심은 데이터 준비와 GPU 처리를 병렬화하는 것입니다. 이상적인 시나리오에서는 GPU가 배치 N을 처리하는 동안 CPU가 이미 배치 N+1을 로드하고 전처리하여 고정 메모리에 준비해 둡니다. GPU가 배치 N 처리를 마치면 배치 N+1을 DMA 복사하여 즉시 계산을 시작합니다. 한편 CPU는 배치 N+2 처리를 진행합니다. 이러한 파이프라인화는 성능에 매우 중요합니다. 대부분의 프레임워크는 다중 워커 사용 시 기본적으로 이를 수행하지만, 실제로 실행되는지 모니터링해야 합니다. 그렇지 않으면 GPU가 더 많은 데이터를 기다리며 반복자 시작 시마다 유휴 상태가 되는 현상을 목격할 수 있습니다.

텐서를 병합하여 작업을 배치로 수행

가능하다면 로더가 샘플 단위가 아닌 배치 단위로 작업을 수행하도록 해야 합니다. 예를 들어 벡터화된 연산을 사용하여 텐서 전체 배치를 한 번에 변환해야 합니다. 이를 위해 사용자 정의 데이터 병합(`collate_fn()`)을 통해 배치를 구성하거나 GPU의 훈련 루프 자체에서 수행할 수 있습니다. 이는 입력 데이터의 각 행에 대해 별도로 연산을 수행하는 것보다 훨씬 효율적입니다. 다만 일부 변환은 샘플별로 수행해야 하므로, 효과적인 배치 및 병합을 위해 작업 부하를 사전에 이해해야 할 때가 있습니다.

데이터 로딩 시 메모리 고정 사용

PyTorch DataLoader에서 고정 메모리(`pin_memory=True`)를 활성화하면 호스트 → GPU(H2D) 전송 속도가 향상되며, 소스 데이터가 고정된 상태에서 진정한 비동기식 데이터 복사(`.to(..., non_blocking=True)`)가 가능해집니다. 고정된 메모리에서의 DMA는 데이터가 RAM에 고정되어 직접 전송 준비가 되어 있으므로 추가 복사 및 페이지 결함 발생을 방지합니다. 이는 GPU로 데이터를 전송할 때 거의 항상 유리합니다. 큰 고정 버퍼의 할당 실패를 방지하려면 높은 사전 가져오기 큐 길이(`ulimit -l`) 또는 컨테이너 사전 가져오기 큐 길이(container `--ulimit memlock`)를 설정하세요.

배치 사전 가져오기

일부 프레임워크에서는 프리페치 큐 길이를 지정할 수 있습니다. 이는 미리 로드해야 할 배치 수를 의미합니다. 기본적으로 PyTorch의 DataLoader는 `prefetch_factor=2` 과 같은 보수적인 값을 사용합니다. 이 경우 PyTorch는 작업자당 두 개의 배치를 프리페치합니다. 내부적으로는 최대 `num_workers * prefetch_factor` 개의 배치를 큐에 유지합니다. 따라서 각 작업자는 차단되기 전에 두 개의 데이터 배치를 로드합니다. 작업 부하에 버스트성 I/O가 있거나, 가끔 작업자가 GPU를 굶주리게 하는 현상을 목격한다면, 예를 들어 `prefetch_factor` 를 4나 8로 늘릴 수 있습니다. 다음은 `pin_memory` 와 `prefetch_factor=4` 를 모두 사용하여 PyTorch `DataLoader` 를 보여주는 PyTorch 코드 조각입니다:

```
import torch
from torch.utils.data import Dataset, DataLoader

# Create a Dataset and DataLoader that prefetches 4 batches
# per worker into pinned CPU memory.
class Synthetic(Dataset):
```

```
        def __init__(self, n, shape): self.n, self.shape = n, shape
        def __len__(self): return self.n
        def __getitem__(self, i):
            # Cheap CPU-side work; replace with real parse/decode
            return torch.ones(self.shape,  dtype=torch.float32)

    B, C, H, W = 32, 3, 224, 224
    dataset = Synthetic(n=100_000, shape=(B, C, H, W))

    loader = DataLoader(
        dataset,
        batch_size=B,
        num_workers=8,
        pin_memory=True,
        persistent_workers=True,
        prefetch_factor=4,
      )

    copy_stream = torch.cuda.Stream()
    compute_stream = torch.cuda.current_stream()

    for batch in loader:
        with torch.cuda.stream(copy_stream):
            batch_gpu = batch.to(device, non_blocking=True)
        # ensure pending H2D completes before compute uses it
        with torch.cuda.stream(compute_stream):
            torch.cuda.current_stream().wait_stream(copy_stream)
            outputs = model(batch_gpu)
```

이 예시에서 8개의 작업자 프로세스 각각은 크기 4의 배치들을 고정 메모리에 미리 로드합니다. 호스트 메모리가 고정되어 있으므로 비동기식 데이터 로더(`.to(device, non_blocking=True)`) 전송은 고속 데이터 복사를 위해 DMA를 활용할 수 있습니다.

