부록. AI 시스템 성능 체크리스트 (175개 이상 항목)

이 포괄적인 체크리스트는 AI 시스템 성능 엔지니어를 위한 광범위한 프로세스 수준 모범 사례와 세부적인 저수준 튜닝 조언을 모두 다룹니다. 각 체크리스트 항목은 AI 시스템에서 최대 성능과 효율성을 끌어내기 위한 실질적인 지침 역할을 합니다.

AI 시스템 디버깅, 프로파일링, 분석 및 튜닝 시 이 가이드를 활용하십시오. 저수준 OS 및 CUDA 조정부터 클러스터 규모 최적화에 이르기까지 이러한 팁을 체계적으로 적용함으로써, AI 시스템 성능 엔지니어는 CUDA, PyTorch, OpenAI의 Triton, TensorFlow, Keras, JAX 등 다양한 AI 소프트웨어 프레임워크를 사용하여 최신 NVIDIA GPU 하드웨어에서 번개처럼 빠른 실행과 비용 효율적인 운영을 동시에 달성할 수 있습니다. 이 체크리스트의 원칙은 NVIDIA의 차세대 하드웨어(GPU, ARM 기반 CPU, CPU-GPU 슈퍼칩, 네트워킹 장비, 랙 시스템 포함)에도 적용됩니다.

성능 튜닝과 비용 최적화 마인드셋

실용적이고 문서화된 반복 과정—심층 작업 전 빠른 성과 —은 엔지니어링 시간을 측정 가능한 ROI로 전환합니다. 가장 큰 실행 시간 및 비용 요인을 목표로 삼고, 항상 자질을 통해 영향을 검증하세요.

자동 튜닝, 프레임워크 업그레이드, 클라우드 가격 조정 레버, 활용도 대시보드를 결합하여 높은 ROI 성과를 달성하고, 결과를 문서화하며 단순하고 유지 보수성 있는 수정 방안을 우선시하세요. 정확도가 허용할 경우 처리량에 민감한 하이퍼파라미터를 조정하세요. 성능 튜닝 및 비용 최적화 마인드셋에 대한 몇 가지 팁은 다음과 같습니다:

비용이 많이 드는 부분을 먼저 최적화하세요

80/20 법칙을 적용하십시오. 실행 시간에 가장 큰 영향을 미치는 요소를 찾아 집중하십시오. 실행 시간의 90%가 몇 개의 커널이나 통신 단계에 소요된다면, 1%의 시간을 차지하는 부분을 미세 최적화하기보다 해당 부분

을 깊이 있게 최적화하는 것이 더 효과적입니다. 각 장의 기법은 가장 중요한 부분에 적용해야 합니다. 예를 들어, 훈련 과정이 데이터 로딩 40%, GPU 연산 50%, 통신 10%로 구성된다면, 먼저 데이터 로딩을 해결하세요. 오버헤드를 절반으로 줄일 수 있을 것입니다. 그다음 GPU 커널 최적화를 살펴보세요.

자질 전후 비교

최적화를 적용할 때마다 그 효과를 측정하세요. 당연한 말처럼 들리지만, 종종 이론에 기반한 조정이 실제로는 도움이 되지 않거나 오히려 해가 될 수 있습니다. 워크로드가 메모리 제한이 아닌 상황에서 훈련 작업에 활성화 체크포인트를 활성화하기로 결정했다고 가정해 보십시오. 이는 메모리를 줄이기 위해 추가 컴퓨팅을 사용함으로써 실제로 작업 속도를 늦출 수 있습니다. 즉, 변경 전후의 처리량, 지연 시간, 활용률과 같은 핵심 지표를 항상 비교해야 합니다. 100회 반복에 걸친 평균 반복 시간과 같은 간단한 타이밍 측정을 위해 내장 프로파일러를 사용하십시오.

적응형 자동 튜닝 피드백 루프를 수용하세요

강화 학습이나 베이지안 최적화와 같은 기법을 활용하여 실시간 성능 피드백을 기반으로 시스템 매개변수를 동적으로 조정하는 고급 자동 튜닝 프레임워크를 구현하세요. 이 접근 방식은 변화하는 워크로드와 운영 조건에 대응하여 시스템 설정을 지속적으로 미세 조정할 수 있게 합니다.

최적화 시간 예산 확보

성능 엔지니어링은 반복자 투자입니다. 수익 감소 법칙이 적용되므로 AMP 활성화나 데이터 프리페치 같은 쉬운 개선점부터 선택하세요. 이로 인해 성능이 쉽게 2배 향상될 수 있습니다. 커스텀 커널 작성 같은 어려운 최적화는 더 작은 성능 향상을 가져올 수 있습니다. 항상 엔지니어링 소요 시간과 실행 시간 단축 및 비용 절감 효과를 비교 평가하세요. 주력 모델 훈련과 같은 대규모 반복 작업의 경우, 5%의 성능 향상만으로도 수백만 달러를 절감할 수 있으므로 몇 주간의 튜닝이 정당화될 수 있습니다. 일회성 또는 소규모 작업 부하의 경우, 더 큰 성과를 목표로 실용적인 접근을 취하십시오.

프레임워크 개선 사항을 지속적으로 확인하세요

혼합 정밀도, 융합 커널, 분산 알고리즘 등 논의한 많은 최적화 기법은 Deep Learning 프레임워크와 라이브러리에서 지속적으로 개선되고 있습니다. 최신 PyTorch나 TensorFlow로 업그레이드하면 새로운 융합 연산이나 개선된 휴리스틱이 적용되어 즉각적인 속도 향상을 얻을 수 있습

니다. 이러한 개선 사항은 사실상 무비용 이득이므로 적극 활용하세요. 성능 관련 변경 사항은 릴리스 노트를 확인하십시오.

공급업체 및 커뮤니티 구성원과 협업하여 공동 설계

하드웨어 벤더 및 광범위한 성능 엔지니어링 커뮤니티와 지속적으로 소통하여 소프트웨어 최적화를 최신 하드웨어 아키텍처에 맞추십시오. 이러한 공동 설계 접근법은 신흥 하드웨어 기능을 활용하도록 알고리즘을 맞춤화함으로써 상당한 성능 향상 기회를 발견할 수 있습니다. 벤더 문서를 정기적으로 검토하고, 포럼에 참여하며, 드라이버나 프레임워크의 베타 버전을 테스트하세요. 이러한 상호작용을 통해 시스템에 통합할 수 있는 새로운 최적화 기회와 모범 사례를 발견할 수 있습니다. 새로운 드라이버 최적화, 라이브러리 업데이트, 하드웨어별 팁을 통합하면 추가적이고 때로는 상당한 성능 향상을 얻을 수 있습니다.

비용 절감을 위한 클라우드 유연성 활용

Cloud 환경에서 실행할 경우, 저렴한 스팟 인스턴스나 예약 인스턴스를 현명하게 활용하십시오. 비용을 대폭 절감할 수 있지만, 스팟 인스턴스는 몇 분 전 통보로 종료될 수 있습니다. 또한 인스턴스 유형을 고려하십시오. 작업 부하가 최신 사양을 절대적으로 필요로 하지 않는다면, 비용의 일부에 불과한 약간 구형 GPU 인스턴스가 더 나은 가격 대비 성능을 제공할 수 있습니다. H800 대 H100에 대한 논의에서 보듯, 노력만 있다면 차선책 하드웨어로도 훌륭한 작업을 수행할 수 있습니다. Cloud에서도 유사한 절충점을 찾을 수 있습니다. CPU 수, CPU 메모리, GPU 수, GPU 메모리, L1/L2 캐시, 통합 메모리, NVLink/NVSwitch 상호 연결, 네트워크 대역폭 및 지연 시간, 로컬 디스크 구성 등 다양한 인스턴스 구성에 대한 벤치마킹을 통해 비용 대비 성능을 평가하세요. 최적화 결정을 안내하기 위해 달러당 처리량과 같은 지표를 계산하세요.

사용률 지표를 모니터링하세요

GPU 사용률, SM 효율성, 메모리 대역폭 사용량, 그리고 다중 노드 환경에서는 네트워크 사용률을 지속적으로 모니터링하세요. DCGM 익스포터, 프로메테우스 등을 활용해 대시보드를 구축하여 어떤 리소스가 저활용되는지 파악할 수 있도록 하세요. GPU 사용률이 50%라면 그 원인을 파헤쳐야 합니다. 데이터 대기/정지 및 느린 동기화 통신이 원인일 가능성이 높습니다. 네트워크 사용률이 10%에 불과하지만 GPU가 데이터를 기다리는 경우, 잠금(lock)과 같은 다른 문제가 있을 수 있습니다. 이러한 지표는 집중해야 할 하위 시스템을 정확히 파악하는 데 도움이 됩니다.

처리량 향상을 위한 하이퍼파라미터 반복자 조정

배치 크기, 시퀀스 길이, MoE 활성 전문가 수 등 일부 모델 하이퍼파라미터는 최종 정확도 저하 없이 처리량 향상을 위해 조정 가능합니다. 예를 들어, 더 큰 배치 크기는 처리량을 높이지만 정확도 유지를 위해 학습률 스케줄 조정이 필요할 수 있습니다. 속도와 정확도의 최적 균형점을 찾기 위해 이러한 값을 조정하는 것을 주저하지 마십시오. 이는 성능 엔지니어링의 일부이기도 합니다. 활성화 체크포인트 사용이나 동일한 유효 배치에 대해 더 많은 계산 단계를 수행하는 등 모델이나 훈련 절차를 효율적으로 조정할 수 있는 경우가 있습니다. 이러한 시나리오를 보완하기 위해 훈련 학습률 스케줄을 미세 조정할 수도 있습니다.

문서화 및 재사용

적용한 최적화 사항과 그 영향을 기록하세요. 코드나 내부 위키형 공유 지식베이스 시스템에 문서화하십시오. 이는 향후 프로젝트를 위한 지식 기반을 구축합니다. 클러스터에서 도움이 되는 중첩 활성화나 특정 환경 변수 설정처럼 재사용 가능한 패턴인 팁이 많습니다. 이러한 기록은 새로운 작업을 시작하거나 신규 팀원을 성능 조정 작업에 합류시킬 때 시간을 절약할 수 있습니다.

복잡성과 최적화 간 균형 유지

필요한 성능을 달성하는 가장 단순한 해결책을 추구하세요. 예를 들어, 네이티브 PyTorch와 `torch.compile` 로 속도 목표를 충족한다면 커스텀 CUDA 커널을 작성할 필요가 없을 수 있습니다. 이는 추가 유지보수를 피하는 데 도움이 됩니다. 지나치게 커스터마이징된 코드로 과도하게 최적화하면 시스템이 취약해질 수 있습니다. 빠르면서도 유지 보수성이 뛰어난 솔루션에 진정한 우아함이 있습니다. 따라서 필요한 성능 향상을 가져오는 최소한의 침습적 최적화를 적용하고, 필요할 때만 더 복잡한 최적화로 확장하세요.

AI 기반 성능 최적화

머신러닝 모델을 활용하여 과거 텔레메트리 데이터를 분석하고 시스템 병목 현상을 예측함으로써, 실시간으로 매개변수를 자동 조정하여 자원 할당과 처리량을 최적화하십시오.

재현성과 문서화 모범 사례

성능 개선은 재현 가능하고 버전 관리되며 지속적으로 점검되지 않으면 효과가 지속되지 않으며, 시간이 지남에 따라 조용히 퇴보합니다. 문서, CI 벤치마크, 공

유된 지식을 속도 향상을 유지하고 온보딩 및 감사를 가속화하는 접착제로 간주하십시오.

버전, 구성, 벤치마크를 소스 제어에 고정하여 실험을 반복 가능하게 하고 회귀를 추적 가능하게 하십시오. 성능 검사를 CI/CD에 통합하고, 종단 간 모니터링 및 경보를 구현하며, 최적화와 보안 및 철저한 문서를 결합하여 지속 가능하고 감사 가능한 관행을 구축하십시오. 재현성 및 문서화를 개선하기 위한 팁 목록은 다음과 같습니다:

엄격한 버전 관리

모든 시스템 구성, 프레임워크/드라이버 버전, OS 설정, 최적화 스크립트, 벤치마크에 대한 포괄적인 버전 관리를 유지하십시오. Git(또는 유사 시스템)을 사용하여 변경 사항을 추적하고 릴리스를 태그하십시오. 이렇게 하면 실험을 정확히 재현할 수 있으며 성능 저하를 쉽게 식별할 수 있습니다.

성능 저하를 위한 지속적인 통합

자동화된 성능 벤치마크와 실시간 모니터링을 CI/CD 파이프라인에 통합하십시오. 이를 통해 코드 업데이트부터 구성 변경까지 모든 변경 사항이 일련의 성능 지표에 대해 검증되어 성능 저하를 조기에 포착하고 일관되고 측정 가능한 성능 향상을 유지할 수 있습니다. MLPerf와 같은 업계 표준 벤치마크를 채택하여 신뢰할 수 있는 성능 기준선을 설정하고 시간 경과에 따른 개선 사항을 추적하십시오.

종단간 워크플로 최적화

데이터 수집 및 전처리부터 훈련 및 추론 배포에 이르기까지 전체 AI 파이프라인에 걸쳐 최적화가 종합적으로 적용되도록 하십시오. 조정된 크로스 시스템 튜닝은 개별적인 조정으로는 놓칠 수 있는 시너지 효과를 발견하여 전반적인 성능 향상을 더욱 크게 이끌어낼 수 있습니다.

자동화된 모니터링 및 진단

하드웨어, 네트워크, 애플리케이션 계층 전반에 걸쳐 실시간 지표를 수집하는 종단 간 모니터링 솔루션을 배포하십시오. 이를 Prometheus/Grafana와 같은 대시보드에 통합하고 자동화된 경보를 구성하여 GPU 사용률의 급격한 하락이나 네트워크 지연 시간의 급증과 같은 이상 현상을 prompt하게 감지하십시오.

결함 내결함성 및 자동 복구

분산 체크포인트, 중복 하드웨어 구성 및 동적 작업 재스케줄링을 활용하여 시스템 설계에 내결함성을 통합하십시오. 이 전략은 하드웨어 또는 네트워크 장애 발생 시에도 가동 중단 시간을 최소화하고 성능을 유지합니다.

컴파일러 및 빌드 최적화

빌드 과정에서 공격적인 컴파일러 플래그와 자질 기반 최적화를 활용하여 코드에서 최대 성능을 끌어내십시오. 빌드 구성을 정기적으로 업데이트하고 조정하며, 각 변경 사항의 영향을 엄격한 벤치마킹을 통해 검증하여 최적의 실행을 보장하십시오.

보안, 규정 준수 및 성능

보안, 규정 준수 및 성능을 통합하고 공동 설계하십시오. 정기적으로 구성을 감사하고, 접근 제어를 시행하며, 암호화, 보안 데이터 채널, 제로 트러스트 네트워킹, 하드웨어 보안 모듈(HSM), 보안 엔클레이브 등 업계 표준 안전 장치를 유지하십시오. 또한 성능 튜닝이 시스템 보안을 결코 저해하지 않도록 하십시오. 마찬가지로 보안이 불필요한 성능 오버헤드를 발생시키지 않도록 하십시오.

포괄적인 문서화 및 지식 공유

모든 최적화 단계, 시스템 구성 및 성능 벤치마크에 대한 상세 기록을 유지하십시오. 팀 협업과 신속한 온보딩을 촉진하기 위한 내부 지식 기반을 구축하여 모범 사례가 프로젝트 전반에 걸쳐 보존되고 재사용되도록 하십시오.

미래 대비 및 확장성 계획

신규 하드웨어 및 소프트웨어 기술을 쉽게 통합할 수 있는 모듈식이며 적응 가능한 시스템 아키텍처를 설계하십시오. 확장성 요구 사항을 지속적으로 평가하고 워크로드 증가에 따라 경쟁력 있는 성능을 유지하기 위해 최적화 전략을 업데이트하십시오.

시스템 아키텍처 및 하드웨어 계획

하드웨어, 상호 연결(interconnects), 데이터 경로( )는 성능과 비용 효율성의 상한선을 설정합니다. 소프트웨어 조정만으로는 자원이 부족한 GPU를 능가할 수 없습니다. 가속기, CPU/DRAM/I/O, 냉각/전원 공급 장치를 워크로드에 맞춰 달러/와트당 유효 처리량(goodput)을 계획하여 초기부터 병목 현상을 방지하십시오.

구체적으로, 순수한 FLOPS가 아닌 달러/와트당 유용한 작업량인 '굿풋(goodput)'을 목표로 설계하십시오. 가속기와 인터커넥트를 워크로드에 맞추고, GPU에 지속적으로 데이터를 공급하기 위해 CPU/메모리/I/O를 적정 규모로 구성하며, 데이터를 로컬에 유지하고, 하드웨어가 최대 클럭을 유지할 수 있도록 전원/냉각을 계획하십시오. GPU를 추가하기 전에 확장 효율성을 평가하십시오. 시스템 아키텍처 최적화와 하드웨어 계획 효율성 향상을 위한 몇 가지 팁은 다음과 같습니다 :

효율성과 처리량 향상을 위한 설계

유용한 처리량을 목표로 삼으십시오. 성능 향상의 모든 부분은 대규모에서 막대한 비용 절감으로 이어집니다. 단순한 원시 FLOPS가 아닌 달러/와트당 생산적인 작업량 극대화에 집중하십시오.

적합한 가속기 선택

우수한 와트당 성능과 메모리 용량을 위해 최신 GPU를 선호하십시오. 최신 아키텍처는 네이티브 FP8 및 FP4 정밀도 지원과 훨씬 빠른 상호 연결 기능을 제공합니다. 이는 구형 GPU 및 시스템 대비 큰 속도 향상을 가져옵니다.

고대역폭 상호 연결 활용

다중 GPU 워크로드에는 PCIe 전용 연결 대신 GB200/GB300 NVL72와 같은 NVLink/NVSwitch 지원 시스템을 사용하십시오. NVLink 5는 최대 1.8TB/s의 양방향 GPU 간 대역폭(PCIe Gen5 대비 14배 이상)을 제공하여 GPU 간 거의 선형적인 확장성을 가능하게 합니다. NVLink 스위치 도메인은 2단계 스위치로 확장하여 하나의 NVLink 도메인에 최대 576개의 GPU를 연결할 수 있습니다. 이를 통해 계층적 콜렉티브 연산이 가능해지며, 랙 간 패브릭으로 전환되기 전까지 최대한 오랫동안 NVLink를 유지합니다.

