제3장. GPU 기반 환경을 위한OS , Docker 및 쿠버네티스 튜닝

고도로 최적화된 GPU 코드와 라이브러리라도 시스템 수준의 병목 현상은 대규모 AI 훈련 성능을 제한할 수 있습니다. 가장 빠른 GPU도 데이터와 명령어를 공급하는 환경만큼만 성능을 발휘합니다. 이 장에서는 GPU가 최대 성능을 발휘할 수 있도록 운영 체제와 컨테이너 런타임을 조정하는 방법을 살펴봅니다.

먼저 기초적인 GPU 소프트웨어 스택을 살펴봅니다. 이후 NUMA 어피니티 및 휴지페이지(hugepages)와 같은 핵심 CPU 및 메모리 최적화 기법에 대해 심층적으로 다룹니다. 이는 데이터가 스토리지에서 CPU를 거쳐 GPU로 효율적으로 흐르도록 보장합니다. 동시에 퍼시스턴스 모드, 멀티프로세스 서비스(MPS), 멀티인스턴스 GPU(MIG) 파티션과 같은 중요한 GPU 드라이버 설정도 논의합니다. 이러한 설정은 오버헤드를 줄이고 자원을 효과적으로 동기화하여 GPU 활용도를 극대화하는 데 도움이 됩니다.

NVIDIA 컨테이너 툴킷, 컨테이너 런타임, 쿠버네티스 토폴로지 매니저, 쿠버네티스 GPU 오퍼레이터와 같은 솔루션을 활용하면 GPU 환경을 위한 통합되고 고도로 최적화된 소프트웨어 스택을 구축할 수 있습니다. 이러한 솔루션은 단일 노드 및 다중 노드 GPU 환경 전반에 걸쳐 효율적인 리소스 할당과 워크로드 스케줄링을 가능하게 하며, GPU 기능이 완전히 활용되도록 보장합니다.

이를 통해 이러한 최적화가 중요한 이유에 대한 직관을 쌓게 될 것입니다. 본질적으로 이러한 최적화는 지연 시간을 최소화하고 처리량을 극대화하며, GPU가 지속적으로 데이터로 공급되어 최고 성능으로 작동하도록 보장합니다. 그 결과 훈련 및 추론 워크로드 모두에 대해 상당한 성능 향상과 높은 유효 처리량 비율을 제공하는 강력하고 확장 가능한 시스템이 구현됩니다.

운영 체제

운영 체제(OS)는 모든 것이 실행되는 기반 입니다. GPU 서버는 일반적으로 최신 GPU 하드웨어를 지원하는 업데이트된 커널이 탑재된 Ubuntu Server LTS 또는 Red Hat과 같은 Linux 배포판을 실행합니다. NVIDIA 드라이버는 커널 모듈을 설치하여 각 GPU마다 하나씩 `/dev/nvidia0`, `/dev/nvidia1`, `/dev/nvidia2`와 같은 장치 파일을 생성합니다. 또한 드라이버 제어 작업을

위한 `/dev/nvidiactl`, 통합 가상 메모리를 위한 `/dev/nvidia-uvm`, 모드 설정 및 버퍼 관리를 위한 `/dev/nvidia-modeset` 도 생성합니다.

운영체제는 CPU 스케줄링, 메모리, 네트워킹, 스토리지를 관리하며, 이 모든 요소는 높은 GPU 처리량을 위해 최적화되어야 합니다. 따라서 운영체제는 GPU 작업에 간섭하지 않도록 구성되어야 합니다. 예를 들어, GPU 노드는 스왑을 비활성화하거나 ` `vm.swappiness` `을 0으로 설정하여 GPU 워크로드에 방해가 될 수 있는 운영체제 주도 메모리 스왑을 방지해야 합니다. 성능 엔지니어의 역할 중 하나는 이러한 운영체제 설정을 조정하여 GPU가 최대 성능을 발휘할 수 있도록 준비하는 것입니다.

GPU 중심 서버는 추가 데몬(배경 프로세스)을 실행할 수 있습니다. 예를 들어 NVIDIA Persistence Daemon은 GPU 작업이 실행되지 않을 때도 GPU 드라이버와 하드웨어 컨텍스트를 로드된 상태로 유지합니다. 또한 Fabric Manager는 GPU 상호 연결 토폴로지를 관리하며, NVIDIA Data Center GPU Manager(DCGM)는 GPU 시스템 상태 지표를 모니터링합니다.

NVIDIA 소프트웨어 스택

다중 페타플롭스급 GPU 클러스터 를 운영하려면 고수준 PyTorch, TensorFlow 또는 JAX 코드 작성 이상의 작업이 필요합니다. GPU 운영을 뒷받침하는 전체 소프트웨어 스택이 존재하며, 각 계층이 성능에 영향을 미칠 수 있습니다. 그림 3-1 은 현대적인 LLM 워크로드를 개발하고 생산화하는 데 사용되는 프레임워크, 라이브러리, 컴파일러, 런타임 및 도구 세트 를 보여줍니다. 여기에는 PyTorch, cuDNN, cuBLAS, CUTLASS, CUDA C++, `nvcc` 및 CUDA 런타임 API(예: CUDA 도구, 드라이버 등)가 포함됩니다.

또한 NVIDIA GPU 및 CUDA 생태계 는 Python 라이브러리를 포용하며, OpenAI의 Triton 도메인 특정 언어(DSL) 및 NVIDIA의 Warp 프레임워크와 같은 프레임워크와 NVIDIA의 CUDA Python, cuTile 및 CUTLASS 라이브러리를 사용하여 Python으로 CUDA 커널을 생성할 수 있게 해줍니다.

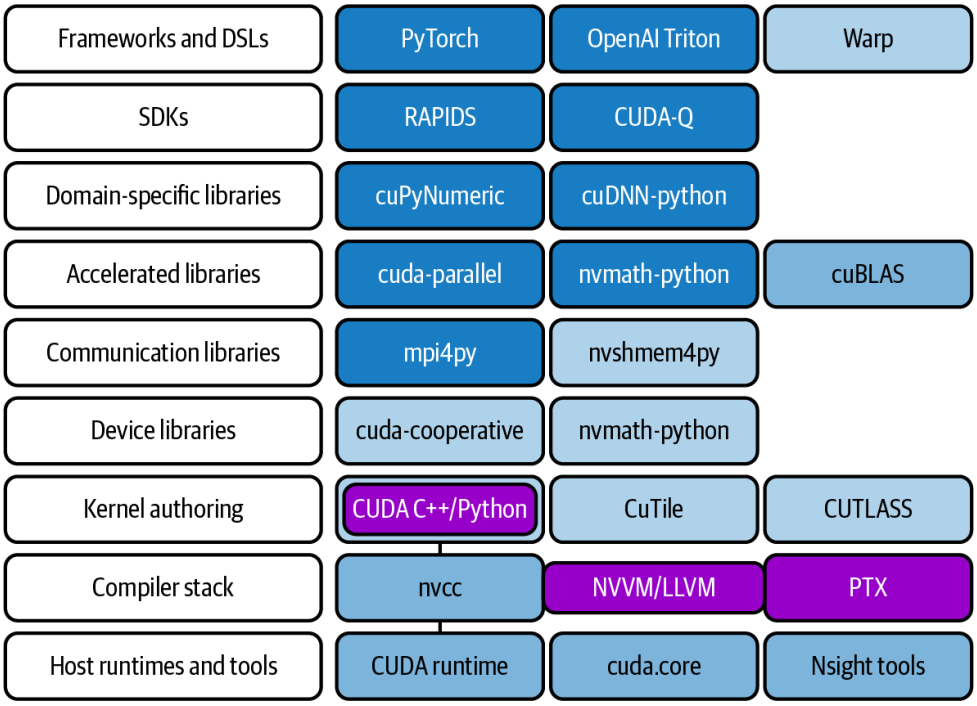

그림 3-1. 현대적인 LLM 워크로드를 개발하고 프로덕션화하는 데 사용되는 프레임워크, 라이브러리, 컴파일러, 런타임 및공통 도구 세트

GPU 드라이버

기초에는 NVIDIA GPU 드라이버( )가 있으며, 이는 Linux OS와 GPU 하드웨어 간 인터페이스를 담당합니다. 이 드라이버는 장치 내 메모리 할당, GPU 코어에서의 작업 스케줄링, 다중 테넌트 사용을 위한 GPU 분할 등 저수준 GPU 작업을 관리합니다.

GPU 드라이버는 GPU 기능을 활성화하고 하드웨어에 지속적으로 작업을 공급합니다. NVIDIA 드라이버를 최신 상태로 유지하는 것이 중요합니다. 새 드라이버 릴리스는 종종 성능 향상을 가능하게 하고 최신 GPU 아키텍처 및 CUDA 기능을 지원합니다.

`nvidia-smi` 과 같은 도구는 드라이버와 함께 제공되며( ), 온도 모니터링, 사용률 측정, 오류 정정 코드(ECC) 메모리 상태 조회, 지속성 모드와 같은 다양한 GPU 모드 활성화 등을 수행할 수 있습니다.

CUDA 툴킷 및 런타임

드라이버 상위에는 NVIDIA CUDA 런타임( )과 CUDA 툴킷(CUDA Toolkit)이라 불리는 라이브러리가 위치합니다. 이 툴킷에는 CUDA C++ 커널 컴파일에 사용되는 CUDA 컴파일러(`nvcc`)가 포함됩니다. 컴파일된 CUDA 프로그램은 CUDA 런타임(`cudart`)과 링크됩니다. CUDA 런타임은 NVIDIA 드라이버와 직접 통신하여 작업을 실행하고 GPU에 메모리를 할당합니다.

또한 CUDA 툴킷은 다양한 최적화 라이브러리를 제공합니다: 신경망 기본 연산을 위한 cuDNN, 선형 대수 연산을 위한 cuBLAS, 다중 GPU 통신을 위한 NCCL 등. 따라서 GPU의 컴퓨트 능력(Compute Capability, CC)을 지원하는 최신 CUDA 툴킷 버전을 사용하는 것이 중요합니다. 최신 툴킷에는 해당 GPU에 특화된 최신 컴파일러 최적화와 라이브러리가 포함되어 있기 때문입니다. CUDA 컴

파일러와 프로그래밍 모델, 그리고 CUDA(및 PyTorch) 최적화에 대해서는 이후 장에서 더 자세히 다룰 예정입니다.

GPU 하드웨어 세대 간 CUDA 전방 및 후방 호환성

의 중요한 특징 NVIDIA의 GPU 프로그래밍 모델은 하드웨어 세대 간 호환성입니다. CUDA 코드를 컴파일하면, 결과 바이너리에는 그림 3-2와 같이 가상 또는 중간 PTX 코드와 물리적 장치 코드(예: ARM, x86, GPU 명령어)가 포함됩니다.

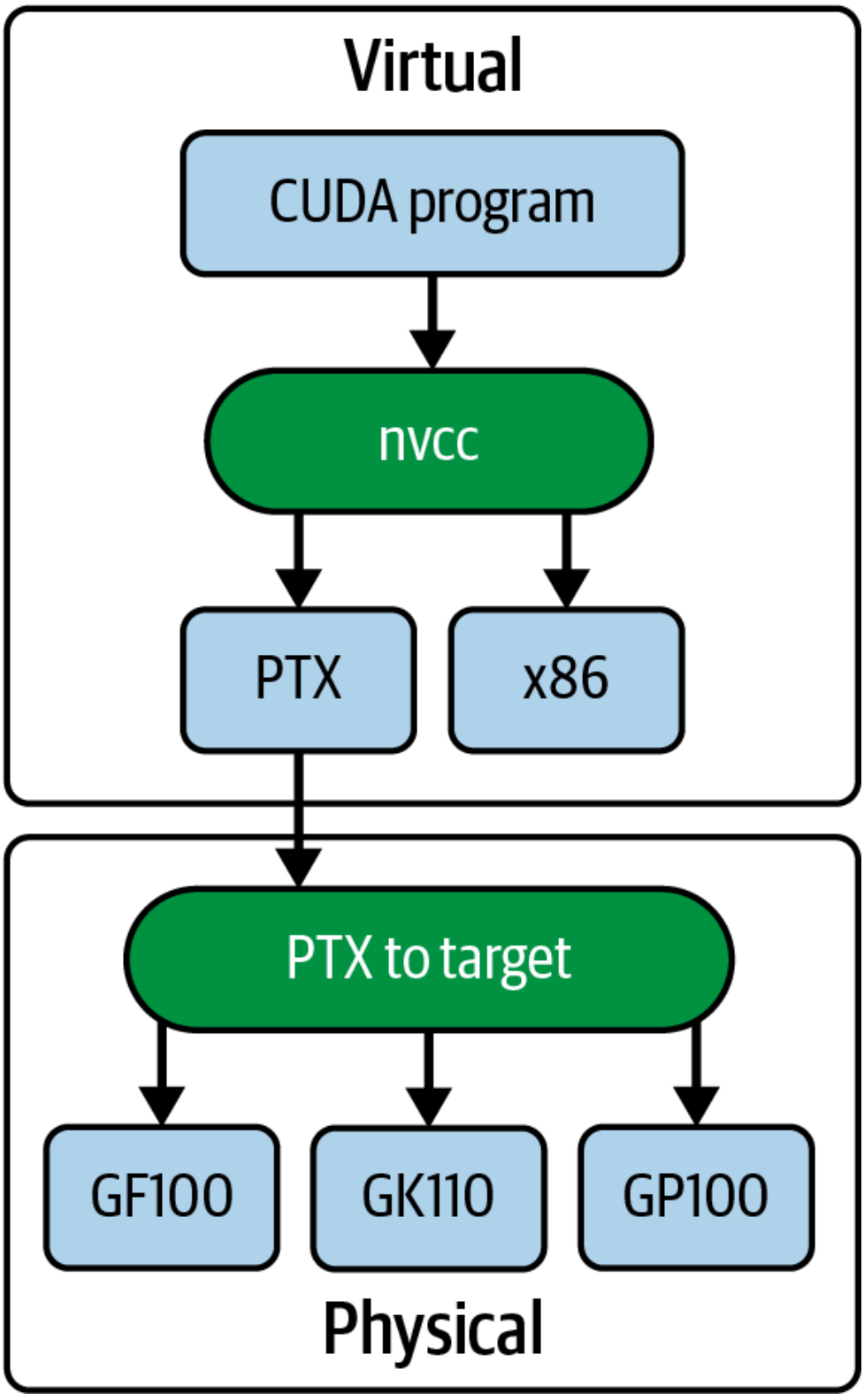

그림 3-2. CUDA 프로그램을 PTX로 컴파일하는 데사용되는 ' `nvcc` ' 명령어—최종적으로 GPU 대상 장치용 저수준 명령어로 변환됨

이를 통해 최신 GPU는 PTX를 JIT(Just-in-Time) 컴파일하여 향후 아키텍처에서도 프로그램이 실행될 수 있게 하고, 최신 GPU는 이전 아키텍처용 구형 바이너리 코드를 실행할 수 있게 합니다. 이러한 호환성은 NVIDIA의 *팻바이너리* 모델을 통해 달성되며, 이 모델은 향후 대비를 위한 PTX와 알려진 아키텍처를 위한 *CUBIN*(아키텍처별 CUDA 장치 코드 바이너리)을 포함합니다.

CUBIN은 에서 `nvcc` 의 `-cubin` 옵션을 사용하여 생성된 바이너리입니다. 이는 특정 NVIDIA 아키텍처를 위한 컴파일된 GPU 스트리밍 어셈블러(SASS) 명령어를 포함합니다. 이 바이너리는 런타임에 CUDA 드라이버가 로드할 수 있도록 팻바이너리로 패키징됩니다. 중간 단계의 전방 호환 표현인 PTX와 달리, CUBIN 바이너리 파일은 알려진 GPU 아키텍처에서 직접 실행이 가능합니다. 팻바이너리에 PTX와 함께 포함될 경우, CUBIN은 미래 GPU를 위한 PTX의 JIT 컴파일과 최신 하드웨어에서 구형 CUBIN 코드 실행을 모두 지원합니다.

요약하면, CUDA는 PTX가 내장될 때 전방 호환성을 제공합니다. 드라이버가 런타임에 새로운 아키텍처를 위해 PTX를 JIT 컴파일할 수 있기 때문입니다( ). CUBIN 객체는 아키텍처 전용이며 향후 GPU 아키텍처와 전방 호환되지 않습니다. 따라서 현재 아키텍처용 SASS와 전방 호환성을 위한 PTX를 모두 포함하는 PTX를 포함하거나 팻 바이너리(일명 "fatbinaries" 또는 간단히 "fatbins")를 배포해야 합니다.

C++ 및 Python CUDA 라이브러리

대부분의 CUDA 툴킷 라이브러리는 C++ 기반이지만, NVIDIA의 현재 Python 지원 옵션에는 CUDA Python(예: 저수준 드라이버 및 런타임 접근), 배열 프로그래밍용 cuPyNumeric, CuTe DSL, cuTile, CuPy, 그리고 Python으로 GPU 커널을 작성하기 위한 NVIDIA Warp가 포함됩니다. CUTLASS는 Python 라이브러리가 아닌 cuBLAS 같은 라이브러리가 내부적으로 사용하는 C++ 템플릿 라이브러리입니다.

대부분의 CUDA 툴킷 라이브러리는 C++ 기반이지만, NVIDIA에서는 C++ 툴킷을 기반으로 하며 "Cu" 접두사가 붙은 Python 기반 라이브러리가 점점 더 많이 등장하고 있습니다. 예를 들어, cuTile과 cuPyNumeric은 2025년 초에 출시된 Python 라이브러리입니다. 이들은 Python 개발자가 CUDA를 사용하여 NVIDIA GPU용 애플리케이션을 구축하는 진입 장벽을 낮추는 것을 목표로 합니다.

cuTile은 GPU에서 대형 행렬을 더 작은 관리 가능한 부분 행렬(*타일)*로 분할하여 작업을 단순화하도록 설계된 Python 라이브러리 입니다. 블록 단위 연산 수행, 메모리 접근 패턴 최적화, GPU 커널 효율적 스케줄링을 용이하게 하는 고수준 타일 기반 추상화를 제공합니다.

대형 행렬을 타일로 분할함으로써 cuTile은 개발자가 저수준 세부 사항을 수동으로 관리할 필요 없이 GPU의 병렬 처리를 최대한 활용할 수 있도록 지원합니다. 이 접근 방식은 행렬 연산이 집중적으로 필요한 애플리케이션에서 캐시 사용률 향상과 전반적인 성능 개선으로 이어질 수 있습니다.

cuPyNumeric은 GPU를 활용하는 인기 있는 `numpy` Python 라이브러리 의 드롭인 대체품(`import cupynumeric as np`)입니다. NumPy와 거의 동일한 함수, 메서드 및 동작을 제공하므로 개발자는 코드를 최소한으로 변경하여 전환할 수 있습니다. 내부적으로 cuPyNumeric은 CUDA를 활용하여 GPU에서 병렬 연산을 수행합니다. 이는 대규모 수치 계산, 행렬 연산, 데이터 분석과 같은 연산 집약적 작업에서 상당한 성능 향상을 가져옵니다.

GPU로 작업을 오프로드함으로써 cuPyNumeric은 대규모 데이터셋을 처리하는 애플리케이션의 계산 속도를 가속화하고 효율성을 향상시킵니다. 완전히 새로운 인터페이스를 배우지 않고도 Python 개발자가 GPU 성능을 활용할 수 있도록 진입 장벽을 낮추는 것이 목표이며, 고성능 컴퓨팅을 위한 NumPy의 강력한 대체재 역할을 합니다.

또 다른 주목할 만한 Python 기반 GPU 프로그래밍 모델은 OpenAI의 오픈소스 Triton 언어 및 컴파일러입니다. Triton은 Python으로 커스텀 GPU 커널을 작성할 수 있게 해주는 Python DSL입니다. NVIDIA 라이브러리는 아니지만, Triton은 개발자가 Python으로 직접 고성능 커널을 작성할 수 있게 함으로써 CUDA를 보완합니다.

Triton과 다양한 Triton 기반 최적화 기법은 후반 장에서 다루겠지만, Triton이 많은 경우 수동으로 작성하는 CUDA C++의 필요성을 줄여준다는 점만 알아두시기 바랍니다. 또한 Triton은 PyTorch 컴파일러 백엔드에 통합되어 GPU 연산을 자동으로 최적화하고 융합하여 성능을 향상시킵니다. 이제 PyTorch에 대한 논의로 넘어가 보겠습니다.