결과적으로 GPU가 현재 배치(배치 N)를 처리하는 동안 DataLoader는 이미 다음 배치(배치 N+1)를 병렬로 준비하고 전송합니다. 이러한 중첩 처리가 핵심입니다. 고정 메모리가 없다면 시스템은 각 전송마다 메모리를 실시간으로 고정해야 하므로 원치 않는 지연이 발생합니다. 본질적으로 고정 메모리는 CPU에서 GPU로의 데이터 전송이 GPU 계산과 더 빠르게 동시 수행되도록 보장하여 전체 처리량을 극대화합니다.

또 다른 옵션은 지속적 작업자(`persistent_workers=True`)를 활성화하여 작업자 가 에포크(*epoch*)를 넘어서도 계속 실행되며 큐를 채우도록 하는 것입니

다. 이는 동일한 데이터셋을 여러 번 반복할 때 가장 효과적이며, 특히 이러한 반복(*에포크)*이 매우 짧을 때 유용합니다. 지속적 작업자는 모듈 임포트, 파일 열기 등으로 인해 작업자 시작 시 상당한 오버헤드가 발생하는 경우에도 도움이 됩니다. 지속적 작업자를 사용하면 각 에포크 경계마다 프로세스를 생성하고 종료하는 비용을 피할 수 있습니다. 작업자가 계속 활성화된 상태를 유지하므로 최소한의 오버헤드로 다음 에포크에 대한 사전 가져오기를 즉시 시작할 수 있습니다.

흔히 발생하는 함정은 파이프라인에 숨겨진 병목 현상을 유발하는 것입니다. 디버그 로깅이나 CPU 집약적 변환을 추가할 때 상대적으로 쉽게 발생할 수 있습니다. 지연은 부하 상태에서만 드러날 수 있습니다. 이를 포착하려면 먼저 에서 DataLoader를 단독으로 자질하세요. 모든 하류 GPU 작업을 비활성화한 상태에서 100개 배치 생성 소요 시간을 측정합니다. 이 기준값을 측정한 후 목표 반복 시간 및 정상 훈련 중 측정된 총 GPU 유휴 시간과 비교하세요.

DataLoader 단독 속도가 너무 느리다면, 요소별 로깅 제거, 변환 단순화, 작업자 추가 등을 통해 Python 파이프라인을 최적화하세요. 분리된 로더 속도와 실제 실행 속도 간 차이가 크다면, 호스트 → 디바이스 전송 또는 커널 실행 오버헤드에 의해 성능이 바운디드되고 있을 가능성이 높습니다.

자질을 위해 DataLoader를 분리하기 위해 GPU 커널을 비활성화하면 CPU 측 커널 실행 오버헤드도 감소합니다. 따라서 "순수한" 데이터 로딩 처리량은 실제 훈련 실행에서 관찰되는 수치보다 낮게 나타나는 경우가 많습니다. 이는 여전히 유용한 기법이지만, 이 점을 염두에 두시기 바랍니다.

GPU 수를 확장할 때 작업자 수 확장

GPU를 추가할 때는 데이터 파이프라인도 확장해야 합니다( ). 그렇지 않으면 디바이스에 데이터 공급이 부족해집니다. 실제로는 DataLoader의 작업자 수나 I/O 대역폭을 늘려 모든 GPU에 데이터를 공급할 수 있도록 해야 합니다. 이는 총 배치 크기를 늘려 더 많은 디바이스에서 각 반복자당 더 많은 샘플을 처리할 수 있도록 하기 위함입니다.

데이터 로딩 파이프라인 리소스를 확장하지 않은 채 컴퓨팅만 확장하면 병목 현상이 데이터 로딩 파이프라인 쪽으로 더욱 이동합니다. 다중 노드 데이터 병렬 구성에서는 각 랭크가 고유한 샤드를 읽습니다. 전체 데이터 로딩 작업 부하는 클러스터 규모에 따라 확장됩니다.

데이터 입력 파이프라인이 GPU 훈련 가속화 시 병목이 되므로 항상 CPU 사용률을 측정하십시오.

필요한 처리량을 유지하려면 앞서 논의한 바와 같이 클러스터 내 다수 노드에 걸친 초대형 데이터 샤딩을 지원하기 위해 병렬화되고 대역폭이 높은 분산형 스토리지 백엔드가 필요합니다. 또한 노드별 데이터셋 샤딩에 관한 이전 논의 내용을 상기하십시오. 특히 노드를 추가할 때 각 노드의 로컬 스토리지가 담당 데이터셋 분량을 처리할 수 있도록 해야 합니다.

NVIDIA DALI를 활용한 다중 모달 데이터 처리

복잡하거나 무거운 데이터 전처리 작업의 경우, NVIDIA는 데이터 로딩 라이브러리(, DALI)를 제공합니다. DALI는 데이터를 GPU로 이동하거나 C++로 작성된 최적화된 CPU 코드를 사용하여 데이터 처리 속도를 가속화합니다. 특히 GPU 가속을 통해 디코딩 및 증강이 가능한 이미지 및 비디오 데이터에 유용합니다.