CPU/GPU 및 메모리 비율 균형 조정

GPU당 충분한 CPU 코어, DRAM 및 스토리지 처리량을 확보하십시오. 예를 들어, 데이터 로딩 및 네트워킹 작업을 위해 GPU당 약 1개의 고속 CPU 코어를 할당하십시오. 시스템 RAM 및 I/O가 GPU당 수백 MB/s 수준의 요구 속도로 GPU에 데이터를 공급할 수 있도록 하여 자원 부족을 방지하십시오.

데이터 로컬리티 계획 수립

다중 노드에서 훈련할 경우 노드 간 통신을 최소화하십시오. 가능한 경우, 완전한 대역폭을 활용하기 위해 긴밀하게 결합된 워크로드를 동일한

NVLink/NVSwitch 도메인에 유지하고, 접근 가능한 최고 속도 인터커넥트를 사용하십시오. 이상적으로는 노드 내 및 랙 내 통신에는 NVLink를, 랙 간 통신에는 InfiniBand를 사용하십시오.

체인 내 병목 현상 방지

가장 느린 링크(CPU, 메모리, 디스크, 네트워크 등)를 식별하여 확장하십시오. 예를 들어, I/O로 인해 GPU 활용도가 낮다면 GPU를 추가하기보다 더 빠른 스토리지나 캐싱에 투자하십시오. 모든 구성 요소가 잘 조화된 종단 간 설계는 GPU 사이클 낭비를 방지합니다.

적절한 클러스터 크기 선택

GPU 추가 시 감소하는 수익률에 주의하십시오. 특정 클러스터 규모를 넘어서면 오버헤드가 증가할 수 있으므로, 속도 향상이 비용을 정당화하는지 확인하십시오. 예를 들어*2N* GPU로 확장하기 전에 *N* GPU에서 95% 사용률에 도달하는 등 활용도를 최적화하는 것이 종종 더 낫습니다.

냉각 및 전력 설계를 고려

데이터센터가 GPU의 열 및 전력 요구사항을 처리할 수 있는지 확인하십시오. GB200/GB300과 같은 고성능 시스템은 매우 높은 TDP를 가집니다. GPU가 스로틀링 없이 부스트 클럭을 유지할 수 있도록 적절한 냉각(액체 기반일 가능성이 높음) 및 전력 공급을 제공하십시오.

통합 CPU-GPU "슈퍼칩" 아키텍처

통합 메모리 및 패키지 내 링크 를 활용하면 적절한 데이터를 적절한 계층에 배치함으로써 더 큰 모델을 수용하고 복사 오버헤드를 줄일 수 있습니다. Grace를 전처리용으로, HBM을 '핫' 텐서용으로 사용하면 슈퍼칩이 더 적은 스톨을 가진 긴밀하게 결합된 엔진으로 변모합니다.

Grace Blackwell 슈퍼칩에서는 CPU와 GPU를 공유 메모리 복합체로 간주하십시오. 핫 웨이트/활성화는 HBM에 보관하고, 오버플로우 또는 빈번하지 않은 데이터는 NVLink-C2C를 통해 Grace LPDDR에 보관하십시오.사전 처리/오케스트레이션에는 패키지 내 Grace CPU를 사용하고, 초대형 모델의 지연 시간을 숨기기 위해 프리페치 또는 파이프라인 관리 메모리를 활용하십시오. 다음과 같이 슈퍼칩 아키텍처의 장점을 활용하십시오:

통합 CPU-GPU 메모리 활용

Grace Blackwell(GB200/GB300) 슈퍼칩의 통합 메모리 공간을 활용하십시오. 두 개의 Blackwell GPU와 72코어 Grace CPU는 NVLink-C2C(900 GB/s)로 연결된 일관성 있는 메모리 풀을 공유합니다. 대형 모델의 경우 CPU의 대용량 메모리(예: 480 GB LPDDR5X)를 확장 메모리로 활용하되, 속도 확보를 위해 "핫" 데이터는 GPU의 HBM에 보관하십시오.

로컬리티를 위한 데이터 배치

통합 메모리를 사용하더라도 데이터 배치 우선순위를 지정하십시오. 모델 가중치, 활성화 값 및 기타 자주 액세스되는 데이터는 로컬 대역폭이 훨씬 높은 GPU HBM3e에 배치하고, 드물게 사용되거나 오버플로 데이터는 CPU RAM에 보관하십시오. 이렇게 하면 900 GB/s NVLink-C2C 링크가 중요한 데이터의 병목 현상이 되지 않습니다.

가능할 경우 CPU-GPU 직접 메모리 접근 활용

GB200 및 GB300과 같은 통합 CPU-GPU 슈퍼칩에서 GPU가 CPU 메모리에 직접 접근할 수 있는 기능을 활용하십시오. GPU는 호스트 PCIe를 통한 스테이징 없이 NVLink-C2C를 통해 Grace LPDDR 메모리를 일관성 있게 읽고 쓸 수 있습니다. 대역폭과 지연 시간은 여전히 HBM보다 낮으므로, 관리 포인터를 미리 가져오고 데이터를 스테이징하며 전송을 파이프라인화하여 지연 시간을 숨기십시오. 따라서 핫 활성화 및 KV 캐시는 HBM에 유지하고, 명시적 프리페치를 통해 CPU 메모리를 하위 계층 캐시로 사용하는 것이 권장됩니다.

Grace CPU를 효과적으로 사용하십시오

패키지 내장 Grace CPU는 72개의 고성능 코어를 제공합니다—이를 활용하세요! 데이터 전처리, 증강 및 기타 CPU 친화적 작업을 이 코어로 오프로드하십시오. NVLink-C2C를 통해 GPU에 신속하게 데이터를 공급할 수 있어, 본질적으로 GPU를 위한 초고속 I/O 및 컴퓨팅 보조 장치 역할을 합니다.

초대형 모델 계획 수립

GPU 메모리를 초과하는 1조 매개변수 모델 훈련의 경우, GB200/GB300 시스템은 모델 메모리 풀의 일부로 CPU 메모리를 활용하여 훈련할 수 있도록 지원합니다. 프레임워크 캐싱 할당기를 선호하고 커스텀 코드에서는 `cudaMallocAsync` 를 사용하여 조각화를 최소화하고 그래프 캡처를 활성화하세요. CUDA 통합 메모리 또는 관리형 메모리 API를 사용하여 오버플로를 우아하게 처리하고, 지연 시간을 숨기기 위해 CPU → GPU 메모

리에서 다가오는 레이어의 명시적 프리페칭(예: `cudaMemPrefetchAsync`)을 고려하십시오.

슈퍼칩 최적화 알고리즘 고려

SuperOffload는 오프로드 및 텐서 캐스트/복사 전략의 효율성 향상에 초점을 맞춘 슈퍼칩 최적화 알고리즘 세트의 예입니다. 이노베이션에는 추측 후 검증(STV), 이종 최적화 계산, ARM 기반 CPU 최적화기가 포함됩니다. NVIDIA 슈퍼칩(예: Grace Hopper, Grace Blackwell, Vera Rubin)을 위해 특별히 설계된 SuperOffload는 기존 오프로드 전략 대비 토큰 처리 처리량과 칩 활용도를 높입니다.

다중 GPU 확장성 및 상호 연결 최적화

확장은 통신 이 빠르고 토폴로지를 인식할 때만 효과가 있습니다. 그렇지 않으면 추가된 GPU들이 서로를 기다리기만 합니다. NVLink/NVSwitch 대역폭, 현대적인 콜렉티브, 패브릭 인식 배치에 의존하여 선형적인 속도 향상에 접근하세요.

구체적으로, NVLink/NVSwitch 도메인(예: NVL72)을 활용하여 거의 선형적인 스케일링을 달성하고 패브릭에 적합한 병렬화 전략을 선택하십시오. 토폴로지 인식 배치, 업데이트된 NCCL 콜렉티브(예: PAT), 텔레메트리 기능을 사용하여 GPU당 약 1.8TB/s 양방향 처리량 대역폭을 효과적으로 활용하고 있는지 확인하십시오. 확장 시 계층적 통신을 계획하십시오. 다음은 상호 연결 및 토폴로지 최적화를 통한 다중 GPU 확장 활용 팁입니다:

고속 전-대-전 토폴로지 설계

예를 들어, 72개의 완전히 상호 연결된 GPU를 갖춘 NVL72 NVSwitch 클러스터에서는 모든 GPU가 다른 GPU와 NVLink 5의 최대 속도로 통신할 수 있습니다. 패브릭 수준에서 NVLink 스위치 도메인은 비차단 방식입니다. 애플리케이션 수준 처리량은 동시 트래픽 및 경로 스케줄링에 따라 달라질 수 있으므로, 쌍별 포화 상태를 가정하기 전에 DCGM NVLink 카운터와 Nsight Systems 추적을 사용하여 동작을 확인하십시오. 데이터 병렬, 텐서 병렬, 파이프라인 병렬화와 같이 하위 상호 연결에서는 병목 현상이 발생할 수 있는 병렬화 전략을 활용하여 이 토폴로지의 이점을 누리십시오.

토폴로지 인식 스케줄링 활용

가능한 경우 항상 다중 GPU 작업을 NVLink 스위치 도메인 내에 배치하십시오. 작업의 모든 GPU를 NVL72 패브릭에 유지하면 통신 집약적 워크로드에 대해 거의 선형적인 확장성을 제공합니다. NVLink 도메인 간 또는

표준 네트워크를 통해 GPU를 혼합하면 병목 현상이 발생하므로 밀접하게 결합된 작업에서는 피해야 합니다.

전례 없는 대역폭 활용

NVLink 5는 GPU당 양방향 900GB/s 대역폭을 제공하며, 이는 이전 세대 대비 GPU당 대역폭이 두 배로 증가한 것입니다. NVL72 랙은 총 랙 내 대역폭으로 약 130TB/s를 제공합니다. 이는 수십 기가바이트의 기울기 데이터도 1.8TB/s 속도로 수 밀리초 내에 올-리듀스(all-reduce)가 가능해 통신 대기 시간을 획기적으로 줄입니다. 기울기 동기화 및 매개변수 샤딩과 같은 훈련 알고리즘을 설계하여 이 상대적으로 여유로운 통신 예산을 최대한 활용하십시오.

현대적 집단 알고리즘 도입

NVSwitch에 최적화된 최신 NVIDIA NCCL 라이브러리를 활용하십시오. 특히 NVLink 스위치 토폴로지를 위해 도입된 병렬 집계 트리(PAT) 알고리즘을 활성화하십시오. 이는 NVL72 토폴로지를 활용하여 다른 트리/링 알고리즘보다 효율적으로 축소 연산을 수행함으로써 동기화 시간을 추가로 단축합니다.

미세 병렬성 고려

전체 대역폭의 전수 통신 연결성을 바탕으로, 이전에는 실현 불가능했던 세분화된 모델 병렬화를 고려하십시오. 예를 들어, 각 GPU가 다른 모든 GPU와 양방향으로 1.8TB/s의 처리량을 가질 경우, 레이어별 병렬화나 다중 GPU 간 텐서 병렬화가 효율적일 수 있습니다. 이전에는 과도한 GPU 간 통신을 피했을 수 있지만, NVL72는 네트워크 한계에 부딪히지 않고 작업을 공격적으로 분할할 수 있게 합니다.

포화 상태 모니터링

NVL72는 극도로 빠르지만 자질 시 링크 활용도를 주시하세요. 예를 들어 극단적인 전수 통신 작업으로 NVSwitch가 포화 상태에 이르면, 기울기 집계 등을 통해 통신을 제한해야 할 수 있습니다. NVIDIA 도구 또는 NVSwitch 텔레메트리 데이터를 활용해 통신이 NVLink 용량 범위 내에 있는지 확인하고 필요 시 패턴을 조정하세요. 예를 들어, 네트워크 경합을 피하기 위해 모든 대 모든 교환을 시간차를 두고 수행할 수 있습니다. DCGM은 NVLink 카운터를 노출하여 링크 균형을 확인하고 집단 연산 중 핫스팟을 탐지하는 데 도움을 줍니다.

향후 확장을 계획하십시오

NVLink 스위치는 단일 랙을 넘어 확장 가능하다는 점을 유의하십시오. 2단계 스위치를 사용하면 하나의 연결된 도메인 내에서 최대 576개의 GPU까지 확장할 수 있습니다. 이러한 초대규모 환경에서 운영할 경우, 먼저 로컬 NVL72 랙 간 집단 연산을 활용한 계층적 통신을 계획하고, 필요한 경우에만 랙 간 상호 연결을 사용하십시오. 이는 랙 내 NVLink 사용을 우선적으로 극대화하여 랙 간 InfiniBand 홉에 의존하기 전에 가장 빠른 링크를 활용하도록 보장합니다.

연합 및 분산 최적화 기회 파악

멀티클라우드 또는 에지-투-클라우드 환경과 같이 이기종 환경을 아우르는 배포의 경우, 적응형 통신 프로토콜과 동적 부하 분산 전략을 채택하십시오. 이는 분산 시스템 전반에서 지연 시간을 최소화하고 처리량을 극대화하여, 리소스의 성능과 용량이 상이한 경우에도 안정적인 성능을 보장합니다.

운영체제 및 드라이버 최적화

OS 지터, NUMA 미스, 드라이버 불일치( )는 처리량을 은밀히 소모하고 조정 불가능한 변동성을 유발합니다. 스택 강화(대용량 페이지, 어피니티, 일관된 CUDA/드라이버, 지속성)는 안정적이고 고성능의 기준점을 제공합니다.

HPC에 최적화된 간결한 Linux를 실행하십시오. NUMA/IRQ 어피니티를 설정하고 THP 및 높은 메모리 잠금(memlock)을 활성화하십시오. 노드 전반에 걸쳐 NVIDIA 드라이버/CUDA를 일관되게 유지하십시오. 시스템 지터를 격리하고, CPU 라이브러리/저장소를 조정하며, 컨테이너 제한을 올바르게 설정하고, 예측 가능한 처리량을 위해 BIOS/펌웨어/NVSwitch 패브릭을 최신 상태로 유지하십시오. 환경에서 탐색해야 할 호스트, OS 및 컨테이너 최적화 사항은 다음과 같습니다:

HPC에 최적화된 Linux 커널 사용

GPU 서버가 고성능 컴퓨팅에 최적화된 최신 안정판 Linux 커널을 실행하도록 하십시오. CPU 또는 I/O를 소모하는 불필요한 백그라운드 서비스를 비활성화하십시오. GPU에 공급하기 위해 CPU 코어를 높은 클럭 상태로 유지하려면 "on-demand"나 "power-save" 대신 "performance" CPU 거버너를 사용하십시오.

성능이 중요한 워크로드에 대해 스왑을 비활성화하십시오

페이지 스래싱을 방지하기 위해 훈련 서버에서 스왑을 비활성화하십시오. 스왑을 반드시 활성화해야 하는 경우, `mlock` 또는 `cudaHostAlloc` 를 사용하여 중요한 버퍼를 RAM에 고정하십시오.

적극적인 사전 할당으로 메모리 조각화 방지

자주 사용되는 텐서를 위해 크고 연속적인 메모리 블록을 사전 할당하여 런타임 할당 오버헤드와 조각화를 줄이십시오. 이 사전 예방적 전략은 장시간 훈련 실행 중 더 안정적이고 효율적인 메모리 관리를 보장합니다.

CPU 라이브러리를 위한 환경 변수 최적화

`OMP_NUM_THREADS` 및 `MKL_NUM_THREADS` 와 같은 매개변수를 하드웨어 구성에 맞게 미세 조정하십시오. 이러한 변수를 조정하면 스레드 경합을 줄이고 CPU에 바운디드한 작업의 병렬 효율성을 향상시킬 수 있습니다.

NUMA 인식 설계

다중 NUMA 서버의 경우 GPU 프로세스/스레드를 로컬 NUMA 노드의 CPU에 고정하십시오. `numactl` 또는 taskset과 같은 도구를 사용하여 각 훈련 프로세스를 할당된 GPU에 가장 가까운 CPU에 바인딩하십시오. 마찬가지로 메모리 할당을 로컬 NUMA 노드(`numactl --membind`)에 바인딩하여 GPU DMA용 호스트 메모리가 가장 가까운 RAM에서 제공되도록 하십시오. 이는 효과적인 PCIe/NVLink 대역폭을 절반으로 줄일 수 있는 비용이 많이 드는 크로스-NUMA 메모리 트래픽을 방지합니다.

네트워크 및 GPU 작업에 IRQ 어피니티 활용

NIC 인터럽트를 NIC와 동일한 NUMA 노드의 CPU 코어에 명시적으로 바인딩하고, 마찬가지로 GPU 드라이버 스레드를 전용 코어에 고정하십시오. 여기에는 `nvidia-persistence` 서비스 데몬과 같은 장기 실행 서비스의 코어도 포함됩니다. 이 전략은 NUMA 간 트래픽을 최소화하고 고부하 상태에서 성능을 안정화합니다.

투명 거대 페이지 활성화

항상 또는 `madvise` 모드에서 투명 대형 페이지(THP)를 활성화하여 대용량 메모리 할당 시 2MB 페이지를 사용하도록 합니다. 이는 프레임워크를 위해 수십~수백 GB의 호스트 메모리를 할당할 때 TLB 스래싱과 커널 오버헤드를 줄입니다.
`/sys/kernel/mm/transparent_hugepage/enabled` 를 확인하여 THP가 활성화되었는지 검증하십시오. THP가 활성화되면 프로세스가 대

용량 할당에 대형 페이지를 사용합니다. 작업 부하가 지연 시간에 민감하고 지터 현상이 관찰될 경우 `madvise` 모드의 THP를 우선적으로 사용하십시오.

최대 고정 메모리 증가

대용량 고정(페이지 잠금) 할당을 허용하도록 OS를 구성하십시오. GPU 애플리케이션은 종종 더 빠른 전송을 위해 메모리를 고정합니다. 데이터 로더가 OS 제한에 걸리지 않고 고정 버퍼를 할당할 수 있도록 `` `ulimit -l unlimited` ``을 설정하거나 높은 값을 지정하십시오. 이는 GPU DMA 속도를 저하시키는 페이지 가능 메모리로의 실패 또는 전환을 방지합니다.