PyTorch와 상위 AI 프레임워크

CUDA 기반의 인기 있는 Python 프레임워크 로는 PyTorch, TensorFlow, JAX, Keras 등이 있습니다. 이 프레임워크들은 NVIDIA GPU의 성능을 활용하면서 Deep Learning을 위한 고수준 인터페이스를 제공합니다. 이 책은 주로 PyTorch의 컴파일 및 그래프 최적화 기능, 특히 `torch.compile` 스택에 초점을 맞춥니다.

PyTorch 컴파일러 스택 은 TorchDynamo, AOT Autograd, 그리고 TorchInductor나 가속 선형 대수(XLA)와 같은 백엔드로 구성되어 모델을 자동으로 캡처하고 최적화합니다. TorchInductor가 가장 흔한 백엔드이며, 내부적으로는 OpenAI의 Triton을 사용합니다. Triton은 커널을 융합하고 특정 GPU 및 시스템 환경에 맞게 커널 자동 튜닝을 수행합니다. 이에 대해서는 14장에서 다루겠습니다.

GPU를 사용하여 PyTorch 텐서에 연산 을 수행할 때, 이는 단일 Python 호출로 보이지만 실제로는 CPU에서 GPU로 이동됩니다. 그러나 이 단일 호출은 그림 3-3과 같이 다양한 CUDA 라이브러리를 활용하는 CUDA 런타임에 대한 일련의 호출로 변환됩니다.

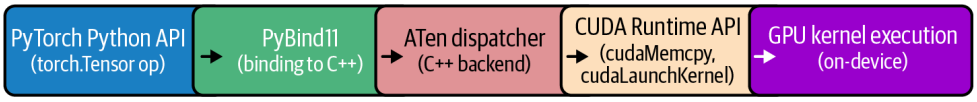

그림 3-3. PyTorch 코드에서 GPU 장치로의흐름

예를 들어 행렬 곱셈을 수행할 때 PyTorch는 이러한 작업을 cuBLAS와 같은 라이브러리에 위임합니다. cuBLAS는 CUDA 툴킷의 일부이며 GPU 실행에 최적화되어 있습니다. 내부적으로 PyTorch는 전파(forward pass) 및 역전파(backward pass)와 같은 연산이 저수준의 최적화된 CUDA 함수와 라이브러리를 사용하여 실행되도록 보장합니다.

요약하면, PyTorch는 직접적인 CUDA 프로그래밍의 복잡성을 추상화하여 직관적인 Python 코드를 작성할 수 있게 하며, 이 코드는 궁극적으로 고도로 최적화된 CUDA 루틴을 호출하여 개발 편의성과 높은 성능을 동시에 제공합니다. CUDA 프로그래밍과 최적화에 대해서는 4장과 5장에서,PyTorch 최적화에 대해서는 9장에서 논의할 것입니다.

이상적인 GPU 기반 개발 환경을 구축하려면 OS, GPU 드라이버, CUDA 툴킷, CUDA 라이브러리, PyTorch 등 모든 구성 요소가 함께 작동해야 합니다. 연구자가 훈련 작업을 제출하면 스케줄러가 노드를 예약하고, OS는 NVIDIA 드라이버를 사용하여 GPU 장치와 메모리를 할당하며, 컨테이너는 최적화되고 하드웨어를 인식하는 CUDA 라이브러리를 포함한 올바른 소프트웨어 환경을 제공합니다. 사용자 코드(예: PyTorch, TensorFlow, JAX)는 이러한 CUDA 라이브러리를 사용하며, 이 라이브러리는 궁극적으로 드라이버 및 하드웨어와 통신합니다.

본 장에서 설명하는 최적화 기법은 이 스택의 각 계층을 최대한 효율적으로 만들기 위해 설계되었습니다. 이를 통해 GPU가 CPU 대기, 메모리 또는 디스크 I/O 대기, 다른 GPU 동기화 대기 대신 실제 유용한 훈련 및 추론 작업에 집중할 수 있도록 지원합니다.

잘 조정된 시스템은 수십 개의 GPU에 분산된 모델이 I/O나 OS 오버헤드로 인해 병목 현상이 발생하지 않도록 보장합니다. 시스템 수준 조정은 모델 최적화에 비해 종종 간과되지만, 시스템 수준 최적화는 상당한 성능 향상을 가져올 수 있습니다. 경우에 따라 OS 수준 구성에 약간의 조정만으로도 두 자릿수 퍼센트의 성능 향상을 얻을 수 있습니다. 대규모 AI 프로젝트 규모에서는 이로 인해 수만 달러에서 수십만 달러에 달하는 컴퓨팅 시간을 절약할 수 있습니다.

GPU 환경을 위한 CPU 및 OS 구성

GPU가 완전한 활용도에 도달하지 못하는 가장 흔한 이유 중 하나는 CPU가 유용한 작업으로 GPU를 지속적으로 공급하지 못하기 때문입니다. 일반적인 훈련 루프에서 CPU는 디스크에서 데이터 로드, 데이터 토큰화, 변환 등을 포함하여 다음 배치 데이터를 준비하는 역할을 담당합니다. 또한 CPU는 GPU 커널을 디스패치하고 스레드 및 프로세스 간 협업을 조정하는 책임도 있습니다.

이러한 호스트 측 작업이 느리거나 OS가 이를 효율적으로 스케줄링하지 못하면 고가의 GPU가 유휴 상태에 빠질 수 있습니다. 트랜지스터를 가만히 두며 다음 작업이나 데이터 배치를 기다리는 셈이죠. 이를 방지하려면 CPU와 OS가 GPU 워크로드를 처리하는 방식을 최적화해야 합니다.

이러한 최적화에는 CPU 어피니티 설정을 통해 크로스 NUMA 노드 트래픽을 방지하여 적절한 코어가 적절한 데이터를 처리하도록 하고, 메모리 할당 전략을 사용하여 NUMA 페널티를 피하며, 불필요한 지연 시간을 제거하기 위한 OS 수준 변경을 적용하는 것이 포함됩니다. 이렇게 하면 GPU가 데이터 부족에 시달리지 않습니다. 이 중 일부는 백그라운드 데몬과 OS 작업을 자체 코어에 격리하고 GPU에 데이터를 공급하는 코어로부터 멀리 떨어뜨리는 것을 포함하며, 이에 대해서는 다음에서 논의하겠습니다.

NUMA 인식 및 CPU 고정

현대 서버 CPU는 수십 개의 코어 를 갖추고 있으며, 종종 여러 NUMA 노드로 분할됩니다. *NUMA* 노드는 물리적으로 가까운 CPU, GPU, 네트워크 인터페이스 컨트롤러(NIC), 메모리를 논리적으로 그룹화한 것입니다. 시스템의 NUMA 아키텍처를 인지하는 것은 성능 튜닝에 중요합니다. 단일 NUMA 노드 내 리소스에 접근하는 것이 다른 NUMA 노드의 리소스에 접근하는 것보다 빠릅니다.

예를 들어, NUMA 노드 0의 CPU에서 실행 중인 프로세스가 NUMA 노드 1의 GPU에 접근해야 하는 경우, 노드 간 링크를 통해 데이터를 전송해야 하므로 더 높은 지연 시간이 발생합니다. 실제로 다른 NUMA 노드로 이동할 때 메모리 접근 지연 시간은 거의 두 배 가까이 증가할 수 있습니다.

와 같은 GH200 및 GB200과 같은 Grace 기반 슈퍼칩에서는 CPU와 GPU가 NVLink-C2C로 연결되어 Grace와 페어링된 가속기 간 최대 ~900GB/s의 일관된 CPU-GPU 메모리 액세스를 제공합니다. Linux는 여전히 CPU DRAM을 CPU NUMA 메모리로, GPU HBM을 장치 메모리로 처리합니다. 따라서 일관성으로 인해 소프트웨어 오버헤드가 감소하더라도 CPU 스레드를 로컬 Grace CPU에 바인딩하고 데이터 지역성을 유지해야 합니다.

많은 듀얼 소켓 시스템에서 원격 메모리 접근 지연 시간은 로컬 메모리 접근보다 훨씬 높을 수 있습니다. 한 실험에서 로컬 NUMA 노드 메모리 접근 지연 시간은 약 80ns인 반면, 원격(크로스-노드) 메모리 접근 지연 시간은 약 139ns였습니다. 이는 지연 시간이 약 75% 증가한 것으로, 로컬과 원격 NUMA 노드 메모리 접근 간의 접근 속도 차이는 매우 큽니다.

프로세스를 GPU와 동일한 NUMA 노드에 있는 CPU( )에 바인딩함으로써 이러한 추가 오버헤드를 피할 수 있습니다. 예를 들어, `numactl --cpunodebind=<node> --membind=<node>`를 사용하여 CPU 스레드와 메모리 할당을 모두 GPU의 로컬 NUMA 노드에 바인딩할 수 있습니다. 이에 대해서는 잠시 후 자세히 알아보겠습니다. 핵심 아이디어는 CPU 실행과 메모리 액세스를 해당 GPU가 서비스하는 로컬 영역에 유지하는 것입니다.

Linux는 기본적인 NUMA 밸런싱 기능을 포함하지만( ), 성능이 중요한 AI 워크로드에는 일반적으로 충분하지 않습니다. 기본적으로 프로세스는 NUMA 노드 간에 마이그레이션될 수 있습니다. 이는 원격 메모리 액세스로 인한 추가 지연 시간을 초래합니다. 따라서 프로세스와 메모리를 로컬 GPU와 동일한 NUMA 노드에 명시적으로 바인딩하는 것이 중요합니다. 이는 `numactl`, `taskset` 또는 cgroups를 사용하여 수행할 수 있으며, 이는 잠시 후 설명하겠습니다.

NUMA 친화성을 명시적으로 지정하려면 프로세스나 스레드를 GPU와 동일한 NUMA 노드에 연결된 특정 CPU에 "고정(pin)"해야 합니다. 이러한 유형의 CPU 친화성을 *CPU 고정(CPU pinning)*이라고 합니다. 노드에 8개의 GPU가 있고, 그 중 4개는 NUMA 노드 0에, 나머지 4개는 NUMA 노드 1에 연결되어 있다고 가정해 보겠습니다.

GPU당 하나씩 총 8개의 훈련 프로세스를 실행할 경우, 각 훈련 프로세스를 GPU와 동일한 NUMA 노드에 연결된 CPU 코어(또는 코어 집합)에 바인딩해야 합니다. 이 경우 GPU 0~3은 NUMA 노드 0에, GPU 4~7은 NUMA 노드 1의 코어에 연결됩니다(그림 3-4 참조).

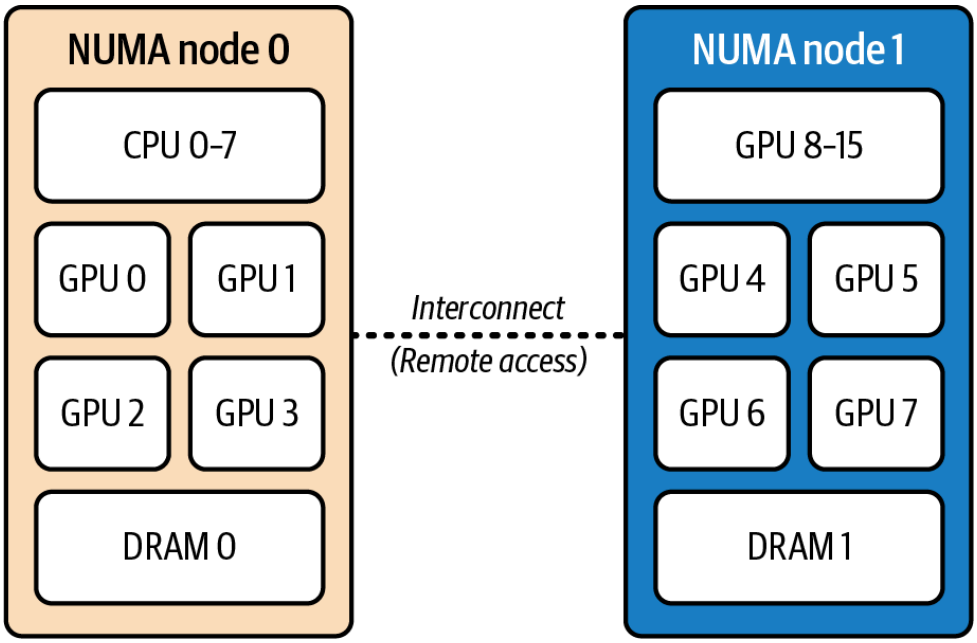

그림 3-4. 노드 내8개의 GPU: 4개는 NUMA 노드 0에, 나머지 4개는 NUMA 노드 1에 연결됨

이렇게 하면 CPU 프로세스가 GPU 4에 데이터를 공급하려 할 때, GPU 4가 NUMA 노드 1에 연결되어 있으므로 NUMA 노드 1에 연결된 CPU에서 실행되어야 합니다. Linux는 이를 수행하는 도구(예: 지정된 NUMA 노드에 고정된 프로세스를 실행하는 `` `numactl --cpunodebind=<node> --membind=<node>` ``)를 제공합니다.

`taskset` 를 사용하여 프로세스를 특정 코어 ID에 고정할 수도 있습니다. 다음은 `numactl` 를 사용하여 `train.py` 스크립트를 GPU 4와 동일한 NUMA 노드 1에서 실행 중인 CPU에 바인딩하는 예시입니다:

```
numactl --cpunodebind=1 --membind=1 \
```

```
    python train.py --gpu 4
```

이 예시는 NUMA 노드 ID를 알고 있으며 스크립트를 단일 GPU에만 바인딩한다는 가정하에 작성되었습니다. `train.py` 를 알 수 없는 NUMA 노드의 여러 GPU에 바인딩하는 것은 다소 복잡합니다. 다음 스크립트는 `nvidia-smi topo` 를 사용하여 토폴로지를 동적으로 쿼리한 후 로컬 NUMA 노드를 기반으로 GPU에 스크립트를 바인딩합니다:

```
#!/bin/bash
for GPU in 0 1 2 3; do
  # Query NUMA node for this GPU
  NODE=$(nvidia-smi topo -m -i $GPU \
         | awk '/NUMA Affinity/ {print $NF}')

  # Launch the training process pinned to that NUMA node
  numactl --cpunodebind=$NODE --membind=$NODE \
    bash -c "CUDA_VISIBLE_DEVICES=$GPU python train.py --gpu $GPU"
done
```

여기서는 `topo -m` 를 사용하여 CPU 및 NUMA 어피니티를 모두 가져옵니다. 그런 다음 NUMA Affinity 열에서 단일 노드 ID를 추출합니다. 마지막으로 해당 노드에 `--cpunodebind` `--membind` 를 해당 노드에 바인딩하여 프로세스의 *스레드와* 메모리 할당이 GPU의 NUMA 도메인 내에 로컬로 유지되도록 합니다.

많은 Deep Learning 프레임워크(포함)는 프로그래밍 방식으로 스레드 어피니티를 설정할 수 있게 합니다. 예를 들어, PyTorch의 DataLoader는 초기화 시 각 작업자 프로세스의 CPU 어피니티를 설정할 수 있도록 ` worker_init_fn ` 를 노출합니다. 다음과 같이 구현됩니다:

```
import os
import re
import glob
import subprocess
import psutil
import ctypes
import torch
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.utils.data import DataLoader, Dataset
from functools import partial

# Optional: NVML is preferred for GPU↔NUMA mapping
try:
    import pynvml as nvml  # pip install nvidia-ml-py3
    _HAS_NVML = True
except Exception:
    _HAS_NVML = False

# --- libnuma for memory binding
```

```
_libnuma = ctypes.CDLL("libnuma.so")
if _libnuma.numa_available() < 0:
    raise RuntimeError("NUMA not available on this system")
_libnuma.numa_run_on_node.argtypes = [ctypes.c_int]
_libnuma.numa_set_preferred.argtypes = [ctypes.c_int]

def parse_physical_cpu_list(phys_str: str):
    """Parse '0-3,8-11' -> [0,1,2,3,8,9,10,11]."""
    cpus = []
    if not phys_str:
        return cpus
    for part in phys_str.split(','):
        part = part.strip()
        if not part:
            continue
        if '-' in part:
            start, end = map(int, part.split('-'))
            cpus.extend(range(start, end + 1))
        else:
            cpus.append(int(part))
    return cpus

def get_numa_cpus_for_node(node: int):
    """Read /sys/devices/system/node/node{node}/cpulist."""
    path = f"/sys/devices/system/node/node{node}/cpulist"
    with open(path, "r") as f:
        return parse_physical_cpu_list(f.read().strip())

def get_numa_cpus_and_memory():
    """Return (current_cpu_mask, preferred_node) from numactl --show."""
    out = subprocess.run(["numactl", "--show"],
        capture_output=True, text=True).stdout
    phys = re.search(r"physcpubind:\s*([\d,\-\s]+)", out).group(1)
    cpus = parse_physical_cpu_list(phys)
    node = int(re.search(r"preferred node:\s*(-?\d+)", out).group(1))
    return cpus, node

def get_gpu_numa_node(device: int) -> int:
    """
    Determine NUMA node for a GPU (prefer NVML; fall back to sysfs;
    final fallback to current preferred node).
    """
    # NVML path (preferred)
    if _HAS_NVML:
        try:
            nvml.nvmlInit()
            props = torch.cuda.get_device_properties(device)
            pci = props.pci_bus_id  # '0000:03:00.0' or '00000000:03:00.0'
            # Normalize to 8-hex-digit domain if needed for NVML
            try:
                domain, bus, devfn = pci.split(':')
                if len(domain) < 8:
                    domain = domain.rjust(8, '0')
```

```
                pci8 = f"{domain}:{bus}:{devfn}"
            except ValueError:
                pci8 = pci
            try:
                handle = nvml.nvmlDeviceGetHandleByPciBusId_v2(pci8)
            except AttributeError:
                handle = nvml.nvmlDeviceGetHandleByPciBusId(pci8)

            # Direct NUMA ID if driver exposes it
            try:
                numa_id = nvml.nvmlDeviceGetNUMANodeId(handle)
                if isinstance(numa_id, int) and numa_id >= 0:
                    return numa_id
            except Exception:
                pass

            # Derive from NVML CPU affinity
            cpu_count = psutil.cpu_count(logical=True)
            elems = (cpu_count + 63) // 64
            mask = nvml.nvmlDeviceGetCpuAffinity(handle, elems)
            cpus = []
            for i, m in enumerate(mask):
                m = int(m)
                for b in range(64):
                    if m & (1 << b):
                        cpu_id = i * 64 + b
                        if cpu_id < cpu_count:
                            cpus.append(cpu_id)
            # Build CPU→NUMA map from sysfs and choose majority node
            cpu2node = {}
            for node_path in sorted(glob.glob("/sys/devices/system/node/no
                node_id = int(os.path.basename(node_path).replace("node",
                with open(os.path.join(node_path, "cpulist"), "r") as f:
                    for c in parse_physical_cpu_list(f.read().strip()):
                        cpu2node[c] = node_id
            counts = {}
            for c in cpus:
                n = cpu2node.get(c)
                if n is not None:
                    counts[n] = counts.get(n, 0) + 1
            if counts:
                return max(counts.items(), key=lambda kv: kv[1])[0]
        except Exception:
            pass

    # sysfs fallback
    try:
        props = torch.cuda.get_device_properties(device)
        pci = props.pci_bus_id
        sysfs_path = f"/sys/bus/pci/devices/{pci}/numa_node"
        with open(sysfs_path, "r") as f:
            val = int(f.read().strip())
            return val if val >= 0 else 0
```

```
    except Exception:
        pass

    # last resort: current preferred node
    _, node = get_numa_cpus_and_memory()
    return node if node >= 0 else 0

def set_numa_affinity(node: int):
    """Bind current process to CPUs and memory of the given NUMA node."""
    cpus = get_numa_cpus_for_node(node)  # IMPORTANT: CPUs of target node
    psutil.Process(os.getpid()).cpu_affinity(cpus)
    _libnuma.numa_run_on_node(node)
    _libnuma.numa_set_preferred(node)
    print(f"PID={os.getpid()} bound to NUMA node {node} (CPUs={cpus})")
    return cpus

def _worker_init_fn(worker_id: int, node: int, cpus: list):
    """Reapply binding in each DataLoader worker (no CUDA calls here)."""
    psutil.Process(os.getpid()).cpu_affinity(cpus)
    _libnuma.numa_run_on_node(node)
    _libnuma.numa_set_preferred(node)
    print(f"Worker {worker_id} (PID={os.getpid()}) bound to NUMA node {node

 # ----- Example usage below -----
class MyDataset(Dataset):
    def __len__(self): return 1024
    def __getitem__(self, idx): return torch.randn(224*224*3, device='cpu'

def main():
    # DDP setup
    dist.init_process_group(backend="nccl", init_method="env://")
    device = torch.cuda.current_device()

    # Determine GPU's NUMA node and bind this process
    gpu_node = get_gpu_numa_node(device)
    cpus = set_numa_affinity(gpu_node)

    # Build dataloader with closure-based worker_init_fn
    dataset = MyDataset()
    init_fn = partial(_worker_init_fn, node=gpu_node, cpus=cpus)
    dataloader = DataLoader(
        dataset,
        batch_size=32,
        num_workers=4,
        pin_memory=True,
        persistent_workers=True,  # reduces worker respawn churn
        worker_init_fn=init_fn,
        prefetch_factor=2,
    )

    # Model and DDP
    model = torch.nn.Linear(224*224*3, 10, bias=True).to("cuda")
    ddp_model = DDP(model, device_ids=[device], static_graph=True)
```

```
    for batch in dataloader:
        batch = batch.to("cuda", non_blocking=True)
        out = ddp_model(batch)
        # ... loss, backward, optimizer ...

if __name__ == "__main__":
    main()
```

이 스크립트는 메인 훈련 프로세스와 각 DataLoader 작업자 프로세스를 GPU의 로컬 NUMA 노드에 바인딩하여 크로스-NUMA 메모리 접근을 방지합니다. DataLoader 내에서 우리는 각 작업자 내부에서 미리 계산된 NUMA 바인딩을 재적용하는 클로저 기반 함수 `worker_init_fn` 를 전달합니다. 그리고 작업자 내에서 어떤 CUDA API도 건드리지 않고 이를 수행합니다.