예를 들어, DALI는 GPU에서 JPEG 이미지를 디코딩하고 무작위 자르기, 크기 조정, 정규화 등의 증강을 모두 GPU에서 수행할 수 있습니다. 이는 GPU에 사용 가능한 사이클이 있다고 가정할 때 CPU보다 종종 더 빠릅니다. 이는 CPU에서 처리를 오프로드하고 필요한 CPU 작업자 수를 줄입니다.

DALI 파이프라인은 연산자들로 구성된 정적 그래프로 선언적으로 정의됩니다. `nvidia.dali.pipeline.Pipeline`를 상속받아 `define_graph()`에서 데이터 소스와 CPU/GPU 연산자를 선언하면, DALI가 자체 스레드 풀과 큐를 활용해 내부적으로 실행, 프리페칭, 스레딩을 처리합니다.

워크로드가 입력에 바운디드일 경우(예: 모델 훈련), DALI 통합으로 처리량이 크게 향상될 수 있습니다. 다만 훈련 루프 자체에 통합해야 하므로 복잡성이 증가하고 학습 곡선이 다소 존재합니다.

분류, 객체 탐지, 분할과 같은 많은 일반적인 워크로드의 경우 NVIDIA DALI는 GPU에서 이미지와 비디오를 디코딩하는 사전 구축된 파이프라인을 제공합니다. 이는 GPU의 미디어 가속 하드웨어를 완전히 활용합니다.

이미지와 동영상을 읽고, 데이터 증강을 수행하며, 객체 탐지 모델을 훈련하는 데이터 파이프라인을 고려해 보십시오. CPU 사용률이 800%(8개 코어가 100%로 작동 중)에 달하는 것을 관찰할 수 있습니다. 그러나 GPU는 여전히 가끔씩 정지 상태에 빠집니다.

DALI를 사용하면 파일 읽기 작업은 CPU(200%, 즉 2코어)에서 수행하고 실제 이미지/동영상 디코딩은 GPU에서 처리함으로써 CPU 사용률을 낮출 수 있습니다. 또한 GPU는 계산과 병렬로 읽기 작업을 수행할 수 있습니다.

실제 성능 향상은 DALI를 워크플로우의 어느 단계에 배치하느냐에 따라 완전히 달라집니다. DALI를 단순히 JPEG 압축 해제에만 사용하고, 원시 픽셀을 즉시 CPU로 넘겨 증강 및 정렬 작업을 수행한다면, 호스트-장치-호스트 간 추가 복사 작업이 발생하여 DALI 사용으로 얻는 성능 이점이 상쇄될 수 있습니다.

DALI에 대한 더 나은 접근법은 GPU 친화적인 전처리 작업을 식별하여 GPU 기반 전처리 계산 그래프에 직접 통합하는 것입니다. 대부분의 전처리 작업은 TorchVision이나 TensorRT 같은 기존 CUDA 기반 라이브러리 , 또는 커스텀 CUDA 커널을 사용해 수행할 수 있습니다. 이렇게 하면 CPU와 GPU 간에 데이터를 지나치게 오가며 이동하는 것을 피할 수 있습니다. 이는 파이프라인에 DALI를 사용하는 것보다 더 높은 엔드투엔드 성능을 낼 수 있으므로 검토해 볼 가치가 있습니다.

항상 그렇듯, 실제 조건에서 엔드투엔드 시스템( )을 벤치마킹하십시오. CPU 전용 파이프라인, DALI 지원 파이프라인, 완전히 융합된 GPU 그래프 구현을 비교하여 모델과 데이터셋에 대해 CPU 절감과 GPU 활용도의 최적 균형을 제공하는 방식을 결정하십시오.

NVIDIA NeMo Curator를 활용한 고품질 LLM 데이터셋 생성

NVIDIA NeMo는 딥 러닝 모델( ) 개발 및 언어 모델 훈련을 위한 툴킷입니다. NeMo 툴킷에는 오픈 소스 NeMo Curator 프레임워크를 포함한 NeMo 라이브러리 및 프레임워크 제품군이 포함되어 있습니다.

Curator는 LLM 훈련을 위한 대규모 다중 모달 데이터셋 준비를 지원합니다. 다양한 출처의 테라바이트급 데이터를 처리할 때 유용합니다. Curator는 데이터 정제, 토큰화, 셔플링 등의 데이터 처리 단계를 지원합니다.

NeMo Curator는 데이터셋 전처리 작업을 여러 GPU 또는 노드에 분산 처리할 수 있습니다. 이는 다중 가속기를 활용해 데이터를 더 빠르게 준비하는 방식으로, 멀티테라바이트 규모의 훈련 데이터셋을 구축할 때 중요한 고려 사항입니다.

또한 Curator는 데이터를 소수의 대용량 파일로 압축 및 패키징하거나, 기계가 더 쉽게 처리할 수 있도록 이진 형식으로 변환할 수 있습니다. 상대적으로 제한적

이고 점점 더 희소해지는 인간 데이터셋을 보완하기 위해 새로운 합성 훈련 데이터셋을 생성할 수도 있습니다.