최신 NVIDIA 드라이버 및 CUDA 스택 사용

모든 노드에서 NVIDIA 드라이버와 CUDA 런타임을 최신 상태로 유지하십시오(테스트된 안정 버전 범위 내에서). 새 드라이버는 성능 향상을 가져올 수 있으며, 새로운 GPU의 컴퓨팅 기능을 사용하려면 필수입니다. 다중 노드 작업에서 불일치를 방지하려면 모든 노드가 동일한 드라이버/CUDA 버전을 사용하도록 하십시오. 부팅 시 GPU에 지속성 모드(`nvidia-smi -pm 1`)를 활성화하여 드라이버가 계속 로드되고 GPU가 재초기화 지연을 겪지 않도록 하십시오. 모든 노드에서 NVIDIA 드라이버 및 툴킷을 업데이트하여 버그 수정 사항과 성능 개선 사항을 적용하십시오.

MIG 구성 사용 시 GPU 지속성 활성화

지속성 모드가 활성화되면 GPU가 "워밍업" 상태로 유지되어 작업 시작 지연 시간을 줄입니다. 이는 특히 멀티 인스턴스 GPU(MIG) 파티셔닝을 사용할 때 중요합니다. 지속성 모드가 없으면 MIG 구성이 작업마다 재설정되지만, 드라이버를 활성화 상태로 유지하면 슬라이스가 보존됩니다. MIG 사용 시 항상 지속성 모드를 구성하십시오.

시스템 작업 격리

각 서버에서 인터럽트 처리 및 백그라운드 데몬과 같은 OS 관리 작업을 위해 코어 하나 또는 소수의 코어를 전용으로 할당하십시오. 이렇게 하면 GPU에 데이터를 공급하는 주요 CPU 스레드가 중단되지 않습니다. CPU 격리 또는 cgroup 고정 기능을 사용하여 구현할 수 있습니다. OS 지터를 제거하면 일관된 처리량을 보장합니다.

시스템 I/O 설정 최적화

작업 부하에서 로깅이나 체크포인트 작업이 많다면, 처리량에 유리한 옵션으로 파일 시스템을 마운트하십시오. 데이터 디스크에는 디스크 디스크(`noatime`) 사용을 고려하고, 스트리밍 읽기를 위해 파일 시스템의 미리 읽기(read-ahead)를 늘리십시오. NVMe SSD의 경우 지연 시간 변동성을 줄이기 위해 디스크 스케줄러가 적절한 디스크 디스크(`mq-deadline`) 또는 디스크 디스크(`noop`)를 사용하도록 설정되었는지 확인하십시오.

정기적인 유지보수 수행

성능 개선을 위해 BIOS/펌웨어를 최신 상태로 유지하십시오. 일부 BIOS 업데이트는 PCIe 대역폭을 향상시키거나 GPU용 IOMMU(입출력 메모리 관리 장치) 문제를 해결합니다. 또한 NVIDIA에서 제공하는 Fabric Manager 업그레이드 등과 같이 해당되는 경우 NIC 및 NVSwitch/Fabric의 펌웨어 업데이트를 주기적으로 확인하십시오. 사소한 펌웨어 조정으로 때로는 모호한 병목 현상이나 안정성 문제를 해결할 수 있습니다.

최대 성능을 위해 Docker 및 쿠버네티스 구성을 조정하십시오

컨테이너에서 실행할 때는 공유 메모리를 위해 ` `--ipc=host` `와 같은 옵션을 추가하고, 메모리 잠금 문제를 방지하기 위해 ` `--ulimit memlock=-1` `를 설정하십시오. 이렇게 하면 컨테이너화된 프로세스가 OS가 부과하는 제한 없이 메모리에 접근할 수 있습니다.

GPU 리소스 관리 및 스케줄링

더 스마트한 배치 및 파티셔닝 은 새 하드웨어 구매 없이 활용도를 높이고 혼합 워크로드의 예측 가능성을 보호합니다. 토폴로지를 고려하고, 적절한 경우 MPS/MIG를 사용하며, 클럭/전력을 제어하여 경합 및 꼬리 지연 시간을 최소화하세요.

GPU/NUMA/NVLink 토폴로지를 고려하여 스케줄링하고, MPS 또는 MIG를 사용하여 신뢰성을 위한 ECC 및 지속성을 유지하면서 소규모 작업의 활용도를 높이십시오. 필요 시 안정성을 위해 클럭 또는 전력 제한을 잠그고, CPU 과도한 할당을 피하며, 경합 없이 ROI를 극대화하기 위해 작업을 지능적으로 압축하십시오. 다음은 GPU 리소스 관리 및 스케줄링 팁입니다:

토폴로지 인식 작업 스케줄링

쿠버네티스 및 SLURM과 같은 오케스트레이터가 NUMA 및 NVLink 경계를 준수하는 노드에 컨테이너를 스케줄링하여 크로스-NUMA 및 크로

스-NVLink 도메인 메모리 액세스를 최소화하도록 하십시오. 이러한 정렬은 지연 시간을 줄이고 전체 처리량을 향상시킵니다.

다중 프로세스 서비스(MPS)

단일 GPU에서 다중 프로세스를 실행할 때 NVIDIA MPS를 활성화하여 활용도를 개선하십시오. MPS는 서로 다른 프로세스의 커널이 시간 분할 대신 GPU에서 동시 실행되도록 합니다. 개별 작업이 GPU를 완전히 포화시키지 못하는 경우 유용합니다. 예를 들어 MPS를 사용해 단일 GPU에서 4개의 훈련 작업을 실행하면 작업이 중첩되어 전체 처리량이 향상됩니다.

다중 인스턴스 GPU(MIG)

고성능 GPU를 여러 작업용 소형 인스턴스로 분할하려면 MIG를 사용하십시오. 소형 모델 추론이나 다수 실험 실행과 같은 경량 워크로드가 많은 경우 GPU를 분할하여 각 작업에 보장된 리소스를 확보할 수 있습니다. 예를 들어 최신 GPU는 여러 MIG 슬라이스(최대 7개)로 분할 가능합니다. 밀접하게 결합된 병렬 작업에는 MIG를 사용하지 마십시오. 이러한 작업은 전체 GPU 접근이 유리합니다. 작업 규모가 전체 GPU보다 작을 때 격리 및 GPU 투자 수익률 극대화를 위해 MIG를 배포하십시오.

MIG 지속성

작업 간 MIG 파티션을 유지하려면 지속성 모드를 켜두십시오. 이렇게 하면 재파티셔닝 오버헤드를 피하고 후속 작업이 지연 없이 예상된 GPU 슬라이스를 볼 수 있습니다. 스케줄링이 예측 가능하도록 클러스터 부팅 시 MIG를 구성하고 활성화 상태로 유지하십시오. 실행 중인 작업 중 MIG 구성을 변경하려면 GPU를 재설정해야 하므로 진행 중인 작업이 중단될 수 있습니다. MIG 장치 파티션은 재부팅 시 GPU에 의해 유지되지 않으므로 유지보수 시간을 계획하십시오. NVIDIA의 MIG Manager를 사용하여 부팅 시 원하는 레이아웃을 자동으로 재구성하십시오.

GPU 클럭 및 전력 설정

실행 간 일관성이 필요한 경우, 자동 부스트(`nvidia-smi -lgc`)를 사용하지 않고 GPU 클럭을 고정된 높은 주파수로 고정하는 것을 고려하십시오. 기본적으로 GPU는 자동 부스트를 사용하며 이는 일반적으로 최적이지만, 고정 클럭은 일시적인 다운클럭킹을 방지할 수 있습니다 `-lmc` 실행 간 일관성이 필요한 경우 고려하십시오. 기본적으로 GPU는 자동 부스트를 사용하며 이는 일반적으로 최적이지만, 고정 클럭은 일시적인 다운클럭킹을 방지할 수 있습니다. 전력 제약이 있는 시나리오에서는 GPU를 안정적인 열/전력 범위 내에 유지하기 위해 약간 언더클럭하거나 전력 제

한을 설정할 수 있습니다. 이는 가끔 발생하는 스로틀링이 문제였던 경우 일관된 성능을 제공할 수 있습니다.

ECC 메모리

특정 사유가 없는 한 데이터 센터 GPU에서 ECC를 활성화 상태로 유지하십시오. 성능 저하 비용은 미미합니다(대역폭 및 메모리 손실 몇 퍼센트 수준). 그러나 ECC는 메모리 오류를 포착하여 장시간 훈련 작업이 손상되는 것을 방지합니다. 대부분의 서버 GPU는 기본적으로 ECC가 활성화된 상태로 출하됩니다. 다주간 훈련을 보호하기 위해 활성화 상태를 유지하십시오.

작업 스케줄러 인식

SLURM이나 쿠버네티스 같은 작업 스케줄러에 GPU 토폴로지를 통합하세요. 저지연 커플링이 필요한 경우 동일한 노드나 동일한 NVSwitch 그룹에 작업을 할당하도록 스케줄러를 구성하세요. 소규모 작업의 MIG 슬라이스 스케줄링에는 쿠버네티스 디바이스 플러그인이나 SLURM Gres를 사용하세요. GPU 인식 스케줄러는 단일 작업이 멀리 떨어진 GPU에 걸쳐 배치되어 대역폭 문제를 겪는 상황을 방지합니다.

CPU 오버서브스크립션

작업 스케줄링 시 데이터 로딩 스레드 등 각 GPU 작업의 CPU 요구량을 고려하십시오. CPU가 처리할 수 있는 양보다 많은 GPU 작업을 노드에 배치하지 마십시오. 모든 GPU가 자원이 부족해지는 CPU 과부하보다 GPU를 유휴 상태로 두는 것이 더 낫습니다. GPU 작업별 CPU 사용률을 모니터링하여 스케줄링 결정에 활용하십시오.

NVSwitch용 NVIDIA 패브릭 관리자 사용

NVSwitch가 탑재된 시스템에서는 GB200/GB300 NVL72 랙이 NVIDIA Fabric Manager의 실행을 보장합니다. 이 서비스는 NVSwitch 토폴로지와 라우팅을 관리합니다. 이 서비스가 없으면 대규모 작업에서 다중 GPU 간 통신이 완전히 최적화되지 않거나 실패할 수 있습니다. Fabric Manager 서비스는 일반적으로 NVSwitch가 장착된 서버에서 기본적으로 실행되지만, 특히 드라이버 업데이트 후에는 활성화 및 실행 상태를 반드시 재확인해야 합니다.

활용도를 위한 작업 패킹

작업 지정을 지능적으로 구성하여 활용도를 극대화하십시오. 예를 들어, 4-GPU 노드에서 CPU 사용량이 적은 2-GPU 작업 두 개가 있다면, 동일한 컴퓨팅 노드 또는 NVLink 지원 랙 내에서 함께 실행하면 리소스를 절약할

수 있으며, 더 빠른 NVLink를 통한 통신도 가능합니다. 반대로, 노드의 메모리 또는 I/O 용량을 초과하는 작업을 함께 배치하지 마십시오. 목표는 경합 없이 하드웨어 활용도를 높이는 것입니다.

I/O 최적화

데이터 공급이 따라가지 못하면 GPU는 유휴 상태가 됩니다— 에서 가장 크고 저렴한 성능 향상은 연산이 아닌 입력 처리 개선에서 비롯됩니다. 병렬 처리, 고정 메모리, 비동기 전송, 고속 스토리지를 통해 모델에 지속적으로 데이터를 공급하세요.

데이터 로더 병렬화, 고정 메모리 및 비동기 전송 활용, 고속 NVMe 저장소(가급적 GPUDirect Storage 사용)를 통해 GPU에 지속적으로 데이터를 공급하세요. 스트라이핑, 캐싱, 압축을 현명하게 수행하십시오. 클러스터 규모에 따라 I/O가 확장되도록 종단 간 처리량을 측정하고, 체크포인트/로그를 비동기적으로 작성하세요. 데이터 파이프라인을 위한 I/O 최적화 팁은 다음과 같습니다:

병렬로 데이터 로드

GPU용 데이터 로딩 및 전처리를 위해 다중 작업자/스레드를 사용하십시오. 기본값인 1~2개의 데이터 로더 작업자는 부족할 수 있습니다. 예를 들어 PyTorch의 `DataLoader(num_workers=N)`를 사용하여 자질하고 데이터 로더 프로세스/스레드 수를 데이터 입력이 더 이상 병목 현상이 되지 않을 때까지 늘리십시오. 고코어 수 CPU는 GPU에 데이터를 공급하기 위해 존재하므로 이를 활용하십시오.

I/O용 호스트 메모리 고정

데이터 전송 버퍼에 고정 메모리(페이지 고정 메모리)를 활성화하세요. PyTorch의 DataLoader처럼 GPU가 직접 DMA할 수 있는 호스트 메모리를 할당하는 옵션이 많은 프레임워크에 있습니다(`pin_memory=True` 참조). 고정 메모리 사용은 H2D 복사 처리량을 크게 향상시킵니다. 비동기 전송과 결합하여 데이터 로딩과 계산을 중첩하세요.

컴퓨팅과 데이터 전송 중첩

입력 데이터를 파이프라인화하세요. GPU가 배치 N을 계산하는 동안 CPU에서 배치 N+1을 로드 및 준비하고, CUDA 스트림과 비차단 `cudaMemcpyAsync` 을 사용하여 백그라운드에서 전송하세요. 이 이중 버퍼링은 지연 시간을 숨깁니다—GPU는 이상적으로 데이터를 기다리지 않습니다. 훈련 루프가 비동기 전송을 사용하도록 하세요. 예를 들어

PyTorch에서는 `non_blocking=True`로 텐서를 GPU로 복사할 수 있습니다. 비동기 전송은 데이터 전송이 백그라운드에서 진행되는 동안 CPU가 계속 실행되도록 합니다. 이는 데이터 전송과 계산을 중첩시켜 성능을 향상시킵니다.

고속 스토리지(NVMe/SSD) 사용

훈련 데이터는 빠른 로컬 NVMe SSD 또는 고성능 병렬 파일 시스템에 저장하십시오. 회전 디스크는 처리량을 심각하게 제한합니다. 사용 가능한 경우 GPUDirect Storage(GDS)를 활성화하여 GPU가 CPU를 우회하고 NVMe 또는 네트워크 스토리지에서 직접 데이터를 스트리밍할 수 있도록 하십시오. 이는 대규모 데이터셋 읽기 시 I/O 지연 시간과 CPU 부하를 추가로 줄여줍니다. 대규모 데이터셋의 경우 각 노드가 데이터의 로컬 복사본 또는 샤드를 보유하도록 고려하십시오. 네트워크 스토리지를 사용할 경우 스트라이핑이 가능한 Lustre와 같은 분산 파일 시스템이나 다수의 클라이언트에 병렬로 서비스를 제공할 수 있는 객체 스토리지를 선호하십시오.

I/O 동시성 및 스트라이핑 조정

단일 파일 접근으로 인한 병목 현상을 피하십시오. 모든 작업자가 하나의 대용량 파일을 사용하는 경우, 여러 저장 대상에 스트라이핑하거나 여러 서버가 동시에 제공할 수 있도록 쪼개십시오. 예를 들어, 데이터셋을 여러 파일로 분할하고 각 데이터 로더 작업자가 서로 다른 파일을 동시에 읽도록 합니다. 이렇게 하면 저장 시스템의 총 대역폭을 극대화할 수 있습니다.

소규모 파일 액세스 최적화

데이터셋이 수백만 개의 소형 파일로 구성될 경우 메타데이터 오버헤드를 완화하십시오. 초당 너무 많은 소형 파일을 열면 파일 시스템의 메타데이터 서버가 과부하될 수 있습니다. 소형 파일을 더 큰 컨테이너(예: `tar` 또는 `RecordIO` 파일)로 압축하거나, 일괄 읽기를 지원하는 데이터 수집 라이브러리를 사용하거나, 클라이언트에서 메타데이터 캐싱이 활성화되었는지 확인하십시오. 이렇게 하면 파일당 오버헤드가 감소하고 에포크 시작 시간이 단축됩니다.

가능한 경우 클라이언트 측 캐싱 사용

캐싱 계층을 적극 활용하십시오. NFS 사용 시 클라이언트 캐시 크기와 유효 기간을 늘리십시오. 분산 파일 시스템의 경우 캐싱 데몬을 고려하거나 데이터 세트 일부를 로컬 디스크에 수동으로 캐싱하는 것도 방법입니다. 목표는 느린 소스에서 동일한 데이터를 반복적으로 읽는 것을 방지하는

것입니다. 각 노드가 서로 다른 시간에 동일한 파일을 처리하는 경우 로컬 캐시는 중복 I/O를 크게 줄일 수 있습니다.

데이터를 현명하게 압축

I/O가 병목 현상인 경우 데이터셋을 압축하여 저장하되, LZ4 또는 Zstd 고속 모드와 같은 경량 압축 방식을 사용하십시오. 이는 일부 CPU를 희생하여 I/O 양을 줄이는 방식입니다. 압축 해제로 인해 CPU가 병목 현상이 되면, 멀티스레드 압축 해제 또는 가속기로의 오프로딩을 고려하십시오. 또한 압축된 데이터를 읽는 스레드와 데이터를 병렬로 압축 해제하는 스레드를 사용하여 압축 해제와 읽기를 중첩하십시오. 최신 GPU는 GPUDirect Storage 및 `cuFile` I/O 스택과 결합 시 GPU 컴퓨팅 리소스(또는 이미지/비주얼 데이터용 전용 디코더)를 활용한 실시간 데이터 압축 해제 기능을 수행할 수 있습니다.

처리량 측정 및 병목 현상 제거

데이터 파이프라인의 처리량을 지속적으로 모니터링하십시오. GPU 활용률이 100%에 근접하지 않으면서 입력 지연이 의심될 경우, 디스크에서 읽는 속도(MB/s)와 데이터 로더 코어의 부하 상태를 측정하십시오. `dstat` 이나 NVIDIA의 DCGM 같은 도구를 사용하면 GPU가 데이터를 기다리는지 확인할 수 있습니다. 프리페치 버퍼를 늘리고, 네트워크 버퍼 크기를 증가시키며, 디스크 RAID 설정을 최적화하는 등 체계적으로 각 구성 요소를 조정하세요. 입력 파이프라인이 GPU가 소비하는 속도만큼 빠르게 데이터를 공급할 수 있을 때까지 이 작업을 반복합니다. 이러한 최적화는 I/O 지연을 제거함으로써 동일한 하드웨어에서 GPU 활용률을 약 70%에서 95% 이상으로 끌어올리는 경우가 많습니다.