시작 시 프로세스는 NVML을 사용하여 현재 GPU를 해당 NUMA 노드 및 CPU 친화성 마스크에 매핑합니다. 가능한 경우 `nvmlDeviceGetNUMANodeId` 를 통해 노드를 직접 읽습니다. 그렇지 않으면 GPU의 CPU 친화성 마스크 (`nvmlDeviceGetCpuAffinity`)에서 이를 추론합니다. NVML을 사용할 수 없거나 노드를 노출하지 않는 경우 `/sys/bus/pci/devices/<PCI_ID>/numa_node` 의 커널 sysfs 항목으로 대체합니다. 최후의 수단으로 프로세스의 현재 선호 노드를 사용합니다.

그런 다음 해당 노드의 CPU 목록을 `/sys/devices/system/node/node<N>/cpulist` 에서 계산하고, `psutil` 를 사용하여 해당 코어에 CPU 어피니티를 적용합니다. 또한 `libnuma` (`numa_run_on_node + numa_set_preferred`)를 사용하여 향후 모든 할당을 해당 노드에 바인딩합니다.

일부 런처, 컨테이너 런타임 또는 커널은 NUMA 정책을 자식 프로세스에 안정적으로 전파하지 못할 수 있으므로, 포크된 각 작업자에서 바인딩을 명시적으로 재적용하고 검증합니다. 상속에만 의존하는 것은 안전하지 않습니다.

`pin_memory=True` 를 설정하고 H2D 복사본에 `non_blocking=True` 를 사용하여 페이지 잠금 호스트 버퍼가 올바른 NUMA 노드에 유지되도록 하십시오. 작업자 재포크와 에포크 간 어피니티 손실을 방지하려면 `persistent_workers=True` 를 선호하십시오. 또한 `worker_init_fn` 에서 `torch.cuda.*` 를 호출하지 마십시오. 대신 클로저나 환경 변수를 통해 GPU 인덱스를 전달하십시오.

결과적으로 데이터 준비 및 배치 로딩이 완전히 로컬 메모리에서 수행됩니다. 이렇게 하면 GPU가 지속적으로 작업 상태를 유지하며 원격 NUMA 이동으로 인한 일시 정지가 필요하지 않습니다. 이 코드를 사용하면 `libnuma` 및 `numactl` 가 설치된 모든 Linux 서버에서 토폴로지 인식 어피니티를 견고하게 구현할 수 있습니다.

기본적으로 `numactl` 는 프로세스에 CPU 및 메모리 정책을 적용하며, 문서에 따르면 해당 정책이 모든 포크된 자식 프로세스에 상속됩니다. 그러나 실제로 Python 프레임워크에서 생성된 스레드나 exec로 실행된 서브프로세스는 모든 커널이나 Linux 배포판에서 동일한 설정을 항상 상속받지 않습니다. 프레임워크 관리형 작업자 프로세스를 사용할 때는 각 작업자 내부에서 CPU 및 메모리 정책을 명시적으로 재설정해야 합니다.

Grace Blackwell(및 Vera Rubin)과 같은 슈퍼칩 아키텍처에서는 CPU와 GPU가 NVLink-C2C를 통해 일관성을 유지합니다. 그러나 Linux는 여전히 CPU DRAM과 GPU HBM을 별개의 풀로 모델링합니다. 로컬 CPU NUMA 노드에 CPU 스레드를 바인딩하는 것은 여전히 지역성 측면에서 유리합니다.

실제 환경에서 핀 설정은 예측 불가능한 CPU 스케줄링 동작을 제거할 수 있습니다. 이는 GPU용 데이터 로딩 스레드와 같은 중요한 스레드가 훈련 또는 추론 도중 OS에 의해 갑자기 다른 NUMA 노드의 코어로 마이그레이션되는 것을 방지합니다. 실제 환경에서는 크로스-NUMA 트래픽과 CPU 코어 마이그레이션을 제거하는 것만으로도 5~10%의 훈련 처리량 향상을 확인할 수 있습니다. 이는 성능 지터와 변동성도 감소시키는 경향이 있습니다.

많은 고성능 AI 시스템 은 CPU 동시 멀티스레딩(SMT, 흔히 *하이퍼스레딩이라* 불림)을 평가하고, 예측 가능한 코어별 성능을 위해 이를 비활성화하기도 합니다. 다만 이 혜택은 워크로드에 따라 달라집니다. 이러한 시스템은 커널 매개변수 `isolcpus` 를 설정하여 일반 스케줄러로부터 격리함으로써 OS 백그라운드 작업 전용으로 소수의 코어를 예약하기도 합니다. 시스템 데몬을 위해 쿠버네티스 CPU 격리 기능을 사용할 수도 있습니다. 이를 통해 남은 코어는 훈련 및 추론 스레드와 유용한 작업에 전적으로 할당됩니다.

의 NVIDIA Grace Blackwell과 같은 통합 CPU-GPU 슈퍼칩의 경우, CPU와 GPU가 NVLink-C2C를 통해 일관된 공유 가상 주소 공간을 노출하는 반면 CPU DRAM과 GPU HBM은 별개의 메모리 풀로 유지되기 때문에 CPU-GPU 간 데이터 전송에 대한 기존 우려 사항 대부분이 완화된다는 점을 유의해야 합니다. 이는 크로스 NUMA 지연과 같은 문제가 최소화되고 데이터가 CPU와 GPU 간에 보다 직접적으로 흐를 수 있음을 의미합니다.

NVIDIA가 Grace Blackwell 아키텍처와 같은 단일 슈퍼칩에 CPU와 GPU를 통합하여 CPU-GPU 병목 현상을 해결한 것은 우연이 아닙니다. 이 설계에서는 CPU와 GPU가 NVLink-C2C를 통해 최대 900GB/s의 속도로 통합된 일관성 메모리를 공유하므로 데이터 전송 오버헤드가 최소화됩니다. NVIDIA가 소프트웨어 및 알고리즘 요구사항과 함께 설계된 이러한 유형의 하드웨어 혁신을 통해 시스템 병목 현상을 지속적으로 해결해 나갈 것으로 예상됩니다.

CPU-GPU 슈퍼칩 아키텍처가 밀접하게 결합되어 있더라도, 통합 시스템이 최고 효율로 작동하도록 하드웨어와 소프트웨어를 적절히 구성하여 스택을 최적화하

는 것은 여전히 중요합니다. 이러한 밀접하게 결합된 아키텍처에서도 GPU를 완전히 활용하기 위해 데이터 처리 시 불필요한 지연을 최소화해야 합니다. 여기에는 다음 섹션에서 살펴볼 것처럼, 휴지페이지 구성, 효율적 프리페칭 사용, 메모리 고정 등이 포함됩니다.

NUMA 친화적 메모리 할당 및 메모리 고정

기본적으로 프로세스는 현재 실행 중인 CPU의 NUMA 노드에서 로컬 메모리( )를 할당합니다. 따라서 프로세스를 NUMA 노드 0에 고정하면 해당 메모리는 자연스럽게 NUMA 노드 0의 로컬 RAM에서 제공되므로 이상적입니다. 그러나 OS 스케줄러가 스레드를 마이그레이션하거나 고정 전에 일부 메모리가 할당된 경우, NUMA 노드 0에서 실행 중인 프로세스가 NUMA 노드 1의 메모리를 사용하는 비이상적인 상황이 발생할 수 있습니다. 이 경우 모든 메모리 액세스가 다른 NUMA 노드로 이동해야 하므로 CPU 고정으로 인한 이점이 사라집니다.

이를 방지하기 위해, 앞서 언급한 바와 같이 ` numactl --membind ` 옵션은 ` `을 통해 특정 NUMA 노드에서 메모리를 할당하도록 강제합니다. 코드 수준에서는 이 구성을 제어할 수 있는 NUMA API나 환경 변수도 존재합니다. 일반적인 규칙은 메모리를 CPU 근처에, CPU는 GPU 근처에 배치하는 것입니다. 이렇게 하면 메모리에서 CPU를 거쳐 GPU로 이어지는 데이터 이동 경로가 모두 단일 NUMA 노드 내에서 이루어집니다. 다음은 이전 예제와 동일하지만, NUMA 노드 1을 포함하는 선호 NUMA 노드에서 메모리 할당을 강제하는 ` --membind=1 ` 옵션을 적용한 경우입니다:

```
numactl --cpunodebind=1 --membind=1 python train.py --gpu 5 &
```

`numactl` 환경에서 프로세스를 실행할 때 ` `를 사용하면 해당 프로세스에 CPU 정책(`--cpunodebind`)과 메모리 정책(`--membind`)이 모두 적용되며, 이 정책은 모든 자식 프로세스에 상속됩니다. 따라서 훈련 스크립트가 포크한 모든 작업자 하위 프로세스는 자동으로 동일한 NUMA 메모리 바인딩을 사용합니다. 단, 포크 기반 모델로 생성되어야 합니다. 스폰(spawn) 시작 방식으로 전환하거나, 다른 방식으로 새 프로그램을 생성(`exec`)하는 경우, 해당 자식 프로세스는 부모의 메모리 정책을 상속받지 않습니다.

또한 고정 메모리(*페이지 고정 메모리라고도 함*) 는 효율적이고 직접적인 GPU 접근에 필수적입니다. 메모리가 고정되면 OS는 해당 메모리를 스왑하거나 이동하지 않습니다. 이는 더 빠른 직접 메모리 접근(DMA) 전송으로 이어집니다. GPU 또는 NIC가 직접 DMA를 수행할 수 있기 때문에 고정된 호스트 메모리에서 GPU로 데이터를 복사하는 것은 일반 페이지 가능 메모리에서 복사하는 것보다 2~3배 빠를 수 있습니다.

설치된 CUDA 유틸리티의 ` bandwidthTest --memory=<pinned or pageable> ` 명령어를 사용하면 CPU 메모리와 GPU 메모리 간 데이터 전송 대역폭( )을 테스트할 수 있습니다.

사실 이는 NVIDIA의 GPUDirect 기술(예: GPUDirect RDMA)의 기반이 되며, 이를 통해 InfiniBand와 같은 NIC가 GPU 메모리와 직접 데이터를 교환할 수 있습니다. 마찬가지로 GPUDirect Storage(GDS)는 추가적인 CPU 오버헤드 없이 NVMe 드라이브가 GPU 메모리로 데이터를 스트리밍할 수 있게 합니다.

Deep Learning 프레임워크는 데이터 로더에 고정 메모리(pinned memory)를 사용할 수 있는 옵션을 제공합니다. 예를 들어, PyTorch의 `DataLoader` 에는 `pin_memory=True` 플래그가 있습니다. 이 플래그가 true로 설정되면 로드된 배치(batch)가 그림 3-5와 같이 고정된 RAM에 배치됩니다.

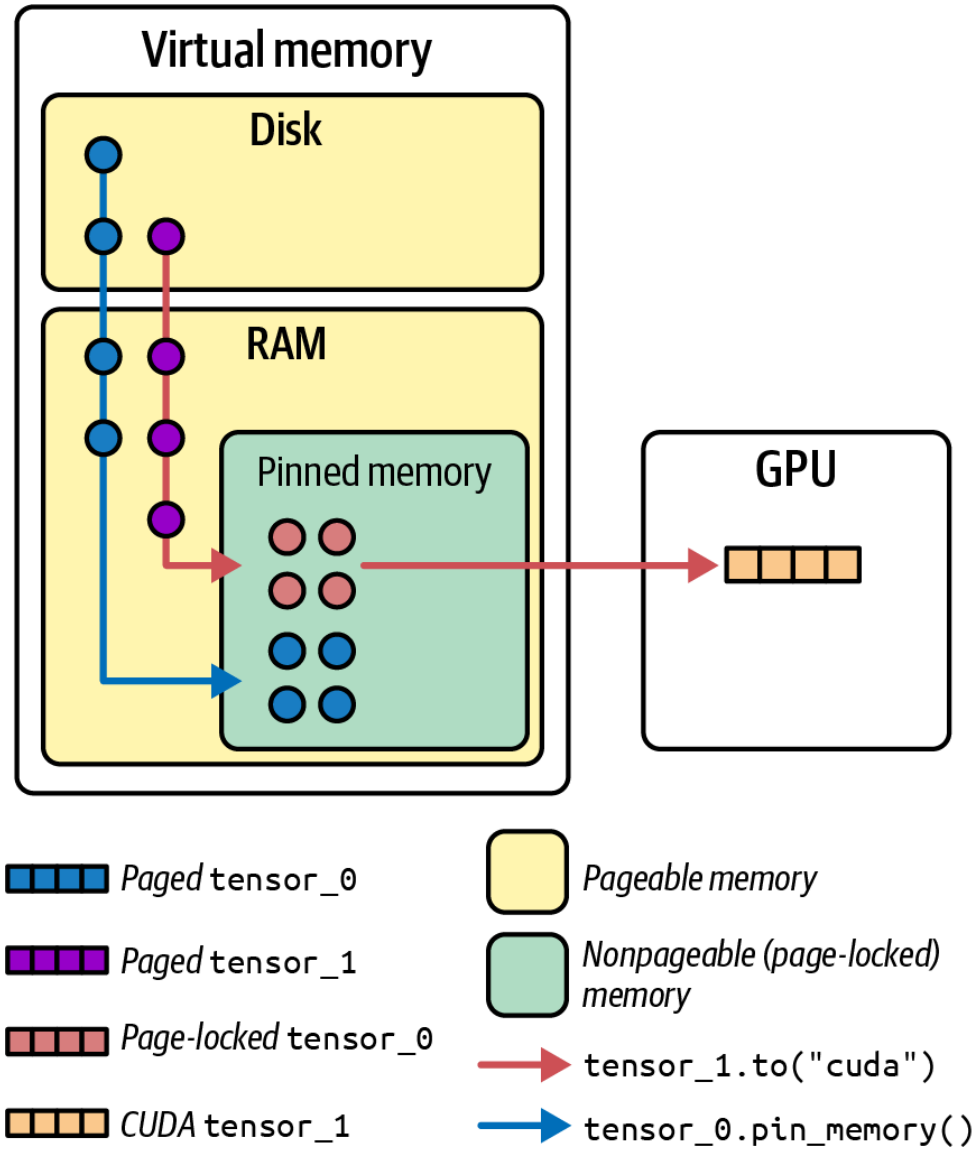

그림 3-5. 고정 메모리(페이지 잠금 또는 비페이지 가능 메모리라고도 함)는 디스크로 스왑 아웃될 수 없는 메모리 유형입니다

메모리 고정(memory pinning)은 `tensor.to(device)` 작업을 가속화합니다. CUDA 드라이버가 실시간으로 페이지를 고정할 필요가 없기 때문입니다. 특히 대량 배치 크기 사용 시나 각 반복에서 많은 데이터를 읽을 때 효과적입니다. 많은 실무자들은 PyTorch에서 `pin_memory=True` 만 활성화해도 데이터 전송 병목 현상을 줄이고 호스트-장치 간 전송 처리량을 증가시켜 성능을 10~20% 까지 향상시킬 수 있음을 확인했습니다.

요약하자면, 데이터 로더가 고정 메모리(예: PyTorch의 `pin_memory=True` `DataLoader`)를 사용하고, 지원되는 하드웨어에서 GPUDirect RDMA 및 GDS

가 활성화되었는지 확인해야 합니다. 이렇게 하면 데이터 전송 지연 시간이 줄어듭니다.

운영체제(OS)는 사용자가 고정(pin)할 수 있는 메모리 양에 제한이 있습니다. 이는 `` `ulimit -l <max locked memory>` `` 명령어로 설정됩니다. 컨테이너화된 환경에서는 컨테이너의 보안 컨텍스트(security context)와 Docker 설정을 적절히 조정할 수 있습니다. 이렇게 하면 컨테이너가 충분한 메모리를 확보할 수 있습니다. `--ulimit memlock` 설정을 적절히 조정할 수 있습니다. 이렇게 하면 컨테이너가 충분한 메모리를 고정할 수 있습니다.

대용량 고정 버퍼를 사용할 계획이라면 `ulimit` 값이 충분히 크거나 무제한으로 설정되어 있는지 확인하십시오. 그렇지 않으면 할당이 실패할 수 있습니다. 일반적으로 대규모 AI 워크로드 및 고성능 컴퓨팅(HPC) 애플리케이션에는 무제한으로 설정합니다.

투명 거대 페이지(Transparent Hugepages)

메모리 고정( ) 및 NUMA 노드 바인딩 외에도 투명 거대 페이지(THP)에 대해 논의해야 합니다. Linux 메모리 관리에서는 일반적으로 4KB 페이지를 사용하지만, 딥 러닝 데이터셋, 사전 가져온 배치, 모델 매개변수 등과 같이 수십 또는 수백 기가바이트의 메모리를 사용하는 프로세스의 경우 수백만 개의 작은 페이지를 관리하는 것은 비효율적입니다.

거대 페이지(2MB 또는 1GB 페이지)는 메모리 덩어리를 더 크게 만들어 가상 메모리 관리의 오버헤드를 줄일 수 있습니다. 주요 이점은 페이지 결함 감소와 번역 보조 버퍼(TLB)에 대한 부담 경감입니다.

TLB는 CPU가 가상 주소를 물리 주소로 매핑하는 데 사용하는 캐시입니다. 더 적고 큰 페이지는 동일한 수의 항목으로 TLB가 더 많은 메모리를 커버할 수 있게 하여 미스(miss)를 줄입니다.

대용량 페이지(Hugepages)는 일반적으로 약 3~5% 수준의 처리량 개선이라는 적당한 성능 향상을 가져옵니다. 이는 페이지 결함 오버헤드와 TLB 부담을 줄여 달성됩니다. 커널이 2MB 페이지로 대용량 할당을 자동으로 처리하므로 대부분의 시스템에서 THP 활성화는 간단한 이득입니다. 매우 큰 메모리 풀이 필요한 시나리오(예: I/O용 사전 할당된 고정 버퍼)에서는 보다 결정론적인 성능을 위해 `vm.nr_hugepages` 또는 `hugetlbfs` 를 사용한 명시적 대형 페이지 할당을 고려할 수도 있습니다.

로 대용량 고정 메모리 영역을 사용할 때는 `ulimit -l` 설정(최대 고정 메모리)을 높은 값으로 올리거나 `unlimited` 를 참조하십시오. 이 제한값이 너무 낮으면 메모리 고정 시도가 실패하여 스왑 가능 메모리로 대체되거나 메모리 부족(OOM) 오류가 발생할 수 있습니다.

THP의 백그라운드 압축은 예측 불가능한 일시 정지를 유발할 수 있으며, 이는 지연 시간에 민감한 LLM 추론 작업 부하에 치명적일 수 있다는 점을 유의해야 합니다. Linux는 기본적으로 가능한 경우 THP를 사용하여 2MB 페이지를 자동 할당하도록 구성됩니다. 이는 대부분의 경우 충분하지만, 작업 부하에 맞게 테스트해 볼 가치가 있습니다.