Curator가 훈련 과정 전에 오프라인으로 중량급 전처리를 수행함으로써, 온라인 훈련 데이터 파이프라인은 준비된 데이터를 읽고 가벼운 '마지막 단계' 셔플링 등을 수행하는 정도로 훨씬 단순해집니다.

NeMo Curator는 데이터 중복 제거 및 문제성 콘텐츠 제거를 통해 데이터 품질 필터링도 강제 적용할 수 있습니다. 이는 LLM 훈련 품질과 성능 모두에 중요합니다. 사전에 잘 구조화되고 전처리되며 정제된 데이터셋을 확보하면 훈련 파이프라인은 일관된 흐름의 잘 구조화되고 균일한 크기(예: 고정 길이로 패딩)의 데이터를 처리할 수 있으며, 실시간 텍스트 토큰화나 복잡한 문자열 처리를 피할 수 있습니다.

NeMo Curator와 같은 도구를 사용할 수 있다면 이를 활용하여 훈련 작업이 주로 GPU 전방 및 후방 통과에 집중되도록 하는 것이 현명합니다. 텍스트 처리나 수백만 개의 임의 크기의 작은 파일 읽기에 시간을 낭비하지 않도록 해야 합니다. NeMo 기반 훈련의 경우, 전처리된 데이터셋은 일반적으로 메모리 매핑 가능한 *.bin* 데이터 파일과 *.idx* 인덱스 파일로 저장됩니다. 이후 NeMo Curator의 인덱싱 도구(`DocumentDataset`)가 분할된 JSONL 또는 Parquet 형식으로 읽고 씁니다. 인덱싱된 데이터셋을 구축할 때 *.bin/.idx* 로의 다운스트림 변환이 처리됩니다.

N개의 에포크에 걸친 훈련 과정에서 런타임 셔플링 비용을 피하기 위해, 데이터를 *N가지* 서로 다른 방식으로 셔플링한 *N개의* 복사본을 저장하는 것을 고려해 볼 수 있습니다. 명백한 타협점은 디스크 공간과 메모리이지만, 이는 고려해 볼 만한 가치가 있는 타협점입니다.

일반적으로 훈련 전에 데이터를 준비하십시오. 원시 텍스트로 훈련하는 경우는 거의 없어야 합니다. 오프라인에서 데이터를 전처리하는 데 시간이 다소 걸릴 수 있지만, 장기적으로 더 빠른 훈련 실행, 신속한 반복, 예측 가능한 확장성으로 보상이 따릅니다.

여기서 설명한 모든 기법은 데이터 로딩 파이프라인이 고가의 GPU 클러스터를 유휴 상태로 방치하지 않도록 설계되었습니다. 데이터 파이프라인이 충분히 빠르게 입력을 공급하지 못하면 최고 성능의 GPU도 무용지물입니다. 따라서 스토리지, 네트워크, CPU, GPU를 포함한 모든 계층에서 종합적이고 전체 스택에 걸친 최적화 접근이 필요합니다.

NeMo의 데이터 로딩은 여전히 CPU에서 실행됩니다. CPU I/O를 우회하려면 GDS 같은 도구와 통합해야 합니다.

대부분의 경우 데이터 파이프라인 최적화가 알고리즘 조정보다 더 큰 개선 효과를 가져옵니다. 제대로 조정되지 않은 입력 파이프라인은 GPU 사용 시간의 50%를 낭비할 수 있는 반면, 알고리즘 최적화는 몇 퍼센트에 불과한 개선만 제공할 수 있습니다.

지속적인 자질 및 튜닝 워크플로

성능 엔지니어링은 반복적인 과정입니다. 분산 훈련 또는 추론 애플리케이션이 확장되거나 수정될 때 효율성을 유지하려면 지속적인 프로파일링 및 튜닝 워크플로를 채택해야 합니다. 이는 정기적으로 성능 데이터를 수집하고, 병목 현상을 식별하며, 최적화를 적용한 후 다시 측정하는 것을 의미합니다.

시간이 지남에 따라 하드웨어 및 소프트웨어 업데이트로 최적 설정이 변경되므로 지속적인 프로파일링과 튜닝이 필요합니다. 이를 대비하기 위해 성능 중심 엔지니어링 팀은 종종 성능 대시보드를 유지하여 시간 경과에 따른 샘플/초와 같은 메트릭을 추적합니다.

훈련 및 추론 워크로드를 자질하는 자동화된 야간 실행을 설정하는 것을 고려하십시오. 이렇게 하면 성능 저하나 개선 사항을 포착하고 이를 코드 변경 사항으로 추적할 수 있습니다.