다중 노드 환경에서 I/O 확장

클러스터 규모에서는 스토리지 시스템이 집계 처리량을 처리할 수 있도록 해야 합니다. 예를 들어, 각 GPU가 200MB/s를 소비하는 8개의 GPU는 노드당 1.6GB/s를 소비합니다. 100개 노드에 걸쳐서는 160GB/s가 필요합니다. 이를 지속할 수 있는 중앙 파일 시스템은 극히 드뭅니다. 데이터 분할을 통해 저장 서버에 분산하거나, 노드별 캐시를 활용하거나, 각 노드의 로컬 디스크에 데이터를 사전 로드하는 방식으로 완화하세요. 고가의 GPU 자원이 부족해지는 상황을 피하기 위해 저장 공간을 희생하고 처리량(예: 데이터 복제본 다중 생성)을 확보하는 것은 종종 가치 있는 선택입니다.

체크포인트 및 로깅 오버헤드 최소화

체크포인트와 로그를 효율적으로 작성하십시오. 가능하면 체크포인트에 비동기 쓰기를 사용하거나, 로컬 디스크에 작성한 후 네트워크 스토리지

로 복사하여 훈련 중단을 방지하십시오. 체크포인트를 압축하거나 스파스 저장 형식을 사용하여 크기를 줄이십시오. 반복 통계 집계를 통해 각 단계의 로깅 빈도를 제한하고, 매 반복마다 기록하지 않고 N번째 반복마다만 기록하십시오. 이는 I/O 오버헤드를 크게 줄일 것입니다.

`cuda-checkpoint` 와 사용자 공간 체크포인트/복원(CRIU)을 활용해 실행 중인 GPU 프로세스를 일시 중지하고 프로세스 이미지를 보존할 수도 있습니다. 재개 시 CUDA 드라이버가 장치 메모리와 CUDA 상태를 복원할 수 있으며, 동일한 장치 유형의 다른 GPU로도 복원 가능합니다. 이는 모델의 상태 딕셔너리나 분할 체크포인트 파일을 대체하기보다는 보완하는 것으로 간주하십시오.

데이터 처리 파이프라인

데이터의 형식, 레이아웃 및 로컬리티 는 대규모 환경에서 파이프라인이 얼마나 원활하게 실행되는지를 결정합니다. 바이너리 형식, 샤딩, 캐싱 및 우선순위 지정 스레드는 I/O를 병목 현상에서 안정적인 흐름으로 전환합니다.

데이터셋을 바이너리 또는 메모리 매핑 형식으로 변환하고, 스토리지와 노드에 샤딩하며, 스레드 우선순위를 높이거나 간단한 증분 작업을 GPU로 이동시켜 정체를 방지하세요. 핫 데이터/KV 상태를 캐싱하고, 적극적으로 프리페치 및 버퍼링하며, 디스크에서 장치까지 파이프라인이 원활하게 유지되도록 배치 크기를 조정하세요. 데이터 처리 개선을 위한 팁은 다음과 같습니다:

바이너리 데이터 형식을 사용하십시오

데이터셋을 TFRecords, LMDB 또는 메모리 매핑 배열과 같은 바이너리 형식으로 변환하세요. 이 변환은 수백만 개의 작은 파일을 처리하는 데 따른 오버헤드를 줄이고 데이터 수집 속도를 높입니다.

파일 시스템을 조정하십시오

`noatime` 로 파일 시스템을 마운트하고 미리 읽기(read-ahead)를 늘리는 것 외에도, I/O 부하를 분산하고 단일 서버의 병목 현상을 방지하기 위해 여러 스토리지 노드에 데이터를 샤딩하는 것을 고려하십시오.

CPU에 의존적인 바운디드 워크로드에 대해 하이퍼스레딩 비활성화

CPU에 크게 바운디드한 데이터 파이프라인의 경우 하이퍼스레딩을 비활성화하면 리소스 경합을 줄이고 더 일관된 성능을 얻을 수 있습니다. 이는 단일 스레드 성능이 중요한 시스템에서 특히 유용합니다.

스레드 우선순위 높이기

`chrt` 이나 `pthread_setschedparam` 같은 도구를 사용해 데이터 로더 및 전처리 CPU 스레드의 스케줄링 우선순위를 높여보세요. 이 스레드들에 더 높은 우선순위를 부여하면 데이터가 최소한의 지연으로 GPU에 공급되어 파이프라인 정지 가능성을 줄일 수 있습니다.

자주 사용되는 데이터 캐시

운영 체제 페이지 캐시 또는 전용 RAM 디스크를 활용하여 자주 액세스하는 데이터를 캐시하십시오. 이 접근 방식은 특정 토큰이나 구문이 반복적으로 액세스되는 NLP와 같은 애플리케이션에서 특히 유용하며, 중복 처리 및 I/O 오버헤드를 줄여줍니다.

데이터 프리페치 및 버퍼링

데이터는 항상 필요한 반복 작업보다 앞서 로드하십시오. 백그라운드 데이터 로더 스레드나 프로세스(예: `prefetch_factor`를 사용한 PyTorch DataLoader)를 활용하세요. 분산 훈련 시에는 `DistributedSampler`를 사용하여 각 프로세스가 고유한 데이터를 받아 중복 I/O를 방지하십시오.

데이터 변환 병렬화

이미지 증강이나 텍스트 토큰화 같은 CPU 전처리 작업이 무거운 경우, 여러 작업자 스레드/프로세스에 분산하세요. GPU가 대기하는 동안 CPU가 병목이 되지 않도록 자질하세요. 병목 현상이 발생하면 작업자 수를 늘리거나 일부 변환 작업을 GPU로 이동하세요. NVIDIA의 DALI 같은 라이브러리를 사용하면 GPU에서 비동기적으로 이미지 작업을 수행할 수 있습니다.

모델 상태 및 출력 캐싱

LLMs으로 추론할 때는 자주 등장하는 토큰에 대해 임베딩과 V 캐시를 캐싱하여 반복적인 재계산을 피하는 것이 유리합니다. 마찬가지로 LLM 훈련 작업이 동일한 데이터셋을 여러 번 재사용하는 경우(*에포크라고 함*), 핫 데이터를 저장하기 위해 OS 페이지 캐시나 RAM을 활용해야 합니다.

노드 간 데이터 분할

다중 노드 훈련 시 각 노드에 데이터의 하위 집합을 할당하여 모든 노드가 단일 소스에서 전체 데이터셋을 읽는 것을 방지하세요. 이는 I/O를 확장합니다. 분산 파일 시스템이나 수동 샤드 할당을 사용하여 각 노드가 서로 다른 파일을 읽도록 하세요. 이는 속도를 높이고 각 노드가 자체 데이터 샤드

를 처리하므로 데이터 병렬 처리와 자연스럽게 일치합니다. DeepSeek의 Fire-Flyer 파일 시스템(3FS)은 분산 데이터셋 샤딩 파일 시스템의 한 예입니다. DeepSeek의 3FS는 각 노드의 NVMe SSD에 데이터셋 샤드를 분산 배치함으로써 초당 멀티테라바이트 처리량을 달성합니다. 동시에 기존 캐싱 방식을 최소화합니다. 이 설계는 각 GPU에 로컬 고속 데이터를 공급하여 I/O 병목 현상을 방지합니다.

파이프라인 모니터링 및 배치 크기 조정

배치 크기를 늘리면 GPU 작업량이 증가하고 I/O 빈도가 감소하여 전체 활용도가 향상되지만, 수렴 속도에 영향을 미치므로 일정 수준까지만 효과적입니다. 반대로 GPU가 데이터를 자주 대기하고 I/O 속도를 높일 수 없는 경우, 배치 크기를 줄여 각 반복 시간을 단축하고 유휴 시간을 줄이거나, 더 작은 배치로 기울기 누적을 수행하여 데이터 읽기를 연속적으로 진행할 수 있습니다. GPU가 거의 항상 바쁘게 작동하는 균형점을 찾아야 합니다.

GPU에서 데이터 증강 적용

증강 작업이 노이즈 추가나 정규화처럼 단순하지만 방대한 데이터에 적용되는 경우, CPU 포화 상태를 피하기 위해 GPU에서 수행하는 것이 유리할 수 있습니다. 데이터 로딩 중 GPU는 종종 활용도가 낮으므로, 로딩 후 작은 CUDA 커널을 사용해 데이터를 증강하는 것이 효율적일 수 있습니다. 다만 파이프라인이 직렬화되지 않도록 주의해야 합니다. 배치 *N이* 훈련 중인 동안 *배치 N+1의* 증강을 중첩 처리하려면 스트림을 활용하세요. NVIDIA DALI 같은 GPU 가속 라이브러리를 사용해 이러한 작업을 비동기적으로 수행하세요. 이는 원활하고 높은 처리량의 데이터 파이프라인 유지에 도움이 됩니다.

종단 간 처리량(예: 초당 토큰 수)에 집중

모델 연산 속도를 높여도 데이터 파이프라인이 처리량을 절반으로 줄인다면 소용없다는 점을 명심하세요. 훈련 루프만 분리해서 자질하지 말고 항상 엔드투엔드(end-to-end)로 자질하세요. Nsight Systems 및 Nsight Compute를 사용하여 커널 타임라인과 스톨을 측정하거나, 프레임워크 수준 속성 분석을 위해 PyTorch 프로파일러를 활용하세요. 그런 다음 합성 데이터와 실제 데이터의 반복자를 비교하여 데이터 로딩이 얼마나 많은 오버헤드를 유발하는지 확인하세요. 이상적인 상태 대비 10% 미만의 오버헤드를 목표로 하세요. 그 이상이라면 파이프라인 최적화에 시간을 투자하세요. 이는 종종 훈련에서 큰 '무료' 속도 향상을 가져옵니다.

성능 자질, 디버깅 및 모니터링

측정하지 않으면 최적화할 수 없습니다. 자질을 통해 컴퓨팅, 메모리, I/O 또는 네트워크 중 어느 부분이 바운디드인지 파악하여 적절한 해결책을 적용하세요. 지속적인 텔레메트리 및 회귀 테스트를 통해 코드, 드라이버, 데이터가 진화함에 따라 개선 효과가 유지되도록 합니다.

구체적으로, Nsight Systems/Compute 및 프레임워크 자질을 NVTX와 함께 사용하여 컴퓨팅, 메모리, I/O 또는 통신 중 어느 부분이 바운디드를 일으키는지 확인하십시오. Python 오버헤드를 줄이고, 활용도 격차를 관찰하며, 랭크 간 작업 균형을 맞추고, 메모리/네트워크/디스크 상태를 추적하며, 성능 회귀 테스트 및 경고를 통해 변경 사항을 제어하십시오. AI 워크로드의 성능을 프로파일링, 모니터링 및 디버깅하려면 다음 지침을 따르십시오:

자질을 통한 병목 현상 및 근본 원인 분석

훈련/추론 작업에 자질을 정기적으로 실행하세요. NVIDIA Nsight Systems를 사용해 CPU 및 GPU 활동 타임라인을 확보하세요. Nsight Compute 또는 PyTorch 자질을 활용해 커널 효율성을 심층 분석할 수도 있습니다. 작업이 컴퓨팅 바운디드인지, 메모리 바운디드인지, 아니면 I/O/통신 대기 상태인지 식별하세요. 최적화 대상을 이에 맞게 설정하세요. 예를 들어 메모리 바운디드 워크로드라면 컴퓨팅 바운디드 최적화 구현보다 메모리 트래픽 감소에 집중하십시오. 머신러닝 기반 분석과 결합하여 성능 병목 현상을 예측하고 선제적으로 방지하세요. 이는 실시간으로 미세 조정 작업을 자동화하는 데 도움이 됩니다. GPUDirect Storage 사용 시 GDS 추적을 활성화하여 `cuFile` 활동과 커널 간격을 연관 분석하세요.

Python 오버헤드 제거

훈련 스크립트의 자질을 확인하여 과도한 루핑이나 로깅과 같은 Python 병목 현상을 식별하고, 이를 벡터화 작업이나 최적화된 라이브러리 호출로 대체하십시오. Python 오버헤드를 최소화하면 CPU가 전체 시스템 성능의 숨겨진 병목이 되지 않도록 보장하는 데 도움이 됩니다.

GPU 활용도 및 유휴 간격 측정

GPU 사용률, SM 효율성, 메모리 대역폭 사용량 등을 지속적으로 모니터링하십시오. 주기적인 사용률 하락이 관찰되면 이벤트와 연관성을 분석하십시오. 예를 들어 5분마다 발생하는 사용률 하락은 체크포인트 저장 시점과 일치할 수 있습니다. 이러한 패턴은 체크포인트 간격 조정이나 비동기

플러시 사용과 같은 최적화 기회를 시사합니다. DCGM이나 데몬 모드의 `nvidia-smi` 과 같은 도구를 활용하여 이러한 메트릭을 시간 경과에 따라 기록하십시오.

NVTX 마커 사용

NVTX 범위나 프레임워크 자질 API로 코드를 계측하여 데이터 로딩, 포워드 패스, 백워드 패스 등 다양한 단계를 표시하세요. 이러한 마커는 Nsight Systems 또는 Perfetto 타임라인에 표시되어 GPU 유휴 시간이나 지연 시간을 파이프라인의 특정 부분에 귀속시키는 데 도움이 됩니다. 이를 통해 개발자에게 코드의 어느 부분이 주의가 필요한지 전달하기가 더 쉬워집니다. PyTorch의 경우 `torch.profiler.record_function()` 를 사용할 수 있습니다.

PyTorch 프로파일을 넘어 커널 프로파일링 및 분석 도구 활용

성능이 중요한 커널의 경우 Nsight Compute를 사용하여 점유율 및 처리량과 같은 커널 수준 메트릭을 검토하거나, Nsight Systems를 사용하여 GPU/CPU 타임라인과 중첩을 분석하세요. 달성한 점유율, 메모리 처리량, 명령어 처리량을 확인하세요. 하드웨어 최대치에 근접한 메모리 대역폭과 같은 메모리 병목 현상의 징후를 찾으세요. 이는 메모리 바운디드 워크로드를 식별하는 데 도움이 됩니다. 프로파일러의 "문제점" 섹션은 커널이 메모리 바운디드인지 연산 바운디드인지, 그 이유를 직접 제시하는 경우가 많습니다. 글로벌 로드 효율이 낮을 경우 메모리 결합을 개선하는 등 코드 변경을 안내하는 데 이 피드백을 활용하세요.

워프 발산 확인

자질을 사용하여 워프 분기 여부를 확인하십시오. 분기 효율성과 분기 지표가 표시될 수 있습니다. 분기란 워프 내 일부 스레드가 분기로 인해 비활성화되어 처리량에 악영향을 미치는 것을 의미합니다. 분기 현상이 심각하다면 커널 코드를 재검토하여 조건문이나 데이터 할당을 재구성하여 워프 내 분기를 최소화하고 각 워프가 균일한 작업을 처리하도록 하십시오.

로드 밸런싱 검증

다중 GPU 작업에서는 랭크 간 자질을 수행하세요. 때로는 한 GPU(랭크 0)가 통계 집계나 데이터 수집 같은 추가 작업을 수행하며 병목 현상의 원인이 되기도 합니다. 각 GPU의 타임라인을 모니터링하세요. 특정 GPU가 지속적으로 지연된다면 해당 추가 작업 부하를 분산시키세요. 예를 들어, 0이 아닌 랭크들이 I/O 및 로깅 책임을 공유하도록 할 수 있습니다. 모든 GPU/랭크가 유사한 작업 부하를 가지도록 하면 가장 느린 랭크가 나머지를 지연시키는 것을 방지할 수 있습니다.

메모리 사용량 모니터링

시간 경과에 따른 GPU 메모리 할당 및 사용량을 추적하십시오. OOM(Out of Memory) 상태에 가까워지지 않도록 주의하십시오. 이는 프레임워크가 텐서를 호스트로 예기치 않게 스왑하게 하여 심각한 성능 저하를 유발할 수 있습니다. 메모리 사용량이 반복자마다 증가한다면 메모리 누수를 확인한 것입니다. 이 경우 `torch.cuda.memory_summary()` 및 Nsight Systems의 GPU 메모리 추적 도구로 자질을 분석하세요. CPU 측면에서는 페이징을 모니터링하세요. 프로세스의 상주 메모리(RES)가 물리적 RAM을 크게 초과해서는 안 됩니다. 페이징이 발생하면 데이터셋 사전 로드 크기를 줄이거나 RAM을 늘리세요.

네트워크 및 디스크 모니터링

분산 작업의 경우 OS 도구를 사용하여 네트워크 처리량과 디스크 처리량을 모니터링하십시오. 실제 처리량이 예상치와 일치하는지 확인하십시오. 예를 들어, 100 Gbps 링크에서 완전히 활용될 경우 12.5 GB/s(12.5 GB/s = 100 Gb/s ÷ 바이트당 8비트)를 확인해야 합니다. 그렇지 않다면 네트워크가 병목 현상이나 잘못된 구성일 수 있습니다. 마찬가지로, 훈련 노드의 디스크 I/O를 모니터링하십시오. 디스크 사용률이 100%로 급증하고 GPU가 유휴 상태인 경우, 데이터를 더 잘 버퍼링하거나 캐싱해야 할 가능성이 높습니다.

이상 현상에 대한 경보 설정

생산 환경이나 장기 실행 훈련 시에는 ECC 오류, 장치 과열 등 GPU 오류와 같은 이벤트에 대한 자동 알림이나 로그를 설정하세요. 이는 비정상적으로 느린 반복자를 식별하는 데 도움이 됩니다. 예를 들어 NVIDIA의 DCGM은 상태 지표를 모니터링할 수 있으며, GPU가 스로틀링을 시작하거나 오류를 발생시키면 조치를 트리거할 수 있습니다. 이는 작업 완료 후가 아닌 즉시 성능 문제(예: 냉각 실패로 인한 스로틀링)를 포착하는 데 도움이 됩니다.