THP를 비활성화할 수 있지만, 이 경우 수동으로 hugepages를 할당하고 제어해야 합니다. 이는 추가적인 복잡성을 초래하지만, 추론과 같은 저지연 워크로드에는 필요할 수 있습니다. THP를 비활성화하면 커널 주도 조각 모음으로 인한 시스템 정체를 피할 수 있습니다.

현대적인 합의는 처리량이 중요한 대부분의 GPU 기반 훈련 워크로드에서는 THP를 활성화하고, 지연 시간이 중요한 추론과 같은 워크로드에서는 THP를 완전히 비활성화하거나 (`transparent_hugepage=never`) `madvise`를 사용하는 것입니다. 이는 많은 랭크(GPU)가 동시에 메모리를 할당하는 분산 훈련 워크로드에도 동일하게 적용됩니다.

CPU/메모리 고정 및 휴지 페이지 외에도 언급할 가치가 있는 몇 가지 OS 수준 조정 사항이 있습니다. 여기에는 스레드 스케줄링, 가상 메모리 관리, 파일 시스템 캐싱, CPU 주파수 설정이 포함되며, 다음 몇 섹션에서 다룰 예정입니다.

스케줄러 및 인터럽트 어피니티

바쁜 시스템에서는 데이터 파이프라인 스레드와 같은 중요한 스레드가 자주 중단되지 않도록 해야 합니다. Linux는 기본적으로 대부분의 경우에 잘 작동하는 완전 공정 스케줄러(CFS)를 사용합니다.

그러나 예를 들어 GPU에 데이터를 공급하는 지연 시간에 매우 민감한 스레드가 있다면, 해당 스레드에 대해 실시간 선입선출(FIFO) 또는 라운드 로빈(RR) 우선순위 스케줄링을 사용하는 것을 고려할 수 있습니다. 이렇게 하면 높은 우선순위 스레드가 일반 우선순위 스레드에 의해 선점되지 않고 실행되도록 보장할 수 있습니다.

다만, 실시간 스레드는 관리가 부실할 경우 다른 프로세스의 자원을 고갈시킬 수 있으므로 주의해서 사용해야 합니다. 실제로 스레드를 전용 코어에 고정(pinned)한 경우 실시간 스레드 우선순위를 조정할 필요가 거의 없지만, 계속 주시하는 것이 좋습니다.

또 다른 방법은 코어를 격리하거나 별도의 CPU 파티션을 생성하여 이러한 전용 컴퓨팅 리소스에 대한 간섭을 더욱 줄이는 것입니다. 이를 위해 `cset`를 사용하거나, 커널 매개변수(예: `isolcpus` 및 `nohz_full`)를 설정하거나, cgroup `cpuset` 격리 기능을 활용할 수 있습니다. 격리 시 운영체제 스케줄러는 해당 CPU 코어를 사용자가 원하는 대로 사용할 수 있도록 남겨둡니다.

프로덕션 환경에서는 cgroup CPU 및 메모리 어피니티 설정을 강력히 권장합니다. 이를 통해 각 AI 워크로드는 자체 물리 코어 및 메모리 영역에 격리됩니다. 이는 워크로드 간 경합과 NUMA 페널티를 방지합니다. `cpuset` cgroups 또는 컨테이너 런타임(`docker --cpuset-cpus`)과 같은 도구를 사용하여 이를 강제 적용해야 합니다.

각 장치의 하드웨어 인터럽트를 동일한 NUMA 노드의 코어에 할당할 수 있습니다. 이렇게 하면 노드 간 인터럽트 처리가 방지되어 추가 지연 시간이 발생하거나 원격 노드에서 유용한 캐시 라인이 제거되는 것을 막을 수 있습니다. 예를 들어, NUMA 노드 0에 있는 GPU 또는 NIC가 인터럽트를 발생시키면, 다른 노드가 이를 처리하지 않도록 노드 0의 코어에 바인딩해야 합니다. 이 바인딩이 없으면 다른 NUMA 노드의 CPU가 인터럽트를 처리할 수 있습니다. 이 경우 캐시 일관성 트래픽과 노드 간 통신이 강제 발생합니다.

실제 성능에 민감한 시스템 에서는 기본 인터럽트 처리 데몬(`irqbalance`)을 비활성화하거나 맞춤형 규칙으로 실행하는 경우가 많습니다. 다른 방법은 `/proc/irq/*/smp_affinity`를 사용하여 각 인터럽트의 어피니티 마스크를 수동으로 설정하는 것입니다. 모든 GPU 및 NIC 인터럽트를 가장 가까운 코어에 고정함으로써 해당 장치 인터럽트가 항상 최적의 NUMA 노드에서 처리되도록 보장할 수 있습니다.

요약하면, 전용 코어, 적절한 스케줄링 우선순위, NUMA 인식 하드웨어 인터럽트 바인딩의 조합은 GPU에 데이터를 공급하는 데이터 로딩 스레드의 지터를 최소화하는 데 도움이 될 수 있습니다.

가상 메모리 및 스와핑

말할 필요도 없지만, 메모리 스왑을 항상 피해야 합니다( ). 프로세스 메모리의 일부라도 디스크로 스왑되면 치명적인, 여러 차원의 속도 저하가 발생합니다. GPU 프로그램은 데이터 캐싱을 위해 호스트 메모리를 많이 할당하는 경향이 있습니다. OS가 일부 데이터를 메모리에서 디스크로 스왑하기로 결정하면, GPU가 해당 데이터에 접근해야 할 때 큰 지연이 발생합니다.

`vm.swappiness=0` 를 설정할 것을 권장합니다. 이는 에서 설명하듯 극심한 메모리 압박이 없는 한 Linux가 스왑을 피하도록 지시합니다. cgroup 제한을 통해 훈련 작업의 메모리를 효과적으로 격리하여 스왑을 방지합니다.

Docker 또는 쿠버네티스를 통해 cgroups v2를 사용하여 메모리와 CPU를 AI 프로세스에 고정해야 합니다. 이렇게 하면 컨테이너화된 환경에서 NUMA 어피니티 및 노스왑 정책이 강제 적용됩니다.

`sudo swapoff -a` 를 사용해 다음 재부팅까지 모든 스왑 장치와 파일을 일시적으로 비활성화할 수도 있습니다. 작업 부하에 충분한 RAM이 있는지 확인하거나 오버커밋을 방지하기 위한 제한을 설정하세요. 그렇지 않으면 OOM 킬러가

프로세스를 종료할 수 있습니다. `vmstat` 또는 `free -m` 로 스왑 사용량을 모니터링하여 스왑이 0으로 유지되는지 확인하세요.

앞서 고정 메모리(pinned memory)와 관련해 언급한 바와 같이, 의 또 다른 관련 설정은 `ulimit -l` 입니다. 메모리가 스왑되는 것을 방지하려면 이 한도를 높게 설정해야 하며, 그렇지 않으면 과도한 메모리 스왑이 발생할 수 있습니다. 일반적으로 메모리를 많이 사용하는 대규모 AI 작업 부하에는 이 한도를 무제한으로 설정합니다.

파일시스템 캐싱 및 쓰기 백업

대규모 훈련 작업 의 모범 사례는 실패한 작업을 알려진 정상 체크포인트에서 재시작해야 할 경우를 대비해 디스크에 자주 체크포인트를 기록하는 것입니다. 그러나 체크포인트 기록 중에는 방대한 데이터가 OS 페이지 캐시를 채워 작업이 중단될 수 있습니다.

저장을 위해 `vm.dirty_ratio` 및 `vm.dirty_background_ratio` 를 조정하여 쓰기 버퍼링용 페이지 캐시 크기를 튜닝할 수 있습니다. 예를 들어, 멀티 GB 규모의 체크포인트의 경우 더 높은 더티 비율을 사용하면 OS가 디스크로 플러시하기 전에 RAM에 더 많은 데이터를 배치할 수 있습니다. 이는 대규모 체크포인트 쓰기를 원활하게 하고 훈련 루프의 정체를 줄여줍니다.

또 다른 방법은 별도의 스레드에서 체크포인트를 수행하는 것입니다. PyTorch의 최신 옵션으로는 클러스터 내 노드에서 분산 체크포인트 파티션을 작성하는 방법이 있습니다. 이 경우 작업 실패 후 재시작 시 체크포인트를 로드할 때 파티션이 결합됩니다.

지연 시간에 민감한 훈련 워크플로우에서는 페이지 캐시를 완전히 우회하는 것이 가장 좋습니다. 예를 들어, `O_DIRECT` 명령어로 체크포인트 파일을 열거나 비동기 I/O를 위해 Linux의 `io_uring` 를 사용하여 페이지 캐시 정체를 방지하세요. 각 체크포인트 작성 후에는 `posix_fadvise(fd, 0, 0, POSIX_FADV_DONTNEED)` 를 호출하여 해당 페이지를 캐시에서 즉시 제거하고 후속 반복 작업 시 메모리 압박을 방지하세요.

CPU 주파수 및 C-상태

기본적으로 많은 컴퓨팅 노드 는 CPU를 절전 모드로 실행합니다. 이 모드는 CPU가 유휴 상태일 때 클럭 속도를 낮추거나 절전 상태로 전환합니다. 이는 에너지 절약, 발열 감소 및 비용 절감에 도움이 됩니다. 모델 훈련 중에는 GPU가 데이터셋의 마지막 배치 작업을 처리하는 동안 CPU가 항상 100% 활용되지는 않을 수 있습니다. 그러나 이러한 전력 관리 기능은 새로운 작업이 도착하여 시스템이 CPU를 다시 깨울 때 추가 지연 시간을 유발할 수 있습니다.

최대 및 일관된 성능을 위해 AI 시스템은 종종 CPU 주파수 거버너를 "성능" 모드로 설정합니다. 이 모드는 CPU를 항상 최대 주파수로 유지합니다. 이는 시스템

관리 도구(`cpupower frequency-set -g performance`) 또는 기본 입출력 시스템(BIOS)에서 설정할 수 있습니다.

마찬가지로, 깊은 C-스테이트를 비활성화하면 코어가 저전력 절전 상태로 진입하는 것을 방지할 수 있습니다. CPU C-스테이트는 시스템의 ACPI 사양에 정의된 절전 모드입니다. CPU 코어가 유휴 상태일 때 에너지를 절약하기 위해 C-스테이트로 진입할 수 있습니다. C-스테이트가 깊을수록 더 많은 전력을 절약하지만 작업이 도착했을 때 코어가 깨어나는 데 더 오랜 시간이 걸릴 수 있습니다. 더 깊은 C-스테이트를 비활성화하면 과도한 지연 시간 급증을 제거할 수 있습니다. C0은 활성 상태이며, C0 이상의 모든 상태는 더 깊은 절전 상태를 나타냅니다.

실제 환경에서는 많은 서버 BIOS/UEFI(통합 확장 펌웨어 인터페이스)가 고성능 자질을 제공하며, 이는 CPU 거버너를 자동으로 '성능' 모드로 설정하고 깊은 C-상태를 비활성화합니다.

본질적으로, 우리는 약간 더 많은 전력 소모를 감수하는 대가로 더 반응성이 뛰어난 CPU 동작을 얻을 수 있습니다. GPU가 주요 전력 소비자인 훈련 시나리오에서는, GPU에 데이터를 공급하는 데 문제가 없다면 CPU 전력 사용량이 약간 증가하는 것은 일반적으로 괜찮습니다. 예를 들어, 데이터 로더 스레드가 데이터를 기다리며 슬립 상태에 들어가고 CPU가 깊은 C6 상태로 진입하면, 에너지 절약을 극대화하기 위해 CPU의 상당 부분이 전원이 차단됩니다.

CPU가 더 깊은 절전 상태로 진입하면 깨어나는데 몇 마이크로초가 소요될 수 있습니다. 이는 긴 시간은 아니지만, 여러 마이크로초가 누적되면 적절히 관리하지 않을 경우 GPU 버블을 유발할 수 있습니다. *버블*은 GPU가 CPU가 데이터 처리를 재개하기를 기다리는 시간대를 의미합니다. CPU를 대기 상태로 유지함으로써 이러한 장애를 줄일 수 있습니다. 많은 서버용 BIOS에는 C-상태를 비활성화하거나 최소한 제한하는 설정이 있습니다.

예상치 못한 지연 시간을 유발할 수 있는 시스템 요소(과도한 컨텍스트 스위칭, CPU 주파수 스케일링, 메모리-디스크 스왑 등)는 항상 비활성화해야 합니다. 이를 통해 OS가 잘못된 코어에 작업을 스케줄링하거나 부적절한 시점에 CPU 사이클을 빼앗지 않도록 하여, CPU가 GPU가 소비할 수 있는 속도만큼 빠르게 데이터를 GPU에 전달할 수 있습니다.

호스트 CPU 메모리 할당기 조정

잘 조정된 GPU 서버에서는 GPU가 대부분의 계산을 처리하므로 CPU 사용률이 매우 높지 않을 수 있습니다. 그러나 CPU 사용률은 안정적으로 유지되며 GPU 활동과 동기화되어야 합니다. CPU는 현재 배치 작업이 GPU에서 처리되는 동안 들어오는 각 배치 작업을 준비하느라 바쁘게 유지되어야 합니다.

높은 GPU 활용도를 유지하려면 적절한 CPU-GPU 작업 인계가 필수적입니다. 호스트의 메모리 할당기(`jemalloc` 또는 `tcmalloc`)를 조정하면 데이터 준비

과정에서 발생하는 예측 불가능한 일시 정지를 제거할 수 있습니다. 이를 통해 의도적인 동기화 지점을 제외하고 GPU가 최대 성능으로 작동하도록 유지됩니다.

튜닝 후 각 GPU 사용률이 100% 근처에서 유지되며 필수 동기화 장벽에서만 하락하는 것을 확인할 수 있습니다. CPU 측 지연으로 인해 GPU가 데이터 대기 상태에 빠지는 일은 없어야 합니다. `jemalloc` 를 사용하면 할당을 CPU별 영역(`narenas`)으로 분할하고, 오프패스 정리(`background_thread`)를 활성화하며, 해제된 페이지가 즉시 OS로 반환되지 않도록 `dirty_decay_ms`/`muzzy_decay_ms` 를 연장할 수 있습니다. 이는 잠금 경쟁과 조각화를 최소화합니다.

`jemalloc` 는 `MALLOC_CONF` 환경 변수를 다음과 같이 조정할 수 있습니다:

```
export MALLOC_CONF="narenas:8,dirty_decay_ms:10000,muzzy_decay_ms:10000
,background_thread:true"
```

마찬가지로 `tcmalloc` 는 `TCMALLOC_MAX_TOTAL_THREAD_CACHE_BYTES` 및 `TCMALLOC_RELEASE_RATE` 환경 변수 조정을 통해 성능이 향상됩니다. 이 설정은 스레드별 캐시를 확대하여 소규모 할당 시 전역 잠금 및 시스템 호출을 회피하게 합니다. 이를 통해 CPU 스레드가 낮은 예측 가능한 지연 시간으로 GPU에 데이터를 공급할 준비 상태를 유지합니다. 설정 방법은 다음과 같습니다:

```
export TCMALLOC_MAX_TOTAL_THREAD_CACHE_BYTES=$((512*1024*1024))
export TCMALLOC_RELEASE_RATE=16
```

요약하면, 할당기 최적화는 할당기 오버헤드와 조각화를 줄일 수 있습니다. 이는 CPU 스레드의 일관된 속도를 유지하고 GPU 공급 시 예상치 못한 정체를 방지합니다. 특정 워크로드와 환경에 맞게 이러한 환경 변수를 실험하고 조정하십시오.

성능 향상을 위한 GPU 드라이버 및 런타임 설정

CPU 측면은 최적화했지만( ), 특히 다중 GPU 및 다중 사용자 시나리오에서 성능에 영향을 미칠 수 있는 GPU 드라이버 및 런타임 설정도 중요합니다. NVIDIA GPU에는 적절히 조정하면 오버헤드를 줄이고 여러 워크로드가 GPU를 공유하는 방식을 개선할 수 있는 몇 가지 조정 항목이 있습니다.

다음으로 GPU 지속성 모드, MPS 파티션, MIG 및 클럭 설정, ECC 메모리, 메모리 부족 시 동작과 같은 몇 가지 고려 사항을 다룰 것입니다.

GPU 지속성 모드

기본적으로 애플리케이션이 GPU를 사용하지 않을 경우( ), 드라이버는 GPU를 저전력 상태로 전환하고 일부 드라이버 컨텍스트를 언로드할 수 있습니다. 이후

애플리케이션이 GPU를 사용하려 할 때 초기화 비용이 발생합니다. 드라이버가 모든 구성 요소를 가동하는 데 1~2초 정도의 시간이 소요될 수 있습니다.

GPU 초기화 오버헤드는 GPU를 주기적으로 해제하고 재획득하는 워크로드의 성능에 부정적인 영향을 미칠 수 있습니다. 예를 들어, 작업이 빈번하게 시작 및 중지되는 훈련 클러스터나 새로운 추론 요청이 도착할 때마다 GPU를 깨워야 하는 저용량 추론 클러스터를 생각해 보십시오. 두 경우 모두 오버헤드로 인해 전체 워크로드 성능이 저하됩니다.

지속성 모드는 `nvidia-persistenced` 데몬을 실행하여 활성화됩니다. ` ` 이 명령은 애플리케이션이 비활성 상태일 때도 GPU 드라이버를 로드된 상태로 유지하고 하드웨어를 준비 상태로 유지합니다. 이는 시스템이 유휴 상태에서 GPU를 완전히 전원 차단하지 않도록 요청하여 전원 게이트를 방지합니다. 지속성은 GPU를 깨어 있게 유지하여 다음 작업의 시작 지연을 제로로 만듭니다. 이는 일반적으로 장시간 실행되는 작업이나 지연 시간에 민감한 워크로드에 권장됩니다. 다음 명령을 사용하여 부팅 시 지속성 데몬을 활성화할 수 있습니다:

```
systemctl enable nvidia-persistenced
```

쿠버네티스 환경에서는 NVIDIA GPU Operator를 구성하여 모든 GPU에 대해 자동으로 지속성 모드를 활성화할 수 있습니다.

AI 클러스터에서는 서버 부팅 시 모든 GPU에 지속성 모드를 활성화하는 것이 일반적입니다. 이렇게 하면 작업이 시작될 때 GPU가 이미 초기화되어 즉시 처리를 시작할 수 있습니다. 수학 연산 속도를 높이지 않으므로 실제 컴퓨팅 속도는 빨라지지 않지만, 작업 시작 지연 시간을 줄이고 콜드 스타트 지연을 방지합니다.

GPU 지속성 모드는 대화형 사용에도 도움이 됩니다. 지속성이 없으면 유휴 시간 후 첫 번째 CUDA 호출이 드라이버가 GPU를 재초기화하는 동안 지연될 수 있습니다. 지속성이 활성화되면 해당 호출이 빠르게 반환됩니다.

지속성 의 유일한 단점은 GPU가 더 높은 준비 상태를 유지하기 때문에 유휴 시 전력 소모가 약간 증가한다는 점입니다. 그러나 대부분의 데이터 센터 GPU에서는 성능 일관성 향상을 위한 수용 가능한 타협점입니다. 관리자가 시스템 관리 권한(`sudo`)으로 GPU 지속성 모드를 설정하면 이점을 누리고 다른 최적화 작업으로 넘어갈 수 있습니다.

MPS

일반적으로 여러 프로세스 가 단일 GPU를 공유할 때 GPU 스케줄러는 이들 프로세스 간에 시간을 분할합니다. 예를 들어, 두 Python 프로세스가 각각 동일한 GPU에서 실행할 커널을 가지고 있다면, GPU는 한 프로세스의 커널을 실행한 후 다른 프로세스의 커널을 실행하는 식으로 번갈아 가며 처리할 수 있습니다. 해당

커널들이 짧고 그 사이에 유휴 시간이 존재한다면, GPU는 작업이 겹치지 않고 "핑퐁"처럼 컨텍스트 전환을 반복하며 활용도가 떨어질 수 있습니다.

NVIDIA의 MPS는 여러 프로세스가 엄격한 시간 분할 없이 GPU에서 동시에 실행될 수 있도록 하는 일종의 통합 기능을 제공합니다. MPS를 사용하면 GPU 리소스(스트리밍 멀티프로세서[SM], 텐서 코어 등)가 사용 가능한 한, 서로 다른 프로세스의 커널을 동시에 실행할 수 있습니다. MPS는 본질적으로 프로세스들의 컨텍스트를 하나의 스케줄러 컨텍스트로 통합합니다. 이를 통해 독립적인 프로세스 간 전환 및 유휴 상태로 인한 전체 비용을 지불하지 않아도 됩니다.