이제 본 장에서 설명한 주제뿐만 아니라 모든 자질 및 디버깅 상황에 광범위하게 적용할 수 있는 일반적인 워크플로와 모범 사례 세트를 살펴보겠습니다:

기준선 설정

단일 GPU 또는 최소한의 설정으로 시작하여 성능을 측정하세요. 예를 들어 초당 샘플 수로 측정되는 훈련 처리량이나 밀리초 단위로 측정되는 추론 지연 시간(바라건대!)을 측정합니다. 그런 다음 단일 노드에서 다중 GPU로 확장하고, 다시 다중 노드로 확장하세요. 매번 전체 처리량과 같은 상위 수준의 지표를 사용하여 성능이 어떻게 확장되는지 분석합니다. 단순한 데이터 병렬 워크로드의 경우 이상적으로는 *N개의* GPU가 *N배의*처리량을 제공해야 합니다. 이보다 훨씬 낮은 성능이 관측된다면 시스템에 과도한 오버헤드가 발생하고 있다는 신호입니다. 다음 단계는 병목 현상을 정량화하는 것입니다. 예를 들어 8개의 GPU가 처리량을 5배만 증가시

킨다면 시스템 효율이 62.5%에 불과함을 의미합니다. 이는 이상적인 상태가 아닙니다.

다중 GPU 실행의 병목 현상 자질

정확한 병목 현상의 원인을 진단하기 위해, 우리는 더 깊이 파고들어 Nsight Systems(`nsys`)와 같은 시스템 자질 도구를 다중 GPU 작업 전체에 적용하여 시간이 어디에 소모되는지 개요를 파악합니다. 첫 번째 단계는 GPU 활용도 타임라인을 살펴보는 것입니다. GPU가 자주 정지하고 있나요? 그렇다면 무엇을 기다리고 있나요? CPU 타임라인도 확인합니다. 메인 프로세스가 다른 작업자 프로세스보다 뒤처지고 있나요? 모든 스레드가 대기 중인 동기화 지점이 있나요? 예를 들어, 모델 훈련 중 그라디언트 올-리듀스 단계에서 GPU가 유휴 상태라면 통신이 병목 현상임을 알 수 있습니다. 각 반복자 시작 시 GPU가 유휴 상태라면 데이터 로딩이나 특정 커널이 병목일 수 있습니다.

필요 시 특정 커널 확대 분석

특정 GPU 작업(네트워크 또는 컴퓨트)이 예상보다 느리게 실행된다고 확인되면, 해당 커널에 대해 Nsight Compute(`ncu`)를 사용하여 커널 효율성을 더 깊이 분석할 수 있습니다. 예를 들어, 앞서 논의한 NCCL 커널의 경우 PCIe를 통해 통신할 때 NVLink 대비 SM 활용도가 60%에 불과하고 메모리 스톨 카운터가 높게 나타났습니다. 통신을 NVLink로 최적화한 후 SM 활용도는 90%로 상승했고 메모리 스톨도 감소했습니다. 이러한 심층 분석을 통해 커널이 네트워크 대역폭 바운디드, 메모리 대역폭 바운디드, 컴퓨팅 바운디드 중 어느 유형에 해당하는지 확인할 수 있습니다. 반드시 이 세 가지 중 하나일 것입니다.

원인 파악

병목 현상을 발견하면 가설적인 원인 집합에 매핑하고 하나씩 검증(또는 배제)하세요. 예를 들어 병목이 네트워크에 의해 바운디드된다면, RDMA를 사용하지 않거나 메시지 크기가 너무 작거나 더 나은 오버래핑이 필요할 수 있습니다. GPU가 다른 GPU를 기다리며 유휴 상태라면, 데이터 불균형으로 인해 한 GPU가 다른 GPU보다 더 많은 작업을 수행하는 스트래글러 상황이 발생했을 수 있습니다. 워크로드가 CPU에 의존하는 경우, 데이터 로더가 올바르게 구성되지 않았거나 GPU에 더 적합한 CPU 집계 작업이 존재할 수 있습니다. GPU 메모리에 병목 현상이 발생하는 경우, 일부 커널이 레지스터와 HBM 메모리 간에 과도한 데이터 전송을 수행할 수 있으므로 배치 크기 축소나 커널 융합 도입 등을 시도해 볼 수 있습니다.

수정 또는 최적화 적용

가설을 검토하고 실제 원인을 파악한 후에는 조치를 취할 차례입니다. 네트워크 및 통신 병목 현상 해결을 위해서는 GPUDirect RDMA가 활성화되었는지 확인하고, 다중 NIC를 사용 중임에도 네트워크 대역폭이 여전히 제한된다면 NIC 간 대역폭(`NCCL_NSOCKS_PERTHREAD`)을 늘리십시오. 또한 후속 장에서 다루는 그라디언트 압축(gradient compression)과 같은 기법을 활용해 데이터 압축을 고려해 보십시오. NUMA 노드 간 통신이 발생하는 경우 계층적 접근 방식을 시도하거나 NCCL 토폴로지를 구성하여 NUMA 노드당 사용 GPU 수를 줄이는 등의 방법을 적용하십시오.

인트라노드 토폴로지 문제의 경우, GPU가 PCIe 스위치에 분산되어 있다면 가능한 경우 작업을 단일 NUMA 노드의 GPU에 바인딩하여 느린 상호 연결을 피하거나, 토폴로지를 더 고려하는 알고리즘을 사용해 보십시오. CPU 및 데이터 문제의 경우 데이터 로더가 너무 느린지 확인하세요. 이 경우 작업자 프로세스/스레드를 추가하거나 DALI 등을 활용해 일부 전처리 작업을 GPU로 이동시키세요. 또는 사전에 더 많은 오프라인 전처리를 수행하세요. 특정 GPU가 추가 검증이나 로깅으로 인해 느려진다면 작업량을 줄이거나 비동기 작업을 통해 중요 경로에서 제외시키세요.