회귀 테스트 수행

CUDA 드라이버, CUDA 버전, AI 프레임워크 버전 또는 훈련 코드 변경 시마다 실행할 벤치마크 작업 세트를 유지하세요. 이전 실행과 성능을 비교하여 성능 저하를 조기에 포착할 수 있습니다. 드라이버 업데이트나 코드 변경으로 인해 의도치 않게 처리량이 감소하는 경우가 드물지 않습니다. 표준 워크로드에 대한 빠른 자질 실행으로 이를 파악하여 조사할 수 있습니다. 예를 들어, 커널이 실수로 텐서 코어를 더 이상 사용하지 않을 수 있습니다. 이는 반드시 조사해야 할 사항입니다.

GPU 프로그래밍 및 CUDA 튜닝 최적화

커널을 메모리 계층 구조 와 하드웨어 기능에 맞추는 것이 지속적이고 큰 성능 향상의 원천입니다. 퓨전, 텐서 코어, CUDA 그래프, 컴파일러 경로(예: `torch.compile` 및 OpenAI의 Triton)는 런치 오버헤드를 유용한 연산으로 변환합니다.

메모리 계층 구조 최적화: 글로벌 로드 통합, 공유 메모리로 타일링, 레지스터/점유율 관리, 전송(예: `cp.async`/TMA)과 연산 중첩 수행. 튜닝된 라이브러리 및 CUDA 그래프 선호, `torch.compile` 및 OpenAI의 Triton을 활용한 퓨전, 루프라인 분석 및 PTX/SASS 검사로 확장성 검증. 다음은 GPU 및 CUDA 프로그래밍 최적화 팁과 기법입니다:

GPU 메모리 계층 구조 이해

GPU의 계층적 메모리 구조를 염두에 두세요: 스레드별 레지스터, 블록/SM별 공유 메모리/L1 캐시, SM 간 L2 캐시, 글로벌 HBM. 상위 계층에서 데이터 재사용을 극대화하세요. 예를 들어, 레지스터와 공유 메모리를 활용해 값을 재사용하고 느린 글로벌 메모리 접근을 최소화하세요. 우수한 커널은 대부분의 데이터가 레지스터에 있거나, 결합(coalescing) 및 캐싱을 통해 HBM에서 효율적으로 로드되도록 보장합니다.

글로벌 메모리 액세스 통합

동일 워프 내 스레드들이 연속된 메모리 주소에 접근하도록 하여 하드웨어가 최소한의 트랜잭션으로 처리할 수 있게 하십시오. 워프 스레드들의 간격이 넓거나 분산된 메모리 접근은 워프당 다중 메모리 트랜잭션을 유발하여 대역폭을 낭비합니다. 데이터 레이아웃이나 인덱스 계산을 재구성하여 워프가 데이터를 로드할 때마다 단일하고 넓은 메모리 트랜잭션으로 수행되도록 하십시오.

데이터 재사용을 위한 공유 메모리 사용

공유 메모리는 매우 높은 대역폭을 가진 수동 관리 캐시와 유사합니다. 자주 사용되는 데이터(예: 행렬 타일)를 공유 메모리에 로드하세요. 그리고 스레드가 해당 타일을 여러 번 처리한 후 다음 작업으로 넘어가도록 합니다. 이 널리 사용되는 타일링 기법은 글로벌 메모리 트래픽을 크게 줄여줍니다. 공유 메모리 뱅크 충돌에 주의하세요. 공유 메모리 접근 패턴을 구성하거나 데이터를 패딩하여 스레드가 동일한 메모리 뱅크를 놓고 경쟁하지 않도록 해야 합니다. 경쟁 시 접근이 직렬화되어 성능이 저하됩니다.

메모리 정렬 최적화

가능한 경우 데이터 구조를 128바이트 단위로 정렬하십시오. 특히 대량 메모리 복사나 벡터화된 로드 작업 시 중요합니다. 정렬되지 않은 액세스는 이론상 병합될 수 있는 경우에도 여러 트랜잭션을 강제할 수 있습니다. 글로벌 메모리 I/O에 float2나 float4 같은 벡터화된 타입을 사용하면 명령어당 여러 값을 로드/저장할 수 있지만, 데이터 포인터가 벡터 크기에 맞게 정렬되었는지 반드시 확인하십시오.

메모리 전송 최소화

필요한 경우에만 대량의 데이터로 GPU에 전송하십시오. 가능하면 여러 작은 전송을 하나의 큰 전송으로 통합하십시오. 예를 들어, 각 반복자마다 전송할 작은 배열이 많다면 하나의 버퍼에 압축하여 한 번에 전송하십시오. 빈번한 작은 메모리 전송(`cudaMemcpy`)은 병목 현상이 될 수 있습니다. 통합 메모리(Unified Memory)를 사용하는 경우 명시적 프리페치(`cudaMemPrefetchAsync`)를 사용하여 데이터가 필요하기 전에 GPU에 스테이징하십시오. 이렇게 하면 중요한 계산 구간에서 필요에 따른 페이지 결함(page fault)을 피할 수 있습니다.

과도한 임시 할당을 피하라

GPU 메모리의 빈번한 할당 및 해제는 성능에 악영향을 미칠 수 있습니다. 예를 들어 커널에서 `cudaMalloc/cudaFree` 나 device malloc을 자주 사용하면 추가 오버헤드가 발생합니다. 대신 메모리 버퍼를 재사용하거나, PyTorch와 같이 GPU 캐싱 할당기를 구현한 대부분의 DL 프레임워크에서 제공하는 메모리 풀을 사용하세요. 커스텀 CUDA 코드를 작성할 경우 메모리 풀과 함께 `cudaMallocAsync` 를 사용하거나, 반복적인 할당/해제의 오버헤드를 피하기 위해 스크래치 메모리 풀을 직접 관리하는 것을 고려하세요.

스레드와 리소스 사용의 균형 유지

점유율과 리소스 간 적절한 균형을 유지하십시오. 점유율 향상을 위해 스레드 수를 늘리면 메모리 지연 시간을 숨기는 데 도움이 되지만, 각 스레드가 너무 많은 레지스터나 공유 메모리를 사용하면 점유율이 떨어집니다. 블록당 스레드 수를 포함한 커널 실행 매개변수를 조정하여 지연 시간을 커버할 만큼 충분한 워프가 실행 중이되, 각 스레드가 레지스터나 공유 메모리 부족으로 고갈되지 않도록 하십시오. 명령어 수준 병렬성(ILP)이 높은 커널에서는 점유율 향상을 위해 레지스터 사용량을 줄이는 것이 오히려 성능 저하를 초래할 수 있습니다. 최대 점유율이 항상 이상적인 것은 아니므로 최적점은 일반적으로 점유율 스펙트럼 중간에 위치합니다.

NVIDIA Nsight Compute 점유율 계산기를 활용하여 다양한 구성을 실험해 보십시오.

레지스터 및 공유 메모리 사용량 모니터링

Nsight Compute와 같은 자질을 사용하여 스레드별 레지스터 및 공유 메모리 사용량을 지속적으로 모니터링하십시오. 점유율이 25% 미만으로 관측될 경우, 사용 가능한 하드웨어 자원을 더 효율적으로 활용하기 위해 블록당 스레드 수를 늘리는 것을 고려하십시오. 그러나 이 조정이 과도한 레지스터 스필링을 유발하지 않는지, 상세한 점유율 보고서와 커널 실행 메트릭을 검토하여 확인하십시오. 레지스터 스필링은 추가적인 메모리 트래픽을 유발하고 전체 성능을 저하시킬 수 있습니다.

메모리 전송과 계산을 중첩

가능한 경우 메모리 전송과 계산을 중첩하십시오. 커널 실행 중 프리페치하기 위해 여러 CUDA 스트림에서 CUDA 스트림 간 전송(`cuda MemcpyAsync`)을 사용하십시오. 공유 메모리로의 대량 이동에는 텐서 메모리 가속기(Tensor Memory Accelerator)를 선호하고, 세분화된 단계별 복사 및 프리페치에는 CUDA 스트림 간 전송(`cp.async`)을 사용하십시오. 이러한 접근 방식은 데이터 전송과 계산을 중첩하여 글로벌 메모리 지연 시간을 효과적으로 가리고, 메모리 작업 완료를 기다리지 않고 GPU 코어가 완전히 활용되도록 합니다.

가능한 경우 대량 프리페칭 사용

예측 가능한 패턴의 경우 PTX `cp.async.bulk.prefetch.tensor.[1–5]d.L2.global*` (또는 `prefetch.global.L2` 계열)를 사용하여 L2로 프리페치하고, TMA(예: `cp.async.bulk.tensor`)를 사용하여 블록을 공유 메모리로 스테이징하십시오. cp.async를 사용하여 글로벌 메모리를 비동기적으로 공유 메모리로 스테이징하고 복사와 계산을 중첩할 수도 있습니다. 사용 전에 데이터를 레지스터에 명시적으로 로드할 수도 있습니다. 이러한 사전적 방법은 글로벌 메모리 액세스로 인한 지연을 줄이고, 중요한 데이터가 필요할 때 바로 레지스터나 공유 메모리 같은 더 빠르고 지연 시간이 짧은 저장소에 존재하도록 보장하여 실행 정지를 최소화하고 커널의 전반적인 효율성을 향상시킵니다.

협동 그룹 활용

전체 블록 단위의 장벽을 강제하는 대신 CUDA의 협력적 그룹을 활용하여 스레드 하위 집합 간 효율적이고 국소적인 동기화를 달성하세요. 이 기법은 동기화에 대한 세밀한 제어를 가능하게 하여 불필요한 대기 시간과 오버헤드를 줄입니다. 데이터를 공유하거나 관련 계산을 수행하는 스레드를

그룹화함으로써, 조정이 필요한 스레드만 동기화할 수 있어 더 효율적인 실행 패턴과 전반적인 처리량 향상을 이끌 수 있습니다.

워프 발산 최적화

코드 구조를 설계할 때 워프 내 스레드가 가능한 한 동일한 실행 경로를 따르도록 합니다. 분기 발생 시 해당 워프의 실행 시간이 두 배로 증가할 수 있습니다(예: 워프 절반(16개 스레드)이 한 분기로, 나머지 절반(16개 스레드)이 다른 분기로 이동). 특정 데이터가 거의 트리거하지 않는 분기가 있다면, 데이터를 "정렬"하거나 그룹화하여 워프가 모든 경우를 동일하게 처리하도록(모두 참 또는 모두 거짓) 고려하십시오. 특정 문제에 대해 분기 없는 솔루션을 생성하려면 ballot 및 shuffle과 같은 워프 수준 기본 연산을 사용하십시오. 워프를 작업 단위로 간주하고, 최대 효율을 위해 32개 스레드 모두가 동일한 작업을 동기화하여 수행하도록 목표하십시오.

워프 수준 연산 활용

적절한 경우 CUDA의 워프 내장 함수를 사용하여 스레드가 공유 메모리에 접근하지 않고도 통신할 수 있도록 하십시오. 예를 들어, 각 스레드가 공유 메모리에 쓰기 대신 `__shfl_sync`를 사용하여 워프 내 모든 스레드에 값을 브로드캐스트하거나 워프 레벨 축소 연산(예: 워프 전체 레지스터 합산)을 수행하세요. 이러한 내장 함수는 느린 메모리를 우회하여 워프 내에서 수행 가능한 축소나 스캔 같은 알고리즘을 가속화합니다. 워프 내에서 이러한 작업을 처리함으로써 공유 메모리 및 전체 블록 동기화와 관련된 지연 시간을 피할 수 있습니다.

동시성을 위해 CUDA 스트림 사용

단일 프로세스/GPU 내에서, 모든 리소스를 사용하지 않는 독립적인 커널을 서로 다른 CUDA 스트림에 런치하여 실행을 중첩하십시오. 예를 들어, 한 스트림이 모델의 한 부분을 계산하는 동안 다른 스트림은 GPU에서의 데이터 전처리나 비동기적 `memcpy` 과 같은 독립적인 커널을 런치하는 식으로 계산을 중첩하십시오. 종속성을 주의 깊게 고려하고 필요 시 CUDA 이벤트를 사용하여 동기화하십시오. 스트림을 적절히 사용하면 자원을 유휴 상태로 두지 않아 GPU 활용도를 높일 수 있습니다. 특히 가벼운 커널이 있는 경우 더욱 효과적입니다.

라이브러리 함수를 우선적으로 사용하십시오

핵심 수학 연산 및 집합 연산에는 가능한 한 NVIDIA 최적화 라이브러리(cuBLAS, cuDNN, Thrust, NCCL 등)를 사용하십시오. 분산 추론 시 GPU 간 포인트투포인트 데이터 이동에는 가능한 경우 NIXL을 활용하십시오. GPU에서 시작하는 세밀한 전송이 필요할 때는 NVSHMEM을 사용할 수

도 있습니다. 이들 라이브러리는 각 GPU 아키텍처에 대해 고도로 최적화되어 이론적 '광속' 피크 성능에 근접합니다. 이를 통해 라이브러리를 재구현하는 수고를 덜 수 있습니다. 예를 들어, 매우 특수한 패턴이 아닌 한 행렬 곱셈에는 커스텀 커널 대신 cuBLAS GEMM을 사용하세요. 해당 라이브러리는 새로운 하드웨어 기능도 투명하게 처리합니다. PyTorch(및 그 컴파일러)와 같은 AI 프레임워크는 내부적으로 이러한 최적화된 라이브러리를 사용합니다.

반복 실행에는 CUDA 그래프 사용

수천 번 실행되는 정적 훈련 루프가 있다면, CUDA 그래프를 사용하여 연산 시퀀스를 그래프로 캡처하고 실행하는 것을 고려하십시오. 이는 각 반복에 대한 CPU 런치 오버헤드를 크게 줄일 수 있으며, 특히 다중 GPU 환경에서 많은 커널과 커널 실행(`memcpy`)을 런치할 때 CPU에 추가 부담을 주고 지연 시간을 증가시킬 수 있습니다.

확장성 한계 확인

커널을 최적화할 때는 문제 규모와 아키텍처 간 확장성을 주기적으로 점검하세요. 커널은 작은 입력에서는 높은 점유율과 성능을 보이지만, L2 캐시 스래싱이나 메모리-캐시 이젝션에 부딪히면서 큰 입력으로 확장되지 못할 수 있습니다. 루프라인 분석을 활용하세요. 달성한 FLOPS와 대역폭을 하드웨어 한계와 비교하여 성능을 최대한 활용하고 있는지 확인하십시오.

고급 커널 분석을 위해 PTX 및 SASS 검사

성능이 중요한 커스텀 CUDA 커널의 경우 Nsight Compute를 사용하여 생성된 PTX 및 SASS를 검토하세요. 이 심층 분석을 통해 메모리 뱅크 충돌이나 중복 계산과 같은 문제를 발견할 수 있으며, 이를 통해 저수준 최적화를 위한 방향을 제시할 수 있습니다.

PyTorch 컴파일러 사용

`torch.compile` PyTorch의 TorchInductor를 활용하여 Python 레벨 연산을 최적화된 커널로 융합하세요. 컴파일러는 CUDA 그래프 통합을 통해 런치 오버헤드를 줄일 수 있습니다. 최적화가 정착되면 일반적으로 약 10%~40%의 성능 향상이 관찰됩니다. 이는 인터프리터 오버헤드를 제거하고 컴파일러 수준의 최적화를 가능하게 합니다.

`torch.compile` 실제 적용 시, 자동 커널 결합 및 NVIDIA GPU 하드웨어(예: 텐서 코어)의 효율적 활용을 통해 컴파일 모드 활성화로 상당한 속도 향상(예: 다수 모델에서 20%~50%)이 달성되었습니다. 모델에 컴파일

모드를 반드시 테스트하십시오. 처리량을 크게 향상시킬 수 있으나, 배포 전 호환성과 정확성을 반드시 확인해야 합니다. 그래프가 안정화되면 CUDA 그래프를 활성화하여 반복자당 CPU 오버헤드를 줄이십시오. 포인터 안정성 제약 조건을 충족시키기 위해 정적 메모리 풀을 유지하십시오.

동적 형상 계획 수립

입력 크기가 변동될 경우, 동적 차원을 주석 처리하려면 `torch._dynamo.mark_dynamic()`를 사용하거나 `torch.export()`로 형상 다형성 그래프를 내보낸 후 컴파일하십시오. 테스트 및 CI에서 문제가 되는 형상 변동을 파악하려면 `"fail_on_recompile"` 및 `torch._dynamo.error_on_graph_break()`를 사용하여 `torch.compiler.set_stance()`로 재컴파일 동작을 제어하십시오. 가능한 경우 정적 형상을 사용하여 CUDA 그래프가 반복당 CPU 오버헤드를 줄일 수 있도록 하십시오.

Triton 커널 활용 `torch.compile`

PyTorch가 연산을 제대로 융합하지 못할 경우, Triton에서 커스텀 GPU 커널을 작성하여 통합하는 것을 고려하십시오. PyTorch는 `torch.library.triton_op`를 통해 커스텀 GPU 커널을 쉽게 등록할 수 있도록 합니다.

가능한 경우 자동 튜닝 사용

라이브러리 자동 튜닝 기능을 활성화하여 저수준 성능을 극대화하십시오. 예를 들어 입력 크기가 고정된 경우 `torch.backends.cudnn.benchmark=True`를 설정하십시오. 이렇게 하면 NVIDIA cuDNN 라이브러리가 여러 컨볼루션 알고리즘을 시도하여 하드웨어에 가장 빠른 알고리즘을 선택합니다. 일회성 오버헤드는 훈련 및 추론을 가속화할 수 있는 최적화된 커널로 이어집니다. 정확한 재현성이 필요하지 않은 경우 `cudnn.deterministic`를 비활성화하여 비결정적 알고리즘을 허용하고 더 빠른 구현을 활용하십시오.