MPS는 언제 유용할까요? 모델 훈련 시 일반적으로 GPU당 하나의 프로세스를 실행한다면 MPS를 사용하지 않을 수 있습니다. 그러나 하나의 대형 GPU에서 다수의 추론 작업을 실행하는 시나리오에서는 MPS가 판도를 바꿉니다. 강력한 GPU 또는 GPU 클러스터를 보유하고 있지만, 추론 작업(또는 다수의 추론 작업 집합)이 이를 완전히 활용하지 못하는 상황을 상상해 보십시오. 예를 들어, 40GB GPU 하나에서 각각 5~10GB 메모리를 사용하고 GPU 컴퓨팅의 30%만 활용하는 네 개의 별도 추론 작업을 실행한다고 가정해 보십시오. 기본적으로 각 추론 작업은 시간 슬라이스를 할당받으므로, 어느 순간에도 단 하나의 작업만 GPU에서 실제로 실행됩니다. 이로 인해 GPU는 평균 70%의 유휴 상태를 유지하게 됩니다.

이러한 추론 작업에 MPS를 활성화하면 GPU가 작업을 교차 실행할 수 있습니다. 즉, 한 작업이 메모리를 대기하는 동안 다른 작업의 커널이 GPU를 채우는 식입니다. 결과적으로 전체 GPU 활용도가 높아집니다. 실제로 두 프로세스가 각각 GPU의 40%를 사용하는 경우, MPS를 적용하면 두 작업을 모두 처리하면서 GPU 활용도가 80~90%에 달할 수 있습니다.

예를 들어, 동일한 GPU에서 순차적으로 실행될 경우 각각 1시간이 소요되는 두 개의 훈련 프로세스는 MPS를 통해 병렬로 함께 실행될 수 있으며, 순차적으로 2시간이 소요되는 대신 총 1시간 조금 넘게 소요됩니다. 예를 들어, 동일한 GPU에서 순차적으로 실행될 경우 각각 1시간이 소요되는 두 개의 훈련 프로세스는 MPS를 통해 함께 실행될 수 있습니다. 이 경우, 두 프로세스는 순차적으로 2시간이 아닌 병렬로 총 1시간 조금 넘게 소요되어 완료됩니다. MPS의 가속 효과는 커널과 동시 클라이언트의 메모리 대역폭이 서로 보완될 때 거의 두 배에 가까운 수준에 이를 수 있습니다. 이를 시각화하기 위해, MPS 없이 프로세스 A와 프로세스 B가 각각 주기적으로 커널을 실행한다고 가정해 보십시오. GPU 스케줄은 그림 3-6과 같이 각 프로세스가 대기하는 간격이 있는 A-B-A-B 형태로 보일 수 있습니다.

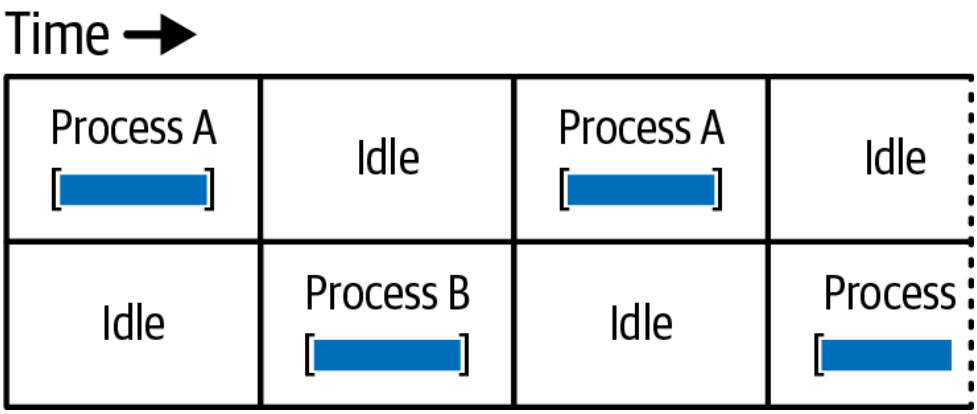

그림 3-6. GPU는 프로세스 A의 커널과 프로세스 B의 커널을 번갈아 실행하며, 한 프로세스가 대기하는 동안 다른 프로세스가 활성 상태인 유휴 간격을 생성함

MPS를 사용하면 스케줄이 A와 B를 중첩시켜 A가 GPU 일부를 사용하지 않을 때마다 B의 작업이 이를 동시에 활용할 수 있게 합니다. 반대의 경우도 마찬가지입니다. 이러한 중첩은 그림 3-7과 같이 유휴 간격을 제거합니다.

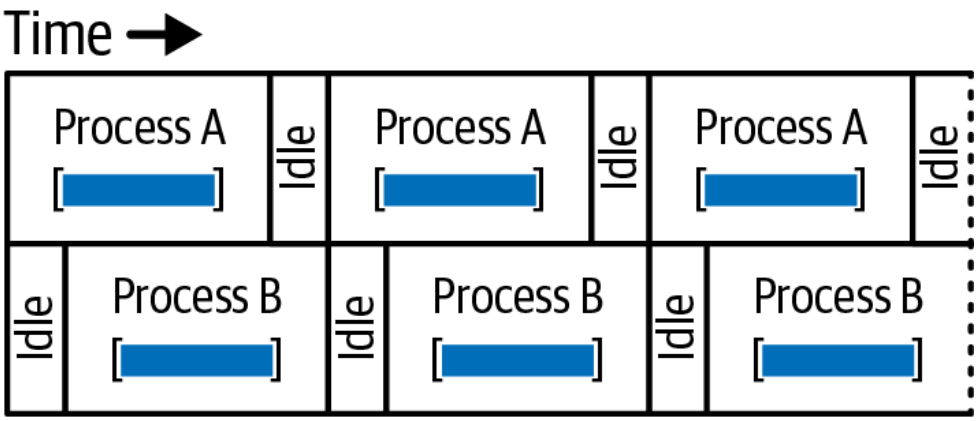

그림 3-7. MPS를 사용하여 프로세스 A와 B의 유휴 간격감소

MPS 설정은 MPS 제어 데몬(`nvidia-cuda-mps-control`)을 실행하는 것으로 시작되며, 이 데몬은 GPU 접근을 중개하는 MPS 서버 프로세스를 시작합니다. 최신 GPU에서는 클라이언트(프로세스)가 컴퓨팅 노드 자체의 최소한의 간섭으로 하드웨어와 직접 통신할 수 있어 MPS가 더욱 간소화되었습니다.

일반적으로 노드에서 MPS 서버를 시작합니다(GPU당 하나 또는 사용자당 하나). 그런 다음 MPS에 연결하는 환경 변수를 사용하여 GPU 작업을 실행합니다. 해당 서버 아래의 모든 작업은 GPU를 동시에 공유합니다.

MPS의 또 다른 기능은 클라이언트별 활성 스레드 비율을 설정할 수 있다는 점입니다. 이는 클라이언트가 사용할 수 있는 SM(기본적으로 GPU 코어) 수를 제한합니다. 예를 들어 두 작업이 각각 GPU 실행 리소스의 최대 50%만 사용하도록 서비스 품질(QoS)을 보장하려는 경우 유용합니다. 이 경우 `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE=50`를 설정하여 클라이언트의 SM 실행 용량을 약 50%로 제한할 수 있습니다. 명시적으로 설정하지 않으면 작업들은 경쟁하여 가능한 모든 GPU 자원을 사용하게 됩니다.

MPS는 GPU 메모리를 분할하지 않으므로 모든 프로세스가 전체 GPU 메모리 공간을 공유합니다. MPS는 주로 컴퓨팅 공유 및 스케줄링에 관한 것입니다. 문제는 한 프로세스가 GPU RAM을 대량으로 요청하여 GPU에서 OOM(Out of

Memory) 오류를 발생시키고, GPU에서 실행 중인 다른 모든 프로세스를 종료시킬 수 있다는 점입니다. 이는 매우 방해가 됩니다. 또한, 하나의 프로그램이 단독으로 GPU를 100% 포화 상태로 만들 경우, MPS가 마법처럼 속도를 높여주지는 않습니다. 100% 활용률을 초과할 수 없기 때문입니다. 개별 작업들이 다른 작업들이 채울 수 있는 여유 공간을 남길 때만 MPS는 이점을 제공합니다.

MPS의 또 다른 한계는 기본적으로 모든 MPS 클라이언트가 동일한 유닉스 사용자 계정으로 실행되어야 한다는 점입니다. 컨텍스트를 공유하기 때문입니다. 다중 사용자 클러스터에서는 MPS가 일반적으로 스케줄러 수준에서 설정되어 한 번에 한 사용자의 작업만 GPU를 공유하도록 합니다. 그렇지 않으면 모든 사용자가 공유하는 시스템 전체 MPS를 구성할 수 있지만, 보안 관점에서 작업이 격리되지 않음을 이해해야 합니다.

최신 NVIDIA 드라이버는 다중 사용자 MPS를 지원하여 서로 다른 Unix 사용자의 프로세스가 단일 MPS 서버를 공유할 수 있습니다. 이는 사용성을 향상시키지만 메모리 격리는 제공하지 않습니다. 강력한 격리가 필요한 경우 MIG를 선호하십시오. MPS의 구체적인 대안 중 하나는 쿠버네티스의 GPU 시간 분할 기능입니다. 쿠버네티스의 시간 분할은 장치 플러그인이 동일한 GPU에 서로 다른 포드를 시간별로 스케줄링할 수 있게 합니다. 예를 들어, 단일 GPU에 시간 분할 복제 계수 4를 구성하면 해당 GPU의 네 개의 포드가 각각 시간 할당을 받을 수 있습니다.

쿠버네티스 타임 슬라이싱은 MPS 없이 자동화된 시간 공유 알고리즘과 유사합니다. 다만 실행이 중첩되지는 않으며, 기본 드라이버보다 더 빠르게 전환할 뿐입니다. 시간 분할은 일부 유휴 시간을 감수하면서 격리를 선호하는 대화형 워크로드에 유용할 수 있습니다. 고처리량 작업의 경우, 다음에서 논의하듯이 MPS와의 중첩 또는 MIG를 통한 GPU 분할이 세분화된 시간 분할보다 일반적으로 더 효과적입니다.

MIG

현대 GPU는 MIG를 통해 하드웨어 수준 에서 여러 인스턴스로 분할할 수 있습니다. MIG는 가상화 기술의 일종이지만 하드웨어에서 구현됩니다. 이 방식은 유연성이 다소 떨어지지만 오버헤드가 매우 낮습니다(몇 퍼센트 수준).

하드웨어적으로 고정 분할되어 있으므로 한 인스턴스가 유휴 상태일 때 다른 인스턴스에 자원을 빌려줄 수 없습니다. MIG는 GPU를 최대 7개의 작은 논리적 GPU로 분할할 수 있게 하며, 각 논리적 GPU는 그림 3-8과 같이 전용 메모리 영역과 컴퓨트 유닛(SM)을 갖습니다.

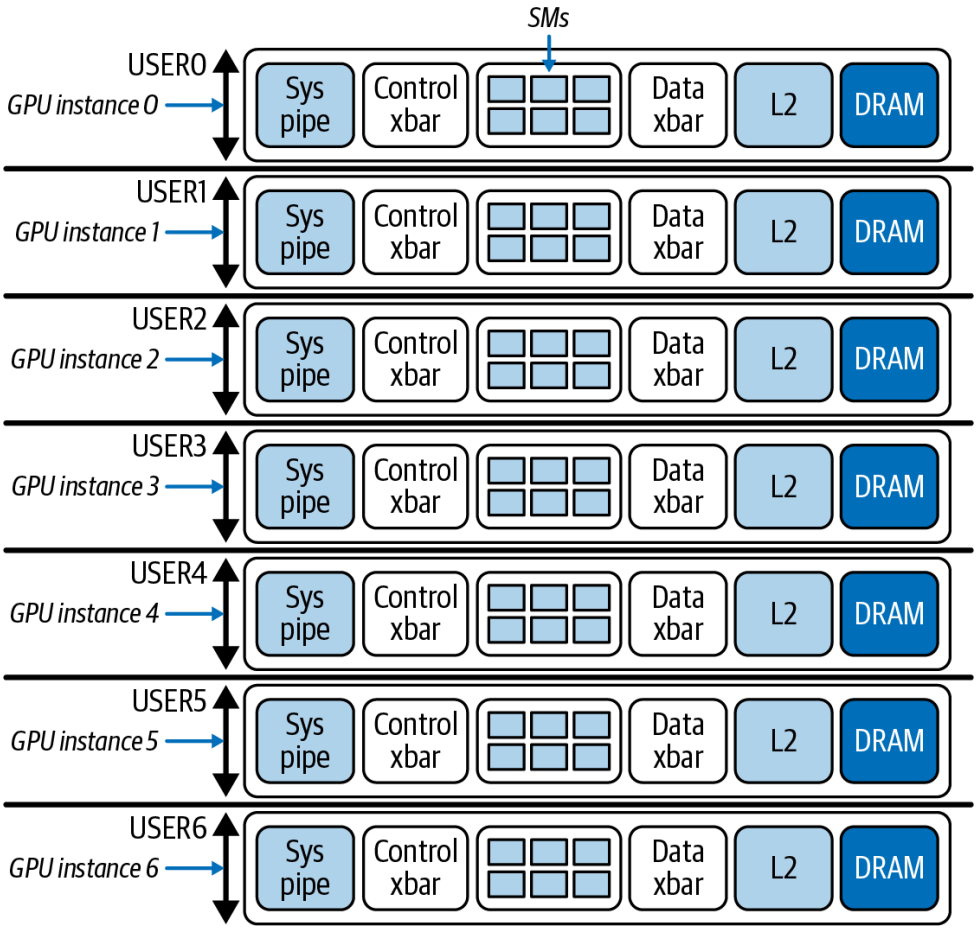

그림 3-8. 최신 GPU의7개 MIG 슬라이스

관례에 따라 NVIDIA의 MIG 자질 명명법 은 접두사 `<X>g to` 를 사용하여 최신 GPU에서 컴퓨트 슬라이스 수(최소 1개, 최대 7개)를 나타냅니다. 각 슬라이스 번호는 해당 파티션에 할당된 SM 그룹 수를 의미합니다. 각 SM 그룹은 전체 SM 수의 약 1/7에 해당하는 슬라이스입니다.

GPU에 132개의 SM이 있다면, 각 1/7 슬라이스는 그룹당 132 SM × 1/7 = 약 19개의 SM을 의미합니다. 따라서 `1g` 은 약 19개의 SM, `2g` 은 약 38개의 SM을 나타내며, 7g까지는 총 약 132개의 SM을 의미합니다.

반면 다소 혼란스럽게도 접미사 `<Y>gb` 는 해당 프로파일에 할당된 HBM GPU RAM의 정확한 용량(기가바이트 단위)을 지정합니다. MIG 자질 값은 각 GPU 세대 및 유형별로 고정되어 있으며 NVIDIA 문서에 명시되어 있습니다. Blackwell B200의 경우 일부 MIG 자질 값이 표 3-1에 나와 있습니다.

표 3-1. Blackwell B200용 MIG 자질 (출처: https://oreil.ly/FsPEx)

자질 이름	메모리 비율	SM 비율	L2 캐시 크기	복사 엔진	인스턴스 수
MIG 1g.23gb	1/8	1/7	1/8	2	7
MIG 1g.45gb	2/8	1/7	2/8	2	4
MIG 2g.45gb	2/8	2/7	2/8	3	3
MIG 3g.90gb	4/8	3/7	4/8	6	2
MIG 4g.90gb	4/8	4/7	4/8	8	1
MIG 7g.180gb	전체	전체	전체	16	1

이 표에는 각 자질별 하드웨어 유닛 수, 복사 엔진 수, L2 캐시 분할 비율도 표시됩니다. 이러한 고정 자질은 GPU의 하드웨어 메모리 컨트롤러와 정렬되어 각 메모리 슬라이스가 연속적인 HBM 채널에 매핑되도록 합니다.

이 2단계 방식은 컴퓨팅 용량(SM 그룹 수)과 메모리 용량(총 GB)을 분리합니다. 관리자는 MIG 자질 조합을 선택할 수 있습니다. 할당된 SM과 HBM의 합계가 GPU 전체 용량과 정확히 일치할 필요는 없습니다. 그러나 특정 조합은 하드웨어 분할에 의해 제약받습니다. 예를 들어 새로운 슬라이스 크기를 임의로 생성할 수 없습니다.

관리자는 `nvidia-smi -mig` 같은 도구나 NVIDIA 쿠버네티스 GPU Operator의 `nvidia.com/mig.config` config map을 사용하여 각 GPU에서 지원되는 MIG 자질(예: `1g.23gb`, `2g.45gb`, `4g.90gb` 등)만 활성화하거나 비활성화할 수 있습니다. MIG 재구성을 위해서는 워크로드를 비우고 MIG의 동적 재구성 기능을 호출하여 변경 사항을 적용해야 합니다.

GPU가 MIG 모드에 진입하면 최신 GPU는 시스템 전체 재부팅 없이 MIG 파티션을 동적으로 생성 및 삭제할 수 있습니다. 기존 워크로드를 드레인한 후 MIG 인스턴스를 실시간으로 조정할 수 있지만, GPU 자체에서 MIG 모드를 활성화하거나 비활성화하려면 해당 GPU를 재설정해야 합니다.

각 MIG 인스턴스는 자체 메모리, 자체 SM(스트림 프로세서), 심지어 별도의 엔진 컨텍스트를 보유하므로 소프트웨어 관점에서 독립된 GPU처럼 작동합니다. MIG의 장점은 강력한 격리성과 각 작업에 대한 보장된 리소스입니다. 예를 들어 각각 10GB의 논리적 GPU 메모리만 필요한 다중 사용자나 다중 서비스가 있다면, 서로의 메모리나 컴퓨팅 자원을 방해하지 않고 하나의 물리적 GPU에 통합할 수 있습니다.

작업은 MIG 장치를 명시적으로 요청할 수 있지만, 모든 슬라이스를 활용하도록 스케줄링하는 데 주의해야 합니다. 예를 들어, 7슬라이스 구성에서 작업이 1슬라이스만 사용한다면 나머지 6슬라이스는 다른 작업으로 채워야 하며, 그렇지 않으면 상당한 유휴 상태가 발생합니다. 클러스터 내 특정 노드는 소규모 추론 작업용으로 MIG를 사용하도록 구성하고, 다른 노드는 대규모 훈련 작업용 비-MIG 워크로드로 구성하는 것도 가능합니다.

운영상 중요한 점은 MIG 사용을 위해 일반적으로 시스템 수준(최소 노드 수준)에서 구성해야 한다는 것입니다. GPU를 MIG 모드로 전환하고 슬라이스를 생성한 후 GPU를 재설정해야 합니다. 이 과정이 완료되면 슬라이스는 각각 고유한 장치 ID를 가진 별도의 장치로 시스템에 인식됩니다.

사전 계획 부족으로 사용 가능한 MIG 슬라이스를 모두 활용하지 못하면, 슬라이스가 분할된 채로 방치되어 자원이 낭비됩니다. 워크로드에 맞춰 파티션 크기를 사전에 계획하고, 워크로드가 변경될 때 파티션 크기를 조정하는 것이 중요합니다. 변경 사항을 반영하려면 GPU를 재설정해야 합니다.

다수의 GPU를 아우르는 대규모 모델 훈련 작업이나 추론 서버의 경우, 전체 GPU 세트에 접근해야 하므로 MIG는 일반적으로 유용하지 않습니다. 반면, 소규모 GPU 파티션을 실행할 수 있는 다중 테넌트 소형 모델 추론 서버에는 MIG와 그 격리 기능이 유용할 수 있습니다.

현재 시점에서 GPU가 MIG 모드일 경우, GPU 간 피어 투 피어 통신(NVLink 포함) 이 비활성화됩니다. 이는 GPU 간 NVLink 및 PCIe P2P 모두에 적용됩니다. MIG 인스턴스는 다른 GPU와 P2P 통신을 수행할 수 없습니다. MIG 인스턴스 간 CUDA IPC도 제한됩니다. 이는 분산 훈련 처리량을 저하시킬 수 있습니다. MIG를 활성화하기 전에 훈련 또는 추론 토폴로지가 GPU 피어 경로에 의존하지 않는지 확인하십시오. 대규모 훈련 작업(및 스파스 MoE 전문가 추론 시스템)은 방대한 GPU 간 통신이 필요하며 일반적으로 MIG에 적합하지 않습니다. GPU MIG 인스턴스 간 통신은 호스트 또는 네트워크 패브릭을 통해 이루어져야 합니다.