동기화가 문제라면, 실행을 의도치 않게 직렬화할 수 있는 불필요한 ` `torch.cuda.synchronize()` ` 호출이나 배리어를 코드에서 제거하세요(자세한 내용은 13장에서 다룹니다). 환경 조정이 필요하다면 필요 시 ` `NCCL_IGNORE_CPU_AFFINITY=1` `를 설정하세요. 또는 CPU 스레드를 다른 토폴로지 구성에 고정하는 등의 방법을 시도해 볼 수 있습니다. 비교적 적은 노력으로 몇 가지 작은 변경만으로 매우 낮은 자원 활용도를 최대 활용도로 전환할 수 있는 경우가 있습니다(물론 말처럼 쉽지는 않지만 긍정적인 태도를 유지하는 것이 좋습니다!):

변경 후 재측정

모든 디버깅 작업과 마찬가지로 한 번에 한두 가지만 변경한 후 측정하는 것이 중요합니다. 그렇지 않으면 어떤 변경이 도움이 되었는지 알 수 없습니다. 현재 구성에서 좋은 확장성을 달성했다면, 좋은 메트릭 값을 기록하고 확장 시에도 이러한 좋은 값을 유지하는 것을 목표로 삼으십시오.

소프트웨어를 최신 상태로 유지하되 항상 검증

NCCL이나 CUDA의 새 버전은 종종 성능 향상을 가져올 수 있는 개선 사항을 제공합니다. 예를 들어, 최신 NCCL은 계층적 처리를 자동으로 수행하거나 통신/계산 중첩 메커니즘을 사용할 수 있습니다. 또는 PyTorch 업데이트는 DDP 오버헤드를 줄이거나 더 효율적인 분산 최적화기를 도입할 수 있습니다. 그러나 각 업데이트는 최적 설정을 변경함으로써 불안정성

을 초래할 수 있습니다. 프로파일링 워크플로를 다시 실행하고 마지막으로 확인된 정상 시스템 구성에서 얻은 우수한 메트릭 값을 유지하고 있는지 확인하십시오.

최신 하드웨어 기능 활용

갑자기 더 많은 통합 메모리, 더 높은 메모리 대역폭, 더 빠른 상호 연결을 갖춘 최신 하드웨어를 제공받았다고 가정해 보겠습니다. 첫 번째 단계는 새로운 개선 사항을 이해하고 이를 활용하는 것입니다. 이제 더 큰 배치 크기의 입력 데이터를 사용하고 더 큰 모델을 메모리에 맞출 수 있습니다. 다만, 점진적으로 확장하고 리소스 사용량을 모니터링해야 합니다. 너무 공격적으로 확장하면 새로 개선된 리소스가 포화 상태에 이르러 프로파일링 및 튜닝 워크플로를 다시 시작해야 할 수 있습니다!

프로덕션 환경에서 모니터링 자동화

대규모 훈련 및 추론 워크로드를 정기적으로 실행한다면, 프로덕션 환경에서 GPU 활용도, 네트워크 처리량, 메모리 처리량을 지속적으로 자질할 수 있는 일관된 모니터링 체계를 구축하는 것이 좋습니다. 이렇게 하면 환경 문제, 커널 업데이트, 데이터 파이프라인 회귀로 인해 작업이나 추론 요청이 예상보다 느리게 실행될 때 신속하게 파악할 수 있습니다. 쿠버네티스 및 기타 작업 스케줄러는 모니터링 도구와 원활하게 통합됩니다. 예를 들어 사용률이 특정 임계값 아래로 떨어지면 경보를 설정하세요.

문서화 및 교육 실시

성능 튜닝은 종종 팀 내 암묵적 지식(예: 어떤 환경 변수가 재정의되는지, 어떤 라이브러리 버전이 버그가 있는지 등)을 필요로 합니다. 이러한 발견 사항을 코드나 구성 파일에 직접 문서화하여 다른 사람이나 미래의 자신이 해당 파일을 열 때마다 이를 상기할 수 있도록 하세요. 예를 들어, "이 클러스터 구성에서 `NCCL_SOCKET_NTHREADS=2` 설정이 멀티노드 처리량을 10% 향상시킨다는 사실을 확인했습니다"라고 기록하세요. 이는 표준 관행이 되길 바랍니다.

이 워크플로를 지속적으로 따르면 본질적으로 피드백 루프가 생성됩니다: 실행 → 측정 → 조정 → 실행 → 측정 → 조정 →... 이 피드백 루프는 GPU를 더 많이 확장하고 더 나은 모델로 전환할 때도 시스템 성능과 효율성을 지속적으로 유지하도록 보장합니다. 시간이 지나 성능이 저하된 후 다시 회복하는 것보다 좋은 성능을 유지하는 것이 훨씬 쉽습니다. 수많은 움직이는 부품들이 최고 수준으로 작동하도록 유지하는 것은 끊임없는 싸움입니다.