읽기 전용 경로 활용

자주 사용되는 상수나 계수를 읽기 전용으로 표시하면 GPU가 전용 L1 읽기 전용 캐시에 이를 캐시할 수 있습니다. CUDA C++에서는 `const __restrict__` 포인터를 사용하여 데이터가 불변임을 암시할 수 있습니다. 최신 GPU 아키텍처에서는 컴파일러가 `const __restrict__`로 지정된 포인터에 대해 캐시된 전역 로드를 생성합니다. AI 프레임워크 및 라이브러리를 사용할 때는 조회 테이블이나 정적 가중치가 장치에 위치하며 상수로 처리되도록 하십시오. 이 최적화는 해당 값들에 대한 전역

메모리 트래픽과 지연 시간을 줄여줍니다. 각 SM이 느린 DRAM에 반복적으로 접근하는 대신 캐시에서 빠르게 가져올 수 있기 때문입니다.

커널 스케줄링 및 실행 최적화

런치 오버헤드와 불필요한 동기화는 유휴 간격을 생성하여 처리량을 저하시킵니다. 소규모 커널을 융합하고 지속적/동적 전략을 사용하면 장치를 지속적으로 활용하고 지연 시간을 숨길 수 있습니다.

동기화를 최소화하고, 작은 커널을 융합하며, 동일한 작업을 반복 실행할 때는 지속적 커널을 사용하여 장치를 바쁘게 유지하십시오. 불규칙한 작업의 경우 GPU 동적 병렬 처리를 고려하되, 오버헤드 증가를 피하기 위해 신중하게 사용하십시오. 다음은 커널 스케줄링 및 실행 개선을 위한 팁입니다:

GPU 동기화 호출 최소화

GPU 진행을 지연시키는 불필요한 전역 동기화를 피하십시오. 동기화(`cudaDeviceSynchronize()`)의 과도한 사용이나 GPU 작업 차단(동기식 메모리 복사 등)은 CPU와 GPU 모두 유용한 작업을 수행할 수 없는 유휴 간격을 발생시킵니다. 반드시 필요한 경우에만 동기화하십시오. 예를 들어 최종 결과 전송 시나 디버깅 시 동기화합니다. 비동기 작업을 대기열에 쌓아두면 GPU를 바쁘게 유지하고 CPU는 추가 작업 준비에 자유롭게 할당됩니다. 이는 보다 연속적인 실행 파이프라인으로 이어집니다.

작은 커널을 융합하여 런치 오버헤드 분산

여러 개의 작은 GPU 커널이 연속적으로 실행되는 경우, 가능한 한 단일 커널에서 작업을 병합하여 실행하는 것을 고려하십시오. 각 커널 실행에는 수십 마이크로초 단위의 고정 비용이 발생하므로, 수동 CUDA 커널 퓨전, XLA 퓨전 또는 NVIDIA CUTLASS/Triton과 같은 커스텀 연산 도구를 통해 작업을 결합하면 처리량을 향상시킬 수 있습니다. 퓨전된 커널은 실제 작업에 더 많은 시간을 할애하고 실행 오버헤드나 메모리 왕복 시간은 줄입니다. 이는 특히 추론이나 전처리 파이프라인에서 요소별 연산 체인을 한 번에 실행할 수 있을 때 유용합니다. 먼저 `torch.compile(mode="reduce-overhead")` 를 시도해 보세요. 컴파일러가 연산 체인을 융합하고 안정적인 영역을 CUDA 그래프로 감쌀 수 있습니다. 이는 CPU 런치 오버헤드를 줄여줍니다. 융합되지 않은 핫스팟의 경우, Triton 커널로 마이그레이션하고 해당되는 경우 비동기 TMA 및 자동 워프 특화 기능을 사용하는 것을 고려하세요.

GPU 내 작업 스케줄링을 위해 GPU 동적 병렬 처리 활용

CUDA의 동적 병렬 처리(Dynamic Parallelism)를 활용하여 GPU 커널이 CPU로 반환하지 않고 GPU 내에서 다른 커널을 런칭하도록 하십시오. 중간 결과에 따라 추가 작업을 생성해야 하는 알고리즘과 같이 예측 불가능하거나 반복적인 작업이 발생하는 시나리오에서 동적 병렬 처리는 CPU 런칭 병목 현상을 제거하여 지연 시간을 줄입니다. 예를 들어, 부모 커널이 하위 커널을 분할하여 장치에서 직접 추가 처리를 실행할 수 있습니다. 이를 통해 전체 워크플로우를 GPU에 유지하여 CPU 개입을 피하고 더 나은 중첩 및 활용을 가능하게 합니다. 그러나 과도하게 사용하면 자체 오버헤드가 발생할 수 있으므로 신중하게 사용하십시오.

반복 작업 부하에 지속적 커널 사용

작업 부하가 동일한 커널을 빠르게 연속 실행하는 경우(작업 큐 처리나 동일한 계산을 가진 배치 스트리밍 등)에는 지속적 커널 전략을 사용하십시오. 지속적 커널은 한 번 실행된 후 활성 상태를 유지하며, 각 작업 단위마다 새 커널을 실행하는 대신 루프 내에서 여러 작업 단위를 처리하기 위해 스레드를 재사용합니다. 이 접근 방식은 더 복잡한 커널 설계를 대가로 스케줄링 오버헤드를 크게 줄입니다. 커널을 활성 상태로 유지함으로써 반복적인 실행 비용을 피하고 더 높은 지속적 점유율을 달성할 수 있습니다. 고성능 분산 훈련 및 추론 시스템은 반복 작업의 처리량 극대화와 지연 시간 최소화를 위해 이 기법을 자주 활용합니다.

스레드 블록 클러스터 평가

스레드 블록을 클러스터링하여 데이터를 가깝게 유지하고 재실행 오버헤드를 줄이십시오. Blackwell에서는 최대 16개의 스레드 블록이 클러스터를 형성할 수 있습니다(비이동성 제한 증가 후). 클러스터 인식 동기화와 공유 메모리 상주성을 활용하여 지속적 스타일 설계에서 지역성을 개선하십시오. Nsight Compute와 같은 커널 수준 자질 도구를 사용하여 점유율과 상주성 간의 절충점을 자질하십시오.

산술 최적화 및 정밀도 감소/혼합 정밀도

낮은 정밀도와 스파시티 를 활용하면 비트 수를 줄여 속도와 메모리 효율을 크게 향상시킬 수 있으며, 정확도 영향은 무시할 수 있을 정도로 미미합니다. 혼합 정밀도(TF32/FP8/INT8)와 융합 스케일링은 하드웨어 연산 경로를 활용하여 비용 대비 처리량을 높입니다.

구체적으로 혼합 정밀도(BF16/FP16)와 텐서 코어를 활용해 큰 이득을 얻고, TF32를 채택해 FP32 속도를 쉽게 향상시키며, 품질이 허용하는 범위에서 FP8/FP4를 평가하십시오. 추론을 위해 구조적 스파시티, 저정밀도 기울기/통신, INT8/INT4 양자화를 활용하세요—정확도 유지를 위해 스케일/활성화를 융합합니다. 다음 최적화 기법은 산술 연산 성능 향상과 축소/혼합 정밀도 활용에 적용됩니다:

혼합 정밀도 훈련 사용

훈련 시 FP16 또는 BF16을 활용하여 연산 속도를 높이고 메모리 사용량을 줄이십시오. 최신 GPU는 FP16/BF16 행렬 연산을 대폭 가속화하는 텐서 코어를 탑재하고 있습니다. 수치적 안정성을 위해 최종 누적이나 가중치 복사본 같은 핵심 부분은 FP32로 유지하되, 대량 계산은 반정밀도로 실행하세요. 이는 정확도 손실을 최소화하면서 약 1.5~3.5배의 가속 효과(모델 및 커널 조합에 따라 다름, 특히 딥 레이어 연산(`matmul`)이 많은 작업에서 더 큰 이득)를 제공하며, 자동 혼합 정밀도(AMP)를 지원하는 대부분의 프레임워크에서 표준으로 채택되었습니다.

경사 누적 및 활성화 체크포인트링 채택

추가 메모리 사용 없이 배치 크기를 효과적으로 늘리기 위한 기울기 누적 활용법을 상세히 설명하고, 매우 깊은 신경망에서 메모리 사용량을 줄이기 위한 활성화 체크포인트 방식을 고려하십시오. 이러한 기법은 GPU 메모리 한도에 근접하거나 초과하는 모델 훈련 시 필수적입니다.

신규 하드웨어에서는 FP16 대신 BF16을 우선적으로 사용

가능하다면 FP16 대신 BF16을 사용하십시오. BF16은 더 넓은 지수 범위를 가지며 손실 스케일링이 필요하지 않습니다. 최신 GPU는 FP16과 동일한 속도로 BF16 텐서 코어를 지원합니다. BF16은 오버플로우/언더플로우 문제를 피하면서도 반정밀도의 성능 이점을 유지하여 훈련을 단순화합니다.

FP8, 새로운 정밀도 및 스케일링 기법 활용

현대 GPU에서 FP8 텐서 코어는 연산 바운디드 커널에서 FP16 또는 BF16 대비 약 2배의 연산 처리량을 제공하면서 동시에 활성화 및 가중치 대역폭을 감소시킵니다. 또한 FP4(NVFP4) 텐서 코어는 FP8 대비 처리량을 두 배로 높이며, 마이크로 텐서 스케일링(정확도 유지를 위한 오류 보정 기술)을 통한 추론에 사용되어 토큰 처리량을 향상시킵니다. 훈련 시에는 NVIDIA Transformer Engine과 함께 FP8을 사용하고, 필요 시 FP16 또는 FP32 누산기를 유지하십시오. 추론 시에는 먼저 FP8을 평가하고, 캘리

브레이션 결과 작업에 적합한 품질이 확인된 후에만 NVFP4를 채택하십시오. 훈련에는 하이브리드 FP8(전방 활성화/가중치에 E4M3, 기울기에 E5M2) 사용을 권장합니다. 구체적으로 전방 전달(예: 활성화 및 가중치)에는 E4M3를, 후방 전달(예: 기울기)에는 E5M2를 사용하는 것을 고려하십시오. 지연 스케일링 창을 256–1024로 설정하는 것이 종종 유리합니다. 추론 시에는 보정 후 NVFP4를 고려하십시오. TE는 PyTorch와 통합되며 최신 GPU 하드웨어에서 지원됩니다. 임의의 FP8 커스텀 연산보다 프레임워크 TE 커널을 우선적으로 사용하십시오. 엔드투엔드 가속 효과는 커널 조합, 메모리 대역폭, 보정에 따라 달라지므로 모델과 워크로드에서 정확도와 성능을 검증하십시오.

텐서 코어 및 텐서 메모리 가속기(TMA) 활용

가능하다면 사용자 정의 CUDA 커널이 행렬 연산에 텐서 코어를 활용하도록 하십시오. 단순화를 위해 CUTLASS 템플릿 사용이 필요할 수 있습니다. 텐서 코어와 TMA를 활용해 공유 메모리로 텐서를 비동기 이동하면 GEMM, 컨볼루션 및 기타 텐서 연산에서 극적인 가속 효과를 얻을 수 있으며, 종종 GPU의 최대 FLOPS에 근접합니다. 데이터는 필요에 따라 FP16/BF16/TF32 형식으로 유지하고, 8 또는 16의 배수인 텐서 코어 타일 크기에 맞춰 정렬하세요.

TF32를 사용하여 손쉽게 속도 향상

32비트 행렬 곱셈의 경우, PyTorch에서 수치적으로 안전한 연산에 대해 TF32(고속 FP32)를 활성화하려면 `torch.set_float32_matmul_precision("high")`를 설정하세요. cuBLAS 및 cuDNN과 같은 라이브러리는 최신 GPU 하드웨어에서 최적의 텐서 코어 코드 경로를 자동으로 선택합니다. "highest"(대신 "high")로 완전 정밀도 FP32를 강제 적용할 경우 성능 영향에 대해 반드시 이해해야 합니다.

구조적 스파시티 활용

최신 NVIDIA GPU는 행렬 곱셈에서 2:4 구조적 스파스성을 지원하여 가중치의 50%를 구조화된 패턴으로 0으로 설정합니다. 이를 통해 하드웨어 처리량이 두 배로 증가합니다. 모델을 정제하여 이 기능을 활용하세요. 가중치를 2:4 스파스성 패턴에 맞게 정제할 수 있다면 해당 레이어의 GEMM 연산 속도가 약 2배 빨라집니다. 구조적 스파스리티 적용 및 스파스 텐서 코어 경로 사용을 보장하려면 NVIDIA SDK 또는 라이브러리 지원을 활용하세요. 모델이 해당 스파스리티를 허용하거나 스파스리티 정규화를 통한 재훈련으로 훈련될 수 있다면, 이는 추가 비용 없이 속도 향상을 가져올 수 있습니다.

가능한 경우 기울기 및 활성화 값의 정밀도를 낮춤

가중치를 높은 정밀도로 유지하더라도, 기울기나 활성화 값을 낮은 정밀도로 압축하는 것을 고려하십시오. 예를 들어, 기울기 통신에 FP16/BF16 또는 FP8을 사용하십시오. 많은 프레임워크가 FP16 기울기 올-리듀스를 지원합니다. 마찬가지로, 활성화 체크포인트 저장 시 FP32 대신 16비트로 저장하면 메모리를 절약할 수 있습니다. FP8 및 FP4 최적화기, 양자화된 기울기에 대한 연구는 계속되고 있습니다. 이는 메모리 및 대역폭 비용을 줄이면서 모델 품질을 유지하는 데 도움이 됩니다. 대역폭이 제한된 환경에서는 특히 기울기 압축이 게임 체인저가 될 수 있습니다. DeepSeek은 제한된 GPU에서 훈련하기 위해 기울기를 압축함으로써 이를 입증했습니다.

추론을 위한 맞춤형 양자화 사용

배포 시에는 가능한 한 INT8 양자화를 사용하십시오. GPU에서의 INT8 추론은 매우 빠르고 메모리 효율적입니다. NVIDIA의 TensorRT 또는 양자화 도구를 사용하여 모델을 INT8로 양자화하고 보정하십시오. 트랜스포머와 같은 많은 신경망은 정확도 저하가 미미한 수준으로 INT8에서 실행될 수 있습니다. 속도 향상은 FP16 대비 2~4배에 달할 수 있습니다. 최신 GPU에서는 특정 모델에 대해 FP8 또는 INT4를 탐색하고 평가하여 추론 처리량을 더욱 향상시킬 수도 있습니다.

가능한 경우 스케일링 및 연산 작업 융합

낮은 정밀도를 사용할 때는 정확도를 유지하기 위해 연산을 융합하는 것을 잊지 마십시오. 예를 들어, Blackwell의 FP4 "마이크로스케일링"은 값 그룹별로 스케일을 유지할 것을 제안합니다. 정밀도 손실을 유발할 수 있는 별도의 패스 사용 대신, 한 번의 패스로 스케일링과 계산을 수행하여 이러한 융합 연산을 통합하십시오 . 이러한 작업의 대부분은 기존 라이브러리에서 처리되므로, 처음부터 구현하기보다는 이를 활용하십시오.

고급 튜닝 전략 및 알고리즘 기법

알고리즘 변경은 작업 속도를 높이기보다 작업량을 줄여 ROI 측면에서 하드웨어 업그레이드( )를 꾸준히 능가합니다. 자동 튜닝, 플래시 어텐션(FlashAttention), 통신/계산 중첩, 샤딩(sharding)은 낭비를 줄이면서 확장성을 확보합니다.

구체적으로, 커널 및 레이어 매개변수를 자동 조정하고, 융합/FlashAttention 커널을 교체하며, 분산 훈련에서 통신과 계산을 중첩하십시오. 파이프라인/텐서 병렬 처리 및 ZeRO 샤딩으로 딥 모델을 확장하고, 약간의 정확도 작업을 큰 처리량

이점으로 교환하기 위해 비동기 업데이트 또는 프루닝/스파스성을 고려하십시오. 다음은 몇 가지 고급 성능 최적화 및 알고리즘 기법입니다:

커널 매개변수를 자동 조정

대상 GPU에 맞게 커스텀 CUDA 커널을 자동 튜닝하세요. 적절한 블록 크기, 타일 크기, 언롤 팩터 등을 선택하는 것이 성능에 영향을 미치며, 최적 설정은 Ampere, Hopper, Blackwell 등 GPU 세대별로 종종 다릅니다. OpenAI Triton과 같은 자동 튜닝 스크립트나 프레임워크를 사용하거나, 전처리 단계에서 무차별 대입 검색을 통해 최적의 런치 구성을 찾으십시오. 이는 정적 "합리적" 설정으로는 놓칠 수 있는 20~30%의 성능 향상을 쉽게 가져올 수 있습니다. 자동 튜닝 루프에서 Triton 기능을 활용하세요. 예를 들어, `num_warps` 및 `num_stages` 를 설정하고, 자동 워프 전문화를 활성화하며, 비동기 TMA 레이아웃을 테스트하세요. 공유 메모리 스테이징에는 텐서 맵 디스크립터 API를 선호하세요. 다른 하드웨어로 마이그레이션할 때는 타일 모양을 재벤치마킹하세요. 최적의 선택은 GPU 세대마다 다를 수 있습니다.

ML 워크로드에서 커널 융합 사용

Deep Learning 라이브러리가 제공하는 융합 커널을 활용하세요. 예를 들어 융합 최적화기를 활성화하면 가중치 업데이트, 모멘텀 등과 같은 요소별 연산이 융합됩니다. 이는 융합 멀티헤드 어텐션 구현과 융합 정규화 커널도 사용합니다. NVIDIA 라이브러리와 Transformer Engine, FasterTransformer 같은 일부 오픈소스 프로젝트는 융합 LayerNorm + 드롭아웃과 같은 일반적인 패턴에 대한 융합 연산을 제공합니다. 이는 런치 오버헤드를 줄이고 메모리를 더 효율적으로 사용합니다.

FlashAttention과 같은 메모리 효율적인 어텐션 활용

트랜스포머 모델을 위한 FlashAttention과 같은 고급 알고리즘을 통합하십시오. FlashAttention은 타일링된 스트리밍 방식으로 어텐션을 계산하여 대규모 중간 행렬 생성을 피함으로써, 특히 긴 시퀀스에서 메모리 사용량을 대폭 줄이고 속도를 높입니다. 표준 어텐션을 FlashAttention으로 대체하면 처리량과 메모리 사용량을 모두 개선하여 동일한 하드웨어에서 더 큰 배치 크기나 시퀀스 길이를 허용할 수 있습니다.