요약하면, 동일한 GPU에서 강력한 격리 상태로 여러 독립적인 작업을 실행해야 할 때만 MIG를 활성화하세요( ). GPU를 아우르는 대규모 분산 훈련이나 추론에는 MIG를 사용하지 마십시오. GPU의 전체 성능과 빠른 상호 연결을 활용해야 하기 때문입니다.

대규모 트랜스포머 기반 모델 훈련 및 추론 환경에서는 MIG를 비활성화할 것입니다. 다만 이 기능이 존재한다는 점은 알아두는 것이 좋습니다. 클러스터가 동적으로 모드를 전환하여 낮 시간대에는 소규모 훈련/추론 실험이 다수 진행될 때 MIG를 실행하고, 밤에는 전체 GPU를 사용하는 대규모 훈련 작업을 위해 MIG를 비활성화하는 방식이 가능할 수 있습니다.

쿠버네티스 장치 플러그인 은 MIG 장치를 다음과 같은 리소스로 나열합니다. 예시: 1/7 GPU 슬라이스(슬라이스 총 23GB)의 경우 `nvidia.com/mig-1g.23gb`

GPU 클럭 속도와 ECC

NVIDIA GPU에는 에서 설명하는 GPU Boost 기능이 있습니다. 이 기능은 전력 및 열 제한 내에서 코어 클럭을 자동으로 조정합니다. 대부분의 경우 GPU가 자체적으로 최적화하도록 두는 것이 좋습니다. 그러나 일부 사용자는 일관성을 위해 클럭을 고정하여 GPU가 항상 고정된 최대 주파수로 작동하도록 선호합니다. 이렇게 하면 실행 간 성능이 안정적이며 전력이나 온도 변동에 영향을 받지 않습니다.

클럭 고정 기능은 벤치마크 수행 시 특히 중요합니다. 후속 실행에서 과열로 인해 성능이 제한될 수 있기 때문입니다. 이를 고려하지 않으면, 이전 실행으로 인한 과열로 후속 GPU가 성능 제한 상태가 되어 발생한 낮은 결과를 오해할 수 있습니다.

구체적으로, NVIDIA의 GPU Boost는 전력/열 제한 범위 내에서 코어 클럭을 상하로 조절합니다. `nvidia-smi -lgc` 를 사용하여 코어 클럭을 고정하고 `-ac` 를 사용하여 메모리 클럭을 고정함으로써 최대 안정 주파수로 클럭을 고정합니다. 이렇게 하면 GPU가 일정한 주파수로 작동하도록 보장되며, 후속 실행에서 GPU의 기본 Boost 기능이 클럭을 낮추는 것을 방지합니다.

이는 주로 벤치마킹 시 결정적이고 재현 가능한 결과를 얻기 위해 중요합니다. 일상적인 훈련 및 추론 작업에서는 상당한 성능 변동이나 GPU 스로틀링이 발생하지 않는 한 자동 부스트 기능을 켜두는 것이 좋습니다.

클럭 잠금은 결정론과 일관성을 극한까지 추구할 때 고려해야 할 사항입니다. 그러나 일반적으로 GPU를 기본 자동 부스트 모드로 두는 것이 무방합니다.

일부 팀은 특히 장시간 작업을 실행할 때 시간이 지남에 따라 발생하는 열에 의한 속도 저하를 방지하기 위해 고의로 GPU를 언더클럭( )합니다. 데이터 센터 GPU는 일반적으로 충분한 온도 여유 공간과 적절한 공기 및 액체 냉각 시스템을 갖추고 있어 이를 수행할 필요가 없지만, 이러한 옵션이 존재한다는 점을 알아두는 것이 좋습니다.

또 다른 방법은 `nvidia-smi -pl` 를 사용하여 GPU의 최대 열 설계 전력(TDP)보다 약간 낮은 전력 제한을 설정하는 것입니다. TDP는 GPU가 지속적인 부하 상태에서 발생시킬 수 있는 최대 열량(와트 단위)을 의미합니다. 이는 과열을 방지하기 위해 반드시 방출해야 하는 열량을 결정합니다.

전력 제한을 TDP보다 낮게 설정하면 GPU 부스트가 열 제한점 아래에서 클럭을 자동 조정합니다. 이는 최대 발열량을 줄이고 스로틀링을 방지하며 성능 저하를 최소화할 수 있습니다.

GPU의 ECC 메모리도 전력 제한( ) 설정 시 고려해야 할 요소입니다. ECC는 예를 들어 우주선으로 인한 단일 비트 메모리 오류가 발생할 경우 메모리를 실시간으로 수정할 수 있도록 보장합니다. 또한 이중 비트 오류가 발생하면 오류를 감지하

여 호출 코드에 오류 신호를 보냅니다. ECC는 일반적으로 NVIDIA 데이터 센터 GPU에서 기본적으로 활성화되어 있습니다.

ECC를 비활성화하면 오류 검사에 필요한 추가 비트로 인해 소량의 메모리를 확보할 수 있습니다. 이는 실시간 오류 검사와 관련된 오버헤드를 줄여 성능 향상을 가져올 수 있으나, 일반적으로 몇 퍼센트에 불과합니다. 그러나 ECC를 끄면 중요한 메모리 오류 보호 기능도 제거되어 시스템 불안정성이나 감지되지 않은 데이터 손상으로 이어질 수 있습니다.

Hopper 및 Blackwell을 포함한 NVIDIA 데이터 센터 GPU의 경우 ECC는 기본적으로 활성화되어 있으며, 신뢰할 수 있는 오류 수정 계산과 데이터 무결성을 보장하기 위해 계속 활성화된 상태를 유지해야 합니다. 대규모 모델에 대한 장시간 훈련 또는 추론 작업에서 단일 메모리 오류는 작업을 완전히 중단시킬 수 있으며, 더 심각한 경우 경고 없이 모델을 조용히 손상시킬 수 있습니다.

중요한 AI 작업 부하에는 항상 ECC를 켜두는 것이 권장됩니다. ECC를 끌 것을 고려할 수 있는 유일한 경우는 연구 환경에서, 제한된 메모리 GPU 클러스터에 모델을 맞추기 위해 추가 메모리 조각이 필요하여 위험을 감수해도 괜찮을 때입니다.

ECC 모드 전환은 GPU 재설정이 필요하며, 해당 GPU에서 실행 중인 작업도 재시작해야 할 가능성이 높습니다. 따라서 자주 전환하고 싶은 토글 기능이 아닙니다. 안정성과 신뢰성을 위해 ECC를 켜두십시오. ECC를 끕니다. ECC를 끕니다. ECC를 끕니다. ECC를 끕니다. ECC를 끕니다. ECC를 끕니다. ECC를 끕니다. ECC를 끕니다. ECC를 끕니다. ECC를 끕니다. ECC를 끕니다. ECC를 끕니다. ECC를 끕니다. ECC를 끕니다. ECC를 끕니다. ECC를 끕니다. E

GPU 메모리 오버서브스크립션, 조각화 및 메모리 부족 처리

CPU RAM과 달리 기본적으로 GPU에는 "스왑" 메모리 라는 개념이 존재하지 않습니다. 사용 가능한 양보다 많은 GPU 메모리를 할당하려고 하면, 친절하지 않은 OOM(메모리 부족) 오류와 함께 더욱 불친절한 프로세스 크래시가 발생합니다. 이 문제를 완화하기 위한 몇 가지 메커니즘이 있습니다: 메모리의 동적 확장을 허용하고, CPU와 GPU 간 통합 메모리를 활용하며, 메모리 풀과 캐싱 할당기를 사용하는 것입니다.

기본적으로 일부 프레임워크(예: TensorFlow)는 시작 시 사용 가능한 GPU 메모리를 모두 확보하여 조각화를 방지하고 성능을 향상시킵니다. 이를 모르고 있다면 GPU를 공유하는 시나리오에서 매우 나쁜 결과를 초래할 수 있습니다. PyTorch는 기본적으로 필요할 때만 GPU 메모리를 할당합니다.

TensorFlow에는 PyTorch와 유사하게 GPU 메모리 사용량을 소규모로 시작하여 필요에 따라 동적으로 확장하는 옵션(`TF_FORCE_GPU_ALLOW_GROWTH=true`)이 있습니다. 그러나 PyTorch와 TensorFlow 모두 GPU가 보유한 메모리보다 더 많은 양을 할당할 수는 없습니다. 다만 이 지연 할당 방식은 다중 테넌트 시나리오에서 더 효과적입니다. 두 프

로세스가 시작부터 동시에 사용 가능한 최대 GPU 메모리를 할당하려 시도하지 않기 때문입니다.

CUDA의 통합 메모리 시스템( )은 메모리를 CPU 또는 GPU 중 어디에 할당할지 사전 정의하지 않고도 할당할 수 있게 합니다. CUDA 런타임이 필요에 따라 페이지 이동을 처리합니다. Hopper 및 Blackwell과 같은 최신 NVIDIA GPU는 페이지 마이그레이션 엔진(PME)을 사용한 온디맨드 페이징을 위한 하드웨어 지원( )을 포함합니다.

PME는 GPU 사용 가능 메모리가 부족해지면 GPU 메모리와 호스트 CPU RAM 간에 메모리 페이지를 자동으로 마이그레이션합니다. 그러나 PME가 유연성을 제공하지만, 작업 부하에 충분한 GPU 메모리를 확보하는 것에 비해 PME에 의존하면 성능 저하가 발생할 수 있습니다.

이 GPU-CPU 메모리 오프로딩은 느릴 수 있습니다( ). 2장에서 배운 바와 같이 CPU 메모리 I/O는 GPU 고대역폭 메모리(HBM) I/O보다 느리기 때문입니다. 이 메커니즘은 주로 GPU RAM에 들어가지 않는 모델을 실행하려는 실무자들을 위한 편의 기능입니다.

성능이 중요한 작업 부하에서는 가능한 한 통합 메모리 오버서브스크립션에 의존하지 않는 것이 좋습니다. 이는 스크립트가 완전히 중단되는 것을 방지하는 안전망 역할을 하지만, GPU 메모리가 오버서브스크립션 상태일 때 작업 실행 속도가 느려집니다.

PyTorch와 같은 라이브러리는 캐싱 할당기를 사용합니다. 따라서 GPU 메모리를 해제할 때 해당 메모리를 즉시 OS에 반환하지 않고, 향후 할당을 위해 재사용할 수 있도록 유지합니다. 이는 메모리 조각화를 방지하고 동일한 메모리 블록을 반복적으로 할당하도록 OS에 요청하는 오버헤드를 줄여줍니다.

PyTorch 할당기는 환경 변수를 통해 구성할 수 있습니다. 예를 들어 `PYTORCH_ALLOC_CONF` (이전 버전: `PYTORCH_CUDA_ALLOC_CONF`)을 사용해 최대 풀 크기를 설정할 수 있습니다. PyTorch 메모리 할당 메커니즘 최적화는 후속 장에서 다룰 예정입니다.

GPU OOM 오류( )가 발생하면(언젠가는 반드시 발생할 것입니다), 이는 메모리 조각화나 과도한 메모리 캐싱이 원인일 가능성이 높습니다. PyTorch의 `torch.cuda.empty_cache()` 를 사용해 캐시를 지울 수 있지만, 이는 거의 항상 작업 부하가 실제로 그만큼의 메모리를 필요로 한다는 의미입니다.

PyTorch는 할당된 메모리와 예약된 메모리를 비교하여 조각화를 진단하는 데 도움이 되는 `torch.cuda.memory_stats()` 및 `torch.cuda.memory_summary()` 같은 도구도 제공합니다. NVIDIA의 Nsight Systems는 또한 메모리 누수, 누수와 연관된 장기 할당, CPU-GPU 상호 연결 활동, GPUDirect Storage 타임라인 추적을 식별하는 데 도움이 되는 GPU 메모리 사용 패턴을 보여줍니다. 또한 Nsight Compute 프로파일러는 점유율, 처리량, NVLink 사용량을 포함한 저수준 커널 분석을 제공합니다. 이 모든 내용은 향후 장에서 다루겠습니다.

Docker는 컨테이너에 GPU를 선택 및 노출하기 위한 `--gpus` 플래그를 제공하지만, GPU 메모리 제한 설정은 지원하지 않습니다. GPU 메모리 또는 컴퓨팅에 대한 하드 격리가 필요한 경우, 장치를 분할하기 위해 MIG를 사용하거나 공정한 공유를 위해 활성 스레드 비율이 적용된 MPS(Multi-Process Service)를 사용하십시오. 엄격한 분할이 필요한 경우 `nvidia.com/mig-2g.45gb`과 같은 MIG 리소스를 사용하여 쿠버네티스에서 제한을 구성하십시오.

다중 테넌트 노드에서는 작업 격리에 유용할 수 있습니다. GPU당 단일 작업 환경에서는 작업이 가능한 한 많은 GPU 메모리를 사용하도록 허용하기 때문에 메모리 제한을 설정하는 경우가 흔하지 않습니다.

일반적으로 GPU 메모리 부족은 애플리케이션 수준에서 관리할 수 있습니다. 예를 들어 데이터 배치 크기, 모델 가중치 정밀도, 또는 가능한 경우 모델 매개변수 수를 줄일 수 있습니다.

모범 사례는 모델 훈련 및 추론 중 `nvidia-smi` 또는 NVML API를 통해 GPU 메모리 사용량 을 모니터링하는 것입니다. 메모리 한계에 근접한 경우 배치 크기 축소, 훈련용 활성화 체크포인트 사용 등 메모리 사용량을 낮추는 방법을 고려하십시오.

또한 CPU 메모리가 스왑되지 않도록 해야 합니다. GPU가 CPU 호스트에서 데이터를 가져오려 할 때마다 호스트 메모리 페이지가 디스크로 스왑되면 훨씬 느린 디스크 I/O로 인해 성능이 병목 현상을 일으키므로, 이는 간접적으로 GPU 활용도와 처리량에 악영향을 미칩니다. 따라서 메모리 고정, `ulimit` 증가, 스왑성(swappiness) 비활성화 등과 같은 이전 조언과 함께 이러한 메모리 감소 모범 사례를 결합하는 것이 중요합니다.

요약하면, 작업 간에 GPU 드라이버를 언로드하지 않고 항상 드라이버를 로드 상태로 유지하도록 설정하는 것이 권장됩니다( ). 이는 GPU 지속성 모드와 유사하지만 더 근본적인 수준에서 적용됩니다. 일부 클러스터는 OS 커널 메모리 확보 및 보안 목적으로 작업이 실행되지 않을 때 드라이버를 언로드하도록 구성됩니다. 그러나 그렇게 할 경우 다음 작업은 GPU 드라이버 재로드 비용을 부담해야 하며, MIG를 사용하는 경우 MIG 슬라이스 재구성 비용도 추가됩니다.

작업 간에 드라이버 및 MIG 구성을 유지하는 것이 좋습니다. GPU 드라이버를 언로드해야 하는 경우는 문제 해결이나 드라이버 업그레이드 시뿐입니다. 따라서 클러스터 관리자는 시스템 부팅 시 NVIDIA 드라이버 모듈이 항상 활성화되도록 설정하는 경우가 많습니다.

GPU를 위한 컨테이너 런타임 최적화

많은 AI 시스템은 오케스트레이션 도구 와 컨테이너 런타임을 사용하여 소프트웨어 환경을 관리합니다. 쿠버네티스와 Docker는 AI 인프라에서 널리 사용됩니다. 컨테이너를 사용하면 CUDA 및 라이브러리 버전을 포함한 모든 종속성이 일관되게 유지됩니다. 이는 "내 컴퓨터에서는 작동하는데"라는 문제를 방지합니다. 컨테이너는 약간의 복잡성과 미미한 오버헤드를 유발하지만, 올바른 구성으로

컨테이너를 사용한 GPU 워크로드에 대해 베어 메탈에 가까운 성능을 얻을 수 있습니다.

노드에서 실행되는 컨테이너는 기존 가상 머신(VM)과 다릅니다. VM과 달리 컨테이너는 호스트 OS 커널을 공유하므로 CPU 및 메모리 작업이 네이티브에 가까운 속도로 수행됩니다. 또한 NVIDIA 컨테이너 툴킷을 사용하면 Docker 컨테이너 내부에서 GPU에 직접 접근할 수 있어 오버헤드가 발생하지 않습니다.

최신 NVIDIA Container Toolkit을 실행하는 현대 GPU의 경우, 적절히 구성된 환경 내 GPU 성능은 컨테이너 외부 베어메탈 호스트에서 직접 코드를 실행할 때와 사실상 동일합니다(2% 미만 차이). 실제로 MLPerf Inference v5.0 결과에는 Red Hat OpenShift와 쿠버네티스가 사용되었으며, 이는 현대 컨테이너 및 오케스트레이션 구성이 효율성이나 지연 시간을 저하시키지 않음을 입증합니다.

NVIDIA 컨테이너 툴킷 및 CUDA 호환성

GPU와 함께 컨테이너 를 사용할 때의 한 가지 과제는 컨테이너 내부의 CUDA 라이브러리가 호스트의 드라이버와 일치하도록 하는 것입니다. NVIDIA는 컨테이너 툴킷과 기본 Docker 이미지를 통해 이 문제를 해결합니다. 호스트는 커널 및 하드웨어와 긴밀하게 통합된 NVIDIA 드라이버를 제공합니다. 컨테이너 내부에서는 일반적으로 특정 버전의 CUDA 런타임 라이브러리를 찾을 수 있습니다.

일반적인 규칙은 호스트의 NVIDIA 드라이버 버전이 컨테이너 내 CUDA 버전이 요구하는 최소 드라이버 버전 이상이어야 한다는 것입니다. CUDA 13.x의 경우, 최소 요구되는 Linux 호스트 드라이버 브랜치는 R580 이상입니다. CUDA 12.x의 경우, 최소 요구되는 Linux 호스트 드라이버 브랜치는 R525 이상입니다. 구형 드라이버와 최신 CUDA 런타임을 함께 사용하면 CUDA 초기화가 실패합니다.

새로운 CUDA 버전마다 최소 NVIDIA 드라이버 버전이 필요합니다. CUDA 툴킷을 업데이트할 때는 항상 NVIDIA 공식 호환성 매트릭스를 참조하고 호스트 드라이버를 업그레이드하십시오.

Docker 및 쿠버네티스 환경에서는 NVIDIA GPU Cloud(NGC) 또는 DockerHub 이미지 저장소에서 NVIDIA 공식 기본 Docker 이미지를 사용하는 것이 가장 간단한 방법입니다. 이러한 이미지(예: `nvcr.io/nvidia/pytorch` 또는 유사한 이미지)는 적절한 버전의 CUDA 런타임, cuDNN, NCCL 등을 번들로 제공합니다. 또한 이러한 Docker 이미지는 CUDA 버전에 따라 최소 요구 CUDA 드라이버를 명시합니다. 이렇게 하면 의존성 문제 없이 최신 하드웨어를 지원받을 수 있습니다.

NVIDIA 컨테이너 런타임

대안으로 NVIDIA의 컨테이너 런타임 은 런타임 시 호스트 드라이버 라이브러리를 컨테이너에 직접 주입할 수 있으므로, 이미지 내에 NVIDIA 드라이버를 포함시킬 필요조차 없습니다. 대신 호스트의 드라이버에 의존하면 됩니다. 이는 컨테이너가 기존 VM처럼 완전히 격리되지 않기 때문에 가능합니다. Docker 컨테이너는 호스트 장치, 볼륨 및 라이브러리를 사용할 수 있습니다.

컨테이너 내부에서 애플리케이션은 CUDA 런타임 라이브러리(예: `libcudart.so` 를 사용하지만, NVIDIA Container Toolkit은 컨테이너 시작 시 호스트의 드라이버 라이브러리(예: `libcuda.so` 및 `libnvidia-ml.so`)를 주입합니다. 호스트 드라이버 라이브러리는 호스트에서 직접 호출되므로 모든 것이 원활하게 작동합니다.

CUDA 런타임 라이브러리(컨테이너)와 NVIDIA 컨테이너 툴킷(호스트)의 분리는 호스트 드라이버가 이미지 내 CUDA 툴킷이 요구하는 최소 버전을 충족하는 한 지원됩니다. 호스트의 구형 드라이버와 컨테이너의 최신 CUDA 버전을 혼용하면 오류가 발생할 수 있습니다. CUDA와 드라이버 버전을 일치시키는 것이 중요합니다.