요약하자면, 성능을 지속적인 테스트와 검증이 필요한 기능처럼 다루어야 합니다. 코드 정확성을 위해 테스트를 작성하듯이, 성능을 위한 테스트 도구도 마련해

야 합니다. 예를 들어, GPU를 두 배로 늘리면 처리량도 대략 두 배로 증가합니까? 그렇지 않다면 프로파일러를 활용하여 프로파일링 및 튜닝 워크플로를 시작하십시오. 고수준 시스템 관점을 위한 Nsight Systems, 저수준 GPU 커널 자질을 위한 Nsight Compute, NCCL 및 PyTorch 로깅을 함께 사용하는 것이 권장됩니다. 이들을 결합하면 문제가 발생할 때 정확히 원인을 파악할 수 있는 포괄적인 도구 세트를 확보할 수 있습니다.

이 과정을 마치면 정교하게 튜닝된 AI 시스템을 갖게 됩니다. 코드나 하드웨어를 업데이트할 때마다 이 과정을 반복하세요. AI처럼 역동적이고 빠르게 변화하는 환경에서는 성능 튜닝이 끝이 없습니다. 하지만 무엇을 살펴야 할지 알면 훨씬 수월해집니다. 바로 그 이유 때문에 여러분은 이 책을 읽고 있는 것입니다!

통신 대 연산 바운디드 워크로드 진단

예를 들어 모델 훈련 워크로드에서 계산 과 통신 중 어느 것이 제한 요소인지 파악하려면, 계산과 통신의 비율을 변경하여 NIC에서 측정된 네트워크 처리량(GB/s)에 미치는 영향을 관찰할 수 있습니다. 훈련 작업의 백워드 패스 자질을 고려해 보세요. 현재 100 GB/s NIC에서 그라디언트 올-리듀스가 60 GB/s만 활용하고 있음을 보여줍니다.

이 시나리오에서 네트워크가 막혔는지, 아니면 GPU가 너무 느린지 파악하려면 통신량을 고정하고 배치 크기를 늘리거나 줄여 계산량을 증가/감소시킬 수 있습니다. 배치 크기를 늘려도 역전파 단계의 그라디언트 올-리듀스 과정에서 전송되는 데이터 양에는 영향을 미치지 않기 때문에 이 방법이 이상적입니다. 이는 그라디언트 수가 배치 크기가 아닌 모델 매개변수 수에 비례하기 때문입니다.

통신량을 고정시킨 상태에서 배치 크기를 절반으로 줄이고, 이로 인해 달성된 네트워크 처리량에 어떤 영향이 있는지 확인하세요. 처리량이 60GB/s로 유지된다면, GPU는 더 많은 작업을 수행할 수 있었지만 네트워크가 이를 허용하지 않았다는 의미입니다. 따라서 네트워크가 병목 요인입니다.

그러나 달성된 네트워크 활용도가 60GB/s 미만(예: 40GB/s)으로 떨어진다면, GPU가 계산을 충분히 빠르게 완료하지 못해 NIC를 바쁘게 유지하지 못함으로써 네트워크를 가동 불능 상태로 만들고 있는 것입니다. 이 경우 네트워크는 GPU로부터 더 많은 데이터를 기다리며 유휴 상태입니다. 따라서 컴퓨팅이 제한 요소이며 네트워크가 아닙니다.

이 가설을 검증하려면 실험을 반대로 진행하여 배치 크기를 두 배로 늘려보세요. 이때 올-리듀스(all-reduce) 기울기 통신량은 동일하게 유지됩니다. 따라서 통신

이 진정한 병목이라면 배치 크기와 연산 작업량이 증가해도 NIC 속도는 60 GB/s로 유지될 것입니다. 하지만 연산이 병목이라면 전체 반복 시간 중 올-리듀스 통신에 소요되는 비율은 증가하는 연산 시간에 비해 상대적으로 줄어들 것입니다.

이 두 실험에서 NIC의 절대적 GB/s와 계산 대비 통신에 소요된 상대적 시간을 관찰하면 정확히 어떤 하위 시스템을 조정해야 하는지 파악할 수 있습니다. 보다 구체적으로, 배치 크기를 증가/감소시키면서 GB/s 대 통신 비율을 그래프로 표시할 수 있습니다. 이를 통해 정확히 어디에 한계선이 있는지 확인하고 워크로드가 통신 바운디드(네트워크)인지 계산 바운디드(GPU SM)인지 판단할 수 있습니다.

Nsight Systems를 사용하여 에서 종단 간 타임라인을 확인하세요. GPU가 유휴 상태이거나 NCCL 대기 시간에 해당하는 컴퓨팅 커널 간 긴 간격 형태로 데이터를 기다리는 경우, 통신 병목 현상이 발생했을 가능성이 높습니다. GPU가 바쁘지만 예상 FLOPS에 도달하지 못하는 경우, 메모리 병목 또는 컴퓨팅 병목일 수 있습니다. Nsight Compute와 PyTorch 자질을 사용하면 커널의 메모리 및 컴퓨팅 효율성을 판단하는 데 도움이 됩니다.