통신과 계산을 중첩시키세요

분산 훈련 시 가능한 경우 네트워크 통신과 GPU 계산을 중첩하십시오. 예를 들어, 그라디언트 올-리듀스(all-reduce)의 경우 각 레이어의 그라디언트가 준비되는 즉시 비동기적으로 올-리듀스를 실행하고, 다음 레이어는

여전히 역전파(backward pass)를 계산 중일 수 있습니다. 이 파이프라이닝은 올바르게 수행될 경우 올-리듀스 지연 시간을 완전히 숨길 수 있습니다. 비동기 NCCL 호출이나 PyTorch의 분산 데이터 병렬(DDP)과 같은 프레임워크 라이브러리를 사용하세요. 이들은 기본적으로 중첩 처리를 제공합니다. 이를 통해 네트워크를 기다리며 GPU가 유휴 상태가 되는 것을 방지할 수 있습니다.

심층 모델에 파이프라인 병렬 처리 활용

모델 규모로 인해 텐서 병렬 처리나 파이프라인 병렬 처리를 통해 GPU 간 파이프라인을 구성해야 할 경우, 모든 파이프라인 단계를 바쁘게 유지할 수 있을 만큼 충분한 마이크로배치를 사용하세요. NVLink/NVSwitch를 활용하여 단계 간 활성화 값을 신속하게 전송하세요. 인터리브 스케줄링을 사용하여 중첩 및 파이프라인 버블을 줄이세요. 일부 프레임워크는 이러한 유형의 스케줄링을 자동화합니다. NVL72 패브릭은 특히 유용합니다. 통신 집약적인 파이프라인 단계조차도 멀티테라바이트 속도로 데이터를 교환할 수 있어 파이프라인 정체를 최소화합니다.

분산 최적화기 샤딩 활용

Zero Redundancy Optimizer(ZeRO)와 같은 메모리 절약 최적화 전략을 사용하세요. 이는 최적화기 상태나 기울기 같은 텐서를 복제하지 않고 GPU 간에 분할합니다. 이를 통해 메모리 및 통신 부하를 분산시켜 극한의 모델 규모까지 확장할 수 있습니다. GPU당 메모리 압박을 줄이고 CPU로의 스왑을 방지하며, 덩어리 단위로 수행할 경우 통신량을 감소시켜 처리량을 향상시킵니다. DeepSpeed, Megatron-LM 등 많은 프레임워크가 이러한 분할 방식을 제공합니다. 대규모 모델에서 이를 활용하면 OOM(Out Of Memory) 발생이나 스왑으로 인한 속도 저하 없이 고속 처리를 유지할 수 있습니다.

가능한 경우 비동기적으로 훈련하십시오

적용 가능한 경우 비동기 업데이트를 고려하십시오. 예를 들어, 작업자들이 업데이트 공유를 위해 항상 서로를 기다리지 않는 스텔레 확률적 경사하강법(SGD)을 사용할 수 있습니다. 이 접근법은 수렴에 영향을 주지 않도록 세심한 튜닝이 필요할 수 있지만 처리량을 증가시킬 수 있습니다. 비동기 훈련은 적절히 수행될 경우 큰 성능 향상을 제공할 수 있습니다.

스파스성과 프루닝을 통합하십시오

대규모 모델은 종종 중복성을 지닙니다. 훈련 중 스파스성을 도입하기 위해 프루닝 기법을 활용하세요. 이는 추론 시 활용할 수 있으며, 지원된다면 훈련 중에도 부분적으로 적용 가능합니다. 현대 GPU 하드웨어는 가속화

된 스파스 행렬 곱셈(2:4)을 지원하며, 향후 GPU는 이 기능을 확장할 가능성이 높습니다. 훈련은 밀집 상태로 유지하고 추론 시에만 프루닝하더라도, 더 작은 모델은 더 빠르게 실행되고 메모리를 덜 사용합니다. 이는 모델 배포의 비용 효율성을 높입니다. 모델 크기를 줄이면서 정확도를 유지하기 위해 로터리 티켓 가설, 디스틸레이션 또는 구조적 프루닝을 탐구하십시오.

분산 훈련 및 네트워크 최적화

클러스터 규모에서는 네트워크 가 병목이 됩니다. 처리하지 않으면 네트워크가 선형 확장을 깨고 비용을 증가시킬 수 있습니다. RDMA/점보 프레임, 계층적 집단 연산, 어피니티, 압축을 통해 대역폭을 보호하고 지연 시간을 제어합니다.

가능한 경우 RDMA(InfiniBand/RoCE)를 사용하십시오. 이더넷 환경에서는 TCP 버퍼를 조정하고, 점보 프레임을 활성화하며, 최신 혼잡 제어 방식을 선택하십시오. NIC/CPU 어피니티를 정렬하고, NCCL 스레드/버퍼(지원되는 경우 SHARP/CollNet 포함)를 조정하며, 기울기를 압축하거나 누적하고, 패브릭을 테스트하여 손실이나 잘못된 구성을 포착하십시오. 다중 GPU 및 다중 노드 모델 훈련과 같은 분산 환경을 위한 네트워크 최적화 지침:

가능한 경우 RDMA 네트워킹 사용

다중 노드 클러스터에 InfiniBand 또는 RoCE를 적용하여 낮은 지연 시간과 높은 처리량을 확보하십시오. NCCL 및 MPI가 훈련 시 RDMA를 사용하도록 설정하십시오. NCCL은 InfiniBand를 자동 감지하며 사용 가능한 경우 GPUDirect RDMA를 사용합니다. RDMA는 커널 네트워킹 스택을 우회하여 기존 TCP 대비 지연 시간을 크게 줄일 수 있습니다. 이더넷만 사용 가능한 경우, RDMA 지원 NIC에서 RoCE를 활성화하여 RDMA와 유사한 성능을 얻으십시오. NVLink 도메인 시스템(NVL72, GB200/GB300 등)에서는 가능한 경우 콜렉티브 연산을 패브릭 내에서 수행하십시오. 호스트 네트워킹은 아일랜드 간 연결에 전용하십시오. NCCL 토폴로지 힌트를 NVLink/NVSwitch 도메인과 일치시키십시오.

이더넷 사용 시 TCP/IP 스택 조정

TCP 기반 클러스터의 경우 네트워크 버퍼 크기를 늘리십시오. 전송/수신 버퍼를 확대하기 위해 `/proc/sys/net/core/{r,w}mem_max` 및 자동 조정 한계값(`net.ipv4.tcp_{r,w}mem`)을 상향 조정하십시오. 이는 10/40/100GbE 링크 포화도를 높이는 데 도움이 됩니다. 모든 노드와 스위치에서 점보 프레임(MTU 9000)을 활성화하여 패킷당 오버헤드를 줄

이고, 이를 통해 처리량을 향상시키며 CPU 사용률을 낮추십시오. 광역 또는 혼잡한 네트워크의 경우 BBR과 같은 최신 TCP 혼잡 제어 방식을 고려하십시오.

NIC에 CPU 어피니티 할당

네트워크 인터럽트와 스레드를 NIC와 동일한 NUMA 노드의 CPU 코어에 고정하십시오. 이렇게 하면 네트워크 트래픽에 대한 크로스-NUMA 페널티를 피하고 네트워킹 스택의 메모리 액세스를 로컬로 유지할 수 있습니다. `/proc/interrupts` 를 확인하고 `irqaffinity` 설정을 사용하여, 예를 들어 NUMA 노드 0의 NIC가 NUMA 노드 0의 코어에서 처리되도록 하십시오. 이는 특히 높은 패킷 속도에서 네트워크 성능과 일관성을 향상시킬 수 있습니다.

환경에 맞게 NCCL 환경 변수를 최적화하십시오

대규모 다중 노드 작업에 대해 NCCL 매개변수를 실험해 보십시오. 예를 들어, NCCL의 GPU당 CPU 스레드 수인 ` `NCCL_NTHREADS` `를 기본값 4에서 8 또는 16으로 증가시켜 CPU 사용량 증가를 대가로 더 높은 대역폭을 확보할 수 있습니다. GPU당 버퍼 크기인 ` `NCCL_BUFFSIZE` `를 기본값 1MB에서 4MB 이상으로 증가시켜 대용량 메시지의 처리량을 향상시킬 수 있습니다. 클러스터가 SHARP 지원 스위치를 사용하는 경우, NCCL SHARP 플러그인을 설치하고 `NCCL_COLLNET_ENABLE=1` 설정으로 CollNet을 활성화한 후, 문서화된 대로 `SHARP_COLL_LOCK_ON_COMM_INIT=1` 및 `SHARP_COLL_NUM_COLL_GROUP_RESOURCE_ALLOC_THRESHOLD=0` 같은 SHARP 플러그인 변수를 사용하십시오. 리덕션 규모가 충분히 크고 네트워크 패브릭이 SHARP 오프로드를 지원하는 경우에만 속도 향상을 기대할 수 있습니다.

느린 네트워크에서는 기울기 누적을 사용하십시오

중간 성능의 상호 연결로 연결된 노드를 너무 많이 확장하여 네트워크가 병목 현상이 되는 경우, 그라디언트 누적을 사용하여 더 적고 더 큰 전체 축소(all-reduce) 작업을 수행하십시오. 동기화 전에 몇 개의 미니배치에 걸쳐 그라디언트를 누적하여 매 배치마다 통신하는 대신 *N배치에 대해 한* 번만 통신하도록 합니다. 이는 약간의 추가 메모리와 일부 모델 정확도 조정을 대가로 네트워크 오버헤드를 크게 줄여줍니다. 통신 비용으로 인해 GPU 추가 시 반환이 감소하는 경우 특히 유용합니다.

올-리듀스 토폴로지를 최적화하십시오

클러스터 토폴로지에 최적화된 올-리듀스 알고리즘을 사용 중인지 확인하십시오. NCCL은 링 또는 트리 알고리즘을 자동으로 선택하지만, 각 노드 내 GPU 간 NVLink와 노드 간 InfiniBand 또는 이더넷과 같은 혼합 인터커넥트 환경에서는 계층적 올-리듀스가 유리할 수 있습니다. 계층적 올-리듀스는 먼저 노드 내에서 올-리듀스 연산을 수행한 후 노드 간으로 진행합니다. 대부분의 프레임워크는 기본적으로 NCCL 기반 계층적 집계를 수행하지만 자질을 통해 확인하세요. 전통적인 MPI 환경에서는 동일한 2단계 리듀션(먼저 노드 내, 다음 노드 간)을 수동으로 수행하는 것을 고려할 수 있습니다.

네트워크 오버서브스크립션 방지

다중 GPU 서버에서는 GPU들의 합산 트래픽이 NIC 용량을 초과하지 않도록 해야 합니다. 예를 들어, 8개의 GPU는 올-리듀스 중 200Gbps 이상의 트래픽을 쉽게 생성할 수 있으므로, 100Gbps NIC 하나만으로는 성능이 제한됩니다. 노드당 다중 GPU로 확장할 경우 노드당 다중 NIC와 200/400Gbps 인피니밴드(InfiniBand)를 고려하십시오. 마찬가지로, NIC와 GPU가 동일한 PCIe 루트 컴플렉스를 공유하는 경우 PCIe 대역폭 제한에 주의하십시오.

통신 압축

단일 노드 메모리와 마찬가지로 네트워크 전송을 위한 데이터 압축을 고려하십시오. 16비트 또는 8비트 기울기 압축, 노드 간 파이프라인 전송을 위한 활성화 값 양자화, 스케칭과 같은 더 특수한 기법 등이 있습니다. 네트워크가 가장 느린 구성 요소라면 데이터 압축/해제에 소요되는 약간 높은 연산 비용도 감수할 가치가 있습니다. NVIDIA의 NCCL은 기본적으로 압축 기능을 제공하지 않지만, 프레임워크에 압축을 통합할 수 있습니다(예: Horovod의 그라디언트 압축 또는 PyTorch의 커스텀 AllReduce 후크). 이는 DeepSeek의 성공 비결 중 하나였습니다—제한된 노드 간 대역폭을 극복하기 위해 그라디언트를 압축한 것입니다.

네트워크 상태 모니터링

분산 훈련을 방해하는 숨겨진 문제가 없는지 확인하세요. 패킷 손실(재전송 또는 시간 초과로 나타남—InfiniBand에서는 재전송 카운터를, 이더넷에서는 TCP 재전송을 확인)을 점검하십시오. 작은 패킷 손실도 혼잡 제어 작동으로 인해 처리량을 심각하게 저하시킬 수 있습니다. 예상 대역폭과 지연 시간을 확보하고 있는지 검증하기 위해 대역 외 네트워크 테스트(iPerf 또는 NCCL 테스트 등)를 사용하십시오. 그렇지 않은 경우 스위치 구성, NIC 펌웨어 또는 CPU 어피니티를 조사하십시오.

효율적인 추론 및 서비스

서비스는 비용과 지연 시간의 균형 게임입니다. 활용도 는 단순히 더 큰 GPU가 아닌 오케스트레이션과 배치 처리를 통해 향상됩니다. 특수 런타임, KV 캐시 전략, 워밍업은 서비스 수준 목표(SLO)를 위반하지 않으면서 높은 처리량을 유지합니다.

자동 확장, 마이크로서비스, 동적/지속적 배치 처리를 통해 수요에 맞춰 오케스트레이션하여 지연 시간 SLO를 위반하지 않으면서 GPU를 지속적으로 가동 상태로 유지하세요. 전용 런타임(vLLM, SGLang, TensorRT-LLM)을 사용하고, NIXL 및 KV 캐시 오프로딩을 활용하여 분산 서비스, 모델 예열을 구현하고, 리소스를 분리하여 꼬리 지연 시간을 제어하세요. 모델 추론 효율성과 성능을 개선하려면 다음 기법을 따르세요:

동적 리소스를 효율적으로 오케스트레이션

커스텀 성능 지표로 강화된 쿠버네티스 같은 고급 컨테이너 오케스트레이션 플랫폼을 통합하세요. 이를 통해 실시간 사용 패턴과 처리량 목표에 기반한 동적 확장 및 워크로드 균형을 구현할 수 있습니다.

추론을 위한 서버리스 아키텍처 채택

추론 워크로드에 서버리스 아키텍처와 마이크로서비스 설계를 적용하세요. 이는 버스트성 트래픽을 효율적으로 처리하고 수요가 낮을 때 스케일 다운하여 유휴 리소스 오버헤드를 줄일 수 있습니다.

배치 및 동시성 최적화

추론 워크로드에 적합한 배치 전략을 찾으십시오. 추론 워크로드에는 동적 또는 연속 배치를 선호하여 들어오는 요청을 자동으로 배치하십시오. 더 큰 배치 크기는 GPU를 바쁘게 유지하여 처리량을 향상시키지만, 너무 크면 지연 시간이 증가할 수 있습니다. 또한 하나의 스트림이 모든 GPU 리소스를 사용하지 않는 경우(예: 두 개의 GPU SM과 텐서 코어를 모두 완전히 사용하기 위한 두 개의 동시 추론 배치) 여러 추론 스트림을 병렬로 실행하십시오.

분산 추론을 위한 NIXL 활용

GPU 또는 노드 간에 대규모 모델을 서비스할 때는 NVIDIA Inference Xfer Library를 사용하여 RDMA를 통해 프리필(prefill) 작업자와 디코드(decode) 작업자 간에 KV 캐시를 스트리밍하십시오. NIXL의 경우 대규모 트랜스포머 기반 KV 캐시가 노드 간에 전송됩니다. NIXL은 분산형 LLM

추론 클러스터에서 프리필 GPU에서 디코드 GPU로 KV 캐시를 스트리밍하기 위한 고처리량, 저지연 API를 제공합니다. 이는 GPUDirect RDMA와 최적 경로를 활용하며 CPU 개입 없이 수행됩니다. 이를 통해 노드 간 분산형 프리필-디코드 서비스의 꼬리 지연(tail latency)이 감소합니다.

필요한 경우 KV 캐시 오프로드

LLM의 어텐션 KV 캐시가 GPU 메모리를 초과할 경우 계층적 오프로딩을 사용하십시오. NVIDIA Dynamo의 분산 KV 캐시 관리자는 덜 자주 액세스되는 KV 페이지를 CPU 메모리, SSD 또는 네트워크 스토리지로 오프로드하며, TensorRT-LLM 및 vLLM과 같은 추론 엔진은 페이지화 및 양자화된 KV 캐시를 지원합니다. 캐시를 재사용하여 메모리 압박과 첫 번째 토큰 지연 시간을 낮추십시오. 오프로드된 미스(miss)는 추가 I/O 지연 시간을 유발하므로 엔드투엔드 영향을 검증하십시오. 이를 통해 GPU 메모리를 초과하는 시퀀스에 대한 추론이 가능해지며, 빠른 NVMe 및 컴퓨트-I/O 중첩 덕분에 성능 저하가 최소화됩니다. 매우 긴 prompt나 채팅을 예상하는 경우 추론 서버가 이를 사용하도록 구성되었는지 확인하십시오. 완전히 실패하는 것보다 디스크로 오프로드하는 것이 낫습니다.

모델을 효율적으로 제공

vLLM, SGLang, NVIDIA Dynamo, NVIDIA TensorRT-LLM과 같은 최적화된 모델 추론 시스템을 사용하여 저지연 및 고처리량으로 대규모 모델을 서비스하십시오. 이들은 양자화, 저정밀도 형식, 퓨전, 고도로 최적화된 어텐션 커널 및 기타 기법을 구현하여 추론 중 GPU 활용도를 극대화해야 합니다. 이러한 라이브러리는 텐서 병렬 처리, 파이프라인 병렬 처리, 전문가 병렬 처리, 컨텍스트 병렬 처리, 추측적 디코딩, 청크별 사전 채우기, 분리된 사전 채우기/디코딩, 동적 요청 배치 등 다양한 고성능 기능을 처리해야 합니다.

꼬리 지연 시간 모니터링 및 조정

실시간 서비스에서는 평균 지연 시간과 (긴) 꼬리 지연 시간(99번째 백분위수)이 모두 중요합니다. 추론 지연 시간 분포의 자질을 확인하십시오. 꼬리 부분이 높다면 예상치 못한 CPU 개입, 가비지 컬렉션(GC) 일시 중지, 과도한 컨텍스트 스위치 등 이상치 원인을 파악하십시오. 추론 서버 프로세스를 특정 코어에 고정하고, 노이즈가 많은 이웃 프로세스로부터 격리하며, 필요한 경우 실시간 스케줄링을 사용하여 더 일관된 지연 시간을 확보하십시오.