핵심 요점은 GPU용 컨테이너 사용 시 하이퍼바이저나 가상화 계층이 개입되지 않는다는 점입니다. 컨테이너는 호스트 커널과 드라이버를 직접 공유하므로, GPU에서 커널이 실행될 때 마치 호스트에서 직접 실행된 것과 같습니다.

다시 말해, Docker 기반 가상화로 인해 성능이 저하되지 않습니다. 단, VMware나 단일 루트 입출력 가상화(SR-IOV) 가상 GPU와 같은 특수한 시나리오를 사용하는 경우는 예외이며, 이 경우 일부 튜닝이 필요합니다. Docker와 NVIDIA를 함께 사용할 때는 기본적으로 베어 메탈 성능과 동등합니다.

NVIDIA Container Toolkit은 Docker뿐만 아니라 containerd 및 Podman과도 호환됩니다. 이는 containerd를 기본 컨테이너 런타임으로 사용하는 현대적인 쿠버네티스 환경에 해당됩니다.

컨테이너 오버레이 파일 시스템 오버헤드 희피

를 Docker 컨테이너에서 실행할 때와 호스트에서 직접 실행할 때의 주요 차이점은 I/O에 있을 수 있습니다. 컨테이너는 종종 호스트 파일 시스템과 컨테이너 파일 시스템 같은 여러 기본 파일 시스템을 투명하게 오버레이하여 단일 통합 뷰로 제공하는 유니온 파일 시스템을 사용합니다.

OverlayFS와 같은 유니온 파일시스템에서는 여러 소스의 파일과 디렉터리가 하나의 파일시스템에 속하는 것처럼 보입니다. 이 메커니즘은 베이스 이미지 레이어의 읽기 전용 파일시스템과 쓰기 가능한 컨테이너 레이어가 결합되는 컨테이너 환경에서 특히 유용합니다.

그러나 오버레이 파일 시스템을 사용할 때는 일부 오버헤드가 발생합니다. 파일 시스템이 반환할 파일 버전을 결정하기 위해 읽기 전용 레이어와 쓰기 가능 레이어 등 여러 하위 레이어를 확인해야 하기 때문에 추가적인 지연이 발생합니다. 이러한 추가 메타데이터 조회와 레이어 병합 로직은 단일 단순 파일 시스템에서 읽는 것에 비해 소량의 오버헤드를 추가할 수 있습니다.

또한 오버레이가 사용하는 쓰기 시 복사(CoW) 메커니즘에 쓰기 시 오버헤드가 발생합니다. CoW는 읽기 전용 레이어(예: 기본 이미지)에서 파일을 수정할 때 해당 파일을 먼저 쓰기 가능 레이어로 복사해야 함을 의미합니다. 그런 다음 원본 읽기 전용 파일이 아닌 복사된 쓰기 가능 파일에 대한 쓰기가 수행됩니다. 앞서 언급한 바와 같이, 수정된 파일을 읽으려면 반환할 올바른 버전을 결정하기 위해 읽기 전용 레이어와 쓰기 가능 레이어를 모두 확인해야 합니다.

모델 훈련은 데이터셋 읽기, 모델 로딩, 모델 체크포인트 쓰기 과정에서 대량의 I/O 작업을 수반합니다. 이를 해결하기 위해 바인드 마운트를 사용해 호스트 디렉터리(또는 네트워크 파일 시스템)를 컨테이너에 마운트할 수 있습니다.

바인드 마운트는 오버레이를 우회하므로 호스트에서 직접 디스크 I/O를 수행하는 것과 유사한 성능을 보입니다. 호스트 파일 시스템이 NVMe SSD나 NFS 마운트와 같은 경우 해당 기본 스토리지 장치의 전체 성능을 활용할 수 있습니다. 우리는 의도적으로 멀티 테라바이트 규모의 데이터셋을 이미지 내에 패키징하지 않습니다. 대신 마운트를 통해 데이터를 가져옵니다.

예를 들어, 호스트의 `/data/dataset` 에 훈련 데이터가 있다면, `-v /data/dataset:/mnt/dataset:ro` 로 컨테이너를 실행합니다. 여기서 *ro* 는 읽기 전용 마운트를 의미합니다. 그러면 훈련 스크립트는 `/mnt/dataset` 에서 읽게 됩니다. 이렇게 하면 호스트 파일 시스템에서 직접 읽게 됩니다.

사실, 컨테이너의 쓰기 가능 레이어에 대한 대용량 데이터 읽기/쓰기는 피하는 것이 권장됩니다. 대신 호스트의 데이터 디렉터리와 출력 디렉터리를 컨테이너에 마운트하세요. 컨테이너의 CoW(Copy-on-Write) 메커니즘 오버헤드로 인해 I/O가 병목 현상을 일으키지 않도록 해야 합니다.

컨테이너 시작 속도 향상을 위한 이미지 크기 축소

이미지가 너무 커서 네트워크를 통해 풀링해야 하는 경우 컨테이너 시작 시간이 상당히 느려질 수 있습니다( ). 하지만 일반적인 장기 실행 훈련 루프에서는 몇 분의 시작 시간이 수시간, 수일, 수개월에 달하는 훈련 시간에 비하면 무시할 수 있습니다. 그래도 불필요한 빌드 도구나 임시 빌드 파일을 포함하지 않아 이미지를 적정 수준으로 가볍게 유지하는 것은 여전히 가치가 있습니다. 이는 디스크 공간을 절약하고 컨테이너 시작 시간을 개선합니다.

일부 HPC 센터는 루트 데몬 없이 사용자 공간에서 이미지를 실행할 수 있고 호스트 파일 시스템을 직접 사용하며 OS가 이미 가지는 오버헤드 외에는 사실상 제로에 가까운 오버헤드를 보이는 특성 때문에 Docker보다 Singularity(Apptainer)를 선호합니다.

어느 쪽이든, Docker나 Apptainer(구 Singularity) 모두 연구 및 벤치마크 결과에 따르면, 제대로 구성되면 이러한 컨테이너 솔루션은 컨테이너에서 실행하는 것과 호스트에서 직접 실행하는 것 사이에 불과 몇 퍼센트 차이밖에 나지 않습니다. 본질적으로, 누군가 GPU 사용률과 처리량 로그를 제공한다면, 그 로그만으로는 작업이 컨테이너에서 실행되었는지 아닌지 구분하기 어려울 것입니다.

토폴로지 인식 컨테이너 오케스트레이션 및 네트워킹을 위한 쿠버네티스

쿠버네티스(K8s라고도 함)는 AI 훈련 및 추론을 위한 널리 사용되는 오픈소스 컨테이너 오케스트레이터입니다. NVIDIA의 쿠버네티스용 디바이스 플러그인은 GPU 하드웨어(`/dev/nvidia0` , `/dev/nvidiactl` 등)를 스케줄러에 알리는 경량 구성 요소입니다. 이 플러그인은 `resources.limits` 에서 `nvidia.com/gpu` 를 요청할 때 해당 디바이스 노드를 포드에 마운트하며, 명시적으로 둘 다 설정하려는 경우 `resources.requests` 에서도 선택적으로 마운트합니다. 이렇게 하면 이 장치 플러그인이 설치된 컨테이너를 쿠버네티스에 배포할 때, 쿠버네티스가 해당 컨테이너에 GPU를 사용할 수 있도록 처리합니다. 또한 이 장치 플러그인은 토폴로지를 인식합니다. 즉, 특정 포드에 대해 동일한 NVLink 스위치 또는 동일한 NUMA 노드에서 여러 GPU를 할당하는 것을 선호할 수 있습니다.

NVIDIA 쿠버네티스 GPU Operator는 드라이버 라이브러리, 앞서 언급한 NVIDIA 쿠버네티스 장치 플러그인, NVIDIA Container Toolkit을 포함한 모든 NVIDIA 소프트웨어의 설치 및 라이프사이클을 자동화합니다. 또한 NVIDIA의 GPU Feature Discovery를 사용하여 각 GPU에 NUMA 노드 및 NVLink/NVSwitch ID를 라벨링하는 노드 라벨링을 담당합니다. 스케줄러는 이러한 라벨을 활용하여 GPU를 작업에 지능적으로 할당할 수 있습니다. GPU Operator는 또한 DCGM을 사용한 GPU 모니터링을 구현합니다.

쿠버네티스를 사용하여 GPU 기반 컨테이너를 오케스트레이션할 때, XML-PH-0000@deepl.internal, NUMA 노드 및 네트워크 대역폭 구성을 포함한 하드웨어 토폴로지를 인식하는 방식으로 컨테이너에 리소스를 할당하기를 원합니다. 그러나 기본적으로 쿠버네티스는 토폴로지를 인식하지 못합니다. 각 GPU를 리소스로 취급하지만 GPU 0과 GPU 1이 동일한 NUMA 노드에 있는지, 동일한 NVLink 상호 연결을 사용하는지 알지 못합니다. 이는 큰 차이를 만들 수 있습니다.

8개의 GPU를 가진 서버를 생각해 보세요. 두 개의 4개 GPU 세트가 각각 NVLink로 연결되어 있습니다. 작업에 4개의 GPU를 쿠버네티스에 요청할 경우, K8s가 모두 NVLink로 상호 연결된 네 개의 GPU를 할당해 주는 것이 이상적입니다. 이렇게 하면 데이터 공유 속도가 빨라지기 때문입니다. 그러나 쿠버네티스가 시스템 내 어디에나 흩어져 있는 임의의 네 개의 GPU를 선택하면, 작업에 한 NVLink 도메인의 GPU 두 개와 다른 NVLink 도메인의 GPU 두 개가 할당될 수 있습니다.

토폴로지 인식 없이 GPU를 할당하면 GPU 간 경로에 느린 상호 연결(예: InfiniBand 또는 이더넷)이 도입됩니다. 이는 GPU 간 대역폭을 절반으로 줄일 수 있습니다. 이 경우 쿠버네티스는 서로 다른 랙과 NUMA 도메인에 걸쳐 있는 네 개의 GPU보다, 모두 NVLink로 상호 연결되고 동일한 NUMA 노드를 사용하는 네 개의 GPU를 할당하는 것이 이상적입니다.

예를 들어, NVLink 5 인터커넥트로 구축된 NVIDIA의 NVL72 랙을 고려해 보십시오. 이 랙은 72개의 GPU를 단일 고대역폭 도메인으로 연결하며, 랙 내부에서 총 약 130TB/s의 처리량(72개 GPU × GPU당 1.8TB/s)을 제공합니다. 이러한 구성에서 쿠버네티스 스케줄러가 토폴로지를 인식하지 못하면 다중 GPU 작업을 서로 다른 NVLink 도메인에 배치하거나 심지어 NVL72 그룹 외부로 배치할 수 있습니다. 시스템 토폴로지를 고려하지 않고 작업에 GPU를 할당하면 NVL72의 막대한 랙 내 대역폭 이점이 무효화됩니다.

자원 경합을 피하려면 필요한 자원을 예약하거나 작업에 전체 노드를 요청해야 합니다. 컨테이너/포드 배치 시에는 쿠버네티스 토폴로지 관리자 구성 요소를 사용하여 포드를 CPU 친화성과 NUMA 노드에 정렬시켜 컨테이너의 CPU를 해당 컨테이너에 할당된 GPU와 동일한 NUMA 노드에 바인딩해야 합니다. 다음으로 이 내용을 논의해 보겠습니다.

쿠버네티스 토폴로지 매니저로 컨테이너 오케스트레이션

쿠버네티스 토폴로지 매니저는 상세한 토폴로지 정보를 제공할 수 있습니다. 예를 들어, GPU 0이 NUMA 노드 0, NVLink 도메인 A, PCIe 버스 Z에 연결되어 있음을 감지할 수 있습니다. 쿠버네티스 스케줄러는 이 정보를 활용하여 효율적인 처리 및 통신을 위해 GPU에 컨테이너를 최적의 방식으로 할당할 수 있습니다.

토폴로지 인식 GPU 스케줄링은 아직 발전 중인 기술입니다. 많은 클러스터에서 관리자는 쿠버네티스 라벨을 사용하여 노드에 GPU 및 시스템 토폴로지를 명시적으로 라벨링합니다. 이러한 라벨은 다중 GPU 포드가 동일한 NVLink 상호 연결을 공유하거나 동일한 NUMA 도메인 내에 위치한 GPU를 가진 서버에 배치되도록 보장합니다.

본 문서의 목적상, 쿠버네티스에서 다중 GPU 작업을 실행하는 경우 토폴로지 인식 스케줄링을 반드시 활성화하십시오. 일반적으로 `--topology-manager-policy to best-effort`, `restricted` 또는, 경우에 따라, `single-numa-node` 를 구성하는 작업이 포함됩니다. 이 정책 구성은 원격 메모리 접근을 피함으로써 다중 GPU 및 CPU + GPU 워크로드의 지연 시간을 낮추는 데 도움이 됩니다. 이는 OS 수준의 NUMA 튜닝을 보완합니다.

또한, 이전 섹션에서 언급한 최신 NVIDIA GPU 장치 플러그인과 NVIDIA 쿠버네티스 GPU Operator를 반드시 사용하십시오. 이들은 토폴로지를 인식하며 동일한 NUMA 노드에 연결된 GPU에 다중 GPU 포드를 패킹하는 기능을 지원합니다. 이는 크로스-NUMA-노드 통신을 최소화하고 다중 노드 GPU 워크로드의 지연 시간을 줄여 성능을 최적화하는 데 도움이 됩니다.

NVLink-5 NVL72 시스템에서 단일 랙 수준 NVLink 도메인은 최대 130TB/s의 총 양방향 GPU-to-GPU 대역폭을 제공하며, 이는 GPU당 약 1.8TB/s에 해당합니다. 집합 연산이 많은 훈련을 스케줄링할 때는 느린 네트워크 패브릭을 통과하기 전에 트래픽을 빠른 NVLink 도메인 내에 유지하는 배치를 우선적으로 선택하십시오.

쿠버네티스 및 SLURM을 활용한 작업 스케줄링

다중 노드 배포 환경에서 작업 스케줄러 는 모든 노드에 걸쳐 자원 활용도를 극대화하는 데 필수적입니다. 일반적으로 훈련 클러스터에는 SLURM(Simple Linux Utility for Resource Management)이 사용되며, 추론 클러스터에는 쿠버네티스가 선호됩니다. 그러나 SLURM과 쿠버네티스를 통합하는 하이브리드 솔루션도 등장했습니다. 오픈소스 Slinky 프로젝트는 훈련 및 추론 워크로드 전반에 걸친 클러스터 관리를 간소화하는 솔루션의 한 예입니다.

이러한 시스템은 작업에 GPU를 할당하고 노드 간 프로세스 실행을 조정합니다. 훈련 작업이 노드당 8개의 GPU를 갖춘 8개 노드를 요청하면 스케줄러는 적합한 노드를 식별하고 `mpirun` 같은 도구나 Docker와 같은 컨테이너 런타임을 사용하여 작업을 시작합니다. 이렇게 하면 각 프로세스가 작업 내 사용 가능한 모든 GPU를 인식합니다. 많은 클러스터는 NVIDIA의 NGC Docker 저장소와 같이 검증된 Docker 저장소를 활용하여 모든 노드에서 GPU 드라이버, CUDA 툴킷, PyTorch 라이브러리 및 기타 Python 패키지를 포함한 일관된 소프트웨어 환경을 보장합니다.

SLURM에서도 유사한 문제가 존재합니다. SLURM은 GPU에 대한 "일반 리소스(generic resources)" 개념을 가지고 있으며, 특정 GPU가 특정 NUMA 노드나 NVLink/NVSwitch에 연결되도록 정의할 수 있습니다. 그러면 작업 요청 시 동일한 NUMA 노드에 연결된 GPU를 요청할 수 있습니다.

적절히 설정되지 않으면 스케줄러가 모든 GPU를 동일하게 취급하여 다중 GPU 컨테이너 요청에 비효율적인 할당을 제공할 수 있습니다. 올바른 구성은 불필요한 크로스-NUMA-노드 및 크로스-NVLink GPU 통신 오버헤드를 방지합니다.

SLURM은 MIG 파티션을 별개의 리소스로 스케줄링하는 기능도 지원합니다. 이는 하나의 GPU에 여러 작업을 집약하는 데 유용합니다. 이는 쿠버네티스가 Kubernetes 장치 플러그인을 사용하여 GPU 슬라이스를 스케줄링하는 방식과 유사합니다. 다음으로 Kubernetes에서 MIG 슬라이스를 사용하는 방법을 논의하겠습니다.

MIG를 통한 GPU 슬라이싱

앞서 소개한 NVIDIA의 MIG 모드( )를 활성화하면 단일 물리 GPU가 MIG 인스턴스라고 하는 더 작고 고정된 하드웨어 격리 파티션으로 분할됩니다. 다음은 두 개의 MIG 인스턴스에 대한 쿠버네티스 포드 구성 예시입니다(이 구성은 각 노드에서 MIG 장치를 인식하도록 NVIDIA 쿠버네티스 장치 플러그인이 설정되어 있다고 가정합니다): `nvidia.com/mig-2g.45gb` (이 구성은 각 노드에서 NVIDIA

쿠버네티스 장치 플러그인이 MIG 장치를 인식하도록 설정되어 있음을 가정합니다):

```
resources:
  limits:
    nvidia.com/mig-2g.45gb: "2"
```

여기서 구성은 하나의 GPU에 최소 두 개의 사용 가능한 `2g.45gb` 인스턴스가 있는 노드에서 포드를 실행하도록 지정합니다. 즉, 각 슬라이스가 SM의 2/7(`2g`)을 차지하는 2개의 슬라이스입니다. GPU에 총 132개의 SM이 있다면, 각 슬라이스는 2/7 × 132 SM = 약 38 SM입니다. 이를 2로 곱하면, 포드는 총 약 76 SM을 할당합니다. 총 메모리 할당량은 GPU RAM 45GB입니다.

스케줄러는 이를 여러 GPU나 노드에 분할할 수 없습니다. 쿠버네티스는 단일 노드가 두 파티션을 모두 제공할 수 있는 경우에만 포드를 스케줄링합니다. 이는 포드가 여러 노드에 걸쳐 실행될 수 없기 때문입니다. 단일 노드에 총 76개 SM과 45GB GPU RAM(앞서 계산한 대로)을 수용할 수 있는 두 개의 사용 가능한 MIG 슬라이스(`2g.45gb`)가 없는 경우, 포드는 쿠버네티스의 ' `Pending` '(스케줄링되지 않음) 상태로 유지되며 따라서 실행되지 않습니다. 다른 노드들이 합쳐서 충분한 MIG 용량을 가지고 있더라도 마찬가지입니다.

이러한 제약은 일반적인 워크로드 요구에 따라 MIG 크기를 계획하는 것이 중요함을 강조합니다. 예를 들어, 많은 작업이 `2g.45gb` 슬라이스를 요청하는 경우, 각 GPU가 7개의 가능한 슬라이스 중 3개의 `2g.45gb` 인스턴스를 호스팅하도록 구성할 수 있습니다. 이렇게 하면 단일 포드를 위해 두 개의 해당 인스턴스가 하나의 GPU에 함께 상주할 수 있습니다.

이 단일 노드 제약 조건으로 인해 클러스터의 여러 노드에 걸쳐 필요한 MIG 리소스가 존재하더라도 포드가 실행되지 않을 수 있습니다. 요청된 MIG 리소스가 단일 노드에서만 사용 가능한 경우에만 요청이 충족됩니다.

MIG의 관리적 단점은 GPU를 MIG 모드와 일반(비-MIG) 모드 간 전환하려면 GPU 재설정 또는 컴퓨팅 노드 재부팅이 필요하다는 점입니다. 따라서 스케줄러가 작업별로 동적으로 쉽게 처리할 수 있는 작업이 아닙니다. 다만 일반적으로 MIG 파티션은 사전에 생성한 후 일정 기간 동안 구성을 유지합니다.

쿠버네티스 환경에서는 NVIDIA Kubernetes GPU Operator의 MIG Manager가 노드에서 MIG 파티션을 자동으로 구성하고 유지할 수 있습니다. 이렇게 하면 재부팅 및 드라이버 재로드 시에도 MIG 슬라이스가 활성화된 상태로 유지됩니다.

한 K8s 노드에는 "mig-enabled" 라벨을, 다른 노드에는 "mig-disabled" 라벨을 지정하여 스케줄러가 작업/포드를 해당 라벨에 따라 배치하도록 할 수 있습니다.

이는 운영상의 세부 사항에 가깝지만, MIG가 동적 스케줄러의 산물이 아닌 진정한 정적 파티션이라는 점을 인지하는 것이 중요합니다.