핵심 교훈

분산 AI의 최고 성능은 GPU 커널과 네트워크 전송부터 CPU 스레드와 스토리지에 이르는 전체 스택을 함께 최적화함으로써 달성됩니다. 이 중 어느 하나라도 취약하면 전체 시스템이 병목 현상을 일으킬 수 있습니다. 스토리지 계층을 조정할 때 기억해야 할 핵심 교훈은 다음과 같습니다:

계산 확장 시 입력 데이터 파이프라인도 함께 확장하세요

GPU를 확장할 때 스토리지와 데이터 로딩을 소홀히 하지 마십시오. 스토리지 시스템이 충분한 대역폭을 제공하는지, 그리고 이 대역폭을 완전히 활용하고 있는지 확인하십시오. GPU 수가 증가함에 따라 데이터 로더 병렬 처리도 함께 늘리십시오. 그렇지 않으면 입력 파이프라인이 따라가지 못해 GPU를 추가해도 속도 향상이 없는 지점에 도달하게 됩니다.

작업에 적합한 도구 사용

NCCL은 모델 훈련에서 흔히 사용되는 확장 가능한 집단 통신(올리드류드 등)을 위해 설계되었습니다. NIXL은 모델 추론에서 흔히 발생하는 고처리량 지점 간 및 스트리밍 전송을 대상으로 합니다. 토큰 스트리밍이 워크로드의 대부분을 차지하는 경우 NIXL을 사용하십시오. 반면, 대량 집단 및 대칭 메모리 패턴에는 NCCL/NVSHMEM을 선호하십시오. GPUDirect RDMA와 GDS는 각각 네트워크 및 스토리지 I/O에 대한 호스트 메모리 바

운스 버퍼를 제거하지만, 전송 스케줄링 및 제어는 여전히 CPU가 담당합니다. 다중 GPU 훈련에는 항상 `DataParallel` 보다 `DistributedDataParallel` 을 사용하십시오. 이러한 전용 라이브러리와 프레임워크는 하드웨어 성능을 극대화하기 위해 존재하며, 철저히 튜닝되었습니다. 재발명하지 말고 이를 활용하십시오.

엔드투엔드 자질 수행

병목 지점이 항상 명확한 것은 아닙니다. Nsight Systems, Nsight Compute, PyTorch 프로파일러와 같은 도구를 사용하여 시간 소모 지점을 확인하세요. GPU는 컴퓨팅 바운디드, 통신 바운디드 또는 I/O 바운디드 상태일 수 있습니다. 앞서 논의한 자질 예시를 따르면 문제의 근원을 찾는 데 도움이 됩니다. 예를 들어, NCCL 커널이 컴퓨팅과 적절히 결합되었는지 확인하고 Nsight Systems에서 GPU 유휴 시간을 점검할 수 있습니다.

결론

고성능 분산 스토리지 시스템은 대규모 복잡한 AI 시스템 튜닝의 핵심 구성 요소입니다. NVMe SSD 및 GDS와 같은 고급 스토리지 기술을 통합하면 데이터 로딩 파이프라인 성능을 개선하고, 훈련 시간을 단축하며, 실험 및 반복 속도를 높일 수 있습니다.

오프라인 사전 처리 데이터 파이프라인, 효율적인 데이터 캐싱, 비동기 통신과 같은 기법을 통해 스토리지 및 I/O 문제를 해결하면 모델 복잡성과 데이터셋 규모가 확대되더라도 현대 AI 배포 환경에서 높은 처리량을 유지할 수 있습니다.

실무자에게 중요한 교훈은 맞춤형 I/O 솔루션을 처음부터 개발할 필요가 없다는 점입니다. NVIDIA와 오픈소스 커뮤니티는 고도로 최적화된 전용 라이브러리와 도구를 제공하므로, 저수준 인프라 관리 대신 모델, 데이터, 애플리케이션 로직에 집중할 수 있습니다.

성능 엔지니어에게는 빠른 데이터 이동이 순수 컴퓨팅 성능만큼 중요하다는 교훈을 줍니다. 세계에서 가장 빠른 GPU라도 스토리지에서 데이터를 계속 기다린다면 거의 이점이 없습니다.

GDS 및 고급 입력 파이프라인과 같은 기술은 데이터 흐름을 원활하게 유지하고 GPU에 작업을 지속적으로 공급하기 위한 풀스택 접근법의 일부입니다. 이러한 기술을 활용하고 지속적으로 자질 및 튜닝을 수행함으로써, 분산 AI 시스템을 대규모로 이론적 한계에 가깝게 끌어올릴 수 있습니다.

다음 장에서는 이 기반을 바탕으로 CUDA 및 PyTorch 최적화 전략과 고급 시스템 튜닝 주제를 심층적으로 다룰 것입니다. 여기서 배운 원칙은 통신/계산을 중첩하고 가능한 가장 빠른 링크를 활용하며 이론적 최대 하드웨어 성능에 접근하는 과정에서 스택의 모든 계층에 계속 적용됩니다. 궁극적으로 이는 더 빠른 인사이트 도출 시간, 자원 활용도 향상, 비용 절감으로 이어질 것입니다.