콜드 스타트 지연 방지 위한 워밍업

모델을 GPU에 로드하고 몇 번의 더미 추론을 실행하여 GPU를 예열하세요. 이렇게 하면 추론 서버에 첫 번째 실제 요청이 들어올 때 발생하는 일회성 콜드 스타트 지연 문제를 방지할 수 있습니다.

서비스 품질(QoS)을 위한 효율적인 리소스 분할

동일한 인프라에서 훈련과 추론 같은 혼합 이종 워크로드 또는 서로 다른 아키텍처의 모델을 실행하는 경우, 지연 시간에 민감한 추론 작업이 우선순위를 확보할 수 있도록 리소스를 분할하는 것을 고려하십시오. 이는 일부 GPU를 추론 전용으로 할당하거나, 전체 GPU가 필요하지 않지만 예측 가능한 지연 시간이 필요한 추론 서비스에 MIG를 사용하여 GPU의 보장된 슬라이스를 제공하는 것을 의미할 수 있습니다. 가능하다면 추론과 훈련을 서로 다른 노드에서 분리하십시오. 훈련은 대용량 I/O 작업이나 갑작스러운 통신 급증으로 지터를 유발할 수 있습니다.

추론 전처리 작업에 그레이스 CPU 활용

Grace Blackwell 시스템에서는 서버급 CPU가 GPU와 동일한 메모리 공간에서 토큰화 및 배치 정렬과 같은 전처리 작업을 매우 빠르게 처리할 수 있습니다. 이러한 작업을 CPU로 오프로드하여 GPU가 직접 사용할 수 있는 공유 메모리에서 데이터를 준비하도록 하십시오. 이를 통해 버퍼 중복을 줄이고 강력한 CPU를 활용하여 추론 파이프라인의 일부를 처리함으로써 GPU가 더 많은 연산 집약적인 신경망 계산에 집중할 수 있도록 합니다.

에지 AI 및 지연 시간에 민감한 배포를 위해 신중하게 조정

특화된 엣지 가속기를 활용하고 중앙 서버와 엣지 장치 간 데이터 전송 프로토콜을 최적화하여 성능 조정을 엣지까지 확장하십시오. 이는 시간에 민감한 애플리케이션에 초저지연성을 달성하는 데 도움이 됩니다.

다중 노드 추론 및 서비스

프리필/디코딩(prefill/decode) 및 디코딩 후 처리( ) 모델 분할을 분리하면 더 큰 컨텍스트와 더 많은 사용자를 높은 점유율로 처리할 수 있습니다. 연속 배치 처리와 계층적 메모리/오프로딩은 긴 prompt와 높은 동시성에서도 흐름을 유지합니다.

구체적으로, 프리필과 디코드를 여러 장치에 분산하고, 요청 간 토큰을 지속적으로 풀링하며, 텐서/파이프라인 병렬 처리를 통해 대형 모델을 분할하세요. 매우 긴 컨텍스트를 위해 계층적 메모리/오프로드를 추가하면 OOM 없이 더 많은 서

비스를 제공할 수 있으며, 약간의 지연 시간을 희생하여 훨씬 더 높은 용량을 확보할 수 있습니다. 다음 성능 팁은 다중 노드 추론 및 서비스에 적용됩니다:

추론 파이프라인 분할

추론 워크플로를 별개의 단계로 분리합니다. 여기에는 입력 prompt를 모든 모델 레이어를 통해 처리하는 "프리필(prefill)" 단계와 토큰 단위로 출력을 생성하는 반복적 "디코드(decode)" 단계가 포함됩니다. 이러한 단계들을 서로 다른 리소스에 할당하여 독립적인 확장이 가능하도록 합니다. 이 2단계 접근 방식은 빠른 작업이 느린 작업에 의해 병목 현상이 발생하는 것을 방지합니다. 대규모 언어 모델의 경우, prompt 인코딩을 위해 전체 모델을 실행한 후 단계별로 자동회귀 디코딩을 처리하는 전략이 있습니다. 각 단계에 특화된 작업자를 할당할 수도 있습니다. 파이프라인을 분할함으로써 GPU가 가장 효율적인 작업 부분에 지속적으로 집중하도록 하여, 한 번의 긴 생성 작업이 뒤따르는 작업을 지연시키는 헤드오브라인 차단 현상을 방지할 수 있습니다.

LLMs에 연속 배치 처리 사용

단순한 요청 배치 처리에서 나아가 연속 배치 전략을 활용하여 고부하 환경에서 처리량을 극대화하십시오. 기존 동적 배치 방식은 유입되는 요청을 그룹화하여 일괄 처리함으로써 GPU 활용도를 개선합니다. 지속적 배치 처리는 이를 한 단계 발전시켜 요청 간 토큰 시퀀스를 실시간으로 동적으로 병합 및 분할합니다. vLLM과 같은 시스템은 토큰 풀링을 구현하는데, 이는 어떤 스레드든 다음 토큰 생성이 준비되는 즉시 다른 준비된 스레드와 그룹화되어 새 배치를 형성합니다. 이 접근법은 GPU를 항상 높은 점유율로 유지하고 유휴 시간을 대폭 줄입니다. 그 결과, 특히 다양한 시퀀스 길이를 가진 다수의 동시 사용자에게 서비스를 제공할 때 토큰 처리량이 크게 향상되고 지연 시간 일관성이 개선됩니다.

GPU 및 노드 간 효율적인 모델 분할

단일 GPU 메모리에 수용하기에 너무 큰 모델의 경우, 모델을 여러 GPU 또는 여러 서버에 분할하여 모델 병렬 추론 기법을 사용하십시오. 이는 텐서 병렬 처리(tensor parallelism)를 통해 수행할 수 있으며, 이는 각 레이어의 가중치와 계산을 여러 장치에 분할하거나, 파이프라인 병렬 처리(pipeline parallelism)를 통해 수행할 수 있으며, 이는 모델의 레이어를 서로 다른 GPU에 호스팅되는 세그먼트로 분할하고 데이터를 순차적으로 스트리밍합니다. 모델 샤딩은 데이터가 샤드 간에 이동해야 하므로 통신 오버헤드와 일부 추가 지연 시간을 발생시키지만, 그렇지 않으면 서비스가 불가능한 수조 개 매개변수 규모의 모델 배포를 가능하게 합니다. 이를 실현하려면 GPU 간에 NVLink 또는 InfiniBand와 같은 고속 상호 연결을 확

보하고, 가능한 경우 통신과 계산을 중첩시켜 수행하십시오. 핵심은 모든 장치가 병렬로 작동하고 단일 단계가 병목 현상이 되지 않도록 부하를 균형 있게 분배하는 것입니다.

확장된 컨텍스트를 위한 메모리 오프로드

계층적 메모리 전략을 활용하여 GPU가 보유한 메모리보다 더 많은 메모리를 요구하는 추론 워크로드를 지원하십시오. 대규모 모델이나 긴 시퀀스 컨텍스트(예: 다중 회화나 대용량 문서)를 처리할 때는 메모리 오프로딩을 적용하십시오. 오래된 어텐션 키-값 캐시 항목이나 접근 빈도가 낮은 모델 가중치처럼 사용 빈도가 낮은 데이터는 GPU 메모리가 부족해지면 CPU RAM이나 NVMe 스토리지로 이동시킬 수 있습니다. 현대적 추론 프레임워크는 이러한 텐서를 자동으로 스왑아웃하고 필요 시 즉시 다시 불러올 수 있습니다. 이는 캐시 미스 시 추가 지연 시간을 발생시키지만, 메모리 부족 오류를 방지하고 극한 상황을 처리할 수 있게 합니다. 신중하게 데이터를 오프로딩하고 프리페칭함으로써, 속도 일부를 희생하는 대신 대규모 작업 세트를 가진 요청을 처리할 수 있는 능력을 확보하여 메모리 제약 하에서 더 나은 전체 처리량을 달성합니다.

전력 및 열 관리

와트당 성능은 전력 효율성( )의 핵심 지표입니다. 열 또는 전력 제한은 튜닝 효과를 무효화하고 하드웨어 수명을 단축시킵니다. 전력 제한, 효율적 패킹, 사전적 냉각은 클럭을 안정화시키면서 에너지 소비를 절감합니다.

속도와 함께 와트당 성능 및 열 특성을 추적하세요: 전력 제한 또는 메모리 바운디드 워크로드의 클럭 속도 저하를 통해 최소한의 처리량 손실로 효율성을 개선합니다. 냉각을 선제적으로 관리하고, 작업을 통합하여 GPU를 거의 최대 용량으로 가동하며, GPU별 전력 소모를 모니터링하고, 비용 절감 효과가 있을 경우 에너지 가격/재생 에너지 요금을 고려하여 스케줄링하세요. AI 시스템의 전력 및 열 특성 관리에 대한 몇 가지 팁은 다음과 같습니다:

가능한 경우 효율적이고 환경 친화적인 에너지를 활용하십시오

성능과 함께 에너지 소비를 추적하고 최적화하십시오. 전력 및 열 제한 관리 외에도 에너지 사용 지표를 모니터링하고 성능과 지속 가능성을 동시에 개선하는 기술을 고려하십시오. 예를 들어 재생 에너지 가용성에 기반한 동적 전력 제한 또는 워크로드 이동을 구현하면 운영 비용과 탄소 발자국을 줄일 수 있습니다. 이러한 이중 초점은 운영 비용을 절감하고 책임감 있고 환경 친화적인 AI 배포를 지원합니다.

열 및 클럭 모니터링

실행 중 GPU 온도와 클럭 주파수를 주시하십시오. GPU가 열 한계(경우에 따라 85°C)에 근접하면 클럭을 저하시키기 시작하여 성능이 저하될 수 있습니다. `nvidia-smi dmon` 또는 원격 측정 기능을 사용하여 클럭이 최대값에서 떨어지는지 확인하십시오. 저하가 감지되면 냉각을 개선하거나 팬 속도를 높이거나 공기 흐름을 개선하거나 전력 제한을 약간 낮추어 안정적인 열 범위 내에서 유지하십시오. 목표는 열로 인한 성능 저하 없이 일관된 성능을 유지하는 것입니다.

에너지 인식 동적 전력 관리 사용

현대 데이터센터는 실시간 에너지 비용과 재생 에너지 가용성에 따라 워크로드를 조정하는 에너지 인식 스케줄링을 점점 더 많이 활용하고 있습니다. 적응형 전력 제한 및 동적 클럭 스케일링을 통합하면 와트당 처리량을 최적화하면서 운영 비용과 탄소 발자국을 줄일 수 있습니다.

와트당 성능 최적화

전력 예산이 제한적이거나(또는 에너지 비용이 높은) 다중 GPU 배포 환경에서는 효율성 조정을 고려하십시오. 많은 워크로드, 특히 메모리 바운디드 워크로드는 성능 손실은 미미하지만 전력 소모는 현저히 낮아지도록 GPU 클럭을 약간 낮춰 실행할 수 있습니다. 예를 들어 커널이 메모리 바운디드 상태라면 GPU를 낮은 클럭으로 고정해도 실행 시간에 영향을 주지 않으면서 전력을 절약할 수 있습니다. 이는 와트당 처리량을 증가시킵니다. `nvidia-smi -pl` 를 사용하여 몇 가지 전력 제한을 테스트하여 와트당 처리량이 개선되는지 확인하십시오. 일부 모델의 경우 전력 제한을 100%에서 80%로 낮추면 전력 사용량을 20% 줄이면서도 거의 동일한 속도를 유지할 수 있습니다.

적응형 냉각 전략 사용

냉각 또는 에너지 공급이 변동하는 환경에서 운영할 경우 클러스터 관리와 연동하여 워크로드를 조정하세요. 예를 들어, 비용 요인이 된다면 하루 중 더 시원한 시간대나 재생 에너지 공급량이 높은 시간에 중량 작업을 스케줄링하세요. 일부 사이트에서는 전기 요금이 저렴한 야간에 실행하도록 비긴급 작업을 대기열에 넣는 정책을 시행합니다. 이는 단일 작업 성능에는 영향을 주지 않으면서 비용을 크게 절감합니다.

워크로드를 통합하십시오

GPU를 낮은 활용도로 다수 가동하기보다 높은 활용도로 가동하십시오. 바쁜 GPU는 유휴 상태나 낮은 활용도의 GPU보다 와트당 작업량 측면에

서 에너지 효율이 더 높습니다. 이는 GPU가 바쁠 때 기본 전력 소모가 더 효율적으로 분산되기 때문입니다. 최소 실행 시간 최적화가 필요하지 않은 한, 두 개의 GPU를 각각 45% 활용도로 병렬 가동하는 것보다 하나의 GPU를 90% 활용도로 연속 작업하는 것이 더 나을 수 있습니다. 사용하지 않을 때는 많은 하드웨어를 낮은 활용도로 가동하는 것보다 전체 노드를 꺼두거나 유휴 상태로 전환하도록 스케줄링을 계획하십시오.

효율적인 냉각 구성

공랭식 시스템의 경우, GPU에 사전 냉각을 위해 고부하 작업 시 GPU 팬을 고정된 높은 속도로 설정하는 것을 고려하십시오. 일부 데이터 센터는 일관성 향상을 위해 팬을 항상 최대 속도로 가동합니다. 데이터 센터의 흡기 온도가 사양 범위 내에 있는지 확인하십시오. 서버 GPU의 먼지나 장애물을 주기적으로 점검하십시오. 막힌 핀은 냉각 효율을 크게 저하시킬 수 있습니다. 수냉식 시스템의 경우 유량이 최적화되고 수온이 제어되는지 확인하십시오.

전력을 주의 깊게 모니터링

GPU별 전력 소모량을 모니터링하는 도구를 활용하십시오. `nvidia-smi` 는 순간 전력 소모량을 보고하여 워크로드의 전력 자질을 이해하는 데 도움이 됩니다. 전력 급증은 특정 단계와 연관될 수 있습니다. 예를 들어, 올-리듀스(all-reduce) 단계는 계산 부하와 전력 소모가 적게 측정되는 반면, 밀집 레이어(dense layer)는 부하와 전력 측정값을 급증시킵니다. 이를 인지하면 워크로드 순서를 조정하여 전력 소모를 평준화할 수 있습니다. 이는 전력 제한 회로에서 클러스터를 운영할 때 중요합니다. 전력 제약 시나리오에서는 전력 한도 초과를 방지하기 위해 동일한 노드에서 전력 급증 작업들을 동시에 실행하지 않도록 주의해야 합니다.

장시간 실행 작업의 복원력 향상

수개월에 걸친 훈련 작업이나 24시간 연중무휴 추론 작업을 실행하는 경우, 열 관리가 하드웨어 수명에 미치는 영향을 고려하십시오. 지속적으로 100% 전력 및 열 한계로 작동하면 시간이 지남에 따라 고장 위험이 미세하게 증가할 수 있습니다. 실제로 데이터센터 GPU는 이러한 유형의 내구성을 위해 설계되었지만, 추가적인 안전을 원한다면 90% 전력 목표로 실행하면 최소한의 속도 저하로 부품 스트레스를 줄일 수 있습니다. 이는 긴 훈련 실행 시간과 하드웨어 마모 감소 사이의 절충점입니다. 특히 해당 하드웨어를 장기간에 걸쳐 여러 프로젝트에 재사용할 경우 더욱 그렇습니다.

결론

이 체크리스트를 반복 가능한 실행 지침으로 활용하십시오: 자질을 수행하고, 적절한 계층에서 올바른 병목 현상을 조정하며, 확장 전에 개선 효과를 검증하십시오. OS와 커널부터 분산 통신 및 서비스에 이르기까지 이러한 관행을 체계적으로 적용함으로써, 규모에 관계없이 빠르고 비용 효율적이며 신뢰할 수 있는 AI 시스템을 구축할 수 있습니다.

이 목록은 포괄적이지만 완전하지는 않습니다. 하드웨어, 소프트웨어, 알고리즘이 진화함에 따라 AI 시스템 성능 엔지니어링 분야도 계속 성장할 것입니다. 여기에 나열된 모든 모범 사례가 모든 상황에 적용되는 것은 아닙니다. 그러나 종합적으로 보면 AI 시스템 성능 엔지니어링 시나리오의 폭넓은 영역을 다루고 있습니다. 이 팁들은 수년간 AI 시스템 성능 최적화를 통해 축적된 실용적 지혜의 상당 부분을 함축하고 있습니다.

AI 시스템을 튜닝할 때는 본 장에 나열된 관련 범주를 체계적으로 검토하고 체크리스트의 각 항목을 실행해야 합니다. 예를 들어, OS가 튜닝되었는지 확인하고, GPU 커널이 효율적인지 검증하며, 라이브러리를 올바르게 사용하고 있는지 점검하고, 데이터 파이프라인을 모니터링하고, 훈련 루프를 최적화하고, 추론 전략을 튜닝하고, 원활하게 확장해야 합니다. 이러한 모범 사례를 따르면 대부분의 성능 문제를 진단 및 해결하고 AI 시스템에서 최대 성능을 끌어낼 수 있습니다.

클러스터를 대폭 확장하기 전에 소규모 노드에서 자질을 수행하여 잠재적인 확장 병목 현상을 파악해야 한다는 점을 기억하십시오. 예를 들어, 8개의 GPU에서 이미 반복의 20%를 차지하는 all-reduce 집단 연산이 있다면, 특히 Grace Blackwell GB200 및 GB300 NVL72, Vera Rubin VR200 및 VR300 NVL 시스템과 같은 단일 컴퓨팅 노드 또는 데이터 센터 랙 시스템의 용량을 초과할 경우 더 큰 규모에서는 상황이 악화될 뿐입니다.

이 체크리스트를 손에 쥐고 새로운 노하우를 발견할 때마다 추가하세요. 이 팁과 모범 사례를 앞선 장에서 얻은 심층적 이해와 결합하면 효율적이고 확장 가능하며 유지 보수성이 뛰어나며 비용 효율적이며 신뢰할 수 있는 AI 시스템을 설계하고 운영할 수 있을 것입니다.

이제 나아가 가장 야심찬 아이디어를 현실로 만들어 보십시오. 최적화를 즐기세요!