MIG 사용 시 지속성 모드(Persistence mode)를 권장합니다. 이렇게 하면 작업이 실행되지 않더라도 GPU에서 MIG 구성이 활성화된 상태로 유지됩니다. 따라서 주기적 작업 실행 전에 GPU가 슬라이스를 계속 재구성할 필요가 없습니다.

쿠버네티스를 위한 네트워크 통신 최적화

쿠버네티스를 사용해 컨테이너 기반 다중 노드 GPU 워크로드 를 실행할 때, 포드 간 통신이 필요합니다. 기본적으로 쿠버네티스에서는 각 포드가 고유 IP를 가지며, 서로 다른 노드의 포드 간에는 오버레이 네트워크나 NAT(네트워크 주소 변환)가 존재할 수 있습니다. 이는 복잡성과 추가 오버헤드를 유발할 수 있습니다.

GPU 클러스터의 경우 성능에 민감한 작업에 호스트 네트워킹을 사용하는 것이 가장 간단한 해결책입니다. 이는 컨테이너 네트워크가 호스트의 네트워크 인터페이스를 직접 사용하므로 격리되지 않음을 의미합니다. 쿠버네티스에서 이를 활성화하려면 포드 사양에 `hostNetwork: true`을 설정합니다. Docker에서는 다음과 같이 실행할 수 있습니다. `--network=host`.

호스트 네트워킹을 사용하면 컨테이너가 추가 변환이나 방화벽 계층 없이 호스트와 정확히 동일한 방식으로 InfiniBand 상호 연결에 접근할 수 있습니다. 이는 모든 MPI 랭크에 대한 포트 매핑 구성을 제거하므로 MPI 작업에 특히 유용합니다.

그러나 보안 정책으로 인해 호스트 네트워킹을 사용할 수 없는 경우, 쿠버네티스 컨테이너 네트워크 인터페이스(CNI) 및 오버레이 네트워크가 필요한 트래픽을 처리할 수 있는지 반드시 확인해야 합니다. 이러한 경우 NCCL 핸드셰이크 및 데이터 교환을 지원하기 위해 특정 포트를 열어야 할 수 있으며, 연결 설정을 돕기 위해 `NCCL_PORT_RANGE` 및 `NCCL_SOCKET_IFNAME` 과 같은 환경 변수를 사용하여 을 설정해야 할 수 있습니다.

오버레이 네트워크에서 운영할 때는 지연 시간이 낮게 유지되고 작업이 커널 공간에서 실행되는 것이 중요합니다. 또한 사용자 공간 프록시가 노드 간 트래픽을 제한하지 않도록 해야 합니다. 이러한 요소들은 성능에 상당한 영향을 미칠 수 있습니다.

쿠버네티스 환경에서 RDMA를 활성화하려면 Mellanox의 쿠버네티스 RDMA 장치 플러그인 설치를 고려하십시오. 이 플러그인은 포드 인터페이스에 InfiniBand 및 GPUDirect RDMA 엔드포인트를 노출하여 저지연, 제로 카피 네트워킹을 가능하게 합니다.

InfiniBand 또는 RoCE 네트워킹을 사용하는 경우, NIC가 지원하는 경우 NVIDIA 드라이버에서 GPUDirect RDMA를 활성화해야 합니다. 이를 통해 GPU가 CPU를 우회하여 노드 간 통신을 수행하며 NIC와 직접 데이터를 교환할 수 있습니다. 이는 다중 노드 환경에서 높은 성능을 유지하는 데 필수적입니다.

쿠버네티스 오케스트레이션 지터 감소

쿠버네티스 와 같은 오케스트레이터를 실행하면 모든 노드에서 백그라운드 프로세스(예: 쿠버네티스 "kubelet"), 컨테이너 런타임 데몬, 그리고 (이상적으로는) 모니터링 에이전트가 실행됩니다. 이러한 서비스는 CPU와 메모리를 소비하지만, 소비량은 단일 코어의 몇 퍼센트 수준입니다. 따라서 데이터 로딩 및 전처리에 해당 코어를 사용하는 GPU 기반 훈련 작업에서 눈에 띄는 시간을 빼앗지는 않습니다.

그러나 훈련 작업이 추론 워크로드도 실행 중인 노드에서 실행될 경우, 실행 시간과 처리량에 약간의 지터(jitter) 또는 예측 불가능한 변동이 발생할 수 있습니다. 이는 다중 테넌시 환경에서는 흔히 발생하는 현상입니다. 동일한 머신에서 다른 컨테이너가 예상치 못하게 많은 CPU나 I/O를 사용하면, 동일한 리소스를 놓고 경쟁하게 되어 훈련이든 추론이든 관계없이 해당 컨테이너에 영향을 미칩니다.

훈련만 수행하거나 추론만 수행하는 동질적 워크로드는 훈련과 추론이 혼합된 이질적 워크로드보다 시스템 관점에서 디버깅 및 튜닝이 훨씬 용이합니다.

자원 보장 개선

의 쿠버네티스는 리소스 경합을 방지하기 위해 파드에 대한 리소스 요청량(request)과 제한량(limit)을 정의할 수 있게 합니다. 예를 들어, 훈련 작업에 16개의 CPU 코어와 64GB의 RAM이 필요하다고 지정할 수 있습니다. 그러면 쿠버네티스는 해당 리소스를 해당 작업 전용으로 예약하고 동일한 CPU에 다른 파드를 스케줄링하지 않도록 합니다.

이러한 제한은 Linux cgroups를 통해 강제 적용되므로, 컨테이너가 할당량을 초과하면 스로틀링되거나 OOM 킬러에 의해 종료될 수 있습니다. 성능이 중요한 작업이 필요한 CPU 리소스에 독점적으로 접근할 수 있도록 보장하고 다른 프로세스가 예약된 코어의 CPU 시간을 빼앗지 못하게 하려면 리소스 요청을 사용하고, 선택적으로 CPU 매니저 기능을 활용해 코어를 고정하는 것이 일반적인 관행입니다.

지터의 또 다른 원인은 배경 커널 스레드와 인터럽트입니다. 이는 2장에서 인터럽트 요청(IRQ) 어피니티 사용과 관련해 논의한 바 있습니다. 쿠버네티스와 유사하게, 다른 포드가 작업과 동일한 네트워크나 디스크를 사용 중이라면 해당 포드들이 작업이 호스팅되는 컴퓨팅 노드에서 많은 인터럽트와 추가 커널 작업을 유발할 수 있습니다. 이는 지터를 발생시켜 작업 성능에 영향을 미칩니다.

이상적으로는 GPU 노드가 작업에 완전히 전용되어야 합니다. 그렇지 않은 경우, 다른 워크로드가 간섭하지 않도록 I/O 및 CPU에 대해 Linux cgroup 컨트롤러를 사용하여 노드를 신중하게 분할해야 합니다.

다행히 쿠버네티스는 CPU 격리 기능을 지원하여 포드가 요청한 전용 CPU 코어와 메모리를 확보하고, 다른 포드가 동일한 CPU 코어에 스케줄링되는 것을 방지합니다. 이를 통해 컨텍스트 스위칭과 리소스 경합으로 인한 추가 오버헤드를 피할 수 있습니다.

실제 환경에서는 성능에 민감한 쿠버네티스 작업이 해당 노드의 모든 CPU와 GPU를 요청해야 합니다. 이렇게 해야 다른 작업이 해당 작업의 리소스를 방해하거나 경쟁하지 않습니다. 말처럼 쉽지는 않지만, 성능과 일관성 측면에서 이상적인 작업 구성입니다.

메모리 격리 및 OOM 킬러 방지

메모리 간섭도 적절히 제한되지 않으면 발생할 수 있습니다. 쿠버네티스는 (Linux cgroups를 사용한) 일류 메모리 격리 기능을 제공합니다. 그러나 제약 없이 탐욕스러운 컨테이너는 호스트에 지나치게 많은 메모리를 할당할 수 있습니다. 이로 인해 호스트는 일부 메모리를 디스크로 스왑하게 됩니다.

호스트에서 무제한 컨테이너 가 너무 많은 메모리를 사용하면, 악명 높은 Linux "OOM 킬러"가 프로세스를 종료하기 시작합니다. 심지어 해당 작업이 과도한 메모리를 사용한 작업이 아니더라도, 잠재적으로 쿠버네티스 작업까지 종료될 수 있습니다.

OOM 킬러는 종료할 포드를 결정할 때 휴리스틱을 사용합니다. 때로는 가장 큰 실행 중인 포드를 종료하기로 결정하는데, 이는 GPU에 공급할 데이터를 CPU RAM에 많이 보유한 대규모 훈련 또는 추론 작업일 가능성이 높습니다. 이를 방지하려면 훈련 또는 추론 컨테이너에 엄격한 메모리 제한을 의도적으로 설정하지 않을 수 있습니다. 이렇게 하면 필요한 경우 사용 가능한 모든 메모리를 사용할 수 있습니다.

적절한 모니터링과 알림을 통해 작업이 예상 범위를 초과하여 메모리를 할당하지 않도록 할 수 있습니다. 메모리 제한을 설정할 경우, 실제 예상 사용량보다 높게 설정하세요. 이렇게 하면 장기 실행 훈련 작업이 3일째 진행 중일 때 OOM 킬러에 의해 종료되는 것을 방지할 수 있는 여유 공간이 생깁니다.

쿠버네티스에서 요청/제한(requests/limits)이 설정되지 않은 포드(Pod)는 '최소 보장(`BestEffort`)'으로 처리되며, 가장 먼저 제거될 가능성이 높습니다. '최대 보장(`Guaranteed`)' QoS를 얻으려면 모든 컨테이너가 CPU와 메모리 모두에 대해 요청(`requests == limits`)을 설정해야 합니다. 높은 제한값만 설정하면 '최대 보장(`Burstable`)' QoS가 적용되며, '최대 보장(`Guaranteed`)'이 아닙니다.

I/O 격리 처리

현재 시점에서 쿠버네티스는 기본적으로 네이티브한 1급 I/O 격리 기능을 제공하지 않습니다. Linux는 cgroup 컨트롤러를 통한 I/O 제어를 지원하지만, 쿠버네티스 자체는 CPU 및 메모리와 같은 방식으로 I/O 제한을 자동으로 강제 적용하지 않습니다.

GPU 노드에서 실행되는 중량급 I/O 워크로드 간 간섭을 방지하려면 노드 수준에서 I/O 제어를 수동으로 구성해야 할 수 있습니다. 이는 cgroup v2 I/O 컨트롤러 조정이나 I/O 자원 분할을 위한 기타 OS 수준 설정을 포함할 수 있습니다. 요약하면, 쿠버네티스는 스케줄링과 리소스 요청을 통해 CPU 경합을 방지하지만, I/O 격리에는 일반적으로 기본 Linux 시스템에 대한 추가적인 수동 튜닝이 필요합니다.

컨테이너 내부에서는 일부 시스템 설정이 호스트에서 상속된다는 점을 유의해야 합니다. 예를 들어 호스트의 CPU 주파수 스케일링이 성능 모드로 설정된 경우 컨테이너도 해당 설정을 상속합니다. 그러나 클라우드 인스턴스와 같은 가상화 환경에서 컨테이너가 실행 중인 경우 이러한 설정을 변경할 수 없을 수 있습니다.

호스트 머신이 항상 튜닝되어 있는지 확인하는 것이 좋습니다. 컨테이너는 hugepage 설정이나 CPU 거버너 제한과 같은 커널 매개변수를 변경할 수 없기 때문입니다. 일반적으로 클러스터 관리자는 기본 OS 이미지를 통해 이러한 매개변수와 설정을 지정합니다. 또는 쿠버네티스 환경에서는 NVIDIA GPU Operator와 같은 도구를 사용하여 각 노드에서 지속성 모드 및 기타 `sysctl` 노브를 설정할 수 있습니다.

핵심 요약

이 장에서 다룬 운영체제, 드라이버, GPU, CPU, 컨테이너 계층 전반에 걸친 최적화 전략의 핵심 요점은 다음과 같습니다:

데이터 및 컴퓨팅 로컬리티는 매우 중요합니다

데이터가 가능한 한 컴퓨팅 유닛에 가깝게 저장되고 처리되도록 하십시오. NVMe SSD 캐시와 같은 로컬 고속 스토리지를 사용하여 지연 시간을 최소화하고 원격 파일 시스템이나 네트워크 I/O에 대한 의존도를 줄이십시오.

NUMA 인식 구성 및 CPU 어피니티 구현

동일 NUMA 노드 내에서 프로세스와 메모리 할당을 정렬하여 CPU-GPU 데이터 흐름을 최적화하십시오. `numactl` 과 같은 도구로 CPU를 고정하고, `taskset` 을 통해 노드 간 메모리 접근을 방지하십시오. 이는 지연 시간 감소와 처리량 향상을 가져옵니다.

GPU 드라이버 및 런타임 효율성 극대화

GPU 드라이버 설정을 미세 조정하세요. 예를 들어 지속성 모드(persistence mode)를 활성화하여 GPU를 대기 상태로 유지합니다. 단일 GPU에서 여러 프로세스의 작업을 중첩 처리하기 위한 다중 프로세스 서비스(MPS) 같은 기능을 고려하세요. 다중 테넌트 환경에서는 MIG 파티션을 활용하여 워크로드를 효과적으로 격리하세요.

데이터를 효과적으로 프리페치 및 배치 처리

사전 데이터 프리페칭과 소규모 I/O 작업을 더 크고 효율적인 읽기로 배치하여 GPU에 지속적으로 데이터를 공급하세요. PyTorch의 DataLoader(`prefetch_factor`)와 같은 프리페칭 메커니즘(`num_workers` 와 함께 사용)을 활용하여 여러 배치를 미리 로드하세요.

데이터 로딩 시 메모리 고정

PyTorch의 DataLoader `pin_memory=True` 를 사용한 데이터 프리페칭과 메모리 고정(pinned)을 결합하면 고정된 CPU 메모리(페이지 잠금 상태, 디스크로 스왑 불가능)를 활용하여 GPU로의 비동기 데이터 전송 속도를 높일 수 있습니다. 그 결과 데이터 로딩과 모델 실행이 중첩되어 유휴 시간이 감소하고 CPU 및 GPU 리소스가 지속적으로 활용됩니다.

메모리 전송을 최적화하십시오

고정된 페이지 잠금 메모리 및 hugepages와 같은 기술을 활용하여 호스트와 GPU 간 데이터 전송을 가속화하세요. 이는 복사 오버헤드를 줄이고 비동기 전송이 계산과 중첩되도록 합니다.

계산과 통신을 중첩

경사 동기화 및 데이터 스테이징과 같은 메모리 작업을 진행 중인 GPU 계산과 중첩시켜 데이터 전송 대기 시간을 줄입니다. 이러한 중첩은 높은 GPU 활용도를 유지하고 전반적인 시스템 효율성을 향상시킵니다.

네트워킹 스택 조정 및 확장

다중 노드 환경에서는 RDMA 지원 네트워크(예: InfiniBand/이더넷)를 사용하고, TCP 버퍼, MTU, 인터럽트 어피니티와 같은 네트워크 설정을 조정하여 분산 훈련 및 추론 중 높은 처리량을 유지하십시오.

일관성을 위해 컨테이너화와 오케스트레이션 활용

NVIDIA Container Toolkit과 함께 Docker 같은 컨테이너 런타임을 사용하고, NVIDIA GPU Operator 및 장치 플러그인과 함께 쿠버네티스 같은 오케스트레이션 플랫폼을 사용하여 드라이버, CUDA 라이브러리, 애플리케이션 코드를 포함한 전체 소프트웨어 스택이 노드 간에 일관되게 유지되도록 하십시오. 이러한 솔루션은 CPU-GPU 어피니티를 조정하고 하드웨어 토폴로지에 기반한 리소스 할당을 관리하는 데 도움이 됩니다.

컨테이너 런타임 오버헤드 제거

컨테이너는 재현성과 배포 용이성을 높이지만, 컨테이너 오버헤드를 최소화하기 위해 CPU 및 GPU 어피니티, 호스트 네트워킹, 리소스 격리가 올바르게 구성되었는지 확인하십시오.

오케스트레이션 및 스케줄링 모범 사례 활용

쿠버네티스와 같은 강력한 컨테이너 오케스트레이터는 효율적인 리소스 할당을 보장하는 핵심 구성 요소입니다. 쿠버네티스 토폴로지 매니저와 같은 고급 스케줄링 기술은 고속 상호 연결을 갖춘 GPU가 함께 클러스터링되도록 보장합니다.

동적 적응성과 확장성을 통한 유연성 추구

오케스트레이션 계층은 작업을 분배하고 노드 간에 워크로드 분할을 동적으로 관리합니다. 이러한 유연성은 훈련 작업의 확장성과 데이터 부하 및 요청 패턴이 크게 달라지는 추론 시나리오에서 효율적인 런타임을 보장하는 데 모두 중요합니다.

지속적이고 점진적으로 조정

시스템 수준의 최적화는 일회성 작업이 아닙니다. 성능 지표를 정기적으로 모니터링하고, 워크로드 변화에 따라 CPU 어피니티, 배치 크기, 프리페치 설정을 조정하며, 이러한 작은 개선 사항을 누적적으로 활용하여 상당한 성능 향상을 달성해야 합니다.

스택 전반에 걸친 병목 현상 감소

궁극적인 목표는 OS와 CPU부터 GPU 드라이버 및 런타임에 이르기까지 모든 구성 요소가 조화를 이루도록 하는 것입니다. CPU 메모리 할당이나 드라이버 초기화와 같은 한 계층의 병목 현상을 제거하면 GPU의 잠재력을 완전히 발휘할 수 있으며, 이는 더 빠른 훈련, 낮은 비용, 더 효율적인 자원 사용으로 직접 연결됩니다.

이러한 전략들은 데이터 전송 마찰을 최소화하고 대기 시간을 줄이며, 효율적인 훈련 및 추론을 위해 하드웨어가 최대한 활용되도록 보장합니다.

결론

이 장에서는 가장 진보된 GPU조차 주변 환경의 비효율성으로 인해 성능이 저하될 수 있음을 보여주었습니다. 잘 조정된 운영체제, 컨테이너 런타임, 클러스터 오케스트레이터, 소프트웨어 스택은 고성능 AI 시스템의 핵심을 이룹니다. NUMA 인식 핀닝 및 로컬 스토리지 솔루션을 통한 데이터와 컴퓨팅의 정렬, 통신과 연산의 중첩, 호스트 시스템과 GPU 드라이버의 미세 조정을 통해 지연 시간을 줄이고 처리량을 높일 수 있습니다.

전체 시스템을 정밀하게 설계된 스포츠카로 생각하십시오. 각 구성 요소(CPU, 메모리, GPU, 네트워크, 컨테이너, 오케스트레이터, 프로그래밍 스택)가 원활하게 협력해야 최대 성능을 발휘할 수 있습니다. 지속성 모드 활성화나 CPU 스케줄링

최적화와 같은 사소한 조정들은 개별적으로는 미미해 보일 수 있지만, 대규모 GPU 클러스터에 걸쳐 결합 및 확장될 경우 상당한 시간과 비용 절감으로 이어질 수 있습니다. 이러한 최적화는 대규모 트랜스포머 모델 훈련 및 복잡한 추론 파이프라인 실행 시 GPU가 지속적으로 최고 효율에 근접하여 작동하도록 보장합니다.

분야가 발전하고 모델이 계속 성장함에 따라 시스템 수준 튜닝의 중요성은 더욱 커질 것입니다. 본 장에서 논의된 기법들은 성능 엔지니어와 시스템 설계자가 하드웨어 잠재력을 최대한 활용할 수 있도록 지원합니다. 이를 통해 더 빠른 반복 주기와 비용 효율적인 AI 배포가 가능해집니다. 궁극적으로 깊이 최적화된 시스템은 연구를 가속화하고 최첨단 AI 애플리케이션을 더 넓은 대중에게 접근 가능하게 만듭니다.

마지막으로, 하드웨어와 소프트웨어 스택이 관리하기 어려운 수많은 연결된 조절 장치처럼 보일 수 있지만, 작은 조정만으로도 상당한 시간과 비용 절감으로 이어질 수 있음을 기억하십시오. 성능 지표를 지속적으로 모니터링하고 스택의 각 계층을 점진적으로 개선함으로써 잠재적 병목 현상을 효율성 향상 기회로 전환할 수 있습니다. 데이터가 여러분을 안내하도록 하십시오. 그러면 AI 시스템의 완전한 잠재력을 발휘할 수 있을 것입니다.